PHYSICALLY-BASED IMAGE SYNTHESIS: FROM THEORY TO IMPLEMENTATION

Matt Pharr and Greg Humphreys

Contents

[Just as] other information should be available to those who want to learn and understand, program source code is the only means for programmers to learn the art from their predecessors. It would be unthinkable for playwrights not to allow other playwrights to read their plays [and] only be present at theater performances where they would be barred even from taking notes. Likewise, any good author is well read, as every child who learns to write will read hundreds of times more than it writes. Programmers, however, are expected to invent the alphabet and learn to write long novels all on their own. Programming cannot grow and learn unless the next generation of programmers have access to the knowledge and information gathered by other programmers before them.

— Erik Naggum

Preface

Rendering is a fundamental component of computer graphics. At the highest level of abstraction, rendering describes the process of converting a description of a three-dimensional scene into an image. Algorithms for animation, geometric modeling, texturing, and other areas of computer graphics all must feed their results through some sort of rendering process so that the results of their work are made visible in an image. Rendering has become ubiquitous; from movies to games and beyond, it has opened new frontiers for creative expression, entertainment, and visualization.

In the early years of the field, research in rendering focused on solving fundamental problems such as determining which objects are visible from a given viewpoint. As these problem have been solved and as richer and more realistic scene descriptions have become available, modern rendering has grown to be built on ideas from a broad range of disciplines, including physics and astrophysics, astronomy, biology, psychology and the study of perception, and pure and applied mathematics. The interdisciplinary nature is one of the reasons rendering is such a fascinating area to study.

This book presents a selection of modern rendering algorithms through the documented source code for a complete rendering system. All of the images in this book, including the ones on the front and back covers, were rendered by this software. The system, `lrt`, is written using a programming methodology called *literate programming* that mixes prose describing the system with the source code that implements it. We believe that the literate programming approach is a valuable way to introduce ideas in computer science and computer graphics. Often, some of the subtleties of an algorithm can be missed until it is implemented; seeing someone else's implementation is a good way to acquire a solid understanding of an algorithm's details. Indeed, we believe that deep understanding of a smaller number of algorithms provides a stronger base for further study of graphics than superficial understanding of many.

Not only does reading an implementation help clarify how an algorithm is implemented in practice, but by showing these algorithms in the context of a complete and non-trivial software system we are also able to address issues in the design and implementation of medium-sized rendering systems. The design of the basic abstractions and interfaces of such a system has substantial implications for how cleanly algorithms can be expressed in it as well as how well it can support later addition of new techniques, yet the trade-offs in this design space are rarely discussed.

`lrt` and this book focus exclusively on so-called *photorealistic rendering*, which can be defined variously as the task of generating images that are indistinguishable from those that a camera would capture taking a photograph of the scene, or as the task of generating an image that evokes the same response from a human observer when displayed as if the viewer was looking at the actual scene. There are many reasons to focus on photorealism. Photorealistic images are necessary for much of the rendering done by the movie special effects industry, where computer generated imagery must be mixed seamlessly with footage of the real world. For other entertainment applications where all of the imagery is synthetic, photorealism is an effective tool to make the observer forget that he or she is looking at an environment that may not actually exist. Finally, photorealism gives us a reasonably well-defined metric for evaluating the quality of the output of the rendering system.

A consequence of our approach is that this book and the system it describes do not exhaustively cover the state-of-the-art in rendering; many interesting topics in photorealistic rendering will not be covered either because they didn't fit well with the architecture of the software system (e.g. finite element radiosity algorithms), or because we believed that the pedagogical value of explaining the algorithm was outweighed by the complexity of its implementation (e.g. Metropolis light transport). We will note these decisions as they come up and provide pointers to further resources so the reader can follow up on topics that are of interest. Many other areas of rendering, such as interactive rendering, visualization, and illustrative forms of rendering (e.g. pen-and-ink styles) aren't covered in this book at all.

Intended Audience

Our primary intended audience is students in upper-level undergraduate or graduate-level computer graphics classes. This book assumes existing knowledge of computer graphics at the level of an introductory college-level course, though certain key concepts from such a course will be presented again here, such as basic vector geometry and transformations. For students who do not have experience with programs that have tens of thousands of lines of source code, the literate programming style gives a gentle introduction to this complexity. We have paid special attention to explaining the reasoning behind some of the key interfaces and abstractions in the system in order to give these readers a sense of why the system was structured the way that it was.

Our secondary, but equally important, audiences are advanced graduate students and researchers, software developers in industry, and individuals interested in the fun of writing their own rendering systems. Though many of the ideas in this manuscript will likely be familiar to these readers, reading explanations of the algorithms we describe in the literate style may provide new perspectives. `lrt` also includes implementations of a number of newer and/or difficult-to-implement algorithms and techniques, including subdivision surfaces, Monte Carlo light transport, and volumetric scattering models; these should be of particular interest even to experienced practitioners in rendering. We hope that it will also be useful for this audience to see one way to organize a complete non-trivial rendering system.

Overview and Goals

`lrt` is based on the *ray tracing* algorithm. Ray tracing is an elegant technique that has its origins in lens-making; Gauss traced rays through lenses by hand in the 1800s. Ray tracing algorithms on computers follow the path of infinitesimal rays of light through the scene up to the first surface that they intersect. This gives a very basic method for finding the first visible object as seen from any particular position and direction. It is the basis for many rendering algorithms.

`lrt` was designed and implemented with three main goals in mind: it should be *complete*, it should be *illustrative*, and it should be *physically based*.

Completeness implies that the system should not lack important features found in high-quality commercial rendering systems. In particular, it means that important practical issues, such as antialiasing, robustness, and the ability to efficiently render complex scenes should be addressed thoroughly. It is important to face these issues from the start of the system's design, since it can be quite difficult to retrofit such functionality to a rendering system after it has been implemented, as these features can have subtle implications for all components of the system.

Our second goal means that we tried to choose algorithms, data structures, and rendering techniques with care. Since their implementations will be examined by more readers than those in most rendering systems, we tried to select the most elegant algorithms that we were aware of and implement them as well as possible. This goal also implied that the system should be small enough for a single person to understand completely. We have implemented `lrt` with a plug-in architecture, with a core of basic glue that pushes as much functionality as possible out to external modules. The result is that one doesn't need to understand all of the various plug-ins in order to understand the basic structure of the system. This makes it easier to delve in deeply to parts of interest and skip others, without losing sight of how the overall system fits together.

There is a tension between the goals of being both complete and illustrative. Implementing and describing every useful technique that would be found in a production rendering system would not only make this book extremely long, but it would make the system more complex than most readers would be interested in. In cases where `lrt` lacks such a useful feature, we have attempted to design the architecture so that feature could be easily added without altering the overall system design. Exercises at the end of each chapter suggest programming projects that add new features to the system.

The basic foundations for physically-based rendering are the laws of physics and their mathematical expression. lrt was designed to use the correct physical units and concepts for the quantities that it computes and the algorithms it is built from. When configured to do so, lrt can compute images that are *physically correct*; they accurately reflect the lighting as it would be in a real-world scene corresponding to the one given to the renderer. One advantage of the decision to use a physical basis is that it gives a concrete standard of program correctness: for simple scenes, where the expected result can be computed in closed-form, it lrt doesn't compute the same result, we know that it must have a bug. Similarly, if different physically-based lighting algorithms in lrt give different results for the same scene, or if lrt doesn't give the same results as another physically based renderer, there is certainly an error in one of them. Finally, we believe that this physically-based approach to rendering is valuable because it is rigorous. When it is not clear how a particular computation should be performed, physics gives an answer that guarantees a consistent result.

Efficiency was secondary to these three goals. Since rendering systems often run for many minutes or hours in the course of generating an image, efficiency is clearly important. However, we have mostly confined ourselves to *algorithmic* efficiency rather than low-level code optimization. In some cases, obvious micro-optimizations take a back seat to clear, well-organized code, though we did make some effort to optimize the parts of the system where most of the computation occurs. For this reason as well as portability, lrt is not presented as a parallel or multi-threaded application, although parallelizing lrt would not be very difficult.

In the course of presenting lrt and discussing its implementation, we hope to convey some hard-learned lessons from some years of rendering research and development. There is more to writing a good renderer than stringing together a set of fast algorithms; making the system both flexible and robust is the hard part. The system's performance must degrade gracefully as more geometry is added to it, as more light sources are added, or as any of the other axes of complexity are pushed. Numeric stability must be handled carefully; stable algorithms that don't waste floating-point precision are critical.

The rewards for going through the process of developing a rendering system that addresses all of these issues are enormous–writing a new renderer or adding a new feature to an existing renderer and using it to create an image that couldn't be generated before is a great pleasure. Our most fundamental goal in writing this book was to bring the opportunity to do this to a wider audience. You are encouraged to use the system to render the example scenes on the companion CD as you progress through the book. Exercises at the end of each chapter suggest modifications to make to the system that will help you better understand its inner workings and more complex projects to extend the system to add new features.

We have also created a web site to go with this book, located at www.pharr.org/lrt. There you will find errata and bug fixes, updates to lrt's source code, additional scenes to render, supplemental utilities, and new plug-in modules. If you come across a bug in lrt or an error in this text that is not listed at the web site, please report it to the e-mail address lrtbugs@pharr.org.

Acknowledgments

Additional Reading

Donald Knuth's article *Literate Programming* (Knuth 1984) describes the main ideas behind literate programming as well as his `web` programming environment. The seminal TEX typesetting system was written with this system and has been published as a series of books (Knuth 1993a; Knuth 1986). More recently, Knuth has published a collection of graph algorithms in *The Stanford Graphbase* (Knuth 1993b). These programs are enjoyable to read and are respectively excellent presentations of modern automatic typesetting and graph algorithms. The website `www.literateprogramming.com` has pointers to many articles about literate programming, literate programs to download as well as a variety of literate programming systems; many refinements have been made since Knuth's original development of the idea.

The only other literate program that we are aware of that has been published as a book is the implementation of the `lcc` C compiler, which was written by Fraser and Hansen and published as *A Retargetable C Compiler: Design and Implementation* (Fraser and Hanson 1995). **Say something nice about this book**

1. Introduction

This chapter provides a high-level top-down description of `lrt`'s basic architecture. It starts by explaining more about the literate programming approach and how to read a literate program. We then briefly describe our coding conventions before moving forward into the high-level operation of `lrt`, where we describe what happens during rendering by walking through the process of how `lrt` computes the color at a single point on the image. Along the way we introduce some of the major classes and interfaces in the system. Subsequent chapters will describe these and other classes and their methods in detail.

1.1 Approaching the System

1.1.1 Literate Programming

In the course of the development of the TEX typesetting system, Donald Knuth developed a new programming methodology based on the simple (but revolutionary) idea that *programs should be written more for people's consumption than for computers' consumption.* He named this methodology *literate programming*. This book (including the chapter you're reading now) is a long literate program.

Literate programs are written in a meta-language that mixes a document formatting language (e.g. LaTeX or HTML) and a programming language (e.g. C++). The meta-language compiler then can transform the literate program into either a document suitable for typesetting (this process is generally called *weaving*, since Knuth's original literate programming environment was called `web`), or into source code suitable for compilation (so-called *tangling*, since the resulting source code is not generally as comprehensible to a human reader than the original literate program was).

The literate programming meta-language provides two important features. The first is a set of mechanisms for mixing English text with source code. This makes the description of the program just as important as its actual source code, encouraging careful design and documentation on the part of the programmer. Second, the language provides mechanisms for presenting the program code to the reader in an entirely different order than it is supplied to the compiler. This feature makes it possible to describe the operation of the program in a very logical manner. Knuth named his literate programming system `web` since literate programs tend to have the form of a web: various pieces are defined and inter-related in a variety of ways such that programs are written in a structure that is neither top-down nor bottom-up.

As a simple example, consider a function `InitGlobals()` that is responsible for initializing all of the program's global variables. If all of the variable initializations are presented to the reader at once, `InitGlobals()` might be a large collection of variable assignments the meanings of which are unclear because they do not appear anywhere near the definition or use of the variables. A reader would need to search through the rest of the entire program to see where each particular variable was declared in order to understand the function and the meanings of the values it assigned to the variables. As far as the human reader is concerned, it would be better to present the initialization code near the code that actually declares and uses the global.

In a literate program, then, one can instead write `InitGlobals()` like this:

```
⟨Function Definitions⟩≡
  void InitGlobals() {
      ⟨Initialize Global Variables⟩
  }
```

Here we have added text to a *fragment* called ⟨*Function Definitions*⟩. (This fragment will be included in a C++ source code file when the literate program is tangled for the compiler.) The fragment contains the definition of the `InitGlobals()` function. The `InitGlobals()` function itself includes another fragment, ⟨*Initialize Global Variables*⟩. At this point, no text has been added to the initialization fragment. However, when we introduce a new global variable `ErrorCount` somewhere later in the program, we can now write:

```
⟨Initialize Global Variables⟩≡
  ErrorCount = 0;
```

Here we have started to define the contents of ⟨*Initialize Global Variables*⟩. When our literate program is turned into source code suitable for compiling, the literate programming system will substitute the code `ErrorCount = 0;` inside the definition of the `InitGlobals()` function. Later on, we may introduce another global `FragmentsProcessed`, and we can append it to the fragment:

⟨*Initialize Global Variables*⟩+≡

```
  FragmentsProcessed = 0;
```

The +≡ symbol after the fragment name shows that we have added to a previously defined fragment. When tangled, the result of the above fragment definitions is the code:

```
void InitGlobals() {
  ErrorCount = 0;
  FragmentsProcessed = 0;
}
```

By making use of the text substitution that is made easy by fragments, we can decompose complex functions into logically-distinct parts. This can make their operation substantially easier to understand. We can write a function as a series of fragments:

⟨*Function Definitions*⟩+≡

```
  void func(int x, int y, double *data) {
        ⟨Check validity of arguments⟩
        if (x < y) {
              ⟨Swap parameter values⟩
        }
        ⟨Do precomputation before loop⟩
        ⟨Loop through and update data array⟩
  }
```

The text of each fragment is then expanded inline in `func()` for the compiler. In the document, we can introduce each fragment and its implementation in turn–these fragments may of course include additional fragments, etc. This style of decomposition lets us write code in collections of just a handful of lines at a time, making it easier to understand in detail. Another advantage of this style of programming is that by separating the function into logical fragments, each with a single and well-delineated purpose, each one can then be written and verified independently–in general, we will try to make each fragment less than ten lines or so of code, making it easier to understand its operation.

Of course, inline functions could be used to similar effect in a traditional programming environment, but using fragments to decompose functions has a few important advantages. The first is that all of the fragments can immediately refer to all of the parameters of the original function as well as any function-local variables that are declared in preceeding fragments; it's not necessary to pass them all as parameters, as would need to be done with inline functions. Another advantage is that one generally names fragments with more descriptive and longer phrases than one gives to functions; this improves program readability and understandability. Because it's so easy to use fragments to decompose complex functions, one does more decomposition in practice, leading to clearer code.

In some sense, the literate programming language is just an enhanced macro substitution language tuned to the task of rearranging program source code provided by the user. The simplicity of the task of this program can belie how different literate programming is from other ways of structuring software systems.

1.1.2 Coding Conventions

We have written `lrt` in C++. However, we have used a subset of the language, both to make the code easier to understand, as well as to improve the system's portability. In particular, we have avoided multiple inheritance and run-time exception handling and have used only a subset of C++'s extensive standard library. Appendix A.1 reviews the parts of the standard library that `lrt` uses in multiple places; otherwise we will point out and document unusual library routines as they are used.

Types, objects, functions, and variables are named to indicate their scope; classes and functions that have global scope all start with capital letters. (The system uses no global variables.) The names of small utility classes, module-local `static` variables, and private member functions start with lower-case letters.

We will occasionally omit short sections of `lrt`'s source code from this document. For example, when there are a number of cases to be handled, all with nearly identical code, we will present one case and note that the code for the remaining cases has been elided from the text.

1.1.3 Code Optimization

As mentioned in the preface, we have tried to make `lrt` efficient by using well-chosen algorithms rather than by having many low-level optimizations. However, we have used a profiler to find which parts of it account for most of the execution time and have performed local optimization of those parts when doing so didn't make the code confusing. We kept a handful of basic optimization principles in mind while doing so:

- On current CPU architectures, the slowest mathematical operations are divides, square-roots, and trigonometric functions. Addition, subtraction, and multiplication are generally ten to fifty times faster than those operations. Code changes that reduce the number of the slower mathematical operations can help performance substantially; for example, replacing a series of divides by a value v with the computing the value $1/v$ and then multiplying by that value.

- Declaring short functions as `inline` can speed up code substantially, both by removing the run-time overhead of performing a function call (which may involve saving values in registers to memory) as well as by giving the compiler larger basic blocks to optimize.

- As the speed of CPUs continues to grow more quickly than the speed at which data can be loaded from main memory into the CPU, waiting for values from memory is becoming a major performance barrier. Organizing algorithms and data structures in ways that give good performance from memory caches can speed up program execution much more than reducing the total number of instructions to be executed. Appendix **??** discusses general principles for memory-efficient programming; these ideas are mostly applied in the ray–intersection acceleration structures of Chapter 4 and the image map representation in Section 11.5.2, though they influence many of the design decisions throughout the system.

1.1.4 Indexing and Cross-Referencing

There are a number of features of the text designed to make it easier to navigate.

Indices in the page margins give the page number where the functions, variables, and methods used in the code on that page are defined (if not on the current or facing page). This makes it easier to refer back to their definitions and descriptions, especially when the book isn't read fromt-to-back. Indices at the end of the book collect all of these identifiers so that it's possible to find definitions starting from their names. Another index at the end collects all of the fragments and lists the page they were defined on and the pages where they were used.

XXX Page number of definition(s) and use in fragments XXX

1.2 Rendering and the Ray–Tracing Algorithm

What it is, why we're doing it, why you care.

1.3 System Overview

`lrt` is written using an *plug-in* architecture. The `lrt` executable consists of the core code that drives the main flow of control of the system, but has no implementation of specific shape or light representations, etc. All of its code is written in terms of the abstract base classes that define the interfaces to the plug-in types. At run-time, code modules are loaded to provide the specific implementations of these base classes needed for the scene being rendered. This method of organization makes it easy to extend the system; substantial new functionality can be added just by writing a new plug-in. We have tried to define the interfaces to the various plug-in types so that they make it possible to write many interesting and useful extensions. Of course, it's impossible to forsee all of the ways that a developer might want to extend the system, so more far-reaching projects may require modifications to the core system.

The source code to `lrt` is distributed across a small directory hierarchy. All of the code for the `lrt` executable is in the `core/` directory. `lrt` supports twelve different types of plug-ins, summarized in the table in Figure 1.1 which lists the abstract base classes for the plug-in types, the directory that the implementaitions of these types that we provide are stored in, and a reference to the section where each interface is first defined. Low-level details of the routines that load these modules are discussed in Appendix D.1.

1.3.1 Phases of Execution

`lrt` has three main phases of execution. First, it reads in the scene description text file provided by the user. This file specifies the geometric shapes that make up the

Base Class	Directory	Section
`Shape`	`shapes/`	3.1
`Primitive`	`accelerators/`	4.1
`Camera`	`cameras/`	6.1
`Film`	`film/`	8.1
`Filter`	`filters/`	7.7
`Sampler`	`samplers/`	7.3
`ToneMap`	`tonemaps/`	8.3
`Material`	`materials/`	10.2
`Light`	`lights/`	13.1
`SurfaceIntegrator`	`integrators/`	16
`VolumeIntegrator`	`integrators/`	16
`VolumeRegion`	`volumes/`	12.3

Figure 1.1: `lrt` supports twelve types of plug-in objects that are loaded at runtime based on which implementations of them are in the scene description file. The system can be extended with new plug-ins, without needing to be reocmpiled itself.

`Camera`	202
`Film`	294
`Filter`	281
`Light`	478
`Material`	375
`Primitive`	130
`Sampler`	237
`Shape`	63
`SurfaceIntegrator`	563
`ToneMap`	310
`VolumeIntegrator`	630
`VolumeRegion`	465

scene, their material properties, the lights that illuminate them, where the virtual camera is positioned in the scene, and parameters to all of the other algorithms that specify the renderer's basic algorithms. Each statement in the input file has a direct mapping to one of the routines in Appendix B that comprise the interface that `lrt` provides to allow the scene to be described. A number of example scenes are provided in the `examples/` directory in the `lrt` distribution and Appendix C has a reference guide to the scene description format.

Once the scene has been specified, the main rendering loop begins. This is the second main phase of execution, and is the one where `lrt` usually spends the majority of its running time. Most of the chapters in this book describe code that will execute during this phase. This step is managed by the `Scene::Render()` method, which will be the focus of Section 1.3.3. `lrt` uses ray tracing algorithms to determine which objects are visible at particular sample points on the image plane as well as how much light those objects reflect back to the image. Computing the light arriving at many points on the image plane gives us a representation of the image of the scene.

Finally, once the second phase has finished computing the image sample contributions, the third phase of execution handles post-processing the image before it is written to disk (for example, mapping pixel values to the range $[0,255]$ if necessary for the image file format being used.) Statistics about the various rendering algorithms used by the system are then printed, and the data for the scene description in memory is de-allocated. The renderer will then resume procesisng statements from the scene description file until no more remain, allowing the user to specify another scene to be rendered if desired.

The cornerstone of the techniques used to do this is the ray tracing algorithm. Ray tracing algorithms take a geometric representation of a scene and a ray, which can be described by its 3D origin and direction. There are two main tasks that ray tracing algorithms perform: to determine the first geometric object that is visible along a determine whether any geometric objects intersect a ray. The first task

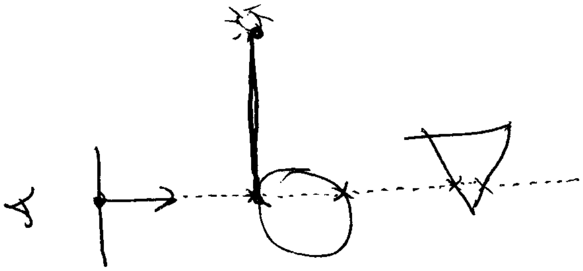


Figure 1.2: Basic ray tracing algorithm: given a ray starting from the image plane, the first visible object at that point can be found by determining which object first intersects the ray. Furthermore, visibility tests between a point on a surface and a light source can also be performed with ray tracing, givng an accurate method for computing shadows.

is useful for solving the hidden-surface problem; if at each pixel we trace a ray into the scene to find the closest object hit by a ray starting from that pixel, we have found the first visible object in the pixel. The second task can be used for shadow computations: if no other object is between a point in the scene and a point on a light source, then illumination from the light source at that point reaches the receiving point; otherwise, it must be in shadow. Figure 1.2 illustrates both of these ideas.

8 Scene

The ability to quickly perform exact visibility tests between arbitrary points in the scene, even in complex scenes, opens the door to many sophisticated rendering algorithms based on these queries. Because ray tracing only requires that a particular shape representation be able to determine if a ray has intersected it (and if so, at what distance along the ray the intersection occured), a wide variety of geometric representations can naturally be used with this approach.

1.3.2 Scene Representation

The `main()` function of the program is in the `core/lrt.cpp` file. It uses the system-wide header `lrt.h`, which defines widely useful types, classes, and functions, and `api.h`, which defines routines related to processing the scene description.

⟨*lrt.cpp**⟩≡

```
  #include "lrt.h"
  #include "api.h"
  ⟨main program⟩
```

`lrt`'s `main()` function is pretty simple; after calling `lrtInit()`, which does system-wide initialization, it parses the scene input files specified by the filenames given as command-line arguments, leading to the creation of a `Scene` object that holds representations of all of the objects that describe the scene and rendering an image of the scene. After rendering is done, `lrtCleanup()` does final cleanup before system exits.

⟨*main program*⟩≡

```
int main(int argc, char *argv[]) {
    ⟨Print welcome banner⟩
    lrtInit();
    ⟨Process scene description⟩
    lrtCleanup();
    return 0;
}
```

If the user ran `lrt` with no command-line arguments, then the scene description is read from standard input. Otherwise we loop through the command line arguments, processing each input filename in turn. No other command line arguments are supported.

⟨*Process scene description*⟩≡

```
if (argc == 1) {
    ⟨Parse scene from standard input⟩
} else {
    ⟨Parse scene from input files⟩
}
```

`lrtCleanup()`	706
`lrtInit()`	706

The `ParseFile()` function parses a text scene description file, either from standard input or from a file on disk; it returns `false` if it was unable to open the file. The mechanics of parsing scene description files will not be described in this book (it is done with straightforward `lex` and `yacc` files.)

⟨*Parse scene from standard input*⟩≡

```
ParseFile("-");
```

If a particular input file can't be opened, the `Error()` routine reports this information to the user. `Error()` is like the `printf()` function in that it first takes a format string that can include escape codes like `%s`, `%d`, `%f`, etc., which have values supplied for them via a variable argument list after the format string.

⟨*Parse scene from input files*⟩≡

```
for (int i = 1; i < argc; i++)
    if (!ParseFile(argv[i]))
        Error("Couldn’t open scene description file \"%s\"\n",
            argv[i]);
```

As the scene file is parsed, objects are created that represent the camera, lights, and the geometric primitives in the scene. Along with other objects that manage other parts of the rendering process, these are all collected together in the `Scene` object, which is allocated by the `GraphicsOptions::MakeScene()` method in Section B.4. The `Scene` class is declared in `core/scene.h` and defined in `core/scene.cpp`.

⟨*Scene Declarations*⟩≡

```
class Scene {
public:
    ⟨Scene Public Methods⟩
    ⟨Scene Data⟩
};
```

We don't include the implementation of the Scene constructor here; it mostly just copies the pointers to these objects that were passed into it.

Each geometric object in the scene is represented by a Primitive, which collects a lower-level Shape that strictly specifies its geometry, and a Material that describes how light is reflected at points on the surface of the object (e.g. the object's color, whether it has a dull or glossy finish, etc.) All of these geometric primitives are collected into a single aggregate Primitive, aggregate, that stores them ina a 3D data structure that makes ray tracing faster by substantially reducing the number of unnecessary ray intersection tests.

⟨*Scene Data*⟩≡

```
Primitive *aggregate;
```

Each light source in the scene is represented by a Light object. The shape of a light and the distribution of light that it emits has a substantial effect on the illumination it casts into the scene. lrt supports a single global light list that holds all of the lights in the scene using the vector class from the standard library. While some renderers support light lists that are specified per-geometric object, allowing some lights to illuminate only some of the objects in the scene, this idea doesn't map well to the physically-based rendering approach taken in lrt, so we only have this global list.

⟨*Scene Data*⟩+≡

```
vector<Light *> lights;
```

The camera object controls the viewing and lens parameters such as camera position and orientation and field of view. A Film member variable inside the camera class handles image storage. The Camera and classes are described in Chapter 6 and film is described in Chapter 8. After the image has been computed, a sequence of imaging operations is applied by the film to make adjustments to the image before writing it to disk.

202	Camera
478	Light
375	Material
130	Primitive
63	Shape
563	SurfaceIntegrator
658	vector
630	VolumeIntegrator
465	VolumeRegion

⟨*Scene Data*⟩+≡

```
Camera *camera;
```

describe this...

⟨*Scene Data*⟩+≡

```
VolumeRegion *volumeRegion;
```

Integrators handle the task of simulating the propagation of light in the scene from the light sources to the primitives in order to compute how much light arrives at the film plane at image sample positions. Their name comes from the fact that their task is to evaluate the value of an integral equation that describes the distribution of light in an environment. SurfaceIntegrators compute reflected light from geometric surfaces, while VolumeIntegrators handle the scattering from *participating media*–particles like fog or smoke in the environment that interact with light. The properties and distribution of the participating media are described by VolumeRegion objects, which are defined in Chapter 12. Both types of integrators are described and implemented in Chapter 16.

⟨*Scene Data*⟩+≡

```
SurfaceIntegrator *surfaceIntegrator;
VolumeIntegrator *volumeIntegrator;
```

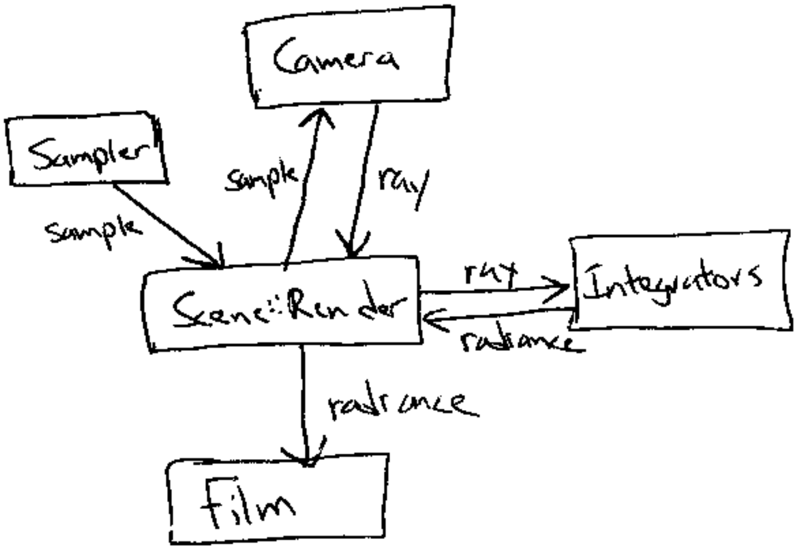


Figure 1.3: Class relationships for main rendering loop, which is in the `Scene::Render()` method in `core/scene.cpp`. The `Sampler` provides a sequence of sample values, one for each image sample to be taken. The `Camera` turns a sample into a corresponding ray from the film plane and the `Integrators` compute the radiance along that ray arriving at the film. The sample and its radiance are given to the `Film`, which stores their contribution in an image. This process repeats until the `Sampler` has provided as many samples as are necessary to generate the final image.

`Camera`	202
`Film`	294
`Integrator`	562
`Sampler`	237
`SurfaceIntegrator`	563
`VolumeIntegrator`	630

The goals of the `Sampler` are subtle, but its implementation can substantially affect the quality of the images that the system generates. First, the sampler is repsonsible for choosing the points on the image plane from which rays are traced into the scene to compute final pixel values. Second, it is responsible for supplying sample positions that are used by the integrators in their light transport computations. For example, some integrators need to choose sample points on light sources as part of the process of computing illumination at a point. Generating good distributions of samples is an important part of the rendering process and is discussed in Chapter 7.

⟨*Scene Data*⟩+≡

```
  Sampler *sampler;
```

1.3.3 Main Rendering Loop

After the `Scene` has been allocated and initialized, its `Render()` method is invoked, starting the second phase of `lrt`'s execution, the main rendering loop. For each of a series of positions on the image plane, this method uses the camera and the sampler to generate a ray out into the scene and then uses the integrators to compute the light arriving along the ray at the image plane. This value is passed along to the film, which records its contribution. Figure 1.3 summarizes the main classes used in this method and the flow of data among them.

⟨Scene Methods⟩≡

```
void Scene::Render() {
        ⟨Allocate and initialize sample⟩
        ⟨Allow integrators to do pre-processing for the scene⟩
        ⟨Get all samples from Sampler and evaluate contributions⟩
        ⟨Clean up after rendering and store final image⟩
}
```

Before rendering starts, this method allocates a `Sample` object for the `Sampler` to use to store sample values for each image sample. Because the number and types of samples that need to be generated for each image sample are partially dependent on the integrators, `Sample` constructor takes pointers to them so that they can inform the `Sample` object about their sample needs. See Section 7.3.1 for more information about how integrators request particular sets of samples at this point.

⟨Allocate and initialize `sample`⟩≡

```
Sample *sample = new Sample(surfaceIntegrator, volumeIntegrator, this);
```

The only other task to complete before rendering can begin is to call the `Preprocess()` methods of the integrators, which gives them an opportunity to do any scene-dependent precomputation that thay may need to do. Because information like the number of lights in the scene, their power and the geometry of the scene aren't known when the integrators are originally created, the `Preprocess()` method gives them an opportunity to do final initialization that depends on this information. For example, the `PhotonIntegrator` in Section 16.6 uses this opportunity to create data structures that hold a representation of the distribution of illumination in the scene.

563	`Integrator::Preprocess()`
611	`PhotonIntegrator`
660	`ProgressReporter`
237	`Sampler`
238	`Sampler::GetNextSample()`
8	`Scene`

⟨Allow integrators to do pre-processing for the scene⟩≡

```
surfaceIntegrator->Preprocess(this);
volumeIntegrator->Preprocess(this);
```

The `ProgressReporter` object tells the user how far through the rendering process we are as `lrt` runs. It takes the total number of work steps as a parameter, so that it knows the total amount of work to be done. After its creation, the main render loop begins. Each time through the loop `Sampler::GetNextSample()` is called and the `Sampler` initializes `sample` with the next image sample value, returning `false` when there are no more samples. The fragments in the loop body find the corresponding camera ray and hand it off to the integrators to compute its contribution, and finally updating the image with the result.

⟨Get all samples from `Sampler` and evaluate contributions⟩≡

```
ProgressReporter progress(sampler->TotalSamples(), "Rendering");
while (sampler->GetNextSample(sample)) {
        ⟨Find camera ray for sample⟩
        ⟨Evaluate radiance along camera ray⟩
        ⟨Add sample contribution to image⟩
        ⟨Free BSDF memory from computing image sample value⟩
        ⟨Report rendering progress⟩
}
```

The main function of the Camera class is to provide a GenerateRay() method, which determines the appropriate ray to trace for a particular sample position on the image plane given the particular image formation process that it is simulating. The sample and a ray are passed to this method, and the fields of the ray are initialized accordingly. An important convention that all Cameras must follow is that the direction components of the rays that they return must be normalized. Most of the Integrators depend on this fact.

The camera also returns a floating-point weight with the ray can be used by Cameras that simulate realistic models of image formation where some rays through a lens system carry more energy than others; for example, in a real camera, less light typically arrives at the edges of the film plane than at the center. This weight will be used later as a scale factor to be applied to this ray's contribution to the image.

```
⟨Find camera ray for sample⟩≡
  RayDifferential ray;
  Float rayWeight = camera->GenerateRay(*sample, &ray);
  ⟨Generate ray differentials for camera ray⟩
```

In order to get better results from some of the texture functions defined in Chapter 11, it is useful to determine the rays that the Camera would generate for samples offset one pixel in the *x* and *y* direction on the image plane. This information will later allow us to compute how quickly a texture is varying with respect to the pixel spacing when projected onto the image plane, so that we can remove detail from it that can't be represented in the image being generated. Doing so eliminates a wide class of image artifacts due to aliasing. While the Ray class just holds the origin and direction of a single ray, RayDifferential inherits from Ray so that it also has those member variables, but it also holds two additional Rays, rx and ry to hold these neighbors.

Camera	202
Camera::GenerateRay()	202
Integrator	562
Ray	36
RayDifferential	37
ential::hasDifferentials	38
RayDifferential::rx	38
RayDifferential::ry	38
Sample::imageX	239
Sample::imageY	239
Spectrum	181

```
⟨Generate ray differentials for camera ray⟩≡
  ++sample->imageX;
  camera->GenerateRay(*sample, &ray.rx);
  --sample->imageX;
  ++sample->imageY;
  camera->GenerateRay(*sample, &ray.ry);
  ray.hasDifferentials = true;
  --sample->imageY;
```

Given a ray, the Scene::Render() method calls Scene::L(), which returns the amount of light arriving at the image along the ray. The implementation of this method will be shown in the next section. The physical unit that describes the strength of this light is *radiance*; it is described in detail in Section 5.2. The symbol for radiance is *L*, thus the name of the method. These radiance values are represented with the Spectrum class, the abstraction that defines the representation of general energy distributions by wavelength–in other words, color.

In addition to returning the ray's radiance, Scene::L() sets the alpha variable passed to it to the *alpha value* for this ray. Alpha is an extra component beyond color that encodes opacity. If the ray hits an opaque object, alpha will be one, indicating that nothing behind the intersection point is visible. If the ray passed through something partially transparent, like fog, but never hit an opaque object `alpha` will be between zero and one. If the ray didn't hit anything, `alpha` is zero. Computing alpha values here and storing an alpha value with each pixel can be useful for a variety of post-processing effects; for example, we can composite a rendered object on top of a photograph, using the pixels in the image of the photograph wherever the rendered image's alpha channel is zero, using the rendered image where its alpha channel is one, and using a mix of the two for the remaining pixels.

Finally, an assertion checks that the returned spectral radiance value doesn't have any floating-point "not a number" components; these are a common side-effect of bugs in other parts of the system, so it's helpful to catch them immediately here.

⟨*Evaluate radiance along camera ray*⟩≡

```
Float alpha;
Spectrum Ls = 0.f;
if (rayWeight > 0.f) Ls = rayWeight * L(ray, sample, &alpha);
⟨Issue warning if unexpected radiance value returned⟩
```

⟨*Issue warning if unexpected radiance value returned*⟩≡

```
if (Ls.IsNaN())
    Error("Not-a-number radiance value returned for image sample");
else if (Ls.y() < 0)
    Error("Negative luminance value, %f, returned for image sample",
        Ls.y());
```

370	BSDF
374	BSDF::FreeAll()
203	Camera::film
294	Film::AddSample()
670	MemoryArena
660	ProgressReporter
15	Scene::L()
181	Spectrum
661	StatsCounter

After we have the ray's contribution, we can update the image. The `Film::AddSample()` method updates the pixels in the image given the results from this sample. The details of this process are explained in Section 7.7.

⟨*Add sample contribution to image*⟩≡

```
camera->film->AddSample(*sample, ray, Ls, alpha);
```

BSDFs describe material properties at a single point on a surface; they will be described in more detail later in this section. In `lrt`, it's necessary to dynamically allocate memory to store the BSDFs used to compute the contribution of sample value here. In order to avoid the overhead of calling the system's memory allocation and freeing routines multiple times for each of them, the `BSDF` class uses the `MemoryArena` class to manage pools of memory for `BSDF`s. Section 10.1.1 describes this in more detail. Now that the contribution for this sample has been computed, it's necessary to tell the `BSDF` class that all of the `BSDF` memory allocated for the sample we just finished is no longer needed, so that it can be reused for the next sample.

⟨*Free BSDF memory from computing image sample value*⟩≡

```
BSDF::FreeAll();
```

So that it's easy for various parts of `lrt` to gather statistics on things that may be meaningful or interesting to the user, a handful of statistics-tracking classes are defined in Appendix A.2.3. `StatsCounter` overloads the `++` operator for indicating that the counter should be incremented. The `ProgressReporter` class indicates how many steps out of the total have been completed with a row of plus signs printed to the screen; a call to its `Update()` method indicates that one of the total number of steps passed to its constructor has been completed.

⟨*Report rendering progress*⟩≡

```
static StatsCounter cameraRaysTraced("Camera",
    "Camera Rays Traced");
++cameraRaysTraced;
progress.Update();
```

At the end of the main loop, `Scene::Render()` frees the sample memory and begins the third phase of `lrt`'s execution with the call to `Film::WriteImage()`, where the imaging pipeline prepares a final image to be stored.

⟨*Clean up after rendering and store final image*⟩≡

```
delete sample;
camera->film->WriteImage();
```

1.3.4 Scene Methods

The `Scene` only has a handful of additional methods; it mostly just holds the variables that represent the scene. The methods it does have generally have little complexity and forward requests on to methods of the `Scene`'s member variables.

BBox	38
Camera	202
Camera::film	203
Film::WriteImage()	294
Intersection	131
Primitive::Intersect()	131
Primitive::IntersectP()	131
ogressReporter::Update()	660
Ray	36
Scene::aggregate	9
StatsCounter	661
SurfaceIntegrator	563

First is the `Scene::Intersect()` method, which traces the given ray into the scene and returns a boolean value indication whether it intersected any of the primitives. If so, it returns information about the closest intersection point in the `Intersection` structure defined in Section 4.1.

⟨*Scene Public Methods*⟩+≡

```
bool Intersect(const Ray &ray, Intersection *isect) const {
    return aggregate->Intersect(ray, isect);
}
```

A closely-related method is `Scene::IntersectP()`, which checks for any intersection along a ray, again returning a boolean result. Because it doesn't return information about the geometry at the intersection point and because it doesn't need to search for the closest intersection, it can be more efficient than `Scene::Intersect()` for rays where this additional information isn't needed.

⟨*Scene Public Methods*⟩+≡

```
bool IntersectP(const Ray &ray) const {
    return aggregate->IntersectP(ray);
}
```

Another useful geometric method, `Scene::WorldBound()`, returns a 3D box that bounds the extent of the geometry in the scene. We won't include its straightforward implementation here.

⟨*Scene Public Methods*⟩+≡

```
const BBox &WorldBound() const;
```

The `Scene`'s method to compute the radiance along a ray, `Scene::L()`, uses a `SurfaceIntegrator` to compute reflected radiance from the first surface that the given ray intersects and stores the result in `Ls`. It then uses the volume integrator's

Transmittance() method to compute how much of that light is extinguished between the point on the surface and the camera due attenuation and scattering of light by participating media, if any. Participating media may also increase light along the ray; the VolumeIntegrator's L() method computes how much light is added along the ray due to volumetric light sources and scattering from particles in the media. Section 16.7 describes the theory of attenuation and scattering from participating media in detail. The net effect of these interactions is returned by this method.

⟨*Scene Methods*⟩+≡

```
Spectrum Scene::L(const RayDifferential &ray,
        const Sample *sample, Float *alpha) const {
    Spectrum Ls = surfaceIntegrator->L(this, ray,
        sample, alpha);
    Spectrum T = volumeIntegrator->Transmittance(this, ray,
        sample, alpha);
    Spectrum Lv = volumeIntegrator->L(this, ray,
        sample, alpha);
    return T * Ls + Lv;
}
```

It's also useful to compute the attenuation of a ray in isolation; the Scene's Transmittance() method returns the reduction in radiance along the ray due to participating media.

36	Ray
37	RayDifferential
8	Scene
181	Spectrum
630	VolumeIntegrator
630	VolumeIntegrator::Transmittance(
16	WhittedIntegrator

⟨*Scene Methods*⟩+≡

```
Spectrum Scene::Transmittance(const Ray &ray) const {
    return volumeIntegrator->Transmittance(this, ray,
        NULL, NULL);
}
```

1.3.5 Whitted Integrator

Chapter 16 has the implementations of many different surface and volume integrators, giving differing levels of accuracy using a variety of algorithms to compute the results. Here we will present a classic surface integrator based on Whitted's ray tracing algorithm. This integrator accurately computes reflected and transmitted light from specular surfaces like glass, mirrors, and water, though it doesn't account for indirect lighting effects. The more complex integrators later in the book build on the ideas in this integrator to implement more sophisticated light transport algorithms.

The WhittedIntegrator is in the whitted.cpp file in the integrators/ directory.

⟨*whitted.cpp**⟩≡

```
#include "lrt.h"
#include "transport.h"
#include "scene.h"
⟨WhittedIntegrator Declarations⟩
⟨WhittedIntegrator Method Definitions⟩
```

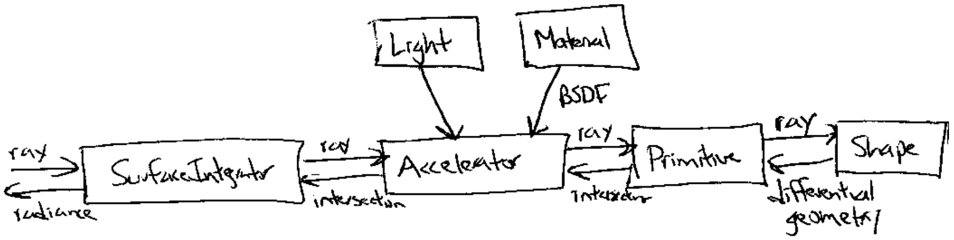


Figure 1.4: Class relationships for surface integration: the main render loop passes a camera ray to the `SurfaceIntegrator`, which has the task of returning the radiance along that ray arriving at the ray's origin on the film plane. The integrator calls back to the `Scene::Intersect()` method fo find the first surface that the ray intersects; the scene in turn passes the request on to an accelerator (which is itself a `Primitive`). The accelerator will perform ray–primitive intersection tests with the `Primitive`s that the ray potentially intersects, and these will lead to the `Shape::Intersect()` routines for the corresponding shapes. Once the `Intersection` is returned to the integrator, it gets the material properties at the intersection point in the form of a `BSDF` and uses the `Light`s in the `Scene` to determine the illumination there. This gives the information needed to compute reflected radiance at the intersection point back along the ray.

`BSDF`	370
`Intersection`	131
`Light`	478
`Primitive`	130
`RayDifferential`	37
`Scene`	8
`Scene::Intersect()`	14
`Spectrum`	181
`SurfaceIntegrator`	563

⟨*WhittedIntegrator Declarations*⟩≡

```
class WhittedIntegrator : public SurfaceIntegrator {
public:
    ⟨WhittedIntegrator Public Methods⟩
private:
    ⟨WhittedIntegrator Private Data⟩
};
```

The key method that all integrators must provide is `L()`, which returns the radiance along a ray. Figure 1.4 summarizes the data-flow among the main classes used during integration at surfaces.

⟨*WhittedIntegrator Method Definitions*⟩≡

```
Spectrum WhittedIntegrator::L(const Scene *scene,
        const RayDifferential &ray, const Sample *sample,
        Float *alpha) const {
    Intersection isect;
    Spectrum L(0.);
    if (scene->Intersect(ray, &isect)) {
        if (alpha) *alpha = 1.;
        ⟨Compute emitted and reflected light at ray intersection point⟩
    }
    else {
        ⟨Handle ray with no intersection⟩
    }
    return L;
}
```

For the integrator to determine what primitive is hit by a ray, it calls the `Scene::Intersect()`

method,

If the ray passed to the integrator's `L()` method intersects a geometric primitive, the reflected radiance is given by the sum of directly emitted radiance from the object if it is itself emissive, and the reflected radiance due to reflection of light from other primitives and light sources that arrives at the intersection point. This idea is formalized by the equation below, which says that outgoing radiance from a point p in direction ω_o, $L_\mathrm{o}(\mathrm{p}, \omega_o)$, is the sum of emitted radiance at that point in that direction, $L_\mathrm{e}(\mathrm{p}, \omega_o)$, plus the incident radiance from all directions on the sphere $\mathcal{S}^2$ around p scaled by a function that describes how the surface scatters light from the incident direction ω_i to the outgoing direction ω_o, $f(\mathrm{p}, \omega_o, \omega_i)$, and a cosine term. We will show a more complete derivation of this equation later, in Sections 5.4.1 and 16.2.

$$L_\mathrm{o}(\mathrm{p}, \omega_o) = L_\mathrm{e}(\mathrm{p}, \omega_o) + \int_{\mathcal{S}^2} L_\mathrm{i}(\mathrm{p}, \omega_i)\, f(\mathrm{p}, \omega_o, \omega_i)\, |\cos\theta_i| \mathrm{d}\omega_i$$

Solving this integral analytically is in general not possible for anything other than the simplest of scenes, so integrators must either make simplifying assumptions or use numerical integration techniques. The `WhittedIntegrator` ignores incoming light from most of the directions and only evaluates $L_\mathrm{i}(\mathrm{p}, \omega_i)$ for the directions to light sources and for the directions of specular reflection and refraction. Thus, it turns the integral into a sum over a small number of directions.

370 BSDF
16 WhittedIntegrator

The Whitted integrator works by recursively evaluating radiance along reflected and refracted ray directions. We keep track of the depth of recursion in the variable `rayDepth` and after a predetermined recursion depth, `maxDepth`, we stop tracing reflected and refracted rays. By default the maximum recursion depth is five. Otherwise, in a scene like a box where all of the walls were mirrors, the recursion might never terminate. These member variables are initialized in the trivial `WhittedIntegrator` constructor, which we will not include in the text.

```
⟨WhittedIntegrator Private Data⟩≡
  int maxDepth;
  mutable int rayDepth;
```

The ⟨*Compute emitted and reflected light at ray intersection point*⟩ fragment is the heart of the Whitted integrator.

```
⟨Compute emitted and reflected light at ray intersection point⟩≡
  ⟨Evaluate BSDF at hit point⟩
  ⟨Initialize common variables for Whitted integrator⟩
  ⟨Compute emitted light if ray hit an area light source⟩
  ⟨Compute reflection by integrating over the lights⟩
  if (rayDepth++ < maxDepth) {
      ⟨Trace rays for specular reflection and refraction⟩
  }
  --rayDepth;
```

To compute reflected light, the integrator must have a representation of the local light scattering properties of the surface at the intersection point as well as a way to determine the distribution of illumination arriving at that point.

To represent the scattering properties at a point on a surface, `lrt` uses a class called `BSDF`, which stands for "Bidirectional Scattering Distribution Function".

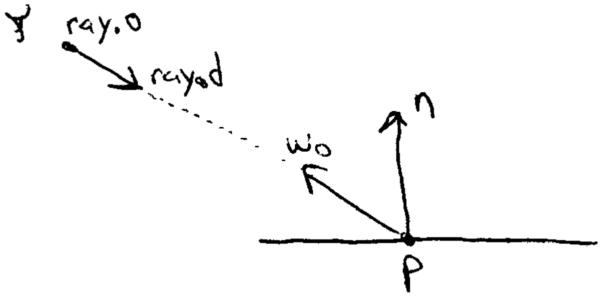


Figure 1.5: Basic setting for the Whitted integrator: p is the ray intersection point and **n** is the surface normal there. The direction in which we'd like to compute reflected radiance is ω_o; its is the vector pointing in the opposite direction of the ray, `-ray.d`.

These functions take an incoming direction and an outgoing direction and return a value that indicates the amount of light that is reflected from the incoming direction to the outgoing direction (actually, BSDF's usually vary as a function of the wavelength of light, so they really return a `Spectrum`). `lrt` provides built-in `BSDF` classes for several standard scattering functions used in computer graphics. Examples of BSDFs include Lambertian reflection and the Torrance-Sparrow microfacet model; these and other BSDFs are implemented in Chapter 9.

BSDF	370
BSDF::dgShading	370
DifferentialGeometry::nn	58
DifferentialGeometry::p	58
Intersection::GetBSDF()	375
Normal	34
Point	33
Ray::d	35
Spectrum	181
Texture	394
Vector	27

The `BSDF` at a surface point provides all information needed to shade that point, but BSDFs may vary across a surface. Surfaces with complex material properties, such as wood or marble, have a different BSDF at each point. Even if wood is modelled as perfectly diffuse, for example, the diffuse color at each point will depend on the wood's grain. These spatial variations of shading parameters are described with `Texture`s, which in turn may be described procedurally or stored in image maps; see Chapter 11.

The `Intersection::GetBSDF()` method returns a pointer to the `BSDF` at the intersection point on the object.

⟨*Evaluate BSDF at hit point*⟩≡

```
BSDF *bsdf = isect.GetBSDF(ray);
```

There are a few quantities that we'll make use of repeatedly in the fragments to come. Figure 1.5 illustrates them. p the world-space position of the ray–primitive intersection and **n** is the surface normal at the intersection point. The normalized direction from the hit point back to the ray origin is stored in `wo`; because `Camera`s are responsible for normalizing the direction component of the rays they generate, there's no need to re-noralize it here. (Normalized directions in `lrt` are generally denoted by the ω symbol, so `wo` is a shorthand we will commonly use for ω_o, the outgoing direction of scattered light.)

⟨*Initialize common variables for Whitted integrator*⟩≡

```
const Point &p = bsdf->dgShading.p;
const Normal &n = bsdf->dgShading.nn;
Vector wo = -ray.d;
```

If the ray happened to hit geometry that is itself emissive, we compute its emitted radiance by calling the `Intersection`'s `Le()` method. This gives us the first term of the outgoing radiance equation above. If the object is not emissive, this method will return a black spectrum.

⟨*Compute emitted light if ray hit an area light source*⟩≡

```
L += isect.Le(wo);
```

For each light, the integrator computes the amount of illumination falling on the surface at the point being shaded by calling the light's `dE()` method, passing it the position and surface normal for the point on the surface. E is the symbol for the physical quantity *irradiance*, and differential irradiance, dE, is the appropriate measure of incident illumination here–radiometric concepts such as energy and differential irradiance are discussed in Chapter 5. This method also returns the direction vector from the point being shaded to the light source, which is stored in the variable `wi`.

The `Light::dE()` method also returns a `VisibilityTester` object, which is a closure representing additional computation to be done to determine if any primitives block the light from the light source. Specifically, the `Spectrum` that is returned from `Light::dE()` doesn't account for any other objects blocking light between the light source and the surface. To verify that there are no such occluders, a shadow ray must be traced between the point being shaded and the point on the light to verify that the path is clear. Because ray tracing is relatively expensive, we would like to defer tracing the ray until we are sure that the BSDF indicates that some of the light from the direction ω_o will be scattered in the direction ω_o. For example, if the surface isn't transmissive, then light arriving at the back side of the surface doesn't contribute to reflection. The `VisibilityTester` encapsulates the state needed to record which ray needs to be traced to do this check. (In a similar manner, the attenuation along the ray to the light source due to participating media is ignored until explicitly evaluated via the `Transmittance()` method.)

373	BSDF::f()
131	Intersection
132	Intersection::Le()
479	Light::dE()
9	Scene::lights
181	Spectrum
182	Spectrum::Black()
27	Vector
479	VisibilityTester

To evaluate the contribution to the reflection due to the light, the integrator multiplies `dE` by the value that the BSDF returns for the fraction of light that is scattered from the light direction to the outgoing direction along the ray. This represents this light's contribution to the reflected light in the integral over incoming directions, which is added to the total of reflected radiance stored in `L`. After all lights have been considered, the integrator has computed total reflection due to *direct lighting*: light that arrives at the surface directly from emissive objects (as opposed to light that has reflected off other objects in the scene before arriving at the point.)

⟨*Compute reflection by integrating over the lights*⟩≡

```
Vector wi;
for (u_int i = 0; i < scene->lights.size(); ++i) {
    VisibilityTester visibility;
    Spectrum dE = scene->lights[i]->dE(p, n, &wi, &visibility);
    if (dE.Black()) continue;
    Spectrum f = bsdf->f(wo, wi);
    if (!f.Black() && visibility.Unoccluded(scene))
        L += f * dE * visibility.Transmittance(scene);
}
```

Figure 1.6:

Before we finish, the integrator also accounts for the contribution of light scattered by perfectly specular surfaces like mirrors or glass. Consider a mirror, for example. The law of mirror reflection says that the angle the reflected ray makes with the surface normal is equal to the angle made by the incident ray (see Figure 1.6). Thus, to compute reflected radiance from a mirror in direction ω_o, we need to know the incident radiance at the surface point p in the direction ω_i. The key insight that Whitted had was that this could be found with arecursive call to the ray tracing routine with a new ray from p in the direction ω_i. Therefore, when a specularly reflective or transmissive object is hit by a ray, new rays are also traced in the reflected and refracted directions and the returned radiance values are scaled by the value of the surface's BSDF and added to the radiance scattered from the original point.

The `BSDF` has a method that returns an incident ray direction for a given outgoing direction and a given mode of light scattering at a surface. Here, we are only interested in perfect specular reflection and transmission, so we use the `BSDF_*` flags to `BSDF::Sample_f()` to indicate that glossy and diffuse reflection should be ignored here. Thus, the two calls to `Sample_f()` below check for specular reflection and transmission and initialize `wi` with the appropriate direction and return the BSDF's value for the directions (ω_o, ω_i). If the value of the BSDF is non-zero, the integrator calls the `Scene`'s radiance function `L()` to get the incoming radiance along the ray, which leads to a call back to the `WhittedIntegrator`'s `L()` method. By continuing this process recursively multiple reflection and refraction are accounted for.

BSDF	370
Scene	8
WhittedIntegrator	16

One important detail in this process is how ray differentials for the reflected and transmitted rays are found; just as having an approximation to the screen-space area of a directly-visible object is cruicial for anti-aliasing textures on the object, if we can approximate the screen-space area of objects that are seen through reflection or refraction, we can reduce aliasing in their textures as well. The fragments that implement the computations to find the ray differentials for these rays are described in Section 10.2.2.

To compute the cosine term of the reflection integral, the integrator calls the `Dot()` function, which returns the dot product between two vectors. If the vectors are normalized, as both `wi` and `n` are here, this is equal to the cosine of the angle between them.

```
⟨Trace rays for specular reflection and refraction⟩≡
  Spectrum f = bsdf->Sample_f(wo, &wi,
      BxDFType(BSDF_REFLECTION | BSDF_SPECULAR));
  if (!f.Black()) {
      ⟨Compute ray differential rd for specular reflection⟩
      L += scene->L(rd, sample) * f * AbsDot(wi, n);
  }
  f = bsdf->Sample_f(wo, &wi,
      BxDFType(BSDF_TRANSMISSION | BSDF_SPECULAR));
  if (!f.Black()) {
      ⟨Compute ray differential rd for specular transmission⟩
      L += scene->L(rd, sample) * f * AbsDot(wi, n);
  }
```

```
⟨Handle ray with no intersection⟩≡
  if (alpha) *alpha = 0.;
  return L;
```

And this concludes the `WhittedIntegrator`'s implementation.

1.4 How To Proceed Through This Book

We have written this text assuming it will be read in roughly front-to-back order. We have tried to minimize the number of forward references to ideas and interfaces that haven't yet been introduced, but assume that the reader is acquianted with the content before any particular point in the text. Because of the modular nature of the system, the most improtant thing to be able to understand an individual section of code is that the reader be familiar with the low-level classes like `Point`, `Ray`, `Spectrum`, etc., the interfaces defined by the abstract base classes listed in Figure 1.1, and the main rendering loop in `Scene::Render()`.

38	BBox
540	BSDF::Sample_f()
334	BSDF_REFLECTION
334	BSDF_SPECULAR
334	BSDF_TRANSMISSION
334	BxDFType
202	Camera
33	Point
36	Ray
237	Sampler
15	Scene::L()
63	Shape
181	Spectrum

Given that knowledge, for example, the reader who doesn't care about precisely how a camera model based on a perspective projection matrix maps samples to rays can skip over the implementation of that camera and can just remember that the `Camera::GenerateRay()` method somehow turns a `Sample` into a `Ray`. Furthermore, some sections go into depth about advanced topics that some readers may wish to skip over (particularly on a first reading); these sections are denoted by an asterisk.

The book is divdided into four main sections of a few chapters each. First, chapters two through four define the main geometric functinoality in the system. Chapter two has the low-level classes like `Point`, `Ray`, and `BBox`; chapter three defines the `Shape` interface, has implementations of a number of shapes, and shows how to perform ray–shape intersection tests; and chapter four has the implementations of the acceleration structures for speeding up ray tracing by avoiding tests with primitives that a ray can be shown to definitely not intersect.

The second main section covers the image formation process. First, chapter five introduces the physical units used to measure light and the `Spectrum` class that represents wavelength-varying distributions (i.e. color). Chapter six defines the `Camera` interface and has a few different camera implementations. The `Sampler` classes that place samples on the image plane are the topic of chapter seven and the overall process of turning the radiance values from camera rays into images suitable for display are explained in chapter eight.

The third section is about light and how light scatters from surfaces and participating media. Chapter nine defines a set of building-block classes that define a variety of types of reflection from surfaces. Materials, described in chapter ten, use these reflection functions to implement a number of different types of surface materials, such as plastic, glass, and metal. Chapter eleven introduces texture, which describes variation in material properties (color, roughness, etc.) over surfaces, and chapter twelve has the abstractions used to describe how light is scattered and absorbed in participating media. Finally, chapter thirteen has the interface for light sources and light source implementations.

The last section brings all of the ideas of the rest of the book together to implement a number of integrators. Chapters fourteen and fifteen introduce the theory of Monte Carlo integration, a statistical technique for estimating the value of complex integrals, and have low-level routines for applying Monte Carlo to illumination and light scattering. The surface and volume integrators of chapter sixteen use Monte Carlo integration to compute more accurate approximations of the light reflection equation defined above than the `WhittedIntegrator` did, using techniques like path tracing, bidirectional path tracing, irradiance caching, and photon mapping. The last chapter of the book has a brief retrospective and discussion of system design decisions along with a number of suggestions for more far-reaching projects than those in the exercises in previous chapters.

`WhittedIntegrator 16`

Additional Reading

In a seminal early paper, Arthur Appel first described the basic idea of ray tracing to solve the hidden surface problem and to compute shadows in polygonal scenes (Appel 1968). Goldstein and Nagle later showed how ray tracing could be used to render scenes with quadric surfaces (Goldstein and Nagel 1971). **(XXX first direct rendering of curved surfaces? XXX)** Kay and Greenberg described a ray tracing approach to rendering transparency (Kay and Greenberg 1979), and Whitted's seminal CACM paper described the general recursive ray tracing algorithm we have outlined in this chapter, accurately simulating reflection and refraction from specular surfaces and shadows from point light sources (Whitted 1980).

Notable books on physically-based rendering and image synthesis include Cohen and Wallace's *Radiosity and Realistic Image Synthesis* (Cohen and Wallace 1993) and Sillion and Puech's *Radiosity and Global Illumination* (Sillion and Puech 1994) which primarily describe the finite-element radiosity method; Glassner's *Principles of Digital Image Synthesis* (Glassner 1995), an encyclopediac two-volume summary of theoretical foundations for realistic rendering; and *Illumination and Color in Computer Generated Imagery* (Hall 1989), one of the first books to present rendering in a physically-based framework. **XXX Advanced Globillum Book XXX**

Many papers have been written that describe the design and implementation of other rendering systems. One type of renderer that has been written about is renderers for entertainment and artistic applications. The REYES architecture, which forms the basis for Pixar's RenderMan renderer, was first described by Cook et al (Cook, Carpenter, and Catmull 1987); a number of improvements to the original algorithm are summarized in (Apodaca and Gritz 2000). Gritz and Hahn describe the BMRT ray tracer (Gritz and Hahn 1996), though mostly focus on the details of implementing a ray tracer that supports the RenderMan interface. The renderer in the Maya modeling and animation system is described by Sung et al (Sung, Craighead, Wang, Bakshi, Pearce, and Woo 1998).

Kirk and Arvo's paper on ray tracing system design was the first to suggest many design principles that have now become classic in renderer design (Kirk and Arvo 1988). The renderer was implemented as a core *kernel* that encapsulated the basic rendering algorithms and interacted with primitives and shading routines via a carefully-constructed object-oriented interface. This approach made it easy to extend the system with new primitives and acceleration methods.

The *Introduction to Ray Tracing* book, which describes the state-of-the-art in ray tracing in 1989, has a chapter by Heckbert that sketches the design of a basic ray tracer (**?**). Finally, Shirley's recent book gives an excellent introduction to ray tracing and includes the complete source code to a basic ray tracer. **XXX cite XXX**

Researchers at Cornell university have developed a rendering testbed over many years; its overall structure is described by Trumbore et al (Trumbore, Lytle, and Greenbert 1993). Its predecessor was described by Hall and Greenberg (Hall and Greenberg 1983). This system is a loosely-coupled set of modules and libraries, each designed to handle a single task (ray–object intersection acceleration, image storage, etc), and written in a way that makes it easy to combine appropriate modules together to investigate and develop new rendering algorithms. This testbed has been quite successful, serving as the foundation for much of the rendering research done at Cornell.

Another category of renderer focuses on physically-based rendering, like `lrt`. One of the first renderers based fundamentally on physical quantities is *Radiance*, which has been used widely in lighting simulation applications. Ward describes its design and history in a paper and a book (Ward 1994b; Larson and Shakespeare 1998). Radiance is designed in the Unix style, as a set of interacting programs, each handling a different part of the rendering process. (This type of rendering architecture, interacting separate programs, was first described by Duff (Duff 1985).).

Glassner's *Spectrum* rendering architecture also focuses on physically-based rendering (Glassner 1993), appraoched through a signal-processing based formulation of the problem. It is an extensible system built with a plug-in architecture; `lrt`'s approach of using parameter/value lists for initializing plug-in objects is similar to *Spectrum*'s. One notable feature of *Spectrum* is that all parameters that describe the scene can be animated in a variety of ways.

Slusallek and Seidel describe the architecture of the *Vision* rendering system, which is also physically based and was designed to be extensible to support a wide variety of light transport algorithms (Slusallek and Siedel 1995; Slusallek and Seidel 1996; Slusallek 1996). In particular, it has the ambitious goal of supporting both Monte Carlo and finite-element based light transport algorithms. Because `lrt` was designed with the fundamental expectation that Monte Carlo algorithms would be used, its design could be substantially more straightforward.

The *RenderPark* rendering system also supports a variety of physically-based rendeirng algorithms, including both Monte Carlo and finite element approaches. It was developed by by Philippe Bekaert, Frank Suykens de Laet, Pieter Peers, and Vincent Masselus, and is available from `http://www.cs.kuleuven.ac.be/cwis/research/graphics/RENDERPARK/`.

The source code to a number of other ray tracers and renderers is available on the web. Notable ones include Mark VandeWettering's *MTV*, which was the first widely-distributed freely-available ray tracer; it was posted to the `comp.sources.unix` newsgroup in 1988. Craig Kolb's *rayshade* had a number of releases during the 1990s; its current homepage is `http://graphics.stanford.edu/ cek/rayshade/rayshade.html`. The *radiance* system is available from `http://radsite.lbl.gov/radiance/HOME.html`. POV-Ray is used by a large number of individuals, primarily for personal purposes; it is available from `http://www.povray.org`. **XXX Photon, 3Dlight, Aqusis. XXX**

A good introduction to the C++ programming language and C++ standard library is the third edition of Stroustroup's *The C++ Programming Language*(Stroustrup 1997).

Exercises

1.1 A good way to gain understanding of the system is to follow the process of computing the radiance value for a single ray in the debugger. Build a version of `lrt` with debugging symbols and set up your debugger to run `lrt` with the **XXXX.lrt** scene. Set a breakpoint in the `Scene::Render()` method and trace through the process of how a ray is generated, how its radiance value is computed, and how its contribution is added to the image.

As you gain more understanding of how the details of the system work, return to this and more carefully trace through particular parts of the process.

2. Geometry and Transformations

We now present the fundamental geometric primitives around which `lrt` is built. Our representation of actual scene geometry (triangles, etc.) is presented in Chapter 3; here we will discuss fundamental building blocks of 3D graphics, such as points, vectors, rays, and transformations. Most of this code is stored in `core/geometry.h` and `core/geometry.cpp`, though transformation matrices, defined in Section 2.6 will be implemented in separate source files.

2.0.1 Affine Spaces

As is typical in computer graphics, `lrt` represents 3D points, vectors, and normal vectors with three floating-point coordinate values: x, y, and z. Of course, these values are meaningless without a *coordinate system* that defines the origin of the space and gives three non-parallel vectors for the x, y, and z axes of the space. Together, the origin and three vectors are called the *frame* that defines the coordinate system. Given an arbitrary point or direction in 3D, its (x,y,z) coordinate values depend on its relationship to the frame. Figure 2.1 shows an example that illustrates this idea in 2D.

A frame's origin P_o and its n linearly independent basis vectors define an n-dimensional *affine space*. All vectors $\mathbf{v}$ in the space can be expressed as a linear combination of the basis vectors. Given a vector $\mathbf{v}$ and the basis vectors $\mathbf{v}_i$, we can compute scalar values s_i such that

$$\mathbf{v} = s_1\mathbf{v}_1 + \cdots + s_n\mathbf{v}_n.$$

The scalars s_i are the *representation* of $\mathbf{v}$ with respect to the basis $\{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}$, and are the coordinate values that we store with the vector. Similarly, for all points

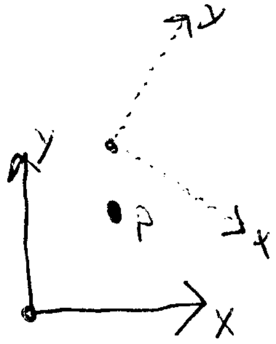


Figure 2.1: In 2D, the (x,y) coordinates of a point p are defined by the relationship of the point to a particular 2D coordinate system. Here, two coordinate systems are shown; the point might have coordinates $(8,8)$ with respect to the coordinate system with its coordinate axes drawn in solid lines, but have coordinates $(2,-4)$ with respect to the coordinate system with dashed axes. In either case, the 2D point p is at the same "absolute" position in space.

p, we can compute scalars s_i such that

$$\mathrm{p} = \mathrm{P}_o + s_1\mathbf{v}_1 + \cdots + s_n\mathbf{v}_n.$$

Thus, although points and vectors are both represented by x, y, and z coordinates in 3D, they are clearly distinct mathematical entities, and are not freely interchangable.

This definition of points and vectors in terms of coordinate systems reveals a paradox: to define a frame we need a point and a set of vectors. But we can only meaningfully talk about points and vectors with respect to a particular frame. Therefore, we need a *standard frame* with origin $(0,0,0)$ and basis vectors $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$. All other frames will be defined with respect this canonical coordinate system. We will call this coordinate system *world space*.

2.0.2 Coordinate System Handedness

XXX Left and right handed coordinate systems: basic idea of what the difference is. `lrt` uses left handed. XXX

There are two different ways that the three coordinate axes can be arranged–having chosen perpindicular x and y, perpindicular z can go in one of two directions. These Two choices have been called *left-handed* and *right-handed*. Figure XXX shows the two possibilities. Idea is that if you take your thumb, index, and middle finger, arrange them as shown in figure XXX, then for a left-handed coordinate system, XXX. This choice has a number of implications in how some of the geometric operations in this chapter are defined...

2.1 Vectors

⟨*Geometry Declarations*⟩≡

```
class COREDLL Vector {
public:
    ⟨Vector Methods⟩
    ⟨Vector Public Data⟩
};
```

A `Vector` in `lrt` represents a direction in 3D space. As described above, we represent vectors with a three-tuple of components that give its representation in terms of the x, y, and z axes of the space it is defined in. The individual components of a vector $\mathbf{v}$ will be written $\mathbf{v}_x$, $\mathbf{v}_y$, and $\mathbf{v}_z$.

⟨*Vector Public Data*⟩≡

```
Float x, y, z;
```

Readers who are experienced in object-oriented design might object to our decision to make the `Vector` data publicly accessible. Typically, data members are only accessible inside the class, and external code that wishes to access or modify the contents of a class must do so through a well-defined API of selector and mutator functions. While we generally agree with this design principle, it is not appropriate here. The purpose of selector and mutator functions is to hide the class's internal implementation details. In the case of `Vector`s, this hiding gains nothing, and adds bulk to the class usage.

By default, the (x,y,z) values are set to zero.

⟨*Vector Methods*⟩≡

```
Vector(Float _x=0, Float _y=0, Float _z=0)
    : x(_x), y(_y), z(_z) {
}
```

2.1.1 Arithmetic

Adding and subtracting vectors is done component-wise. The usual geometric interpretation of vector addition and subtraction is shown in Figures 2.2 and 2.3.

⟨*Vector Methods*⟩+≡

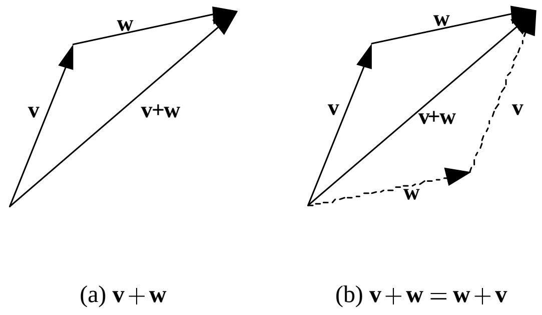
```
Vector operator+(const Vector &v) const {
    return Vector(x + v.x, y + v.y, z + v.z);
}

Vector& operator+=(const Vector &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

The code for subtracting two vectors is similar, and therefore not shown here.

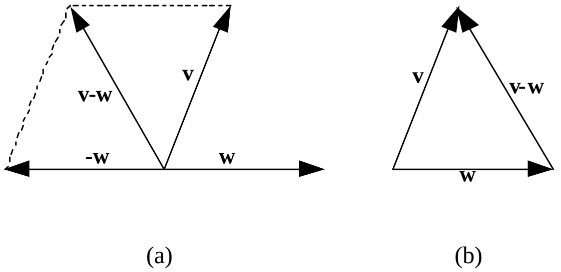
2.1.2 Scaling

We can also multiply a vector component-wise by a scalar, thereby changing its length. Three functions are needed in order to cover all of the different ways that



(a) $\mathbf{v}+\mathbf{w}$ (b) $\mathbf{v}+\mathbf{w}=\mathbf{w}+\mathbf{v}$

Figure 2.2: Vector addition. Notice that the sum $\mathbf{v}+\mathbf{w}$ forms the diagonal of the parallelogram formed by **v** and **w**. Also, the figure on the right shows the commutativity of vector addition.



(a) (b)

Figure 2.3: Vector subtraction. The difference $\mathbf{v}-\mathbf{w}$ is the other diagonal of the parallelogram formed by **v** and **w**.

this operation may be written in source code (i.e. v*s, s*v, and v *= s.)

⟨*Vector Methods*⟩+≡
```
Vector operator*(Float f) const {
    return Vector(f*x, f*y, f*z);
}

Vector &operator*=(Float f) {
    x *= f; y *= f; z *= f;
    return *this;
}
```

⟨*Geometry Inline Functions*⟩+≡
```
inline Vector operator*(Float f, const Vector &v) { return v*f; }
```

Similarly, a vector can be divided component-wise by a scalar. The code for scalar division is similar to scalar multiplication, though division of a scalar by a vector is not well-defined, so is not permitted.

In these methods, we use a single division to compute the scalar's reciprocal, then perform three component-wise multiplications. This is a useful trick for avoiding expensive division operations. It is a common misconception that these sorts of optimizations are unnecessary because the compiler will perform the necessary analysis. Compilers are frequently unable to perform optimizations that require symbolic manipulation of expressions. For example, given two floating point numbers, the quantities a+b and b+a are not candidates for common subexpression elimination, because the IEEE floating point representation cannot guarantee that the two sums will be identical. In fact, some programmers carefully order their floating point addition so as to minimize roundoff error, and it would be a shame for the compiler to undo all that hard work by rearranging a summation.

27 Vector

⟨*Vector Methods*⟩+≡
```
Vector operator/(Float f) const {
    Float inv = 1.f / f;
    return Vector(x * inv, y * inv, z * inv);
}

Vector &operator/=(Float f) {
    Float inv = 1.f / f;
    x *= inv; y *= inv; z *= inv;
    return *this;
}
```

The Vector class also provides a unary negation operator. This returns a new vector pointing in the opposite direction of the original one.

⟨*Vector Methods*⟩+≡
```
Vector operator-() const {
    return Vector(-x, -y, -z);
}
```

Some routines will find it useful to be able to easily loop over the components of a Vector; the Vector class also provides a C++ operator so that given a vector v, then `v[0] == v.x` and so forth. For efficiency, it doesn't check that the offset `i` is within the range $[0,2]$, but will trust calling code to get this right. This non-check is an example of a tradeoff between convenience and performance. While it places an additional burden on the caller, correct code will run faster. One possibility to avoid having to make this tradeoff would be to wrap the range check in a macro that disables the check when `lrt` is compiled with optimizations enabled.

Why not just use assert() here? These get turned off when you compile in optimized mode. Seems wrong. Thoughts?

$\langle$*Vector Methods*$\rangle+\equiv$

```
Float operator[](int i) const { return (&x)[i]; }
Float &operator[](int i) { return (&x)[i]; }
```

2.1.3 Normalization

It is often necessary to *normalize* a vector; that is, to compute a new vector pointing in the same direction but with unit length. A normalized vector is often called a *unit vector*. The method to do this is called `Vector::Hat()`, which is a common mathematical notation for a normalized vector: $\hat{v}$ is the normalized version of $\mathbf{v}$. `Vector::Hat()` simply divides each component by the length of the vector, denoted in text by $\|\mathbf{v}\|$.

Vector 27

Note that `Vector::Hat()` returns a new vector; it does *not* normalize the vector in place.

$\langle$*Vector Methods*$\rangle+\equiv$

```
Float LengthSquared() const { return x*x + y*y + z*z; }
Float Length() const { return sqrtf( LengthSquared() ); }
Vector Hat() const { return (*this)/Length(); }
```

2.1.4 Dot and Cross Product

Two other useful operations on vectors are the dot product (also known as the scalar or inner product) and the cross product. For two vectors $\mathbf{v}$ and $\mathbf{w}$, their *dot product* $(\mathbf{v}\cdot\mathbf{w})$ is defined as

$$\mathbf{v}_x\mathbf{w}_x + \mathbf{v}_y\mathbf{w}_y + \mathbf{v}_z\mathbf{w}_z$$

$\langle$*Geometry Inline Functions*$\rangle+\equiv$

```
inline Float Dot(const Vector &v1, const Vector &v2) {
    return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z;
}
```

The dot product has a simple relationship to the angle between the two vectors **maybe a figure here?**:

$$(\mathbf{v}\cdot\mathbf{w}) = \|\mathbf{v}\|\,\|\mathbf{w}\|\cos\theta, \tag{2.1.1}$$

where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$. It follows from this that $(\mathbf{v}\cdot\mathbf{w})$ is zero if and only if $\mathbf{v}$ and $\mathbf{w}$ are perpendicular, provided that neither $\mathbf{v}$ nor $\mathbf{w}$ is *degenerate*– equal to $(0,0,0)$. A set of two or more mutually-perpendicular vectors is said to be *orthogonal*. An orthogonal set of unit vectors is called *orthonormal*.

It immediately follows from equation 2.1.1 that if **v** and **w** are unit vectors, their dot product is exactly the cosine of the angle between them. As the cosine of the angle between two vectors often needs to be computed in computer graphics, we will frequently make use of this property.

A few basic properties directly follow from the definition. If **u**, **v**, and **w** are vectors and s is a scalar value, then

$$\begin{aligned}(\mathbf{u}\cdot\mathbf{v}) &= (\mathbf{v}\cdot\mathbf{u})\\ (s\mathbf{u}\cdot\mathbf{v}) &= s(\mathbf{v}\cdot\mathbf{u})\\ (\mathbf{u}\cdot(\mathbf{v}+\mathbf{w})) &= (\mathbf{u}\cdot\mathbf{v})+(\mathbf{u}\cdot\mathbf{w})\end{aligned}$$

We will frequently need to compute the absolute value of the dot product as well; the `AbsDot()` function does this for us so that we don't need a separate call to `fabsf()`.

⟨*Geometry Inline Functions*⟩+≡

```
inline Float AbsDot(const Vector &v1, const Vector &v2) {
    return fabsf(v1.x * v2.x + v1.y * v2.y + v1.z * v2.z);
}
```

The *cross product* is another useful vector operation. Given two vectors in 3D, the cross product $\mathbf{v}\times\mathbf{w}$ is a vector that is perpendicular to both of them. Note that this new vector can point in one of two directions; the coordinate system's handedness decides which is appropriate (recall the discussion in Section 2.0.2.) Given orthogonal vectors **v** and **w**, then $\mathbf{v}\times\mathbf{w}$ should return a vector such that $(\mathbf{v},\mathbf{w},\mathbf{v}\times\mathbf{w})$ form a coordinate system of the appropriate handedness.

27 `Vector`

In a left-handed coordinate system, the cross product is defined as:

$$\begin{aligned}(\mathbf{v}\times\mathbf{w})_x &= (\mathbf{v}_y\mathbf{w}_z)-(\mathbf{v}_z\mathbf{w}_y)\\ (\mathbf{v}\times\mathbf{w})_y &= (\mathbf{v}_z\mathbf{w}_x)-(\mathbf{v}_x\mathbf{w}_z)\\ (\mathbf{v}\times\mathbf{w})_z &= (\mathbf{v}_x\mathbf{w}_y)-(\mathbf{v}_y\mathbf{w}_x)\end{aligned}$$

An easy way to remember this is to compute the determinant of the matrix:

$$\mathbf{v}\times\mathbf{w} = \begin{vmatrix} i & j & k\\ \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z\\ \mathbf{w}_x & \mathbf{w}_y & \mathbf{w}_z\end{vmatrix}$$

where i, j, and k represent the axes $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$, respectively. Note that this equation is merely a memory aid and not a rigorous mathematical construction, since the matrix entries are a mix of scalar and vector entries.

⟨*Geometry Inline Functions*⟩+≡

```
inline Vector Cross(const Vector &v1, const Vector &v2) {
    return Vector((v1.y * v2.z) - (v1.z * v2.y),
                  (v1.z * v2.x) - (v1.x * v2.z),
                  (v1.x * v2.y) - (v1.y * v2.x));
}
```

From the definition of the cross product, we can derive:

$$\|\mathbf{v}\times\mathbf{w}\| = \|\mathbf{v}\|\,\|\mathbf{w}\|\sin\theta, \tag{2.1.2}$$

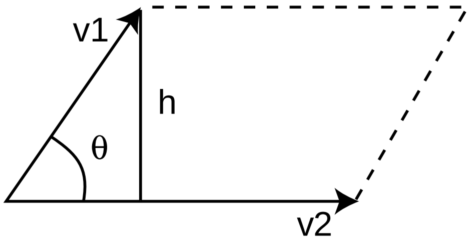


Figure 2.4: The area of a parallelogram with edges given by vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ is equal to $\mathbf{v_2}h$. The cross product can easily compute this value as $\mathbf{v_1}\times\mathbf{v_2}$.

where θ is the angle between $\mathbf{v}$ and $\mathbf{w}$. An important implication of this is that the cross product of two perpendicular unit vectors is itself a unit vector. Note also that the result of the cross product is a degenerate vector if $\mathbf{v}$ and $\mathbf{w}$ are parallel.

This definition also shows a convenient way to compute the area of a parallelogram–see Figure 2.4. If the two edges of the parallelogram are given by vectors $\mathbf{v_1}$ and $\mathbf{v_2}$, and has height h, the area is given by $\|\mathbf{v_2}\|\, h$. Since $h = \sin\theta\,\|\mathbf{v_1}\|$, we can use Equation 2.1.2 to see that the area is $\mathbf{v_1}\times\mathbf{v_2}$.

Vector 27

2.1.5 Coordinate system from a vector

We will frequently want to construct a local coordinate system given only a single vector. Because the cross product of two vectors is orthogonal to both, we can simply apply it twice to get a set of three orthogonal vectors for our coordinate system. Note that the two vectors generated by this technique are only unique up to a rotation about the given vector. This function assumes that the vector passed in, `v1`, has already been normalized.

We first construct a perpendicular vector by zeroing one of the two components of the original vector and swapping the remaining two. Inspection of the two cases should make clear that `v2` will be normalized and that the dot product $(v_1 \cdot v_2)$ will be equal to zero. Given these two perpendicular vectors, a single cross product gives us the third, which by definition will be be perpendicular to the first two.

$\langle$*Geometry Inline Functions*$\rangle+\equiv$

```
inline void CoordinateSystem(const Vector &v1, Vector *v2,
        Vector *v3) {
    if (fabsf(v1.x) > fabsf(v1.y)) {
        Float invLen = 1.f / sqrtf(v1.x*v1.x + v1.z*v1.z);
        *v2 = Vector(-v1.z * invLen, 0.f, v1.x * invLen);
    }
    else {
        Float invLen = 1.f / sqrtf(v1.y*v1.y + v1.z*v1.z);
        *v2 = Vector(0.f, v1.z * invLen, -v1.y * invLen);
    }
    *v3 = Cross(v1, *v2);
}
```

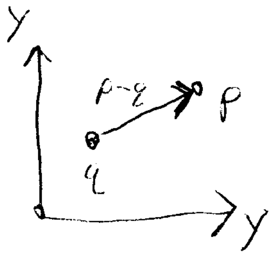


Figure 2.5: Obtaining the vector between two points. The vector p − q is the component-wise subtraction of the points p and q.

2.2 Points

⟨Geometry Declarations⟩+≡

```
class COREDLL Point {
public:
    ⟨Point Methods⟩
    ⟨Point Public Data⟩
};
```

27 Vector

A point is a zero-dimensional location in 3D space. The `Point` class in `lrt` represents points in the obvious way: using x, y, and z coordinates with respect to their coordinate system. Although the same (x,y,z) representation is used for vectors, the fact that a point represents a position, whereas a vector represents a direction, leads to a number of important differences in how they are treated.

⟨Point Public Data⟩≡

```
Float x,y,z;
```

⟨Point Methods⟩≡

```
Point(Float _x=0, Float _y=0, Float _z=0)
    : x(_x), y(_y), z(_z) {
}
```

There are certain `Point` methods which either return or take a `Vector`. For instance, one can add a vector to a point, offsetting it in the given direction and obtaining a new point. Alternately, one can subtract one point from another, obtaining the vector between them, as shown in Figure 2.5.

⟨Point Methods⟩+≡

```
Point operator+(const Vector &v) const {
    return Point(x + v.x, y + v.y, z + v.z);
}

Point &operator+=(const Vector &v) {
    x += v.x; y += v.y; z += v.z;
    return *this;
}
```

⟨*Point Methods*⟩+≡

```
Vector operator-(const Point &p) const {
    return Vector(x - p.x, y - p.y, z - p.z);
}

Point operator-(const Vector &v) const {
    return Point(x - v.x, y - v.y, z - v.z);
}

Point &operator-=(const Vector &v) {
    x -= v.x; y -= v.y; z -= v.z;
    return *this;
}
```

The distance between two points is easily computed by subtracting to compute a vector and then finding the length of that vector.

⟨*Geometry Inline Functions*⟩+≡

```
inline Float Distance(const Point &p1, const Point &p2) {
    return (p1 - p2).Length();
}
inline Float DistanceSquared(const Point &p1, const Point &p2) {
    return (p1 - p2).LengthSquared();
}
```

Point	33
Vector	27

Although it doesn't make sense mathematically to weight points by a scalar or add two points together, the `Point` class still allows these operations in order to be able to compute weighted sums of points, which is mathematically meaningful as long as the weights used all sum to one. The code for scalar multiplication and addition with `Points` is identical to `Vectors`, so it is not shown here.

2.3 Normals

⟨*Geometry Declarations*⟩+≡

```
class COREDLL Normal {
public:
    ⟨Normal Methods⟩
    ⟨Normal Public Data⟩
};
```

A *surface normal* (or just *normal*) is a vector that is perpendicular to a surface at a particular position. It can be defined as the cross product of any two non-parallel vectors that are tangent to the surface at a point. Although normals are superficially similar to vectors, it is important to distinguish between the two of them: because normals are defined in terms of their relationship to a particular surface, they behave differently than vectors in some situations, particularly when applying transformations. This difference is discussed in Section 2.7.

The implementations of `Normals` and `Vectors` are very similar: like vectors, normals are represented by three `Floats` `x`, `y`, and `z`, they can be added and subtracted to compute new normals and they can be scaled and normalized. However, a normal cannot be added to a point and one cannot take the cross product of two normals. Note that in an unfortunate turn of terminology normals are *not* necessarily normalized.

The `Normal` provides an extra constructor that initializes a `Normal` from a `Vector`. Because `Normals` and `Vectors` are different in subtle ways, we want to make sure that this conversion doesn't happen when we don't intend it to. Fortunately, the C++ `explicit` keyword ensures that conversion between two compatible types only happens when that is the intent. The `Vector` also provides a constructor that converts the other way.

⟨*Normal Methods*⟩+≡

```
explicit Normal(const Vector &v)
  : x(v.x), y(v.y), z(v.z) {}
```

⟨*Vector Methods*⟩+≡

```
explicit Vector(const Normal &n);
```

⟨*Geometry Inline Functions*⟩+≡

```
inline Vector::Vector(const Normal &n)
  : x(n.x), y(n.y), z(n.z) { }
```

Thus, given the declarations `Vector v; Normal n;`, the assignment `n = v` is illegal, so it is necessary to explicitly convert the vector, as in `n = Normal(v)`.

The `Dot()` and `AbsDot()` functions are also overloaded to compute dot products between the various possible combinations of normals and vectors. This code won't be included in the text.

34	`Normal`
33	`Point`
36	`Ray`
27	`Vector`

We won't include implementations of all of the various other `Normal` methods here, since they are otherwise similar to those for vectors.

2.4 Rays

⟨*Geometry Declarations*⟩+≡

```
class COREDLL Ray {
public:
    ⟨Ray Public Methods⟩
    ⟨Ray Public Data⟩
};
```

A *ray* is a semi-infinite line specified by its origin and direction. We represent a `Ray` with a `Point` for the origin, and a `Vector` for the direction. A ray is denoted as r; it has origin o(r) and direction **d**(r), as shown in Figure 2.6.

Because we will be referring to these variables often throughout the code, the origin and direction members of a `Ray` are named simply `o` and `d`.

⟨*Ray Public Data*⟩≡

```
Point o;
Vector d;
```

Notice that we again choose to make the data publicly available for convenience.

The *parametric form* of a ray expresses it as a function of a scalar value t, giving the set of points that the ray passes through:

$$\mathrm{r}(t) = \mathrm{o}(\mathrm{r}) + t\mathbf{d}(\mathrm{r}) \qquad 0 \le t \le \infty \tag{2.4.3}$$

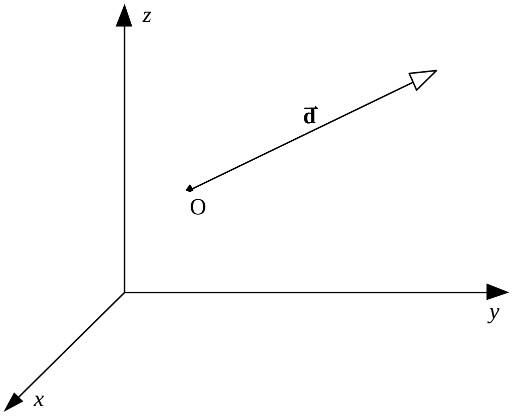


Figure 2.6: A ray is a semi-infinite line defined by its origin $\mathrm{o}(\mathrm{r})$ and its direction $\mathbf{d}(\mathrm{r})$.

Point	33
RAY_EPSILON	37
Vector	27

The Ray also includes fields to restrict the ray to a particular segment along its infinite extent. These fields, called mint and maxt, allow us to restrict the ray to a potentially finite segment of points $[\mathrm{r}(\mathrm{mint}), \mathrm{r}(\mathrm{maxt}))$. Notice that these fields are declared as mutable, meaning that they can be changed even if the Ray structure that contains them is const. **why is this useful?**

⟨*Ray Public Data*⟩+≡

```
  mutable Float mint, maxt;
```

For simulating motion blur, each ray may have a unique time value associated with it. The rest of the renderer is responsible for constructing a representation of the scene at the appropriate time for each ray.

⟨*Ray Public Data*⟩+≡

```
  Float time;
```

Constructing Rays is straightforward. The default constructor relies on the Point and Vector constructors to set the origin and direction to $(0,0,0)$. Alternately, a particular point and direction can be provided. Also note that mint is initialized to a small constant rather than 0. The reason for this is discussed in Section **XXX**–it is a classic ray tracing hack to avoid false self-intersections due to floating point precision limitations.

It's weird that the default ray constructor makes a degenerate ray with direction (0,0,0). Should we either fix this or say something?

⟨*Ray Public Methods*⟩≡

```
  Ray(): mint(RAY_EPSILON), maxt(FLT_MAX), time(0.f) {}
  Ray(const Point &origin, const Vector &direction,
      Float start = RAY_EPSILON, Float end = FLT_MAX, Float t = 0.f)
      : o(origin), d(direction), mint(start), maxt(end), time(t) {
  }
```

The constant to use for the initial mint is arbitrary; no single constant will solve the false self-intersection problem. It needs to be small enough to not miss true intersections, but large enough to overcome most precision errors. For any given constant, it is easy to construct a scene that will not work. There are more sophisticated techniques for solving the false self-intersection problem; see **BLAH AND BLAH**.

⟨*Global Constants*⟩≡

```
  #define RAY_EPSILON 1e-3f
```

Because a ray can be thought of as a function of a single parameter *t*, the `Ray` class overloads the function application operator for rays. This way, when we need to find the point at a particular position along a ray, we can write code like:

```
Ray r(Point(0,0,0), Vector(1,2,3));
Point p = r(1.7);
```

⟨*Ray Public Methods*⟩+≡

```
  Point operator()(Float t) const { return o + d * t; }
```

2.4.1 Ray differentials

In order to be able perform better anti-aliasing with the texture functions defined in Chapter 11, `lrt` keeps track of some additional information with each camera ray that it traces. In Section 11.2, this information will be used by the `Texture` class to estimate the projected area on the image plane of a part of the scene. From this, the `Texture` can compute the texture's average value over that area, leading to a better final image.

33	`Point`
36	`Ray`
394	`Texture`
27	`Vector`

A `RayDifferential` is a subclass of `Ray` that merely carries along additional information about two auxiliary rays. These two extra rays represent camera rays offset one pixel in the *x* and *y* direction from the main ray. By determining the area that these three rays project to on an object being shaded, the `Texture` can estimate an area to average over for proper anti-aliasing.

Because the `RayDifferential` class inherits from `Ray`, geometric interfaces in the system are written to take `const Ray &` values, so that either a `Ray` or `RayDifferential` can be passed to them and the routines can just treat either as a `Ray`. Only the routines related to anti-aliasing and texturing require `RayDifferential` parameters.

⟨*Geometry Declarations*⟩+≡

```
  class COREDLL RayDifferential : public Ray {
  public:
      ⟨RayDifferential Methods⟩
      ⟨RayDifferential Public Data⟩
  };
```

⟨*RayDifferential Methods*⟩≡

```
  RayDifferential() { hasDifferentials = false; }
  RayDifferential(const Point &org, const Vector &dir) : Ray(org, dir) {
      hasDifferentials = false;
  }
```

Note that we again use the `explicit` keyword to prevent `Ray`s from accidentally being converted to `RayDifferential`s. The constructor sets `hasDifferentials` to false initially, because the neighboring rays are not yet known. These fields are initialized by the renderer's main loop, in the code fragment ⟨*Generate ray differentials for camera ray*⟩, on page 12.

⟨*RayDifferential Methods*⟩+≡
```
explicit RayDifferential(const Ray &ray) : Ray(ray) {
    hasDifferentials = false;
}
```

⟨*RayDifferential Public Data*⟩≡
```
bool hasDifferentials;
Ray rx, ry;
```

2.5 Three-dimensional bounding boxes

Why isn't the naming in 3D consistent with the naming in 2D? We should fix this.

⟨*Geometry Declarations*⟩+≡
```
class COREDLL BBox {
public:
    ⟨BBox Public Methods⟩
    ⟨BBox Public Data⟩
};
```

Ray	36
RayDifferential	37

The scenes that `lrt` will render will often contain objects that are computationally expensive to process. For many operations, it is often useful to have a three-dimensional *bounding volume* that encloses an object. If a ray does not pass through a particular bounding volume, `lrt` can avoid processing all of the objects inside of it.

The measurable benefit of this technique is related to two factors: the expense of processing the bounding volume compared to the expense of processing the objects inside of it, and the tightness of the fit. If we have a very loose bound around an object, we will often incorrectly determine that its contents need to be examined further. However, in order to make the bounding volume a closer fit, it may be necessary to make the volume a complex object itself, and the expense of processing it increases.

There are many choices for bounding volumes; we will be using *axis-aligned bounding boxes* (AABBs). Other popular choices are spheres and *oriented bounding boxes* (OBBs). An AABB can be described by one of its vertices and three lengths, each representing the distance spanned along the *x*, *y*, and *z* coordinate axes. Alternatively, two opposite vertices of the box describe it. We chose the two-point representation for `lrt`'s `BBox` class **(WHY)**; it stores the positions of the vertex with minimum *x*, *y*, and *z* values, and the one with maximum *x*, *y*, and *z*. A 2D illustration of a bounding box and its representation is shown in Figure 2.7.

The default `BBox` constructor sets the extent to be degenerate; by violating the invariant that `pMin.x <= pMax.x`, etc., it ensures than any operations done with this box will have the correct result for a completely empty box.

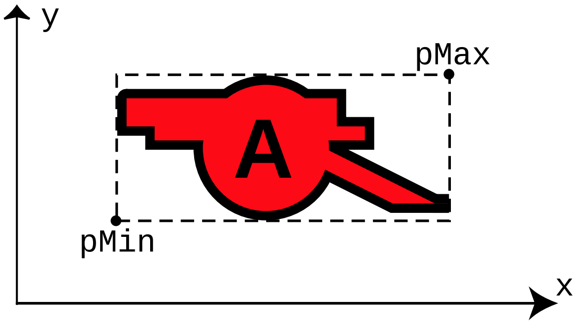


Figure 2.7: An example axis-aligned bounding box. The `BBox` stores only the coordinates of the minimum and maximum points of this box; all other box corners are implicit in this representation.

⟨*BBox Public Methods*⟩≡

```
BBox() {
    pMin = Point( INFINITY,  INFINITY,  INFINITY);
    pMax = Point(-INFINITY, -INFINITY, -INFINITY);
}
```

⟨*BBox Public Data*⟩≡

```
Point pMin, pMax;
```

38	`BBox`
678	`INFINITY`
33	`Point`

It is also useful to be able to initialize a `BBox` to enclose a single point.

⟨*BBox Public Methods*⟩+≡

```
BBox(const Point &p) : pMin(p), pMax(p) { }
```

If the caller passes two corner points (`p1` and `p2`) to define the box, since `p1` and `p2` are not necessarily chosen so that `p1.x <= p2.x` etc, we need to find their minimum and maximum values component-wise.

⟨*BBox Public Methods*⟩+≡

```
BBox(const Point &p1, const Point &p2) {
    pMin = Point(min(p1.x, p2.x),
                 min(p1.y, p2.y),
                 min(p1.z, p2.z));
    pMax = Point(max(p1.x, p2.x),
                 max(p1.y, p2.y),
                 max(p1.z, p2.z));
}
```

Given a bounding box and a point, the `BBox::Union()` methods computes and returns a new bounding box that encompasses that point as well as the space that the original box encompassed.

⟨*BBox Method Definitions*⟩≡

```
BBox Union(const BBox &b, const Point &p) {
    BBox ret = b;
    ret.pMin.x = min(b.pMin.x, p.x);
    ret.pMin.y = min(b.pMin.y, p.y);
    ret.pMin.z = min(b.pMin.z, p.z);
    ret.pMax.x = max(b.pMax.x, p.x);
    ret.pMax.y = max(b.pMax.y, p.y);
    ret.pMax.z = max(b.pMax.z, p.z);
    return ret;
}
```

We can similarly construct a new box that bounds the space encompassed by two other bounding boxes. The definition of this function is similar to the `Union()` method above that takes a `Point`; the difference is the `pMin` and `pMax` of the second box are used for the `min()` and `max()` tests, respectively.

⟨*BBox Public Methods*⟩+≡

```
friend BBox Union(const BBox &b, const BBox &b2);
```

We can easily determine if two `BBox`es overlap by seeing if their extents overlap in *x*, *y*, and *z*.

BBox	38
BBox::pMax	39
BBox::pMin	39
Point	33
Vector	27

⟨*BBox Public Methods*⟩+≡

```
bool Overlaps(const BBox &b) const {
    bool x = (pMax.x >= b.pMin.x) && (pMin.x <= b.pMax.x);
    bool y = (pMax.y >= b.pMin.y) && (pMin.y <= b.pMax.y);
    bool z = (pMax.z >= b.pMin.z) && (pMin.z <= b.pMax.z);
    return (x && y && z);
}
```

Three simple 1D containment tests tell us if a given point is inside the bounding box.

⟨*BBox Public Methods*⟩+≡

```
bool Inside(const Point &pt) const {
    return (pt.x >= pMin.x && pt.x <= pMax.x &&
            pt.y >= pMin.y && pt.y <= pMax.y &&
            pt.z >= pMin.z && pt.z <= pMax.z);
}
```

The `BBox::Expand()` method pads the bounding box by a constant factor, and `BBox::Volume()` returns the volume of the space inside the box.

⟨*BBox Public Methods*⟩+≡

```
void Expand(Float delta) {
    pMin -= Vector(delta, delta, delta);
    pMax += Vector(delta, delta, delta);
}
```

⟨*BBox Public Methods*⟩+≡

```
Float Volume() const {
    Vector d = pMax - pMin;
    return d.x * d.y * d.z;
}
```

The Bbox::MaximumExtent() method tells the caller which of the three axes is longest. This is very useful, for example, when deciding along which axis to subdivide when building a k-D tree.

⟨*BBox Public Methods*⟩+≡
```
int MaximumExtent() const {
    Vector diag = pMax - pMin;
    if (diag.x > diag.y && diag.x > diag.z)
        return 0;
    else if (diag.y > diag.z)
        return 1;
    else
        return 2;
}
```

Finally, the BBox provides a method that returns the center and radius of a sphere that bounds the bounding box. In general, this may give a far looser fit than a sphere that bounded the original contents of the BBox directly, though it's a useful method to have available. For example, in chapter 15, we use this method to get a sphere that completely bounds the scene in order to generate a random ray that is likely to intersect the scene geometry.

Maybe this method should move to that chapter?

⟨*BBox Method Definitions*⟩+≡
```
void BBox::BoundingSphere(Point *c, Float *rad) const {
    *c = .5f * pMin + .5f * pMax;
    *rad = Distance(*c, pMax);
}
```

38	BBox
39	BBox::pMax
39	BBox::pMin
33	Point
27	Vector

2.6 Transformations

In general, a *transformation* **T** can be described as a mapping from points to points and from vectors to vectors:

$$\mathrm{p}' = \mathbf{T}(\mathrm{p}) \qquad \mathbf{v}' = \mathbf{T}(\mathbf{v})$$

The transformation **T** may be an arbitrary procedure. However, we will consider a subset of all possible transformations in this chapter. In particular, they will be:

- Linear: If **T** is an arbitrary linear transformation and s is an arbitrary scalar, then $\mathbf{T}(s\mathbf{v}) = s\mathbf{T}(\mathbf{v})$ and $\mathbf{T}(\mathbf{v}_1 + \mathbf{v}_2) = \mathbf{T}(\mathbf{v}_1) + \mathbf{T}(\mathbf{v}_2)$. These two properties can greatly simplify reasoning about transformations.

- Continuous: roughly speaking, **T** maps the neighborhoods around p and **v** to ones around p′ and **v**′.

- One-to-one and invertible: for each p, **T** maps p to a single unique p′. Furthermore, there exists an inverse transform $\mathbf{T}^{-1}$ that maps p′ back to p.

We will often want to take a point, vector, or normal defined with respect to one coordinate frame and find its coordinate values with respect to another frame.

Using basic properties of linear algebra, a 4*x*4 matrix can express the linear transformation of a point or vector from one frame to another. Furthermore, such a 4*x*4 matrix suffices to express all linear transformations of points and vectors within a fixed frame, such as translation in space or rotation around a point. Therefore, there are two different (and incompatible!) ways that a matrix can be interpreted:

1. *Transformation of the frame*: given a point, the matrix could express how to compute a *new* point in the same frame that represents the transformation of the original point (e.g. by translating it in some direction.)

2. *Transformation from one frame to another*: a matrix can express how a point in a new frame is computed given a point in an original frame.

In general, transformations make it possible to work in the most convenient coordinate space. For example, we can write routines that define a virtual camera assuming that the camera is located at the origin, looks down the z axis, and has the y axis pointing up and the x axis pointing right. These assumptions greatly simplify the camera implementation. Then to place the camera at any point in the scene looking in any direction, we just construct a transformation that maps points in the scene's coordinate system to the camera's coordinate system.

2.6.1 Homogeneous coordinates

Given a frame defined by $(\mathrm{p}, \mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3})$, there is ambiguity between the representation of a point $(\mathrm{p}_x, \mathrm{p}_y, \mathrm{p}_z)$ and a vector $(\mathbf{v}_x, \mathbf{v}_y, \mathbf{v}_z)$ with the same (x, y, z) coordinates. Using the representations of points and vectors introduced at the start of the chapter, we can write the point as the inner product $[s_1\, s_2\, s_3\, 1][\mathbf{v_1}\, \mathbf{v_2}\, \mathbf{v_3}\, \mathrm{p}]^T$ and the vector as the inner product $[s_1'\, s_2'\, s_3'\, 0][\mathbf{v_1}\, \mathbf{v_2}\, \mathbf{v_3}\, \mathrm{p}]^T$ These four-vectors of three s_i values and a zero or one are *homogeneous* representations of the point and the vector. The fourth coordinate of the homogeneous representation is sometimes called the *weight*. For a point, its value can be any scalar other than zero: the homogeneous points $[1, 3, -2, 1]$ and $[-2, -6, 4, -2]$ describe the same Cartesian point $(1, 3, -2)$. In general, homogeneous points obey the identity:

$$(x, y, z, w) = \left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}\right)$$

We will use these facts to see how a transformation matrix can describe how points and vectors in one frame can be mapped to another frame. Consider a matrix $\mathbf{M}$ that describes the transformation from one coordinate system to another:

$$\mathbf{M} = \begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{23} & m_{33} \end{pmatrix}$$

Then if the transformation represented by $\mathbf{M}$ is applied to the x axis $(1, 0, 0)$, we have:

$$\mathbf{M}[1\,0\,0\,0]^T = [m_{00}\, m_{10}\, m_{20}\, m_{30}]^T.$$

Directly reading the columns of the matrix shows how the basis vectors and the origin of the current coordinate system are transformed by the matrix.

$$\begin{array}{rcl}\mathbf{x} & = & [1\,0\,0\,0]^T \\ \mathbf{y} & = & [0\,1\,0\,0]^T \\ \mathbf{z} & = & [0\,0\,1\,0]^T \\ \mathrm{p} & = & [0\,0\,0\,1]^T\end{array}$$

In general, by characterizing how the basis is transformed, we know how any point or vector specified in terms of that basis is transformed. Because points and vectors in the current coordinate system are expressed in terms of the current coordinate system's frame, applying the transformation to them directly is equivalent to applying the transformation to the current coordinate system's basis and finding their coordinates in terms of the transformed basis.

We will not use homogeneous coordinates explicitly in our code; there is no `Homogeneous` class. However, the various transformation routines in the next section will implicitly convert points, vectors, and normals to homogeneous form, transform the homogeneous points, and then convert them back before returning the result. This isolates the details of homogeneous coordinates in one place (namely, the implementation of transformations), and leaves the rest of the system clean.

```
⟨Transform Declarations⟩≡
  class COREDLL Transform {
  public:
      ⟨Transform Public Methods⟩
  private:
      ⟨Transform Private Data⟩
  };
```

675 Matrix4x4

A transformation is represented by the elements of the matrix `m[4][4]`, a `Reference<>` to a `Matrix4x4` object. The automatic reference-counting template class `Reference<>` is described in Appendix A.3.2; it tracks how many objects hold a reference to the reference-counted object and automatically frees its memory when no more references are held.

The low-level `Matrix4x4` class is defined in Appendix A.4.2. `m` is stored in *row-order* form, so element `m[i][j]` corrsponds to $m_{i,j}$, where i is the row number and j is the column number. For convenience, the `Transform` also stores the inverse of the matrix `m` in the `Transform::m_inv` member; it will be handy to have the inverse easily available. Note that it would be possible to compute the inverse of the matrix lazily, in case it is not needed. We don't do this because in practice we find that the inverse of the matrix is almost always needed. Also, most of the transformations in `lrt` explicitly provide their inverse, so actual matrix inversion is rarely required.

`Transform` stores references to matrices rather than storing them directly, so that multiple `Transforms` can point to the same matrices. This means that any instance of the `Transform` class takes up very little memory on its own. If a huge number of shapes in the scene have the same object-to-world transformation, then they can all have their own `Transform` objects but share the same `Matrix4x4`s. Since we only store a pointer to the matrices and a reference count, a second transform that can re-use an existing matrix saves 72 bytes of storage over an implementation where each shape has its own `Matrix4x4`. This savings can be substantial in large scenes.

However, we lose a certain amount of flexibility by allowing matrices to be shared between transformations. Specifically, the elements of a `Matrix4x4` cannot be modified after it is created. This isn't a problem in practice, since the transformations in a scene are typically created when `lrt` parses the scene decscription file and don't need to change later at rendering time.

⟨*Transform Private Data*⟩≡

```
Reference<Matrix4x4> m, m_inv;
```

2.6.2 Basic operations

When a new `Transform` is created, it will default to the *identity transformation*: the transformation that maps each point and each vector to itself. This is represented by the *identity matrix*:

$$\mathbf{I} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Matrix4x4	675
Reference	664
Transform	43

Note that we rely on the default `Matrix4x4` constructor to fill in the identity matrix.

⟨*Transform Public Methods*⟩≡

```
Transform() {
    m = m_inv = new Matrix4x4;
}
```

We can also construct a `Transform` from a given matrix. In this case, we must explicitly invert the given matrix.

⟨*Transform Public Methods*⟩+≡

```
Transform(Float mat[4][4]) {
    m = new Matrix4x4(mat[0][0], mat[0][1], mat[0][2], mat[0][3],
                      mat[1][0], mat[1][1], mat[1][2], mat[1][3],
                      mat[2][0], mat[2][1], mat[2][2], mat[2][3],
                      mat[3][0], mat[3][1], mat[3][2], mat[3][3]);
    m_inv = m->Inverse();
}
```

⟨*Transform Public Methods*⟩+≡

```
Transform(const Reference<Matrix4x4> &mat) {
    m = mat;
    m_inv = m->Inverse();
}
```

Finally, the most commonly used constructor will simply take a reference to the transformation matrix along with an explictly provided inverse. This is far superior to always computing the inverse in the constructor, because many geometric transformations have very simple inverses and we can avoid the expense of computing a

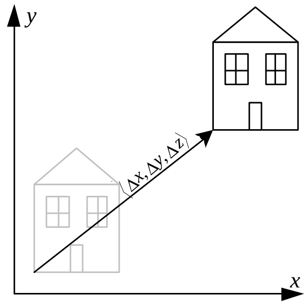


Figure 2.8: Translation in 2D.

generic 4×4 matrix inverse. Of course, this places the burden on the caller to make sure that the supplied inverse is correct.

⟨*Transform Public Methods*⟩+≡

```
Transform(const Reference<Matrix4x4> &mat,
          const Reference<Matrix4x4> &minv) {
    m = mat;
    m_inv = minv;
}
```

```
675 Matrix4x4
664 Reference
 43 Transform
```

2.6.3 Translations

One of the simplest transformations is the translation $\mathbf{T}(\Delta x,\Delta y,\Delta z)$. When applied to a point p, it translates p's coordinates by Δx, Δy, and Δz, as shown in Figure 2.8. As an example, $\mathbf{T}(2,2,1)(x,y,z) = (x+2,y+2,z+1)$.

The translation has some simple properties:

$$\begin{aligned}
\mathbf{T}(0,0,0) &= \mathbf{I} \\
\mathbf{T}(x_1,y_1,z_1)\times\mathbf{T}(x_2,y_2,z_2) &= \mathbf{T}(x_1+x_2,y_1+y_2,z_1+z_2) \\
\mathbf{T}(x_1,y_1,z_1)\times\mathbf{T}(x_2,y_2,z_2) &= \mathbf{T}(x_2,y_2,z_2)\times\mathbf{T}(x_1,y_1,z_1) \\
\mathbf{T}^{-1}(x,y,z) &= \mathbf{T}(-x,-y,-z)
\end{aligned}$$

Translation only affects points, leaving vectors unchaged. In matrix form, the translation transformation is:

$$\mathbf{T}(\Delta x,\Delta y,\Delta z) = \begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

When we consider the operation of a translation matrix on a point, we see the value of homogeneous coordinates. Consider the product of the matrix for

$\mathbf{T}(\Delta x, \Delta y, \Delta z)$ with a point p in homogeneous coordinates $[x y z 1]$:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x+\Delta x \\ y+\Delta y \\ z+\Delta z \\ 1 \end{pmatrix}$$

As expected, we have computed a new point with its coordinates offset by $(\Delta x, \Delta y, \Delta z)$. However, if we apply **T** to a vector **v**, we have:

$$\begin{pmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 0 \end{pmatrix}$$

The result is the same vector **v**. This makes sense, because vectors represent directions, so a translation leaves them unchanged.

We will define a routine that creates a new `Transform` matrix to represent a given translation–it is a straightforward application of the translation matrix equation. These routines fully initialize the `Transform` that is returned, also initializing the matrix that represents the inverse of the translation.

`Matrix4x4`	675
`Transform`	43
`Vector`	27

⟨*Transform Method Definitions*⟩+≡

```
Transform Translate(const Vector &delta) {
    Matrix4x4 *m, *minv;
    m = new Matrix4x4(1, 0, 0, delta.x,
                      0, 1, 0, delta.y,
                      0, 0, 1, delta.z,
                      0, 0, 0,       1);
    minv = new Matrix4x4(1, 0, 0, -delta.x,
                         0, 1, 0, -delta.y,
                         0, 0, 1, -delta.z,
                         0, 0, 0,        1);
    return Transform(m, minv);
}
```

2.6.4 Scaling

Another basic transformation is the *scale transform*. This has the effect of taking a point or vector and multiplying its components by scale factors in x, y, and z: $\mathbf{S}(2,2,1)(x,y,z) = (2x, 2y, z)$. It has the following basic properties:

$$\begin{aligned} \mathbf{S}(1,1,1) &= \mathbf{I} \\ \mathbf{S}(x_1,y_1,z_1) \times \mathbf{S}(x_2,y_2,z_2) &= \mathbf{S}(x_1x_2, y_1y_2, z_1,z_2) \\ \mathbf{S}^{-1}(x,y,z) &= \mathbf{S}\left(\frac{1}{x}, \frac{1}{y}, \frac{1}{z}\right) \end{aligned}$$

We can differentiate between *uniform scaling*, where all three scale factors have the same value and *non-uniform scaling*, where they may have different values.

The general scale matrix is

$$\mathbf{S}(x,y,z) = \begin{pmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

⟨*Transform Method Definitions*⟩+≡

```
Transform Scale(Float x, Float y, Float z) {
    Matrix4x4 *m, *minv;
    m = new Matrix4x4(x, 0, 0, 0,
                      0, y, 0, 0,
                      0, 0, z, 0,
                      0, 0, 0, 1);
    minv = new Matrix4x4(1.f/x, 0, 0, 0,
                         0, 1.f/y, 0, 0,
                         0, 0, 1.f/z, 0,
                         0, 0, 0, 1);
    return Transform(m, minv);
}
```

675 Matrix4x4
43 Transform

2.6.5 X, Y, and Z axis rotations

Another useful type of transformation is rotation. In general, we can define an arbitrary axis from the origin in any direction and then rotate around that axis by a given angle. The most common rotations of this type are around the x, y, and z coordinate axes. We will write these rotations as $\mathbf{R}_x(\theta)$, or $\mathbf{R}_y(\theta)$, etc. The rotation around an arbitrary axis (x,y,z) is denoted by $\mathbf{R}_{(x,y,z)}(\theta)$.

Rotations also have some basic properties:

$$\begin{aligned} \mathbf{R}_a(0) &= \mathbf{I} \\ \mathbf{R}_a(\theta_1) \times \mathbf{R}_a(\theta_2) &= \mathbf{R}_a(\theta_1 + \theta_2) \\ \mathbf{R}_a(\theta_1) \times \mathbf{R}_a(\theta_2) &= \mathbf{R}_a(\theta_2) \times \mathbf{R}_a(\theta_1) \\ \mathbf{R}_a^{-1}(\theta) &= \mathbf{R}_a(-\theta) = \mathbf{R}_a^T(\theta) \end{aligned}$$

where $\mathbf{R}^T$ is the matrix transpose of $\mathbf{R}$. This last property, that the inverse of $\mathbf{R}$ is equal to its transpose, stems from the fact that $\mathbf{R}$ is an *orthonormal matrix*; its upper 3×3 components are all normalized and orthogonal to each other. Fortunately, the transpose is much easier to compute than a full matrix inverse.

For a left-handed coordinate system, the matrix for rotation around the x axis is:

$$\mathbf{R}_x(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Figure 2.9 gives an intuition for how this matrix works. It's easy to see that it leaves the x axis unchanged:

$$\mathbf{R}_x(\theta) \cdot [1\,0\,0\,0]^T = [1\,0\,0\,0]$$

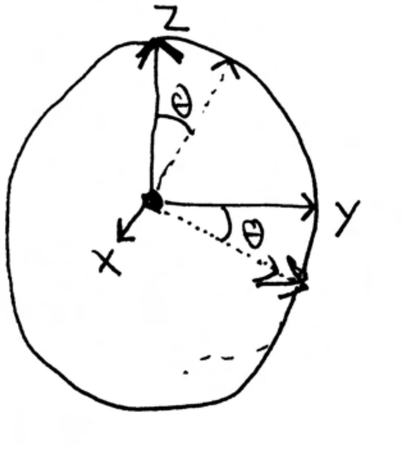


Figure 2.9: Rotation by an angle θ about the *x* axis leaves the *x* coordinate unchanged. The *y* and *z* axes are mapped to the vertices given by the dashed lines; *y* and *z* coordinates move accordingly.

Matrix4x4 675
Transform 43

It maps the *y* axis $(0,1,0)$ to $(0,\cos\theta,\sin\theta)$ and the *z* axis to $(0,-\sin\theta,\cos\theta)$. In general, by reading the columns of $\mathbf{R}_x(\theta)$ we can easily find the vectors that the original coordinate axes transform to. The *y* and *z* axes remain in the same plane, perpendicular to the *x* axis, but are rotated by the given angle. An arbitrary point in space is similarly rotated about *x* while staying in the same *yz* plane as it was originally.

The implementation of the `RotateX()` creation function is straightforward.

⟨*Transform Method Definitions*⟩+≡

```
Transform RotateX(Float angle) {
    Float sin_t = sinf(Radians(angle));
    Float cos_t = cosf(Radians(angle));
    Matrix4x4 *m = new Matrix4x4(1,     0,      0, 0,
                                 0, cos_t, -sin_t, 0,
                                 0, sin_t,  cos_t, 0,
                                 0,     0,      0, 1);
    return Transform(m, m->Transpose());
}
```

Similarly, for rotation around *y* and *z*, we have

$$\mathbf{R}_y(\theta) = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad \mathbf{R}_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The implementations of `RotateY()` and `RotateZ()` follow directly and are not included here.

2.6.6 Rotation around an arbitrary axis

We also provide a routine to compute the transformation that represents rotation around an arbitrary axis. The usual derivation of this matrix is based on computing

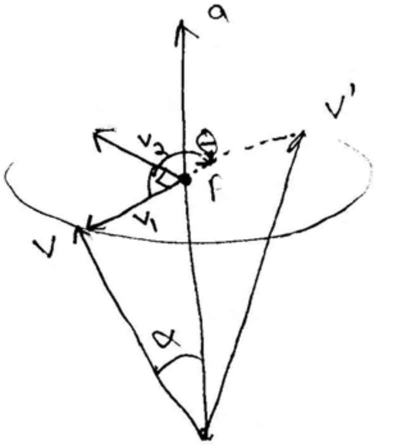


Figure 2.10: Rotation about an arbitrary axis **a**: ...

rotations that map the given axis to a fixed axis (e.g. *z*), performing the rotation there, and then rotating the fixed axis back to the original axis. A more elegant derivation can be constructed with vector algebra.

Consider a normalized direction vector **a** that gives the axis to rotate around by angle θ, and a vector **v** to be rotated (see Figure 2.10). First, we can compute the point p along the axis **a** that is in the plane through the end-point of **v** and is perpendicular to **a**. Assuming **v** and **a** form an angle α, we have:

$$\mathrm{p} = \mathbf{a}\cos\alpha = \mathbf{a}(\mathbf{v}\cdot\mathbf{a}).$$

We now compute a pair of basis vectors $\mathbf{v_1}$ and $\mathbf{v_2}$ in this plane. Trivially, one of them is

$$\mathbf{v_1} = \mathbf{v} - \mathbf{p}$$

and the other can be computed with a cross product

$$\mathbf{v_2} = (\mathbf{v_1} \times \mathbf{a}).$$

Because **a** is normalized, $\mathbf{v_1}$ and $\mathbf{v_2}$ have the same length, equal to the distance from **v** to p. To now compute the rotation by θ degrees about the point p in the plane of rotation, the rotation formulas above give us

$$\mathbf{v}' = \mathrm{p} + \mathbf{v_1}\cos\theta + \mathbf{v_2}\sin\theta.$$

To convert this to a rotation matrix, we apply this formula to the basis vectors $\mathbf{v_1} = (1,0,0)$, $\mathbf{v_2} = (0,1,0)$, and $\mathbf{v_3} = (0,0,1)$ to get the values of the rows of the matrix. The result of all this is encapsulated in the function below.

Should this say colums, not rows?

⟨*Transform Method Definitions*⟩+≡

```
Transform Rotate(Float angle, const Vector &axis) {
    Vector a = axis.Hat();
    Float s = sinf(Radians(angle));
    Float c = cosf(Radians(angle));
    Float m[4][4];

    m[0][0] = a.x * a.x + (1.f - a.x * a.x) * c;
    m[0][1] = a.x * a.y * (1.f - c) - a.z * s;
    m[0][2] = a.x * a.z * (1.f - c) + a.y * s;
    m[0][3] = 0;

    m[1][0] = a.x * a.y * (1.f - c) + a.z * s;
    m[1][1] = a.y * a.y + (1.f - a.y * a.y) * c;
    m[1][2] = a.y * a.z * (1.f - c) - a.x * s;
    m[1][3] = 0;

    m[2][0] = a.x * a.z * (1.f - c) - a.y * s;
    m[2][1] = a.y * a.z * (1.f - c) + a.x * s;
    m[2][2] = a.z * a.z + (1.f - a.z * a.z) * c;
    m[2][3] = 0;

    m[3][0] = 0;
    m[3][1] = 0;
    m[3][2] = 0;
    m[3][3] = 1;

    Matrix4x4 *mat = new Matrix4x4(m);
    return Transform(mat, mat->Transpose());
}
```

Matrix4x4	675
Transform	43
Vector	27

2.6.7 The look-at transformation

The *look-at transformation* is particularly useful for placing a camera in the scene. The caller specifies the desired position of the camera, a point the camera is looking at, and an “up” vector that orients the camera along the viewing direction implied by the first two parameters. All of these values are given in world-space coordinates. The look-at construction then gives a transformation between camera space and world space (see Figure 2.11).

In order to find the entries of the look-at transformation, we use principles described earlier in this section: the columns of a transformation matrix give the effect of the transformation on the basis of a coordinate system.

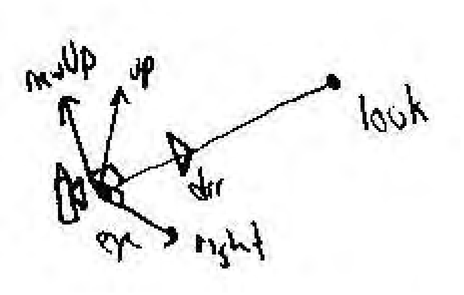


Figure 2.11: Given an camera position, the position being looked at from the camera, and an “up” direction, the look-at transformation describes a transformation from a viewing coordinate system where the camera is at the origin looking down the $+z$ axis and the $+y$ axis is along the up direction.

⟨*Transform Method Definitions*⟩+≡

```
Transform LookAt(const Point &pos, const Point &look,
        const Vector &up) {
    Float m[4][4];
    ⟨Initialize fourth column of viewing matrix⟩
    ⟨Initialize first three columns of viewing matrix⟩
    Matrix4x4 *camToWorld = new Matrix4x4(m);
    return Transform(camToWorld->Inverse(), camToWorld);
}
```

675	Matrix4x4
676	Matrix4x4::Inverse()
33	Point
43	Transform
27	Vector

The easiest column is the fourth one, which gives the point that the camera-space origin, $[0\,0\,0\,1]^T$, maps to in world space. This is clearly just the coordinates of the camera position, supplied by the user.

⟨*Initialize fourth column of viewing matrix*⟩≡

```
m[0][3] = pos.x;
m[1][3] = pos.y;
m[2][3] = pos.z;
m[3][3] = 1;
```

The other three columns aren’t much more difficult. First, `LookAt()` computes the normalized direction vector from the camera location to the look-at point; this gives the vector coordinates that the z axis should map to and thus, the third column of the matrix. (Camera space is defined with the viewing direction down the $+z$ axis.) The first column, giving the world space direction that the $+x$ axis in camera space maps to, is found by taking the cross product of the user-supplied “up”’ vector with the recently computed viewing direction vector. Finally, the “up” vector is recomputed by taking the cross product of the viewing direction vector with the x axis vector, thus ensuring that the y and z axes are perpendicular and we have an orthonormal viewing coordinate system.

⟨*Initialize first three columns of viewing matrix*⟩≡

```
  Vector dir = (look - pos).Hat();
  Vector right = Cross(dir, up.Hat());
  Vector newUp = Cross(right, dir);
  m[0][0] = right.x;
  m[1][0] = right.y;
  m[2][0] = right.z;
  m[3][0] = 0.;
  m[0][1] = newUp.x;
  m[1][1] = newUp.y;
  m[2][1] = newUp.z;
  m[3][1] = 0.;
  m[0][2] = dir.x;
  m[1][2] = dir.y;
  m[2][2] = dir.z;
  m[3][2] = 0.;
```

2.7 Applying Transforms

Cross()	31
Vector	27
Vector::Hat()	30

We can now define routines that perform the appropriate matrix multiplications to transform points and vectors. We will overload the function application operator to describe these transformations; this lets us write code like:

```
Point P = ...;
Transform T = ...;
Point new_P = T(P);
```

2.7.1 Points

The point transformation routine takes a point (x, y, z) and implicitly represents it as the homogeneous column vector $[x, y, z, 1]^T$. It then transforms the point by premultiplying this vector with its transformation matrix. Finally, it divides by w to convert back to a non-homogeneous point representation.

For efficiency, it skips the divide by the homogeneous weight w when $w = 1$, which is common for most of the transformations that we'll be using–only the projective transformations defined in Chapter 6 will require this divide.

⟨*Transform Inline Functions*⟩≡

```
inline Point Transform::operator()(const Point &pt) const {
    Float x = pt.x, y = pt.y, z = pt.z;

    Float xp = m->m[0][0]*x + m->m[0][1]*y + m->m[0][2]*z +
        m->m[0][3];
    Float yp = m->m[1][0]*x + m->m[1][1]*y + m->m[1][2]*z +
        m->m[1][3];
    Float zp = m->m[2][0]*x + m->m[2][1]*y + m->m[2][2]*z +
        m->m[2][3];
    Float wp = m->m[3][0]*x + m->m[3][1]*y + m->m[3][2]*z +
        m->m[3][3];

    if (wp == 1.) return Point(xp, yp, zp);
    else          return Point(xp, yp, zp)/wp;
}
```

We also provide transformation methods that let the caller pass in a pointer to an object for the result. This saves the expense of returning structures by value on the stack. Note that we copy the original (x,y,z) coordinates to local variables in case the result pointer points to the same location as `pt`. This way, these routines can be used even if a point is being transformed in place. This is known as *argument aliasing*.

33 `Point`
43 `Transform`

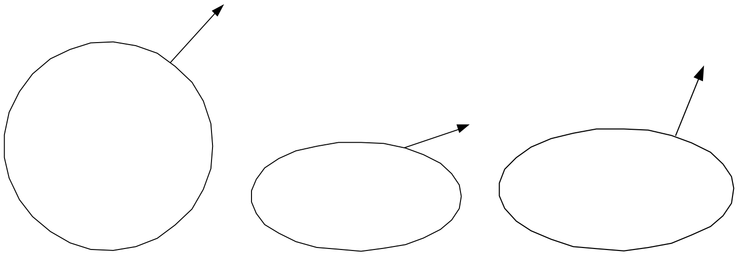
why do we do this copy in the non-in-place version?

⟨*Transform Inline Functions*⟩+≡

```
inline void Transform::operator()(const Point &pt,
        Point *ptrans) const {
    Float x = pt.x, y = pt.y, z = pt.z;
    ptrans->x = m->m[0][0]*x + m->m[0][1]*y + m->m[0][2]*z +
        m->m[0][3];
    ptrans->y = m->m[1][0]*x + m->m[1][1]*y + m->m[1][2]*z +
        m->m[1][3];
    ptrans->z = m->m[2][0]*x + m->m[2][1]*y + m->m[2][2]*z +
        m->m[2][3];
    Float w   = m->m[3][0]*x + m->m[3][1]*y + m->m[3][2]*z +
        m->m[3][3];
    if (w != 1.) *ptrans /= w;
}
```

2.7.2 Vectors

We compute the transformations of vectors in a similar fashion. However, the multiplication of the matrix and the row vector is simplified since the implicit homogeneous w coordinate is zero.



(a) Original object

(b) Scaled object with incorrect normal

(c) Scaled object with correct normal

Figure 2.12: Transforming surface normals. The circle in (a) is scaled by 50% in the *y* direction. Note that simply treating the normal as a direction and scaling it in the same manner, as shown in (b), will lead to incorrect results.

⟨*Transform Inline Functions*⟩+≡

```
inline Vector Transform::operator()(const Vector &v) const {
    Float x = v.x, y = v.y, z = v.z;
    return Vector(m->m[0][0]*x + m->m[0][1]*y + m->m[0][2]*z,
                  m->m[1][0]*x + m->m[1][1]*y + m->m[1][2]*z,
                  m->m[2][0]*x + m->m[2][1]*y + m->m[2][2]*z);
}
```

Point	33
Transform	43
Vector	27

There is also a method allowing the caller to pass a pointer to the result object. The code to do this has a similar design to the Point transformation code, and is not shown here. This code will also be omitted for subsequent transformation methods.

2.7.3 Normals

Normals do not transform in the same way that vectors do, as shown in Figure 2.12. Although tangent vectors transform in the straightforward way, normals require special treatment. Because the normal vector **n** and any tangent vector **t** are orthogonal by construction, we know that

$$\mathbf{n} \cdot \mathbf{t} = \mathbf{n}^T \mathbf{t} = 0.$$

When we transform a point on the surface by some matrix **M**, the new tangent vector $\mathbf{t}'$ at the transformed point is simply **Mt**. The transformed normal $\mathbf{n}'$ should be equal to **Sn** for some 4×4 matrix **S**. To maintain the orthogonality requirement, we must have:

$$\begin{aligned} 0 &= (\mathbf{n}')^T \mathbf{t}' \\ &= (\mathbf{S}\mathbf{n})^T \mathbf{M}\mathbf{t} \\ &= (\mathbf{n})^T \mathbf{S}^T \mathbf{M}\mathbf{t} \end{aligned}$$

This condition holds if $\mathbf{S}^T\mathbf{M} = \mathbf{I}$, the identity matrix. Therefore, $\mathbf{S}^T = \mathbf{M}^{-1}$, so $\mathbf{S} = \mathbf{M}^{-1^T}$, and we see that normals must be transformed by the inverse transpose of the transformation matrix. This is the main reason why Transforms maintain their inverses.

⟨*Transform Public Methods*⟩+≡

```
Transform GetInverse() const {
    return Transform(m_inv, m);
}
```

Note that we do not explicitly compute the transpose of the inverse when transforming normals; we simply iterate through the inverse matrix in a different order (compare to the code for transforming Vectors).

⟨*Transform Inline Functions*⟩+≡

```
inline Normal Transform::operator()(const Normal &n) const {
    Float x = n.x, y = n.y, z = n.z;
    return Normal(m_inv->m[0][0] * x + m_inv->m[1][0] * y +
                      m_inv->m[2][0] * z,
                  m_inv->m[0][1] * x + m_inv->m[1][1] * y +
                      m_inv->m[2][1] * z,
                  m_inv->m[0][2] * x + m_inv->m[1][2] * y +
                      m_inv->m[2][2] * z);
}
```

34	Normal
36	Ray
43	Transform
27	Vector

2.7.4 Rays

Transforming rays is straightforward: we just transform the constituent origin and direction.

⟨*Transform Inline Functions*⟩+≡

```
inline Ray Transform::operator()(const Ray &r) const {
    Ray ret;
    (*this)(r.o, &ret.o);
    (*this)(r.d, &ret.d);
    ret.mint = r.mint;
    ret.maxt = r.maxt;
    ret.time = r.time;
    return ret;
}
```

2.7.5 Bounding Boxes

The easiest way to transform an axis-aligned bounding box is to transform all eight of its corner vertices and then compute a new bounding box that encompasses those points. We will present code for this method below; one of the exercises for this chapter is to find a way to do this more efficiently.

⟨*Transform Method Definitions*⟩+≡

```
BBox Transform::operator()(const BBox &b) const {
    const Transform &M = *this;
    BBox ret(        M(Point(b.pMin.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMin.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMin.x, b.pMax.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMax.y, b.pMin.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMin.y, b.pMax.z)));
    ret = Union(ret, M(Point(b.pMax.x, b.pMax.y, b.pMax.z)));
    return ret;
}
```

2.7.6 Composition of Transformations

Having defined how the matrices representing individual types of transformations are constructed, we can now consider an aggregate transformation resulting from a series of individual transformations. Finally, we can see the real value of representing transformations with matrices.

Consider a series of transformations **ABC**. We'd like to compute a new transformation **T** such that applying **T** gives the same result as applying each of **A**, **B**, and **C** in order; i.e. $\mathbf{A}(\mathbf{B}(\mathbf{C}(\mathrm{p}))) = \mathbf{T}(\mathrm{p})$. Such a transformation **T** can be computed by multiplying the matrices of the transformations **A**, **B**, and **C** together. In `lrt`, we can write:

BBox	38
Matrix4x4	675
Matrix4x4::Mul()	676
Point	33
Reference	664
Transform	43

```
Transform T = A * B * C;
```

Then we can apply `T` to `Points` `p` as usual `Point pp = T(p)` instead of applying each transformation in turn: `Point pp = A(B(C(p)))`.

We use the C++ `*` operator to compute the new transformation that results from post-multiplying the current transformation with a new transformation `t2`. In matrix multiplication, the (i, j)th element of the resulting matrix `ret` is the inner product of the ith row of the first matrix with the jth column of the second.

The inverse of the resulting transformation is equal to the product of `t2.m_inv * m_inv`; this is a result of the matrix identity

$$(\mathbf{AB})^{-1} = \mathbf{B}^{-1}\mathbf{A}^{-1}.$$

⟨*Transform Method Definitions*⟩+≡

```
Transform Transform::operator*(const Transform &t2) const {
    Reference<Matrix4x4> m1 = Matrix4x4::Mul(m, t2.m);
    Reference<Matrix4x4> m2 = Matrix4x4::Mul(t2.m_inv, m_inv);
    return Transform(m1, m2);
}
```

2.7.7 Transformations and Coordinate System Handedness

Certain types of transformations change a left-handed coordinate system into a right-handed one, or vice-versa. Some routines will need to know if the handedness of the source coordinate system is different from that of the destination. In particular, routines that want to ensure that a surface normal always points “outside” of a surface might need to invert the normal after transformation if the handedness changes. See section 2.8 for an example.

Fortunately, it is easy to tell if handedness changes. This happens only when the determinant of the transformation’s upper-left 3×3 submatrix is negative.

⟨*Transform Method Definitions*⟩+≡

```
bool Transform::SwapsHandedness() const {
        Float det =  ((m->m[0][0] *
                       (m->m[1][1] * m->m[2][2] -
                        m->m[1][2] * m->m[2][1])) -
                      (m->m[0][1] *
                       (m->m[1][0] * m->m[2][2] -
                        m->m[1][2] * m->m[2][0])) +
                      (m->m[0][2] *
                       (m->m[1][0] * m->m[2][1] -
                        m->m[1][1] * m->m[2][0])));
    return det < 0.f;
}
```

63 Shape
43 Transform

2.8 Differential Geometry

We will wrap up this chapter by developing a self-contained representation for the geometry of a particular point on a surface (typically the point of a ray intersection). This abstraction needs to hide the particular type of geometric shape the point lies on, allowing the shading and geometric operations in the rest of `lrt` to be implemented generically, without the need to distinguish between different shape types such as spheres and triangles.

The information needed to do this includes:

- The 3D point p.
- The surface normal **n** at the point.
- (u,v) coordinates from the parameterization of the surface.
- The parametric partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$.
- The partial derivatives of the change in surface normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$.
- A pointer to the `Shape` that the differential geometry lies on; the shape class will be introduced in the next chapter. See Figure 2.13 for a depiction of these values.

This representation assumes that shapes have a parametric description–i.e. that for some range of (u,v) values, points on the surface are given by some function f

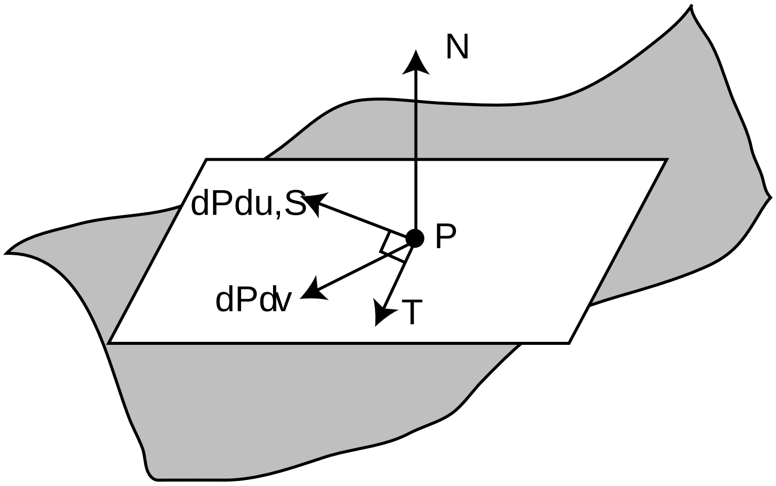


Figure 2.13: The local differential geometry around a point p. The tangent vectors **s** and **t** are orthogonal vectors in the plane that is tangent to the surface at p. The parametric partial derivatives of the surface, $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$, also lie in the tangent plane but are not necessarily orthogonal. The surface normal **n**, is given by the cross product of $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$. The vectors $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$ (not shown here) record the differential change in surface normal as we move in u and v along the surface.

`Normal`	34
`Point`	33
`Shape`	63
`Vector`	27

such that $P = f(u,v)$. Although this isn't true for all shapes, all of the shapes that `lrt` supports do have at least a local parametric description, so we will stick with the parametric representation since this assumption will be helpful to us elsewhere (e.g. for anti-aliasing of textures in Chapter 11.)

```
⟨DifferentialGeometry Declarations⟩≡
  struct DifferentialGeometry {
      DifferentialGeometry() { u = v = 0.; shape = NULL; }
      ⟨DifferentialGeometry Public Methods⟩
      ⟨DifferentialGeometry Public Data⟩
  };
```

`DifferentialGeometry::nn` is a unit-length version of the same normal.

```
⟨DifferentialGeometry Public Data⟩≡
  Point p;
  Normal nn;
  Float u, v;
  const Shape *shape;
```

We also need to store the partial derivatives of the surface parameterization and the surface normal.

```
⟨DifferentialGeometry Public Data⟩+≡
  Vector dpdu, dpdv;
  Vector dndu, dndv;
```

The `DifferentialGeometry` constructor only needs a few parameters–the point of interest, the partial derivatives, and the (u,v) coordinates. It computes the normal as the cross product of the partial derivatives and initializes `s` to be the normalized $\partial \mathrm{p}/\partial u$ vector. It then computes `t` by crossing `s` with `n`, which gives us a vector that is perpendicular to both of them and thus lies in the tangent plane.

⟨*DifferentialGeometry Method Definitions*⟩≡

```
DifferentialGeometry::DifferentialGeometry(const Point &P,
        const Vector &DPDU, const Vector &DPDV, const Vector &DNDU,
        const Vector &DNDV, Float uu, Float vv,
        const Shape *sh)
    : p(P), dpdu(DPDU), dpdv(DPDV), dndu(DNDU), dndv(DNDV) {
    ⟨Initialize DifferentialGeometry from parameters⟩
    ⟨Adjust normal based on orientation and handedness⟩
}
```

⟨*Initialize* `DifferentialGeometry` *from parameters*⟩≡

```
nn = Normal(Cross(dpdu, dpdv)).Hat();
u = uu;
v = vv;
shape = sh;
```

The surface normal has special meaning to `lrt`; we assume that for closed shapes, the normal is oriented such that it points to the “outside” of the shape. For example, this assumption will be used later when we need to decide if a ray is entering or leaving the volume enclosed by a shape. Furthermore, for geometry used as an area light source, light is emitted only from the side of the two-sided surface where the normal points; the other side is black.

31	`Cross()`
58	`DifferentialGeometry`
34	`Normal`
33	`Point`
63	`Shape`
27	`Vector`

Because normals have this special meaning, `lrt` provides a way for the user to reverse the orientation of the normal, flipping it to point in the opposite direction. The `ReverseOrientation` directive in `lrt`’s input file flips the normal to point in the opposite, non-default direction. Therefore, we need to check if the given `Shape` has this flag set, and switch the normal’s direction if so.

One other factor plays into the orientation of the normal and must be accounted for here. If the `Shape`’s transformation matrix has switched the handedness of the object coordinate system from `lrt`’s default left handed coordinate system to a right handed one, we need to switch the orientation of the normal as well. To see this, consider a scale matrix $\mathbf{S}(1,1,-1)$. We would naturally expect this scale to switch the direction of the normal, though because we compute the normal by $\mathbf{n} = \partial \mathrm{p}/\partial u \times \partial \mathrm{p}/\partial v$, it can be shown that

$$\mathbf{s}(1,1,-1)\partial \mathrm{p}/\partial u \times \mathbf{s}(1,1,-1)\partial \mathrm{p}/\partial v = \partial \mathrm{p}/\partial u \times \partial \mathrm{p}/\partial v = \mathbf{n} \neq \mathbf{s}(1,1,-1)\mathbf{n}.$$

Therefore, we also need to manually flip the normal’s direction if the transformation switches the handedness of the coordinate system, since the flip won’t be accounted for by the computation of the normal’s direction using the cross product.

We only swap the normal’s directon if one but not both of these two conditions is met; if both were met, their effect would cancel out. The exclusive or operation lets us easily test this condition.

⟨*Adjust normal based on orientation and handedness*⟩≡

```
if (shape->reverseOrientation ^
    shape->transformSwapsHandedness)
    nn *= -1.f;
```

The functionality described in the text below is gone. This explanation needs to be moved to the BSDF code, which DOES do this stuff...

It is useful to be able to transform direction vectors from world space to the coordinate frame defined by the three basis directions **s**, **t**, and **n**. This maps the object's surface normal to the direction $(0,0,1)$, and can help to simplify shading computations by letting us think of them in a standard coordinate system. It is easy to show that given three orthogonal vectors **s**, **t**, and **n** in world-space, the matrix **M** that transforms vectors in world space to the local differential geometry space is:

$$\mathbf{M} = \begin{pmatrix} \mathbf{s}_x & \mathbf{s}_y & \mathbf{s}_z \\ \mathbf{t}_x & \mathbf{t}_y & \mathbf{t}_z \\ \mathbf{n}_x & \mathbf{n}_y & \mathbf{n}_z \end{pmatrix} = \begin{pmatrix} \mathbf{s} \\ \mathbf{t} \\ \mathbf{n} \end{pmatrix}$$

To confirm this yourself, consider the value of $\mathbf{Mn} = (\mathbf{s} \cdot \mathbf{n}, \mathbf{t} \cdot \mathbf{n}, \mathbf{n} \cdot \mathbf{n})$. Since **s**, **t**, and **n** are all orthonormal, the x and y components of **Mn** are zero. Since **n** is normalized, $\mathbf{n} \cdot \mathbf{n} = 1$. Thus, $\mathbf{Mn} = (0,0,1)$. In this case, we don't need to compute the inverse transpose of **M** to transform normals (recall the discussion of transforming normals in Section 2.7.3 on page 54.) Because **M** is an orthonormal matrix (its rows and columns are mutually orthogonal and are normalized), its inverse is equal to its transpose, so it is its own inverse transpose already.

Shape::reverseOrientation 64
transformSwapsHandedness 64

The function that takes vectors back from local space to world space just implements the transpose to invert **M** and does the appropriate dot products:

Further Reading

DeRose, Goldman, and their collaborators have argued for an elegant "coordinate free" approach to describing vector geometry for graphics, where the fact that positions and directions happen to be represented by (x,y,z) coordinates with respect to a particular coordinate system is de-emphasized and where points and vectors themselves record which coordinate system they are expressed in terms of (Goldman 1985; DeRose 1989; Mann, Litke, and DeRose 1997). This makes it possible for a software layer to ensure that common errors like adding a vector in one coordinate system to a point in another coordinate system are transparently handled by transforming them to a common coordinate system first.

Schneider and Eberly's *Geometric Tools for Computer Graphics* is strongly influenced by the coordinate-free approach and covers the topics of this chapter in much greater depth. It is also full of useful geometry for graphics (Schneider and Eberly 2003).

A classic introduction to the topics of this chapter is *Mathematical Elements for Computer Graphics* by Rogers and Adams (Rogers and Adams 1990). Note that they use a row-vector representation of points and vectors, though, which means that our matrices would be transposed when expressed in their framework, and that they multiply points and vectors by matrices to transform them, p**M**, rather than multiplying matrices by points as we do, **M**p.

There are many excellent books on linear algebra and vector geometry. We have found Lang's (Lang 1986) and Buck's (Buck 1978) to be good references on these respective topics.

Homogeneous coordinates lead to projective geometry, an elegant framework for XXX. Stolfi's book XXX (Stolfi 1991).

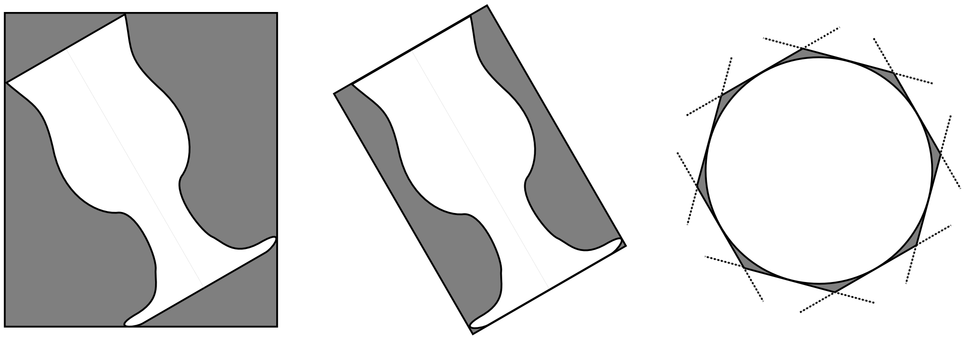
Akenine–Möller and Haines for graphics-based introduction to linear algebra (Akenine-Möller and Haines 2002), lots of ray bounds stuff and ray–obb stuff.

The subtleties of how normal vectors are transformed were first widely understood after articles by Wallis and Turkowski (Wallis 1990; Turkowski 1990c).

Exercises

2.1 (Jim Arvo) Find a more efficient way to transform axis-aligned bounding boxes by taking advantage of the symmetries of the problem: because the eight corner points are linear combinations of three axis-aligned basis vectors and a single corner point, their transformed bounding box can be found much more efficiently than by the method we presented (Arvo 1990).

2.2 Instead of boxes, we could compute tighter bounds by using the intersections of many non-orthogonal slabs. Extend our bounding box class to allow the user to specify a bound comprised of arbitrary slabs.



Axis-aligned bounding box Non-axis-aligned bounding box Arbitrary bounding slabs

2.3 Derive the matrices for rotation in a right-handed coordinate system.

3. Shapes

130 Primitive
663 ReferenceCounted

Shapes in lrt are the basic representations of geometry in a scene. Each specific shape in lrt (sphere, triangle, cone, etc.) is a subclass of the Shape base class. Thus, we can provide a generic interface to shapes that hides details about the specific type of shape. This abstraction makes extending the geometric capabilities of the system quite straightforward; the rest of lrt doesn't need to know what specific shape it is dealing with. The Shape class is purely geometric; it contains no information about the appearance of an object. The Primitive class, introduced in Chapter 1, holds additional information about a shape such as its material properties. The basic interface for Shapes is in the source file core/shape.h, and various common shape function definitions are in core/shape.cpp.

3.1 Basic Shape Interface

The Shape class in lrt is *reference counted*. This means that lrt keeps track of the number of outstanding pointers to a particular shape, and only deletes the shape when that reference count goes to zero. Although not foolproof or completely automatic, this is a form of *garbage collection* which relieves us from having to worry about freeing memory at the wrong time. The ReferenceCounted class handles all of this for us; its implementation is presented in section A.3.2.

```
⟨Shape Declarations⟩≡
  class COREDLL Shape : public ReferenceCounted {
  public:
      ⟨Shape Interface⟩
      ⟨Shape Public Data⟩
  };
```

All shapes are defined in object coordinate space; for example, all spheres are defined in a coordinate system where the center of the sphere is at the origin. In order to place a sphere at another position in the scene, a transformation that describes the mapping from object space to world space must be provided. The Shape class stores both this transformation and its inverse. Shapes also take a boolean parameter, reverseOrientation, that records whether their surface normal directions should be reversed from their default orientations. This is useful because the orientation of the surface normal is used to determine which side of a shape is "outside". Its value is set via the ReverseOrientation statement in lrt input files. Shapes also store the result of the Transform::SwapsHandedness() call for their object to world transformation; this value is needed by the DifferentialGeometry constructor each time a ray intersection is found, so lrt computes it once and stores the result.

⟨*Shape Method Definitions*⟩≡

```
Shape::Shape(const Transform &o2w, bool ro)
    : ObjectToWorld(o2w), WorldToObject(o2w.GetInverse()),
    reverseOrientation(ro),
    transformSwapsHandedness(o2w.SwapsHandedness()) {
}
```

BBox	38
Shape	63
Transform	43
Transform::GetInverse()	55
sform::SwapsHandedness()	57

⟨*Shape Public Data*⟩≡

```
const Transform ObjectToWorld, WorldToObject;
const bool reverseOrientation, transformSwapsHandedness;
```

3.1.1 Bounding

Each Shape subclass must be capable of bounding itself with a bounding box. There are two different bounding methods. The first, ObjectBound(), returns a bounding box in the shape's object space, and the second, WorldBound(), returns a bounding box in world space. The implementation of the first method is left up to each individual shape, but there is a default implementation of the second method which simply transforms the object bound to world space. Shapes that can easily compute a tighter world-space bound should override this method, however. An example of such a shape is a triangle; see Figure 3.1.

⟨*Shape Interface*⟩+≡

```
virtual BBox ObjectBound() const = 0;
```

⟨*Shape Interface*⟩+≡

```
virtual BBox WorldBound() const {
    return ObjectToWorld(ObjectBound());
}
```

3.1.2 Refinement

Not every shape needs to be capable of determining whether a ray intersects it. For example, a complex surface might first be tessellated into triangles, which can then be intersected directly. Another possibility is a shape that is a place-holder for a large amount of geometry that is stored on disk. We could store just the filename

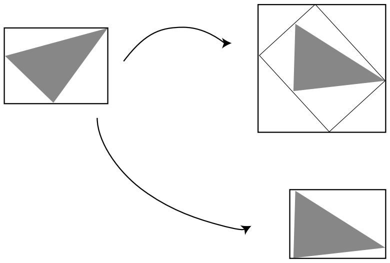


Figure 3.1: If we compute a world-space bounding box of a triangle by transforming its object-space bounding box to world space and then finding the bounding box that encloses the resulting bounding box, a sloppy bound may result (top). However, if we first transform the triangle's vertices from object space to world space and then bound those vertices (bottom), we can do much better.

of the geometry file and the bounding box of the geometry inside of it in memory, and read the geometry in from disk only if a ray pierces the bounding box.

The default implementation of the `Shape::CanIntersect()` function indicates that a shape *can* provide an intersection, so only shapes that are non-intersectable need to override this method.

⟨*Shape Interface*⟩+≡

```
virtual bool CanIntersect() const { return true; }
```

664	`Reference`
63	`Shape`
658	`vector`

If the shape can not be intersected directly, it must provide a `Shape::Refine()` method that splits the shape into a group of new shapes, some of which may be intersectable and some of which may need further refinement. The default implementation of the `Shape::Refine()` method issues an error message; thus, shapes that are intersectable (which is the common case) do not have to provide an empty instance of this method. `lrt` will never call `Shape::Refine()` if `Shape::CanIntersect()` returns true.

⟨*Shape Interface*⟩+≡

```
virtual void Refine(vector<Reference<Shape> > &refined) const {
    Severe("Unimplemented Shape::Refine() method called");
}
```

I think there should be more here, but not sure what to say.

3.1.3 Intersection

The `Shape` class provides two different intersection routines. The first, `Shape::Intersect()`, returns information about a single ray–shape intersection corresponding to the first intersection in the $[\mathrm{mint}, \mathrm{maxt}]$ parametric range along the ray. The other, `Shape::IntersectP()` is a predicate function that determines whether or not an intersection occurs, without returning any details about the intersection itself. Some shapes may be able to provide a very efficient implementation for `IntersectP()` that can determine whether an intersection exists without computing it at all.

There are a few important things to keep in mind when reading (and writing) intersection routines:

- The Ray structure contains Ray::mint and Ray::maxt variables which define a ray *segment*. Intersection routines should ignore any intersections that do not occur along this segment.

- If an intersection is found, its parametric distance along the ray should be stored in the pointer t_hitp that is passed into the intersection routine. If multiple intersections are present, the closest one should be returned.

- Information about an intersection position is stored in the DifferentialGeometry structure, which completely captures the local geometric properties of a surface. This type will be used heavily throughout lrt, and it serves to cleanly isolate the geometric portion of the ray tracer from the shading and illumination portions. The differential geometry class was defined in Section 2.8 on page 57.[1]

- The rays passed into intersection routines will be in world space, so shapes are responsible for transforming them to object space if needed for intersection tests. The differential geometry returned should be in world space.

DifferentialGeometry	58
Ray	36
Shape	63

Rather than making the intersection routines pure virtual functions, the Shape class provides default implementations of the intersect routines that print an error message if they are called. All Shapes that return true from Shape::CanIntersect() must provide implementations of these functions; those that return false can depend on lrt to not call these routines on non-intersectable shapes. If these were *pure* virtual functions, then each non-intersectable shape would have to implement a similar default function.

Why not the obvious default IntersectP that just calls Intersect and throws away the resulting DifferentialGeometry?

⟨*Shape Interface*⟩+≡

```
virtual bool Intersect(const Ray &ray, Float *t_hitp,
        DifferentialGeometry *dg) const {
    Severe("Unimplemented Shape::Intersect() method called");
    return false;
}

virtual bool IntersectP(const Ray &ray) const {
    Severe("Unimplemented Shape::IntersectP() method called");
    return false;
}
```

[1] Almost all ray tracers use this general idiom for returning geometric information about intersections with shapes. As an optimization, many will only partially initialize the intersection information when an intersection is found, storing just enough information so that the rest of the values can be computed later when actually needed. This approach saves work in the case where a closer intersection is later found with another shape. In our experience, the extra work to compute all the information isn't substantial, and for renderers that have complex scene data management algorithms (e.g. discarding geometry from main memory when too much memory is being used and writing it to disk), the deferred approach may fail because the shape is no longer in memory.

3.1.4 Shading Geometry

Some shapes (notably triangle meshes) supports the idea of having two types of differential geometry at a point on the surface: the true geometry, which accurately reflects the local properties of the surface, and the *shading geometry*, which may have normals and tangents that are different than the true differential geometry. For triangle meshes, the user can provide normal vectors and tangents at the vertices of the mesh which are interpolated to give normals and tangents at points across the faces of triangles.

The `GetShadingGeometry()` method of the `Shape` returns the shading geometry for `DifferentialGeometry` returned by the `Intersect()` routine. By default, the shading geometry matches the true geometry, so the default implementation just copies the true geometry. One subtlety is that an object to world transformation is passed to this routine; it is important that if it needs to transform data from its object space to world space as part of computing the shading geometry, it must use this transformation rather than the `Shape::ObjectToWorld` transformation. This is an artifact from how object instancing is implemented in `lrt` (See Section 4.1.2.)

⟨*Shape Interface*⟩+≡

```
virtual void GetShadingGeometry(const Transform &obj2world,
        const DifferentialGeometry &dg,
        DifferentialGeometry *dgShading) const {
    *dgShading = dg;
}
```

58	`DifferentialGeometry`
63	`Shape`
43	`Transform`

3.1.5 Surface Area

In order to properly use `Shape`s as area lights, we need to be able to compute the surface area of a shape in object space. As with the intersection methods, this method will only be called for intersectable shapes.

⟨*Shape Interface*⟩+≡

```
virtual Float Area() const {
    Severe("Unimplemented Shape::Area() method called");
    return 0.;
}
```

3.1.6 Sidedness

Many rendering systems, particularly those based on scan-line or z-buffer algorithms, support the concept of shapes being “one-sided”; the shape is visible if seen from the front, but disappears when viewed from behind. In particular, If a geometric object is closed and always viewed from the outside, then the back-facing shapes can be discarded without changing the resulting image. This optimization can substantially improve the speed of these types of algorithms. The potential for improved performance is substantially reduced when using this technique with ray tracing, however, since we would need to perform the ray–object intersection before determining the surface normal to do the backfacing test. Furthermore, it can lead to a physically inconsistent scene description if one-sided objects are not in fact closed. (For example, a surface might block light when a shadow ray is traced from a light source to a point on another surface, but not if the shadow ray is traced in the other direction.) Therefore, `lrt` doesn't support this feature.

3.2 Spheres

```
⟨Sphere Declarations⟩≡
  class COREDLL Sphere: public Shape {
  public:
      ⟨Sphere Public Methods⟩
  private:
      ⟨Sphere Private Data⟩
  };
```

Spheres are a special case of a general type of surface called *quadrics*. Quadrics are surfaces described by quadratic polynomials in x, y, and z. They are the simplest type of curved surface that is useful to a ray tracer, and are an interesting introduction to more general ray intersection routines. `lrt` supports six types of quadrics: spheres, cones, disks (a special case of a cone), cylinders, hyperboloids, and paraboloids.

Shape 63

Most mathematical surfaces can be described in one of two main ways: in *implicit form* and in *parametric form*. An implicit function describes a 3D surface as:

$$f(x,y,z) = 0$$

The set of all points (x, y, z) that fulfill this condition define the surface. For a unit sphere at the origin, the familiar implicit equation is $x^2 + y^2 + z^2 - 1 = 0$. Only the set of (x,y,z) one unit from the origin satisfies this constraint, giving us the unit sphere's surface.

Many surfaces can also be described parametrically using a function to map the 2D plane to 3D points on the surface. For example, a sphere can be described as a function of 2D spherical coordinates (θ, ϕ) where θ ranges from 0 to π and ϕ ranges from 0 to 2π:

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ y &= r \sin\theta \sin\phi \\ z &= r \cos\theta \end{aligned}$$

We can transform this function $f(\theta, \phi)$ into a function $f(u, v)$ over $[0,1]^2$ with the substitution

$$\begin{aligned} \phi &= u \cdot \phi_{\text{max}} \\ \theta &= \theta_{\text{min}} + v \cdot (\theta_{\text{max}} - \theta_{\text{min}}) \end{aligned}$$

This form is particularly useful for texture mapping, where we can directly use the (u, v) values to map a texture defined over $[0,1]^2$ to the sphere.

As we describe the implementation of the sphere shape, we will make use of both the implicit and parametric descriptions of the shape, depending on which is a more natural way to approach the particular problem we're facing.

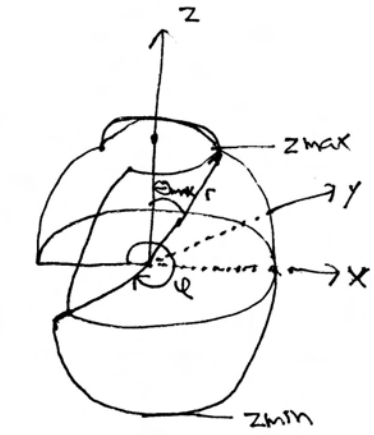


Figure 3.2: Basic setting for the sphere shape. It has a radius of *r* and **XXX**. A partial sphere may be described by specifying a maximum ϕ value.

3.2.1 Construction

Our `Sphere` class specifies a shape that is centered at the origin in object space; to place it elsewhere in the scene, the user must apply an appropriate transformation when specifying the sphere in the input file.

The radius of the sphere can have an arbitrary value, and the sphere's extent can be truncated in two different ways. First, minimum and maximum *z* values may be set; the parts of the sphere below and above these, respectively, are cut off. Second, if we consider the parameterization of the sphere in spherical coordinates we can set a maximum ϕ value. The sphere sweeps out ϕ values from 0 to the given ϕ_{max} such that the section of the sphere with spherical ϕ values above this ϕ is also removed.

677	`Clamp()`
677	`Radians()`
63	`Shape`
68	`Sphere`
43	`Transform`

⟨*Sphere Method Definitions*⟩≡

```
Sphere::Sphere(const Transform &o2w, bool ro, Float rad,
               Float z0, Float z1, Float pm)
    : Shape(o2w, ro) {
    radius = rad;
    zmin = Clamp(min(z0, z1), -radius, radius);
    zmax = Clamp(max(z0, z1), -radius, radius);
    thetaMin = acosf(zmin/radius);
    thetaMax = acosf(zmax/radius);
    phiMax = Radians(Clamp(pm, 0.0f, 360.0f));
}
```

⟨*Sphere Private Data*⟩≡

```
Float radius;
Float phiMax;
Float zmin, zmax;
Float thetaMin, thetaMax;
```

3.2.2 Bounding

Computing a bounding box for a sphere is straightforward. We will use the values of z_{min} and z_{max} provided by the user to tighten up the bound when less than an entire sphere is being rendered. However, we won't do the extra work to look at θ_{max} and see if we can compute a tighter bounding box when that is less than 2π. This is left as an exercise.

⟨*Sphere Method Definitions*⟩+≡

```
BBox Sphere::ObjectBound() const {
    return BBox(Point(-radius, -radius, zmin),
        Point( radius,  radius, zmax));
}
```

3.2.3 Intersection

The task of deriving an intersection test is simplified by the fact that the sphere is centered at the origin. However, if the sphere has been transformed to another position in world space, then we need to transform rays to object space before intersecting them with the sphere, using the world to object transformation. Once we have a ray in object space, we can go ahead and perform the intersection computation in object space.[2]

BBox	38
DifferentialGeometry	58
Point	33
Ray	36
Shape::WorldToObject	64
Sphere	68
Sphere::radius	69
Sphere::zmax	69
Sphere::zmin	69

The entire intersection method is shown below.

⟨*Sphere Method Definitions*⟩+≡

```
bool Sphere::Intersect(const Ray &r, Float *t_hitp,
        DifferentialGeometry *dg) const {
    Float phi;
    Point phit;
    ⟨Transform Ray to object space⟩
    ⟨Compute quadratic sphere coefficients⟩
    ⟨Solve quadratic equation for t values⟩
    ⟨Compute sphere hit position and ϕ⟩
    ⟨Test sphere intersection against clipping parameters⟩
    ⟨Find parametric representation of sphere hit⟩
    ⟨Initialize DifferentialGeometry from parametric information⟩
    ⟨Update t_hitp for quadric intersection⟩
    return true;
}
```

We start by transforming the given world-space ray to the sphere's object space. The remainder of the intersection test will take place in that coordinate system.

⟨*Transform* Ray *to object space*⟩≡

```
Ray ray;
WorldToObject(r, &ray);
```

[2] This is something of a classic theme in computer graphics: by transforming the problem to a particular restricted case, we can more easily and efficiently do an intersection test (i.e. lots of math cancels out since the sphere is always at $(0,0,0)$. No overall generality is lost, since we can just apply an appropriate translation to the ray to account for spheres at other positions.

If we have a sphere centered at the origin with radius r, its implicit representation is

$$x^2 + y^2 + z^2 - r^2 = 0.$$

By substituting the parametric representation of the ray (Equation 2.4.3) into the implicit sphere equation, we have:

$$(\mathrm{o(r)}_x + t\mathbf{d}(\mathrm{r})_x)^2 + \left(\mathrm{o(r)}_y + t\mathbf{d}(\mathrm{r})_y\right)^2 + \left(\mathrm{o(r)}_z + t\mathbf{d}(\mathrm{r})_z\right)^2 = r^2.$$

Note that all elements of this equation besides t are known values. The t values where the equation holds give the parametric positions along the ray where the implicit sphere equation holds and thus the points along the ray where it intersects the sphere.

We can expand this equation and gather the coefficients for a general quadratic in t:

$$At^2 + Bt + C = 0.$$

where[3]

$$\begin{aligned} A &= \mathbf{d}(\mathrm{r})_x^2 + \mathbf{d}(\mathrm{r})_y^2 + \mathbf{d}(\mathrm{r})_z^2 \\ B &= 2(\mathbf{d}(\mathrm{r})_x \mathrm{o(r)}_x + \mathbf{d}(\mathrm{r})_y \mathrm{o(r)}_y + \mathbf{d}(\mathrm{r})_z \mathrm{o(r)}_z) \\ C &= \mathrm{o(r)}_x^2 + \mathrm{o(r)}_y^2 + \mathrm{o(r)}_z^2 - r^2 \end{aligned}$$

69 Sphere::radius

This directly translates to this fragment of source code.

⟨*Compute quadratic sphere coefficients*⟩≡

```
  Float A = ray.d.x*ray.d.x + ray.d.y*ray.d.y + ray.d.z*ray.d.z;
  Float B = 2 * (ray.d.x*ray.o.x + ray.d.y*ray.o.y +
                 ray.d.z*ray.o.z);
  Float C = ray.o.x*ray.o.x + ray.o.y*ray.o.y +
            ray.o.z*ray.o.z - radius*radius;
```

We know there are two possible solutions to this quadratic equation, giving zero, one, or two non-imaginary t values where the ray intersects the sphere:

$$\begin{aligned} t_0 &= \frac{-B - \sqrt{B^2 - 4AC}}{2A} \\ t_1 &= \frac{-B + \sqrt{B^2 - 4AC}}{2A} \end{aligned}$$

We provide a `Quadratic()` utility function that solves a quadratic equation, returning `false` if there are no real solutions and returning `true` and setting `t0` and `t1` appropriately if there are solutions.

[3] Some raytracers require that the direction vector of a ray be normalized, meaning $A = 1$ This can lead to subtle errors, however, if the caller forgets to normalize the ray direction. Of course, these errors can be avoided by normalizing the direction in the ray constructor, but this wastes effort when the provided direction is *already* normalized. To avoid this needless complexity, `lrt` never insists on vector normalization unless it is mathematically necessary.

⟨*Solve quadratic equation for* `t` *values*⟩≡
```
Float t0, t1;
if (!Quadratic(A, B, C, &t0, &t1))
    return false;
⟨Compute intersection distance along ray⟩
```

⟨*Global Inline Functions*⟩≡
```
inline bool Quadratic(Float A, Float B, Float C, Float *t0,
        Float *t1) {
    ⟨Find quadratic discriminant⟩
    ⟨Compute quadratic t values⟩
}
```

If the discriminant ($B^2 - 4AC$) is negative, then there are no real roots and the ray must miss the sphere.

⟨*Find quadratic discriminant*⟩≡
```
Float discrim = B * B - 4.f * A * C;
if (discrim < 0.) return false;
Float rootDiscrim = sqrtf(discrim);
```

The usual version of the quadratic equation can give poor numeric precision when $B \approx \pm\sqrt{B^2 - 4AC}$ due to cancellation error. It can be rewritten algebraically to a more stable form:

$$\begin{aligned} t_0 &= \frac{q}{A} \\ t_1 &= \frac{C}{q} \end{aligned}$$

where

$$q = \begin{cases} -.5(B - \sqrt{B^2 - 4AC}) & : \quad B < 0 \\ -.5(B + \sqrt{B^2 - 4AC}) & : \quad \text{otherwise} \end{cases}$$

⟨*Compute quadratic* `t` *values*⟩≡
```
Float q;
if (B < 0) q = -.5f * (B - rootDiscrim);
else       q = -.5f * (B + rootDiscrim);
*t0 = q / A;
*t1 = C / q;
if (*t0 > *t1) swap(*t0, *t1);
return true;
```

Given the two intersection t values, we need to check them against the ray segment from `mint` to `maxt`. Since t_0 is guaranteed to be less than t_1 (and `mint` less than `maxt`), if t_0 is greater than `maxt` or t_1 is less than `mint`, then it is certain that both hits are out of the range of interest. Otherwise, t_0 is the tentative hit distance. If may be less than `mint`, however, in which case we ignore it and try t_1. If that is also out of range, we have no valid intersection. If there is an intersection, `thit` holds the distance to the hit.

⟨*Compute intersection distance along ray*⟩≡

```
if (t0 > ray.maxt || t1 < ray.mint)
    return false;
Float thit = t0;
if (t0 < ray.mint) {
    thit = t1;
    if (thit > ray.maxt) return false;
}
```

3.2.4 Partial Spheres

Now that we have the distance along the ray to the intersection with a full sphere, we need to handle partial spheres, specified with clipped z or ϕ ranges. Intersections that are in clipped areas need to be ignored.

We start by computing the object space position of the intersection, `phit` and the ϕ value for the hit point. Taking the parametric equations for the sphere,

$$\frac{y}{x} = \frac{r \sin\theta \sin\phi}{r \sin\theta \cos\phi} = \tan\phi$$

so $\phi = \arctan \frac{y}{x}$.

58	DifferentialGeometry
678	M_PI
69	Sphere::phiMax
69	Sphere::zmax
69	Sphere::zmin

⟨*Compute sphere hit position and* ϕ⟩≡

```
phit = ray(thit);
phi = atan2f(phit.y, phit.x);
if (phi < 0.) phi += 2.f*M_PI;
```

We remap the result of the C standard library's `atan2f` function to a value between 0 and 2π, to match the sphere's original definition.

We can now test the hit point against the specified minima and maxima for z and ϕ. If the t_0 intersection wasn't actually valid, we try again with t_1.

⟨*Test sphere intersection against clipping parameters*⟩≡

```
if (phit.z < zmin || phit.z > zmax || phi > phiMax) {
    if (thit == t1) return false;
    if (t1 > ray.maxt) return false;
    thit = t1;
    ⟨Compute sphere hit position and φ⟩
    if (phit.z < zmin || phit.z > zmax || phi > phiMax)
        return false;
}
```

At this point, we are sure that the ray hits the sphere, and we can fill in the `DifferentialGeometry` structure. We compute parametric u and v values by scaling the previously-computed ϕ value for the hit to lie between 0 and 1 and by computing a θ value between 0 and 1 for the hit point, based on the range of θ values for the given sphere. Then, we compute the parametric partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$

⟨Find parametric representation of sphere hit⟩≡

```
Float u = phi / phiMax;
Float theta = acosf(phit.z / radius);
Float v = (theta - thetaMin) / (thetaMax - thetaMin);
⟨Compute sphere ∂p/∂u and ∂p/∂v ⟩
⟨Compute sphere ∂n/∂u and ∂n/∂v ⟩
```

Computing the partial derivatives of a point on the sphere is a short exercise in algebra. Using the parametric definition of the sphere, we have:

$$\begin{aligned} x &= r \sin\theta \cos\phi \\ &= r \sin(\theta_{\mathrm{min}} + v(\theta_{\mathrm{max}} - \theta_{\mathrm{min}})) \cos(\phi_{\mathrm{max}} u) \end{aligned}$$

Consider the first component of $\partial \mathrm{p}/\partial u$, $\partial x/\partial u$: **These equations could use a bit more explanation at each step, like what variable depends on what, which ones can be pulled out of the partial, etc**

$$\begin{aligned} \frac{\partial x}{\partial u} &= \frac{\partial}{\partial u}(r \sin\theta \cos\phi) \\ &= r \sin\theta \frac{\partial}{\partial u}(\cos\phi) \\ &= r \sin\theta (-\phi_{\mathrm{max}} \sin\phi) \end{aligned}$$

Sphere::phiMax	69
Sphere::radius	69
Sphere::thetaMax	69
Sphere::thetaMin	69
Vector	27

Using a substitution based on the parametric definition of the sphere's y coordinate, this simplifies to

$$\partial x/\partial u = -\phi_{\mathrm{max}} y.$$

Similarly

$$\partial y/\partial u = \phi_{\mathrm{max}} x,$$

and

$$\partial z/\partial u = 0.$$

A similar process gives us $\partial \mathrm{p}/\partial v$.

$$\begin{aligned} \frac{\partial \mathrm{p}}{\partial u} &= (-\phi_{\mathrm{max}} y, \phi_{\mathrm{max}} x, 0) \\ \frac{\partial \mathrm{p}}{\partial v} &= (\theta_{\mathrm{max}} - \theta_{\mathrm{min}})(z \cos\phi, z \sin\phi, -r \sin\theta) \end{aligned}$$

⟨Compute sphere ∂p/∂u and ∂p/∂v ⟩≡

```
Float invzradius = 1.f / sqrtf(phit.x*phit.x + phit.y*phit.y);
Float cosphi = phit.x * invzradius, sinphi = phit.y * invzradius;
Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0);
Vector dpdv = (thetaMax-thetaMin) *
    Vector(phit.z * cosphi, phit.z * sinphi,
        -radius * sinf(thetaMin + v * (thetaMax - thetaMin)));
```

3.2.5 ***ADV***: Partial Derivatives of Normal Vectors

It is useful to determine how the normal changes as we move along the surface in the u and v directions. For example, some of the anti-aliasing techniques in Chapter 10 will use this information. The differential changes in normal $\partial\mathbf{n}/\partial u$ and $\partial\mathbf{n}/\partial v$ are given by the *Weingarten equations* from differential geometry. They are:

$$\begin{aligned}\frac{\partial\mathbf{n}}{\partial u} &= \frac{fF-eG}{EG-F^2}\frac{\partial \mathrm{p}}{\partial u}+\frac{eF-fE}{EG-F^2}\frac{\partial \mathrm{p}}{\partial v}\\ \frac{\partial\mathbf{n}}{\partial v} &= \frac{gF-fG}{EG-F^2}\frac{\partial \mathrm{p}}{\partial u}+\frac{fF-gE}{EG-F^2}\frac{\partial \mathrm{p}}{\partial v}\end{aligned}$$

where E, F, and G are coefficients of the *first fundamental form* and are given by

$$\begin{aligned}E &= \left|\frac{\partial \mathrm{p}}{\partial u}\right|^2\\ F &= \left(\frac{\partial \mathrm{p}}{\partial u}\cdot\frac{\partial \mathrm{p}}{\partial v}\right)\\ G &= \left|\frac{\partial \mathrm{p}}{\partial v}\right|^2.\end{aligned}$$

These are easily computed with the $\partial\mathrm{p}/\partial u$ and $\partial\mathrm{p}/\partial v$ values found above. e, f, and g are coefficients of the *second fundamental form*,

$$\begin{aligned}e &= \left(N\cdot\frac{\partial^2 \mathrm{p}}{\partial u^2}\right)\\ f &= \left(N\cdot\frac{\partial^2 \mathrm{p}}{\partial u\partial v}\right)\\ g &= \left(N\cdot\frac{\partial^2 \mathrm{p}}{\partial v^2}\right).\end{aligned}$$

The two fundamental forms have basic connections with the local curvature of a surface; see a differential geometry textbook such as Gray's (Gray 1993) for details. To find e, f, and g, we need to compute the second order partial derivatives $\partial^2\mathrm{p}/\partial u^2$ etc.

For spheres, a little more algebra gives the required second derivatives:

$$\begin{aligned}\frac{\partial^2 \mathrm{p}}{\partial u^2} &= -\phi_{\mathrm{max}}^2(x,y,0)\\ \frac{\partial^2 \mathrm{p}}{\partial u\partial v} &= (z_{\mathrm{max}}-z_{\mathrm{min}})z\phi_{\mathrm{max}}(\sin\phi,-\cos\phi,0)\\ \frac{\partial^2 \mathrm{p}}{\partial v^2} &= -(\theta_{\mathrm{max}}-\theta_{\mathrm{min}})^2(x,y,z)\end{aligned}$$

⟨*Compute sphere* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$⟩≡

```
Vector d2Pduu = -phiMax * phiMax * Vector(phit.x, phit.y, 0);
Vector d2Pduv = (zmax - zmin) * phit.z * phiMax *
    Vector(sinphi, -cosphi, 0.);
Vector d2Pdvv = -(thetaMax - thetaMin) * (thetaMax - thetaMin) *
    Vector(phit.x, phit.y, phit.z);
⟨Compute coefficients for fundamental forms⟩
⟨Compute ∂n/∂u and ∂n/∂v from fundamental form coefficients⟩
```

⟨*Compute coefficients for fundamental forms*⟩≡

```
Float E = Dot(dpdu, dpdu);
Float F = Dot(dpdu, dpdv);
Float G = Dot(dpdv, dpdv);
Vector N = Cross(dpdu, dpdv);
Float e = Dot(N, d2Pduu);
Float f = Dot(N, d2Pduv);
Float g = Dot(N, d2Pdvv);
```

⟨*Compute* $\partial\mathbf{n}/\partial u$ *and* $\partial\mathbf{n}/\partial v$ *from fundamental form coefficients*⟩≡

```
Float invEGF2 = 1.f / (E*G - F*F);
Vector dndu = (f*F - e*G) * invEGF2 * dpdu +
    (e*F - f*E) * invEGF2 * dpdv;
Vector dndv = (g*F - f*G) * invEGF2 * dpdu +
    (f*F - g*E) * invEGF2 * dpdv;
```

DifferentialGeometry	58
Dot()	30
Shape::ObjectToWorld	64
Sphere::phiMax	69
Sphere::thetaMax	69
Sphere::thetaMin	69
Sphere::zmax	69
Sphere::zmin	69
Vector	27

3.2.6 DifferentialGeometry Initialization

Now that we have computed the surface parameterization and all the relevant partial derivatives, we can construct the DifferentialGeometry structure for this intersection.

⟨*Initialize* DifferentialGeometry *from parametric information*⟩≡

```
*dg = DifferentialGeometry(ObjectToWorld(phit), ObjectToWorld(dpdu),
    ObjectToWorld(dpdv), ObjectToWorld(dndu), ObjectToWorld(dndv),
    u, v, this);
```

Since there is an intersection, we update the ray's t_hitp value to hold the hit distance along the ray, which was stored in thit. This will allow subsequent intersection tests to terminate early if the potential hit would be farther away than the existing intersection.

⟨*Update* t_hitp *for quadric intersection*⟩≡

```
*t_hitp = thit;
```

The sphere's IntersectP() routine is almost identical to Sphere::Intersect(), but it does not fill in the DifferentialGeometry structure. Because Intersect and IntersectP are always so closely related, we will not show IntersectP for the remaining shapes.

```
⟨Sphere Method Definitions⟩+≡
  bool Sphere::IntersectP(const Ray &r) const {
      Float phi;
      Point phit;
      ⟨Transform Ray to object space⟩
      ⟨Compute quadratic sphere coefficients⟩
      ⟨Solve quadratic equation for t values⟩
      ⟨Compute sphere hit position and ϕ⟩
      ⟨Test sphere intersection against clipping parameters⟩
      return true;
  }
```

3.2.7 Surface Area

To compute the surface area of quadrics, we use a standard formula from integral calculus. If we revolve a curve $y = f(x)$ from $y = a$ to $y = b$ around the x axis, the surface area of the resulting swept surface is

$$2\pi \int_a^b f(x)\sqrt{1 + (f'(x))^2}\, dx,$$

where $f'(x)$ denotes the derivative $\frac{df}{dx}$[4]. Since most of our surfaces of revolution are only partially swept around the axis, we will instead use the formula:

$$\phi_{\text{max}} \int_a^b f(x)\sqrt{1 + (f'(x))^2}\, dx.$$

33	Point
36	Ray
68	Sphere
69	Sphere::phiMax
69	Sphere::radius
69	Sphere::zmax
69	Sphere::zmin

Our sphere is a surface of revolution of a circular arc. So the function that defines the profile curve of the sphere is

$$f(x) = \sqrt{r^2 - x^2},$$

and its derivative is

$$f'(x) = -\frac{x}{\sqrt{r^2 - x^2}}.$$

Recall that the sphere is clipped at z_{min} and z_{max}. The surface area is therefore

$$\begin{aligned}
\text{A} &= \phi_{\text{max}} \int_{z_0}^{z_1} \sqrt{r^2 - x^2}\sqrt{1 + \frac{x^2}{r^2 - x^2}}\, dx \\
&= \phi_{\text{max}} \int_{z_0}^{z_1} \sqrt{r^2 - x^2 + x^2}\, dx \\
&= \phi_{\text{max}} \int_{z_0}^{z_1} r\, dx \\
&= \phi_{\text{max}} r(z_1 - z_0)
\end{aligned}$$

For the full sphere $\phi_{\text{max}} = 2\pi$, $z_{\text{min}} = -r$ and $z_{\text{max}} = r$, so we have the standard formula $A = 4\pi r^2$, showing that our formula is correct.

```
⟨Sphere Method Definitions⟩+≡
  Float Sphere::Area() const {
      return phiMax * radius * (zmax-zmin);
  }
```

[4] See Anton for a derivation (Anton, Bivens, and Davis 2001).

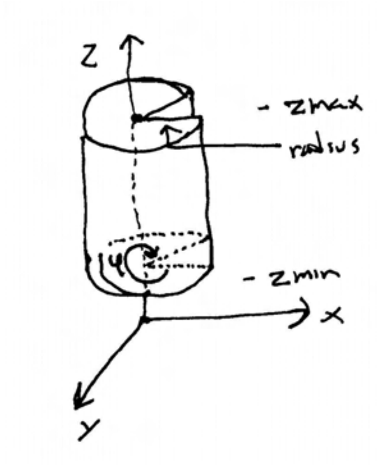


Figure 3.3: Basic setting for the cylinder shape. It has a radius of *r* and is covers a range of heights along the z-axis. A partial cylinder may be swept by specifying a maximum ϕ value.

3.3 Cylinders

Shape 63

```
⟨cylinder.cpp*⟩≡
  #include "shape.h"
  ⟨Cylinder Declarations⟩
  ⟨Cylinder Method Definitions⟩
```

```
⟨Cylinder Declarations⟩≡
  class COREDLL Cylinder: public Shape {
  public:
      ⟨Cylinder Public Methods⟩
  protected:
      ⟨Cylinder Private Data⟩
  };
```

3.3.1 Construction

Another useful quadric is the cylinder; `lrt` provides cylinder `Shapes` that are centered around the *z* axis. The user supplies a minimum and maximum *z* value for the cylinder, as well as a radius and maximum ϕ sweep value (See figure 3.3). In parametric form, a cylinder is described by the equations:

$$\begin{aligned} \phi &= u\phi_{\max} \\ x &= r\cos\phi \\ y &= r\sin\phi \\ z &= z_{\min} + v(z_{\max} - z_{\min}) \end{aligned}$$

⟨*Cylinder Method Definitions*⟩≡

```
Cylinder::Cylinder(const Transform &o2w, bool ro, Float rad,
           Float z0, Float z1, Float pm)
    : Shape(o2w, ro) {
    radius = rad;
    zmin = min(z0, z1);
    zmax = max(z0, z1);
    phiMax = Radians(Clamp(pm, 0.0f, 360.0f));
}
```

⟨*Cylinder Private Data*⟩≡

```
Float radius;
Float zmin, zmax;
Float phiMax;
```

3.3.2 Bounding

As we did with the sphere, we compute a conservative bounding box for the cylinder using the z range but without taking into account the maximum ϕ.

⟨*Cylinder Method Definitions*⟩+≡

```
BBox Cylinder::ObjectBound() const {
    Point p1 = Point(-radius, -radius, zmin);
    Point p2 = Point( radius,  radius, zmax);
    return BBox(p1, p2);
}
```

38	BBox
677	Clamp()
78	Cylinder
33	Point
677	Radians()
63	Shape
43	Transform

3.3.3 Intersection

We can intersect a ray with a cylinder by substituting the ray equation into the cylinder's implicit equation, similarly to the sphere case. The implicit equation for an infinitely long cylinder centered on the z axis with radius r is

$$x^2 + y^2 - r^2 = 0.$$

Substituting the ray equation, 2.4.3, we have:

$$(\mathrm{o}(\mathrm{r})_x + t\mathbf{d}(\mathrm{r})_x)^2 + \left(\mathrm{o}(\mathrm{r})_y + t\mathbf{d}(\mathrm{r})_y\right)^2 = r^2$$

When we expand this and find the coefficients of the quadratic equation $At^2 + Bt + C$, we get:

$$\begin{aligned} A &= \mathbf{d}(\mathrm{r})_x^2 + \mathbf{d}(\mathrm{r})_y^2 \\ B &= 2(\mathbf{d}(\mathrm{r})_x \mathrm{o}(\mathrm{r})_x + \mathbf{d}(\mathrm{r})_y \mathrm{o}(\mathrm{r})_y) \\ C &= \mathrm{o}(\mathrm{r})_x^2 + \mathrm{o}(\mathrm{r})_y^2 - r^2 \end{aligned}$$

⟨*Compute quadratic cylinder coefficients*⟩≡

```
Float A = ray.d.x*ray.d.x + ray.d.y*ray.d.y;
Float B = 2 * (ray.d.x*ray.o.x + ray.d.y*ray.o.y);
Float C = ray.o.x*ray.o.x + ray.o.y*ray.o.y - radius*radius;
```

The solution process for the quadratic equation is similar for all quadric shapes, so some fragments from the Sphere intersection method will be re-used below. The fragments that are re-used from Sphere::Intersect() are marked with an arrow.

⟨*Cylinder Method Definitions*⟩+≡

```
  bool Cylinder::Intersect(const Ray &r, Float *t_hitp,
          DifferentialGeometry *dg) const {
      Float phi;
      Point phit;
->
      ⟨Transform Ray to object space⟩
      ⟨Compute quadratic cylinder coefficients⟩
->
      ⟨Solve quadratic equation for t values⟩
      ⟨Compute cylinder hit point and φ⟩
      ⟨Test cylinder intersection against clipping parameters⟩
      ⟨Find parametric representation of cylinder hit⟩
->
      ⟨Initialize DifferentialGeometry from parametric information⟩
->
      ⟨Update t_hitp for quadric intersection⟩
      return true;
  }
```

Cylinder	78
Cylinder::phiMax	79
Cylinder::zmax	79
Cylinder::zmin	79
DifferentialGeometry	58
M_PI	678
Point	33
Ray	36
Sphere	68

3.3.4 Partial Cylinders

As with the sphere, we invert the parametric description of the cylinder to compute a ϕ value by inverting the x and y parametric equations to solve for ϕ. In fact, the result is the same as for the sphere.

⟨*Compute cylinder hit point and* ϕ⟩≡

```
  phit = ray(thit);
  phi = atan2f(phit.y, phit.x);
  if (phi < 0.) phi += 2.f*M_PI;
```

We now make sure that the hit is in the specified z range, and that the angle is acceptable. If not, we reject the hit and try with t_1, if we haven't already.

⟨*Test cylinder intersection against clipping parameters*⟩≡

```
  if (phit.z < zmin || phit.z > zmax || phi > phiMax) {
      if (thit == t1) return false;
      thit = t1;
      if (t1 > ray.maxt) return false;
      ⟨Compute cylinder hit point and φ⟩
      if (phit.z < zmin || phit.z > zmax || phi > phiMax)
          return false;
  }
```

Again the u value is computed by scaling ϕ to lie between 0 and 1. Straightforward inversion of the parametric equation for the cylinder's z value gives us the v parametric coordinate.

$\langle$*Find parametric representation of cylinder hit*$\rangle\equiv$

```
  Float u = phi / phiMax;
  Float v = (phit.z - zmin) / (zmax - zmin);
```

  $\langle$*Compute cylinder* $\partial \mathrm{p}/\partial u$ *and* $\partial \mathrm{p}/\partial v$ $\rangle$
  $\langle$*Compute cylinder* $\partial \mathbf{n}/\partial u$ *and* $\partial \mathbf{n}/\partial v$ $\rangle$

The partial derivatives for a cylinder are quite easy to derive: they are

$$\begin{aligned}\frac{\partial \mathrm{p}}{\partial u} &= (-\phi_{\mathrm{max}} y, \phi_{\mathrm{max}} x, 0)\\ \frac{\partial \mathrm{p}}{\partial v} &= (0, 0, z_{\mathrm{max}} - z_{\mathrm{min}})\end{aligned}$$

$\langle$*Compute cylinder* $\partial \mathrm{p}/\partial u$ *and* $\partial \mathrm{p}/\partial v$ $\rangle\equiv$

```
  Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0);
  Vector dpdv(0, 0, zmax - zmin);
```

We again use the Weingarten equations to compute the parametric change in cylinder normal. The relevant partial derivatives are

$$\begin{aligned}\frac{\partial^2 \mathrm{p}}{\partial u^2} &= -\phi_{\mathrm{max}}^2 (x, y, 0)\\ \frac{\partial^2 \mathrm{p}}{\partial u \partial v} &= (0,0,0)\\ \frac{\partial^2 \mathrm{p}}{\partial v^2} &= (0,0,0)\end{aligned}$$

78	Cylinder
79	Cylinder::phiMax
79	Cylinder::radius
79	Cylinder::zmax
79	Cylinder::zmin
27	Vector

$\langle$*Compute cylinder* $\partial \mathbf{n}/\partial u$ *and* $\partial \mathbf{n}/\partial v$ $\rangle\equiv$

```
  Vector d2Pduu = -phiMax * phiMax * Vector(phit.x, phit.y, 0);
  Vector d2Pduv(0, 0, 0), d2Pdvv(0, 0, 0);
```

  $\langle$*Compute coefficients for fundamental forms*$\rangle$
  $\langle$*Compute* $\partial \mathbf{n}/\partial u$ *and* $\partial \mathbf{n}/\partial v$ *from fundamental form coefficients*$\rangle$

3.3.5 Surface Area

A cylinder is just a rolled up rectangle. If you unroll the rectangle, its height is $z_{\mathrm{max}} - z_{\mathrm{min}}$, and its width is $r\phi_{\mathrm{max}}$:

$\langle$*Cylinder Method Definitions*$\rangle+\equiv$

```
  Float Cylinder::Area() const {
      return (zmax-zmin)*phiMax*radius;
  }
```

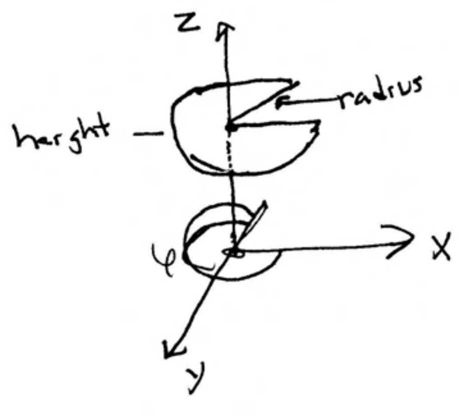


Figure 3.4: Basic setting for the disk shape. The disk has radius *r* and is located at height *h* along the z-axis. A partial disk may be swept by specifying a maximum ϕ value. **figure should use *h* not "height".**

3.4 Disks

Shape 63

```
⟨disk.cpp*⟩≡
  #include "shape.h"
  ⟨Disk Declarations⟩
  ⟨Disk Method Definitions⟩
```

```
⟨Disk Declarations⟩≡
  class COREDLL Disk : public Shape {
  public:
      ⟨Disk Public Methods⟩
  private:
      ⟨Disk Private Data⟩
  };
```

The disk is an interesting quadric since it has a particularly straightforward intersection routine that avoids solving the quadratic equation. In `lrt`, a `Disk` is a circular disk of radius *r* at height *h* along the *z* axis. In order to make partial disks, the caller may specify a maximum ϕ value beyond which the disk is cut off (Figure 3.4). In parametric form, it is described by:

$$
\begin{aligned}
\phi &= u\,\phi_{\mathrm{max}} \\
x &= r\,(1-v)\,\cos\phi \\
x &= r\,(1-v)\,\sin\phi \\
z &= h
\end{aligned}
$$

3.4.1 Construction

⟨Disk Method Definitions⟩≡

```
  Disk::Disk(const Transform &o2w, bool ro, Float ht, Float r, Float tmax)
      : Shape(o2w, ro) {
      height = ht;
      radius = r;
      phiMax = Radians(Clamp(tmax, 0.0f, 360.0f));
  }
```

⟨Disk Private Data⟩≡

```
  Float height, radius, phiMax;
```

3.4.2 Bounding

The bounding method is quite straightforward; we create a bounding box centered at the height of the disk along *z*, with extent of `radius` in both the *x* and *y* directions.

⟨Disk Method Definitions⟩+≡

```
  BBox Disk::ObjectBound() const {
      return BBox(Point(-radius, -radius, height),
                  Point(radius, radius, height));
  }
```

38	BBox
677	Clamp()
58	DifferentialGeometry
82	Disk
33	Point
677	Radians()
36	Ray
63	Shape
43	Transform

3.4.3 Intersection

Intersecting a ray with a disk is also quite easy. We intersect the ray with the $z = h$ plane that the disk lies in and then see if the intersection point lies inside the disk. Again, the re-used chunks are marked with an arrow.

⟨Disk Method Definitions⟩+≡

```
  bool Disk::Intersect(const Ray &r, Float *t_hitp,
          DifferentialGeometry *dg) const {
->
      ⟨Transform Ray to object space⟩
      ⟨Compute plane intersection for disk⟩
      ⟨See if hit point is inside disk radius and φmax ⟩
      ⟨Find parametric representation of disk hit⟩
->
      ⟨Initialize DifferentialGeometry from parametric information⟩
->
      ⟨Update t_hitp for quadric intersection⟩
      return true;
  }
```

The first step is to compute the parametric *t* value where the ray intersects the plane that the disk lies in. Using the same approach as we did for intersecting rays with boxes, we want to find *t* such that the *z* component of the ray's position is equal to the height of the disk. Thus,

$$h = \mathrm{o}(\mathrm{r})_z + t * \mathbf{d}(\mathrm{r})_z$$

and

$$t = \frac{h - \mathrm{o}(\mathrm{r})_z}{\mathbf{d}(\mathrm{r})_z}$$

We first check whether the ray is parallel to the disk's plane, in which case we report no intersection. We then see if t is inside the legal range of values [`mint`,`maxt`]. If not, we can return false.

whasssup with this magic constant? Make a "CloseToZero" function?

⟨*Compute plane intersection for disk*⟩≡

```
  if (fabsf(ray.d.z) < 1e-7) return false;
  Float thit = (height - ray.o.z) / ray.d.z;
  if (thit < ray.mint || thit > ray.maxt)
      return false;
```

We now compute the point `phit` where the ray intersects the plane. Once the plane intersection is known, we return `false` if the distance from the hit to the center of the disk is more than `radius`. We optimize this process by actually computing the squared distance to the center, taking advantage of the fact that the x and y coordinates of the center point $(0,0,\mathrm{height})$ are zero, and the z coordinate of `phit` is equal to `height`.

Disk::height	83
Disk::phiMax	83
Disk::radius	83
M_PI	678
Point	33
Vector	27

⟨*See if hit point is inside disk radius and* ϕ_{max} ⟩≡

```
  Point phit = ray(thit);
  Float dist2 = phit.x * phit.x + phit.y * phit.y;
  if (dist2 > radius * radius)
      return false;
```

  ⟨*Test disk* ϕ *value against* ϕ_{max} ⟩

If the distance check passes, we perform the final test, making sure that the ϕ value of the hit point is between zero and $\phi_{\max}$ specified by the caller. Inverting the disk's parameterization gives us the same expression for ϕ as the other quadric shapes.

⟨*Test disk* ϕ *value against* ϕ_{max} ⟩≡

```
  Float phi = atan2f(phit.y, phit.x);
  if (phi < 0) phi += 2. * M_PI;
  if (phi > phiMax)
      return false;
```

If we've gotten this far, we know that there is an intersection with the disk. The parameter `u` is scaled to reflect the partial disk specified by $\phi_{\max}$ and `v` is computed by inverting the parametric equation. The equations for the partial derivatives at the hit point can be derived with a similar process to that used for the previous quadrics. Because the normal of a disk is the same everywhere, the partial derivatives $\partial\mathbf{n}/\partial u$ and $\partial\mathbf{n}/\partial v$ are both trivially $(0,0,0)$.

⟨*Find parametric representation of disk hit*⟩≡

```
  Float u = phi / phiMax;
  Float v = 1.f - (sqrtf(dist2) / radius);
  Vector dpdu(-phiMax * phit.y, phiMax * phit.x, 0.);
  Vector dpdv(-phit.x / (1-v), -phit.y / (1-v), 0.);
  Vector dndu(0,0,0), dndv(0,0,0);
```

3.4.4 Surface Area

Disks have trivial surface area, since they're just portions of a circle:

$$A = \frac{\phi_{\max}}{2} r^2$$

⟨*Disk Method Definitions*⟩+≡

```
Float Disk::Area() const {
    return phiMax * 0.5f * radius * radius;
}
```

3.5 Other Quadrics

`lrt` supports three more quadrics: cones, paraboloids, and hyperboloids. They are implemented in the source files `shapes/cone.cpp`, `shapes/paraboloid.cpp` and `shapes/hyperboloid.cpp`. We won't include their full implementations here, since the techniques used to derive their quadratic intersection coefficients, parametric coordinates and partial derivatives should now be familiar. However, we will briefly describe the implicit and parametric forms of these shapes.

82	`Disk`
83	`Disk::phiMax`
83	`Disk::radius`

3.5.1 Cones

The implicit equation of a cone centered on the z axis with radius r and height h is

$$\left(\frac{hx}{r}\right)^2 + \left(\frac{hy}{r}\right)^2 - (z-h)^2 = 0.$$

Cones are also described parametrically:

$$\begin{aligned}
\phi &= u\,\phi_{\max} \\
x &= r\,(1-v)\,\cos\phi \\
y &= r\,(1-v)\,\sin\phi \\
z &= v\,h
\end{aligned}$$

The partial derivatives at a point on a cone are

$$\begin{aligned}
\frac{\partial \mathrm{p}}{\partial u} &= (-\phi_{\max} y, \phi_{\max} x, 0) \\
\frac{\partial \mathrm{p}}{\partial v} &= \left(-\frac{x}{1-v}, \frac{y}{1-v}, h\right)
\end{aligned}$$

and the partial second derivatives are

$$\begin{aligned}
\frac{\partial^2 \mathrm{p}}{\partial u^2} &= -\phi_{\max}^2 (x, y, 0) \\
\frac{\partial^2 \mathrm{p}}{\partial u \partial v} &= \frac{\phi_{\max}}{1-v}(y, -x, 0) \\
\frac{\partial^2 \mathrm{p}}{\partial v^2} &= (0, 0, 0).
\end{aligned}$$

3.5.2 Paraboloids

The implicit equation of a paraboloid centered on the *z* axis with radius *r* and height *h* is:

$$\frac{hx^2}{r^2}+\frac{hy^2}{r^2}-z=0$$

and its parametric form is

$$\begin{aligned}
\phi &= u\,\phi_{\mathrm{max}} \\
z &= v\,(z_{\mathrm{max}}-z_{\mathrm{min}}) \\
r &= r_{\mathrm{max}}\sqrt{z/z_{\mathrm{max}}} \\
x &= r\cos\phi \\
y &= r\sin\phi
\end{aligned}$$

The partial derivatives are:

$$\begin{aligned}
\frac{\partial \mathrm{p}}{\partial u} &= (-\phi_{\mathrm{max}}\,y, \phi_{\mathrm{max}}\,x, 0) \\
\frac{\partial \mathrm{p}}{\partial v} &= (z_{\mathrm{max}}-z_{\mathrm{min}})(x/z, y/z, 1)
\end{aligned}$$

and

$$\begin{aligned}
\frac{\partial^2 \mathrm{p}}{\partial u^2} &= -\phi_{\mathrm{max}}^2(x, y, 0) \\
\frac{\partial^2 \mathrm{p}}{\partial u \partial v} &= \phi_{\mathrm{max}}(z_{\mathrm{max}}-z_{\mathrm{min}})(-y/z, x/z, 0) \\
\frac{\partial^2 \mathrm{p}}{\partial v^2} &= -2(z_{\mathrm{max}}-z_{\mathrm{min}})^2(x/z^2, y/z^2, 0)
\end{aligned}$$

3.5.3 Hyperboloids

Finally, the implicit form of the hyperboloid is

$$x^2+y^2-z^2=-1$$

and the parametric form is

$$\begin{aligned}
\phi &= u\,\phi_{\mathrm{max}} \\
x_r &= (1-v)\,x_1+v\,x_2 \\
y_r &= (1-v)\,y_1+v\,y_2 \\
x &= x_r\cos\phi-y_r\sin\phi \\
y &= x_r\sin\phi+y_r\cos\phi \\
z &= (1-v)\,z_1+v\,z_2
\end{aligned}$$

The partial derivatives are:

$$\begin{aligned}
\frac{\partial \mathrm{p}}{\partial u} &= (-\phi_{\mathrm{max}}\,y, \phi_{\mathrm{max}}\,x, 0) \\
\frac{\partial \mathrm{p}}{\partial v} &= ((x_2-x_1)\cos\phi-(y_2-y_1)\sin\phi, (x_2-x_1)\sin\phi+(y_2-y_1)\cos\phi, z_2-z_1)
\end{aligned}$$

and

$$
\begin{aligned}
\frac{\partial^2 \mathrm{p}}{\partial u^2} &= -\phi_{\max}^2(x, y, 0)\\
\frac{\partial^2 \mathrm{p}}{\partial u \partial v} &= \phi_{\max}(-\partial y/\partial v, \partial x/\partial v, 0)\\
\frac{\partial^2 \mathrm{p}}{\partial v^2} &= (0, 0, 0)
\end{aligned}
$$

3.6 Triangles and Meshes

```
⟨trianglemesh.cpp*⟩≡
  #include "shape.h"
  #include "paramset.h"
  ⟨TriangleMesh Declarations⟩
  ⟨TriangleMesh Method Definitions⟩
```

```
⟨TriangleMesh Declarations⟩≡
  class COREDLL TriangleMesh : public Shape {
  public:
      ⟨TriangleMesh Public Methods⟩
  protected:
      ⟨TriangleMesh Data⟩
  };
```

63 Shape

The triangle is one of the most commonly used shapes in computer graphics. lrt supports triangle meshes, where a number of triangles are stored together so that their per-vertex data can be shared among multiple triangles. Single triangles are simply treated as degenerate meshes.

The arguments to the TriangleMesh constructor are as follows:

- nt Number of triangles.
- nv Number of vertices.
- vi Pointer to an array of vertex indices. For the ith triangle, its three vertex positions are P[vi[3*i]], P[vi[3*i+1]], and P[vi[3*i+2]].
- P Array of nv vertex positions.
- N An optional array of normal vectors, one per vertex in the mesh. If present, these are interpolated across triangle faces to compute the triangles shading differential geometry.
- S An optional array of tangent vectors, one per vertex in the mesh. These are also used to compute shading geometry.
- uv An optional array of a parametric (u, v) value for each vertex.

We just copy the relevant information and store it in the `TriangleMesh` object. In particular, must make our own copies of `vi` and `P`, since the caller retains ownership of the data being passed in.

Triangles have a dual role among the primitives in `lrt`: not only are they a user-specified primitive, but other primitives may tessellate themselves into triangle meshes; for example, subdivision surfaces end up creating a mesh of triangles to approximate the smooth limit surface. Ray intersections are performed against these triangles, rather than directly against the subdivision surface.

Because of this second role, it's important that a routine that is creating a triangle mesh be able to specify the parameterization of the triangles. If a triangle was created by evaluating the position of a parametric surface at three particular (u,v) coordinate values, for example, those (u,v) values should be interpolated to compute the (u,v) value at ray intersection points inside the triangle; hence the `uv` parameter.

⟨*TriangleMesh Method Definitions*⟩≡

```
TriangleMesh::TriangleMesh(const Transform &o2w, bool ro,
        int nt, int nv, const int *vi, const Point *P,
        const Normal *N, const Vector *S, const Float *uv)
    : Shape(o2w, ro) {
    ntris = nt;
    nverts = nv;
    vertexIndex = new int[3 * ntris];
    memcpy(vertexIndex, vi, 3 * ntris * sizeof(int));
    ⟨Copy uv, N, and S vertex data, if present⟩
    ⟨Transform mesh vertices to world space⟩
}
```

`Normal`	34
`Point`	33
`Shape`	63
`Transform`	43
`TriangleMesh`	87
`Vector`	27

The ⟨*Copy* `uv`*,* `N`*, and* `S` *vertex data, if present*⟩ fragment just allocates the appropriate amount of space and copies the data directly, if it is present. Its implementation isn't included here.

⟨*TriangleMesh Data*⟩≡

```
int ntris, nverts;
int *vertexIndex;
Point *p;
Normal *n;
Vector *s;
Float *uvs;
```

Unlike the other shapes that leave the primitive description in object space and then transform incoming rays from world space to object space, triangle meshes transform the shape into world space and save the work of transforming the incoming rays into the object space or the intersection's differential geometry out to world space. This is a good idea because this operation can be performed once at startup, avoiding transforming rays many times during rendering. Taking this with quadrics is be more complicated, though is possible—see the exercises for hints on how to do it. (Normal and **s** tangent vectors for shading geometry are left in object space, since the `GetShadingGeometry()` must transform them to world space with the transformation matrix supplied to that method, which may not necessarily be the one stored by the `Shape`.)

⟨*Transform mesh vertices to world space*⟩≡

```
for (int i  = 0; i < nverts; ++i)
    p[i] = ObjectToWorld(P[i]);
```

The object-space bound of a triangle mesh is easily found by computing a bounding box that encompasses all of the vertices of the mesh. Because the vertex positions `p` were transformed to world space in the constructure, the implementation here has to transform them back to object space before computing their bound.

⟨*TriangleMesh Method Definitions*⟩+≡

```
BBox TriangleMesh::ObjectBound() const {
    BBox bobj;
    for (int i = 0; i < nverts; i++)
        bobj = Union(bobj, WorldToObject(p[i]));
    return bobj;
}
```

The `TriangleMesh` shape is one of the shapes that can usually compute a better world space bound than can be found by transforming its object-space bounding box to world space. Its world space bounce can be directly computed from the world-space vertices.

⟨*TriangleMesh Method Definitions*⟩+≡

```
BBox TriangleMesh::WorldBound() const {
    BBox worldBounds;
    for (int i = 0; i < nverts; i++)
        worldBounds = Union(worldBounds, p[i]);
    return worldBounds;
}
```

The `TriangleMesh` shape does not directly compute intersections. Instead, it splits itself into many separate `Triangle`s, each representing a single triangle. All of the individual reference the shared set of vertices in `p`, avoiding per-triangle replication of the shared data. It overrides the `Shape::CanIntersect()` method to indicate that `TriangleMesh`es cannot be intersected directly.

38	`BBox`
664	`Reference`
63	`Shape`
64	`Shape::ObjectToWorld`
64	`Shape::reverseOrientation`
64	`Shape::WorldToObject`
90	`Triangle`
87	`TriangleMesh`
88	`TriangleMesh::ntris`
88	`TriangleMesh::nverts`
88	`TriangleMesh::p`
40	`Union()`
658	`vector`

⟨*TriangleMesh Public Methods*⟩+≡

```
bool CanIntersect() const { return false; }
```

When `lrt` encounters a shape that cannot be intersected directly, it calls its `Refine()` method. `Shape::Refine()` is expected to produce a list of simpler shapes in the “`refined`” `vector`. The implementation here is simple; we just make a new `Triangle` for each of the triangles in the mesh.

⟨*TriangleMesh Method Definitions*⟩+≡

```
void TriangleMesh::Refine(vector<Reference<Shape> > &refined) const
{
    for (int i = 0; i < ntris; ++i)
        refined.push_back(new Triangle(ObjectToWorld, reverseOrientation,
                (TriangleMesh *)this, i));
}
```

3.6.1 Triangle

⟨*TriangleMesh Declarations*⟩+≡

```
class COREDLL Triangle : public Shape {
public:
    ⟨Triangle Public Methods⟩
private:
    ⟨Triangle Data⟩
};
```

The `Triangle` doesn't store much data; just a pointer to the parent `TriangleMesh` that it came from and a pointer to its three vertex indices in the mesh.

⟨*Triangle Public Methods*⟩≡

```
Triangle(const Transform &o2w, bool ro, TriangleMesh *m, int n)
        : Shape(o2w, ro) {
    mesh = m;
    v = &mesh->vertexIndex[3*n];
}
```

Note that the implementation stores a pointer to the first vertex *index*, instead of storing three pointers to the vertices themselves. This reduces the amount of storage required for each `Triangle` significantly.

`BBox`	38
`Point`	33
`Reference`	664
`Shape`	63
`Shape::WorldToObject`	64
`Transform`	43
`TriangleMesh`	87
`TriangleMesh::p`	88

⟨*Triangle Data*⟩≡

```
Reference<TriangleMesh> mesh;
int *v;
```

As with `TriangleMesh`es, it is possible to compute better world space bounding boxes for individual triangles by bounding the world space vertices directly.

⟨*TriangleMesh Method Definitions*⟩+≡

```
BBox Triangle::ObjectBound() const {
    ⟨Get triangle vertices in p1, p2, and p3⟩
    return Union(BBox(WorldToObject(p1), WorldToObject(p2)),
        WorldToObject(p3));
}
```

⟨*TriangleMesh Method Definitions*⟩+≡

```
BBox Triangle::WorldBound() const {
    ⟨Get triangle vertices in p1, p2, and p3⟩
    return Union(BBox(p1, p2), p3);
}
```

⟨*Get triangle vertices in* `p1`*,* `p2`*, and* `p3`⟩≡

```
const Point &p1 = mesh->p[v[0]];
const Point &p2 = mesh->p[v[1]];
const Point &p3 = mesh->p[v[2]];
```

3.6.2 Triangle Intersection

An algorithm for ray–triangle intersection can be computed using *barycentric coordinates*. Barycentric coordinates provide a way to parameterize a triangle in

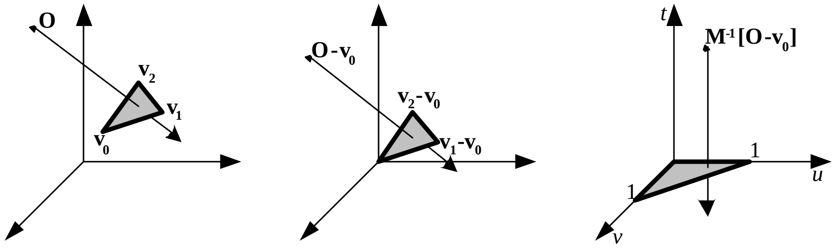


Figure 3.5: Transforming the ray into a more convenient coordinate system for intersection. First, a translation is applied to make a corner of the triangle coincide with the origin. Then, the triangle is rotated and scaled to a unit right-triangle. **The axis labels don't match the text.**

terms of two variables, b_1 and b_2:

$$\mathrm{p}(b_1, b_2) = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2$$

The conditions on b_1 and b_2 are that $b_1 \geq 0$, $b_2 \geq 0$, and $b_1 + b_2 \leq 1$. This is the parametric form of a triangle. The barycentric coordinates are also a natural way to interpolate across the surface of the triangle; given values defined at the vertices a_0, a_1, and a_2 and given the barycentric coordinates for a point on the triangle, we can compute an interpolated value of a at that point as $(1 - b_1 - b_2)a_0 + b_1 a_1 + b_2 a_2$. (See Section **??** on page ?? for a texture that interpolates shading values over a triangle mesh in this manner.)

To derive an algorithm for intersecting a ray with a triangle, we insert the parametric ray equation into the triangle equation.

$$\mathrm{o}(\mathrm{r}) + t\mathbf{d}(\mathrm{r}) = (1 - b_1 - b_2)\mathrm{p}_0 + b_1\mathrm{p}_1 + b_2\mathrm{p}_2 \tag{3.6.1}$$

Following the technique described by Möller and Trumbore(Möller and Trumbore 1997), we use the shorthand notation $\mathbf{E}_1 = \mathrm{p}_1 - \mathrm{p}_0$, $\mathbf{E}_2 = \mathrm{p}_2 - \mathrm{p}_0$, and $\mathbf{T} = \mathrm{o}(\mathrm{r}) - \mathrm{p}_0$. We can now rearrange the terms of Equation 3.6.1 to obtain the matrix equation:

$$\begin{bmatrix} -\mathbf{d}(\mathrm{r}) & \mathbf{E}_1 & \mathbf{E}_2 \end{bmatrix} \begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \mathbf{T} \tag{3.6.2}$$

Solving this linear system will give us both the barycentric coordinates of the intersection point (which can easily be used to compute the 3D intersection point) as well as the distance along the ray.

Geometrically, we can interpret this system as a translation of the triangle to the origin, and a transformation of the triangle to a unit triangle in y and z, keeping the ray direction aligned with x, as shown in Figure 3.5.

We can easily solve equation 3.6.2 using Cramer's rule. Note that we are introducing a bit of notation for brevity here; we write $\begin{vmatrix} \mathbf{a} & \mathbf{b} & \mathbf{c} \end{vmatrix}$ to mean the determinant of the matrix having **a**, **b**, and **c** as its columns. Cramer's rule gives:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\begin{vmatrix} -\mathbf{d}(\mathrm{r}) & \mathbf{E}_1 & \mathbf{E}_2 \end{vmatrix}} \begin{bmatrix} \begin{vmatrix} \mathbf{T} & \mathbf{E}_1 & \mathbf{E}_2 \end{vmatrix} \\ \begin{vmatrix} -\mathbf{d}(\mathrm{r}) & \mathbf{T} & \mathbf{E}_2 \end{vmatrix} \\ \begin{vmatrix} -\mathbf{d}(\mathrm{r}) & \mathbf{E}_1 & \mathbf{T} \end{vmatrix} \end{bmatrix} \tag{3.6.3}$$

This can be rewritten as $\begin{vmatrix} \mathbf{A} & \mathbf{B} & \mathbf{C} \end{vmatrix} = -(\mathbf{A}\times\mathbf{C})\cdot\mathbf{B} = -(\mathbf{C}\times\mathbf{B})\cdot\mathbf{A}$. We can thus rewrite Equation 3.6.3 as:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{(\mathbf{d}(\mathrm{r})\times\mathbf{E}_2)\cdot\mathbf{E}_1} \begin{bmatrix} (\mathbf{T}\times\mathbf{E}_1)\cdot\mathbf{E}_2 \\ (\mathbf{d}(\mathrm{r})\times\mathbf{E}_2)\cdot\mathbf{T} \\ (\mathbf{T}\times\mathbf{E}_1)\cdot\mathbf{d}(\mathrm{r}) \end{bmatrix} \tag{3.6.4}$$

If we use the substitution $\mathbf{s}_1 = \mathbf{d}(\mathrm{r})\times\mathbf{E}_2$ and $\mathbf{s}_2 = \mathbf{t}\times\mathbf{E}_1$ we can make the common subexpressions more explicit:

$$\begin{bmatrix} t \\ b_1 \\ b_2 \end{bmatrix} = \frac{1}{\mathbf{s}_1\cdot\mathbf{E}_1} \begin{bmatrix} \mathbf{s}_2\cdot\mathbf{E}_2 \\ \mathbf{s}_1\cdot\mathbf{T} \\ \mathbf{s}_2\cdot\mathbf{d}(\mathrm{r}) \end{bmatrix} \tag{3.6.5}$$

`DifferentialGeometry`	58
`Ray`	36
`Triangle`	90

In order to compute $\mathbf{E}_1$, $\mathbf{E}_2$, and $\mathbf{T}$ we need 9 subtractions. To compute $\mathbf{s}_1$ and $\mathbf{s}_2$, we need two cross products, which is a total of 12 multiplications and 6 subtractions. Finally, to compute t, b_1, and b_2, we need 4 dot products (12 multiplications and 8 additions), 1 reciprocal, and 3 multiplications. Thus, the total cost of ray–triangle intersection is 1 divide, 27 multiplies, and 17 adds (counting adds and subtracts together). Note that some of these operations can be avoided if it is determined mid-calculation that the ray does not intersect the triangle.

```
⟨TriangleMesh Method Definitions⟩+≡
  bool Triangle::Intersect(const Ray &ray, Float *t_hitp,
          DifferentialGeometry *dg) const {
      ⟨Compute s1⟩
      ⟨Compute first barycentric coordinate⟩
      ⟨Compute second barycentric coordinate⟩
      ⟨Compute t to intersection point⟩
      ⟨Fill in DifferentialGeometry from triangle hit⟩
      *t_hitp = t;
      return true;
  }
```

First, we compute the divisor from Equation 3.6.5. We find the three mesh vertices that make up this particular `Triangle`, and then compute the edge vectors and divisor. Note that if the divisor is zero, this triangle is degenerate and therefore cannot intersect a ray.

⟨*Compute* $\mathbf{s}_1$⟩≡

```
⟨Get triangle vertices in p1, p2, and p3⟩
Vector E1 = p2 - p1;
Vector E2 = p3 - p1;
Vector S_1 = Cross(ray.d, E2);
Float divisor = Dot(S_1, E1);
if (divisor == 0.)
    return false;
Float invDivisor = 1.f / divisor;
```

We can now compute the desired barycentric coordinate b_1. Recall that barycentric coordinates that are less than zero or greater than one represent points outside the triangle, so those are non-intersections.

⟨*Compute first barycentric coordinate*⟩≡

```
Vector T = ray.o - p1;
Float b1 = Dot(T, S_1) * invDivisor;
if (b1 < 0. || b1 > 1.)
    return false;
```

The second barycentric coordinate, b_2, is computed in a similar way:

⟨*Compute second barycentric coordinate*⟩≡

```
Vector S_2 = Cross(T, E1);
Float b2 = Dot(ray.d, S_2) * invDivisor;
if (b2 < 0. || b1 + b2 > 1.)
    return false;
```

```
31 Cross()
58 DifferentialGeometry
30 Dot()
35 Ray::o
27 Vector
```

Now that we know the ray intersects the triangle, we compute the distance along the ray at which the intersection occurs. This gives us one last opportunity to exit the procedure early, in the case where the t value falls outside our `Ray::mint` and `Ray::maxt` bounds.

⟨*Compute* `t` *to intersection point*⟩≡

```
Float t = Dot(E2, S_2) * invDivisor;
if (t < ray.mint || t > ray.maxt)
    return false;
```

We now have all the information we need to compute the `DifferentialGeometry` structure for this intersection. In contrast to previous shapes, we don't need to transform the partial derivatives to world-space, since the triangle's vertices were already transformed to world-space. Like the disk, the triangle's normal partial derivatves are also both $(0,0,0)$.

⟨*Fill in* `DifferentialGeometry` *from triangle hit*⟩≡

```
⟨Compute triangle partial derivatives⟩
⟨Interpolate (u,v) triangle parametric coordinates⟩
*dg = DifferentialGeometry(ray(t), dpdu, dpdv, Vector(0,0,0),
    Vector(0,0,0), tu, tv, this);
```

In order to generate consistent tangent vectors over triangle meshes, it is necessary to compute the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ using the parametric (u,v) values at the triangle vertices, if provided. Although the partial derivatives are the same at all points on the triangle, the implementation here just recomputes them each time an intersection is found. Although this results in redundant computation, the storage savings for large triangle meshes can be substantial.

The triangle is the set of points

$$\mathrm{p_P} + u\partial \mathrm{p}/\partial u + v\partial \mathrm{p}/\partial v,$$

for some $\mathrm{p_P}$, where u and v range over the parametric coordinates of the triangle. We also know the three vertex positions $\mathrm{p_i}$, $i = 0, 1, 2$ and the texture coordinates (u_i, v_i) at each vertex. From this it follows that

$$\mathrm{p_i} = \mathrm{p_P} + u_i\partial \mathrm{p}/\partial u + v_i\partial \mathrm{p}/\partial v.$$

This can be written in matrix form:

$$\begin{pmatrix} \mathrm{p_0} \\ \mathrm{p_1} \\ \mathrm{p_2} \end{pmatrix} = \begin{pmatrix} u_0 & v_0 & 1 \\ u_1 & v_1 & 1 \\ u_2 & v_2 & 1 \end{pmatrix} \begin{pmatrix} \partial \mathrm{p}/\partial u \\ \partial \mathrm{p}/\partial v \\ \mathrm{p_P} \end{pmatrix}$$

In other words, there is a unique affine mapping from the two-dimensional (u, v) space to points on the triangle (such a mapping exists even though the triangle is specified in 3D space, because it is planar.) To compute expressions for $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$, we just need to solve the matrix equation. We subtract the bottom row of each matrix from the top two rows, giving:

What happens to $\mathrm{p_P}$ from the previous equation?

$$\begin{pmatrix} \mathrm{p_0} - \mathrm{p_2} \\ \mathrm{p_1} - \mathrm{p_2} \end{pmatrix} = \begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix} \begin{pmatrix} \partial \mathrm{p}/\partial u \\ \partial \mathrm{p}/\partial v \end{pmatrix}$$

So

$$\begin{pmatrix} \partial \mathrm{p}/\partial u \\ \partial \mathrm{p}/\partial v \end{pmatrix} = \begin{pmatrix} u_0 - u_2 & v_0 - v_2 \\ u_1 - u_2 & v_1 - v_2 \end{pmatrix}^{-1} \begin{pmatrix} \mathrm{p_0} - \mathrm{p_2} \\ \mathrm{p_1} - \mathrm{p_2} \end{pmatrix}$$

Inverting a 2×2 matrix is straightforward; we just inline the computation directly in the code:

The points don't match the math! It looks like we've rotated the triangle here from the math. I don't want to touch this; shorty can you fix this so it matches the math and then test it?

```
⟨Compute triangle partial derivatives⟩≡
  Vector dpdu, dpdv;
  Float uvs[3][2];
  GetUVs(uvs);
  ⟨Compute deltas for triangle partial derivatives⟩
  Float determinant = du1 * dv2 - dv1 * du2;
  if (determinant == 0) {
      ⟨Handle zero determinant for triangle partial derivative matrix⟩
  }
  else {
      Float invdet = 1.f / determinant;
      dpdu = Vector((dx1 * dv2 - dv1 * dx2) * invdet,
          (dy1 * dv2 - dv1 * dy2) * invdet,
          (dz1 * dv2 - dv1 * dz2) * invdet);
      dpdv = Vector((du1 * dx2 - dx1 * du2) * invdet,
          (du1 * dy2 - dy1 * du2) * invdet,
          (du1 * dz2 - dz1 * du2) * invdet);
  }
```

why don't the points being subtracted match up with the math? Can we do dx1, dx2, etc as `Vectors` here? Also need to fix up idrafted dndu/dndv code equivalently.

32	`CoordinateSystem()`
31	`Cross()`
90	`Triangle`
96	`Triangle::GetUVs()`
87	`TriangleMesh`
27	`Vector`
30	`Vector::Hat()`

```
⟨Compute deltas for triangle partial derivatives⟩≡
  Float du1 = uvs[1][0] - uvs[0][0];
  Float du2 = uvs[2][0] - uvs[0][0];
  Float dv1 = uvs[1][1] - uvs[0][1];
  Float dv2 = uvs[2][1] - uvs[0][1];
  Float dx1 = p2.x - p1.x;
  Float dx2 = p3.x - p1.x;
  Float dy1 = p2.y - p1.y;
  Float dy2 = p3.y - p1.y;
  Float dz1 = p2.z - p1.z;
  Float dz2 = p3.z - p1.z;
```

Finally, it is necessary to handle the case when the matrix is singular and therefore cannot be inverted. Note that this only happens when the user-supplied per-vertex parameterization values are degenerate. In this case, the `Triangle` just chooses an arbitrary coordinate system, making sure that it is orthonormal:

```
⟨Handle zero determinant for triangle partial derivative matrix⟩≡
  CoordinateSystem(Cross(E2, E1).Hat(), &dpdu, &dpdv);
```

To compute the (u, v) parametric coordinates at the hit point, the barycentric interpolation formula is applied to the (u, v) parametric coordinates at the vertices.

```
⟨Interpolate (u,v) triangle parametric coordinates⟩≡
  Float b0 = 1 - b1 - b2;
  Float tu = b0*uvs[0][0] + b1*uvs[1][0] + b2*uvs[2][0];
  Float tv = b0*uvs[0][1] + b1*uvs[1][1] + b2*uvs[2][1];
```

The utility `GetUVs()` routine returns the (u, v) coordinates for the three vertices of the triangle, either from the `TriangleMesh`, if it has them, or returning defaults

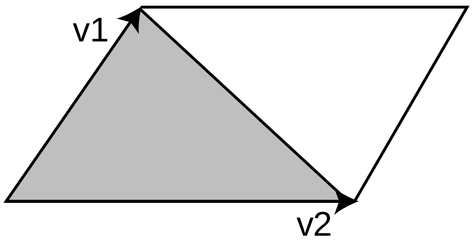


Figure 3.6: The area of a triangle with two edges given by vectors v_1 and v_2 is one half of the area of the parallelogram. The parallelogram area is given by the length of the cross product of v_1 and v_2.

if none were specified with the mesh.

⟨*TriangleMesh Method Definitions*⟩+≡

```
void Triangle::GetUVs(Float uv[3][2]) const {
    if (mesh->uvs) {
        uv[0][0] = mesh->uvs[2*v[0]];
        uv[0][1] = mesh->uvs[2*v[0]+1];
        uv[1][0] = mesh->uvs[2*v[1]];
        uv[1][1] = mesh->uvs[2*v[1]+1];
        uv[2][0] = mesh->uvs[2*v[2]];
        uv[2][1] = mesh->uvs[2*v[2]+1];
    } else {
        uv[0][0] = 0.; uv[0][1] = 0.;
        uv[1][0] = 1.; uv[1][1] = 0.;
        uv[2][0] = 1.; uv[2][1] = 1.;
    }
}
```

`Triangle` 90
`TriangleMesh::uvs` 88

3.6.3 Surface Area

Recall from Section 2.1 that the area of a parallelogram is given by the length of the cross product of the two vectors along its sides. From this, it's easy to see that given the vectors for two edges of a triangle, its area is $\frac{1}{2}$ of the area of the parallelogram given by those two vectors–see Figure 3.6.

⟨*TriangleMesh Method Definitions*⟩+≡

```
Float Triangle::Area() const {
    ⟨Get triangle vertices in p1, p2, and p3⟩
    return 0.5f * Cross(p2-p1, p3-p1).Length();
}
```

3.6.4 Shading Geometry

text here

need xrefs in the code here

```
⟨Triangle Public Methods⟩+≡
  virtual void GetShadingGeometry(const Transform &obj2world,
          const DifferentialGeometry &dg,
          DifferentialGeometry *dgShading) const {
      if (!mesh->n && !mesh->s) {
          *dgShading = dg;
          return;
      }
      ⟨Initialize Triangle shading geometry with n and s⟩
  }
```

```
⟨Initialize Triangle shading geometry with n and s⟩≡
  ⟨Compute barycentric coordinates for point⟩
  ⟨Use n and s to compute shading tangents for triangle, ss and ts⟩
  Vector dndu, dndv;
  ⟨Compute ∂n/∂u and ∂n/∂v for triangle shading geometry⟩
  *dgShading = DifferentialGeometry(dg.p, ss, ts,
      dndu, dndv, dg.u, dg.v,
      dg.shape, dg.dudx, dg.dvdx, dg.dudy,
      dg.dvdy);
```

Recall that the (u, v) parametric coordinates in the `DifferentialGeometry` for a triangle are computed with barycentric interpolation of parametric coordinates at the triangle vertices.

58	DifferentialGeometry
400	DifferentialGeometry::dudx
400	DifferentialGeometry::dudy
400	DifferentialGeometry::dvdx
400	DifferentialGeometry::dvdy
58	DifferentialGeometry::p
58	DifferentialGeometry::shape
58	DifferentialGeometry::u
58	DifferentialGeometry::v
43	Transform
90	Triangle
90	Triangle::mesh
88	TriangleMesh::n
88	TriangleMesh::s
27	Vector

$$
\begin{aligned}
u &= b_0 u_0 + b_1 u_1 + b_2 u_2 \\
v &= b_0 v_0 + b_1 v_1 + b_2 v_2
\end{aligned}
$$

Because b_i are barycentric coordinates, $b_0 = 1 - b_1 - b_2$. Here, u, v, u_i and v_i are all known, u and v from the `DifferentialGeometry` and u_i and v_i from the `Triangle`. We can substitue for the b_0 term and rewrite the above equations, giving a linear system in two unknowns b_1 and b_2.

$$
\begin{pmatrix} u_1 - u_0 & u_2 - u_1 \\ v_1 - v_0 & v_2 - v_1 \end{pmatrix}
\begin{pmatrix} b_1 \\ b_2 \end{pmatrix}
=
\begin{pmatrix} u - u_0 \\ v - v_0 \end{pmatrix}
$$

This is a linear system of the basic form $Ab = C$. We can solve for b by inverting A, giving the two barycentric coordinates

$$
b = A^{-1} C.
$$

The closed form solution for this is implemented in the utility routine `SolveLinearSystem2x2()`.

```
⟨Compute barycentric coordinates for point⟩≡
  Float b[3];
  ⟨Initialize A and C matrices for barycentrics⟩
  if (!SolveLinearSystem2x2(A, C, &b[1])) {
      ⟨Handle degenerate parametric mapping⟩
  }
  else
      b[0] = 1.f - b[1] - b[2];
```

⟨*Initialize* `A` *and* `C` *matrices for barycentrics*⟩≡

```
Float uv[3][2];
GetUVs(uv);
Float A[2][2] = { { uv[1][0] - uv[0][0], uv[2][0] - uv[0][0] },
                  { uv[1][1] - uv[0][1], uv[2][1] - uv[0][1] } };
Float C[2] = { dg.u - uv[0][0], dg.v - uv[0][1] };
```

If the determinant of *A* is zero, the solution is undefined and `SolveLinearSystem2x2()` returns false. This case happens if all three triangle vertices had the same texture coordinates, for example. In this case, the barycentric coordinates are all set arbitrarily to $\frac{1}{3}$.

⟨*Handle degenerate parametric mapping*⟩≡

```
b[0] = b[1] = b[2] = 1.f/3.f;
```

⟨*Use* `n` *and* `s` *to compute shading tangents for triangle,* `ss` *and* `ts`⟩≡

```
Normal ns;
Vector ss, ts;
if (mesh->n) ns = (b[0] * mesh->n[v[0]] + b[1] * mesh->n[v[1]] +
    b[2] * mesh->n[v[2]]).Hat();
else   ns = dg.nn;
if (mesh->s) ss = (b[0] * mesh->s[v[0]] + b[1] * mesh->s[v[1]] +
    b[2] * mesh->s[v[2]]).Hat();
else   ss = dg.dpdu.Hat();
ts = obj2world(Cross(ss, ns)).Hat();
ss = obj2world(Cross(ts, ns)).Hat();
```

`Cross()`	31
`DifferentialGeometry::u`	58
`DifferentialGeometry::v`	58
`Normal`	34
`SolveLinearSystem2x2()`	675
`Vector`	27

make sure not to include the heightfield on the CD.

3.7 ***ADV***: Subdivision Surfaces

We will wrap up this chapter by defining a shape that implements *subdivision surfaces*, which are particularly well-suited to describing complex smooth shapes. The subdivision surface for a particular mesh is defined by repeatedly subdividing the faces of the mesh into smaller faces, then changing the new vertex locations using weighted combinations of the old vertex positions.

For appropriately chosen subdivision rules, this process converges to give a smooth *limit surface* as the number of subdivision steps goes to infinity. In practice, just a few levels of subdivision typically suffice to give a good approximation of the limit surface. Figure 3.7 shows the effect of applying one set of subdivision rules to a tetrahedron; on the left is the original control mesh, and one, two, three, and four levels of subdivision are shown moving from left to right.

Though originally developed in the 1970s, subdivision surfaces have recently received a fair amount of attention in computer graphics thanks to some key advantages over polygonal and spline-based representations of surfaces. The advantages of subdivision include:

- Subdivision surfaces are smooth, as opposed to polygon meshes which appear faceted when viewed close up, regardless of how finely they are modeled.

Figure 3.7: tetra control mesh and 4 levels of subdivision.

- A lot of existing infrastructure in modeling systems can be retargeted to subdivision. The classic toolbox of techniques for modeling polygon meshes can be applied to modeling subdivision control meshes.
- Subdivision surfaces are well-suited to describing objects with complex topology, since we can start with a control meshes of arbitrary (manifold) topology. Parametric surface models generally don't handle complex topology well.
- Subdivision methods are often generalizations of spline-based surface representations, so spline surfaces can often just be run through general subdivision surface renderers.
- It is easy to add detail to a localized region of a subdivision surface, simply by adding faces to appropriate parts of the control mesh. This is much less easily done with spline representations.

Here, we will describe an implementation of *Loop subdivision surfaces*[5]. The Loop rules are based on triangular faces in the control mesh; faces with more than three vertices are just triangulated at the start. At each subdivision step, all faces split into four child faces (Figure 3.8). New vertices are added along all of the edges of the original mesh, with positions computed using weighted averages of nearby vertices. Furthermore, the position of each original vertex is updated with a weighted average of its position and its new neighbors' positions.

3.7.1 Mesh Representation

[5] Don't be fooled by the name. These surfaces are not ‘loopy”; they are named after the inventor of the subdivision rules, Charles Loop.

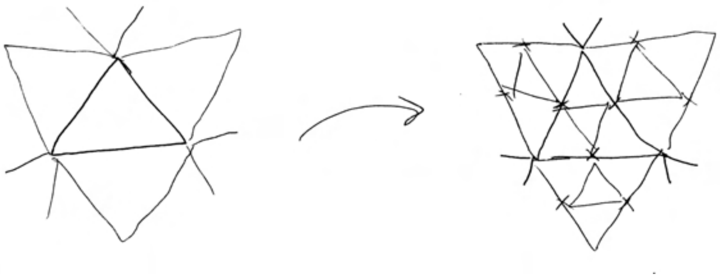


Figure 3.8: Basic refinement process for Loop subdivision: the control mesh on the left has been subdivided once to create the new mesh on the right. Each triangular face of the mesh has been subdivided into four new faces by splitting each of the edges and connecting the new vertices with new edges.

```
⟨LoopSubdiv Declarations⟩≡
  class COREDLL LoopSubdiv : public Shape {
  public:
      ⟨LoopSubdiv Public Methods⟩
  private:
      ⟨LoopSubdiv Private Methods⟩
      ⟨LoopSubdiv Private Data⟩
  };
```

Point	33
SDFace	102
SDVertex	101
Shape	63
Transform	43
TriangleMesh	87

We will start by describing the data structures used to represent the subdivision mesh. These data structures need to be carefully designed in order to support all of the operations necessary to cleanly implement the subdivision algorithm. The parameters to the LoopSubdiv constructor specify a triangle mesh in exactly the same format used in the TriangleMesh constructor (see Section 3.6 on page 87.): each face is described by three integer vertex indices, giving offsets into the vertex array P for the face's three vertices. We will need to process this data to determine which faces are adjacent to each other, which faces are adjacent to which vertices, etc.

```
⟨LoopSubdiv Method Definitions⟩≡
  LoopSubdiv::LoopSubdiv(const Transform &o2w, bool ro, int nfaces,
          int nvertices, const int *vertexIndices,
          const Point *P, int nl)
      : Shape(o2w, ro) {
      nLevels = nl;
      ⟨Allocate LoopSubdiv vertices and faces⟩
      ⟨Set face to vertex pointers⟩
      ⟨Set neighbor pointers in faces⟩
      ⟨Finish vertex initialization⟩
  }
```

We will shortly define SDVertex and SDFace structures, which hold data for vertices and faces in the subdivision mesh. We start by allocating one instance of the SDVertex class for each vertex in the mesh and an SDFace for each face. For now, these are mostly uninitialized.

⟨*Allocate* `LoopSubdiv` *vertices and faces*⟩≡

```
int i;
SDVertex *verts = new SDVertex[nvertices];
for (i = 0; i < nvertices; ++i) {
    verts[i] = SDVertex(P[i]);
    vertices.push_back(&verts[i]);
}
SDFace *fs = new SDFace[nfaces];
for (i = 0; i < nfaces; ++i)
    faces.push_back(&fs[i]);
```

The `LoopSubdiv` destructor, which we won't include here, just deletes all of the faces and vertices allocated above.

⟨*LoopSubdiv Private Data*⟩≡

```
int nLevels;
vector<SDVertex *> vertices;
vector<SDFace *> faces;
```

The Loop subdivision scheme, like most other subdivision schemes, assumes that the control mesh is *manifold*, i.e. no more than two faces share any given edge. Such a mesh may be closed or open: a *closed mesh* has no boundary, and all faces have adjacent faces across each of their edges. An *open mesh* has some faces that do not have all three neighbors. The `LoopSubdiv` implementation supports both closed and open meshes.

100	`LoopSubdiv`
33	`Point`
102	`SDFace`
658	`vector`

In the interior of a triangle mesh, most vertices are adjacent to six faces and have six neighbor vertices directly connected to them with edges. On the boundaries of an open mesh, most vertices are adjacent to three faces and four vertices. The number of vertices directly adjacent to a vertex is called the vertex's *valence*. Interior vertices with valence other than six, or boundary vertices with valence other than four are called *extraordinary vertices*; otherwise they are called *regular*. Loop subdivision surfaces are smooth everywhere except at their extraordinary vertices.

Each `SDVertex` stores its position `P`, a boolean that indicates whether it is a regular or extraordinary vertex, and a boolean that records if it lies on the boundary of the mesh. It also holds a pointer to one of the faces adjacent to it; later we will use this pointer to start an iteration over all of the faces adjacent to the vertex by following pointers stored in each `SDFace` to record which faces are adjacent. **¡—- This sentence is pretty garbled.** Finally, we have a pointer to store the new `SDVertex` for the next level of subdivision, if any.

⟨*LoopSubdiv Local Structures*⟩+≡

```
struct SDVertex {
    ⟨SDVertex Constructor⟩
    ⟨SDVertex Methods⟩
    Point P;
    SDFace *startFace;
    SDVertex *child;
    bool regular, boundary;
};
```

The constructor for `SDVertex` does the obvious initialization; note that `SDVertex::startFace` is initialized to `NULL`.

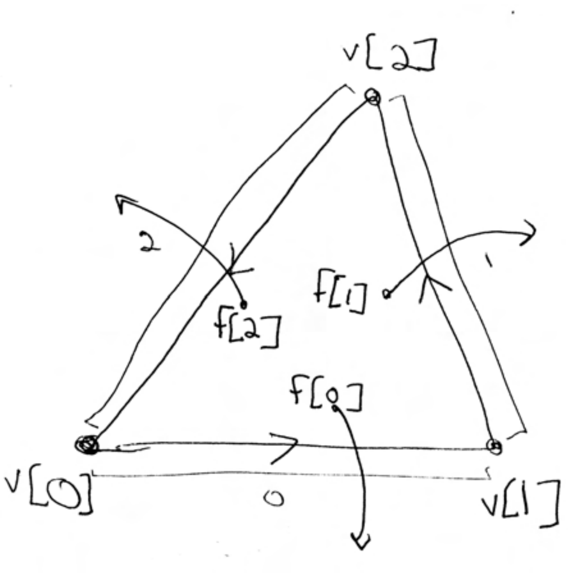


Figure 3.9: Each triangular face stores three pointers to `SDVertex` objects `v[i]` and three pointers to neighboring faces `f[i]`. Neighboring faces are indexed using the convention that the *i*th edge is the edge from `v[i]` to `v[(i+1)%3]`, and the neighbor across the *i*th edge is in `f[i]`.

```
Point  33
SDVertex  101
```

⟨*SDVertex Constructor*⟩≡

```
SDVertex(Point pt = Point(0,0,0))
    : P(pt), startFace(NULL), child(NULL),
    regular(false), boundary(false) {
}
```

The `SDFace` structure is where we maintain most of the topological information about the mesh. Because all faces are triangular, we always store three pointers to the vertices for this face and three pointers to the faces adjacent to this one. (The face neighbor pointers will be `NULL` if the face is on the boundary of an open mesh.)

The face neighbor pointers are indexed such that if we label the edge from `v[i]` to `v[(i+1)%3]` as the *i*th edge, then the neighbor face across that edge is stored in `f[i]`–see Figure 3.9. This labeling convention is important to keep in mind; later when we are updating the topology of a newly subdivided mesh, we will make extensive use of it to navigate around the mesh. Similarly to the `SDVertex` class, we also store pointers to child faces at the next level of subdivision.

⟨*LoopSubdiv Local Structures*⟩+≡

```
struct SDFace {
    ⟨SDFace Constructor⟩
    ⟨SDFace Methods⟩
    SDVertex *v[3];
    SDFace *f[3];
    SDFace *children[4];
};
```

The `SDFace` constructor is straightforward–it simply sets pointers to `NULL`–so it is not shown here.

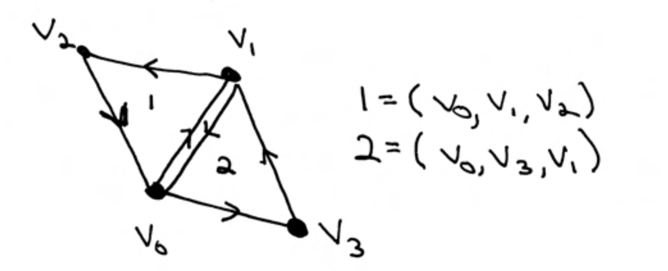


Figure 3.10: All of the faces in the input mesh must be specified so that each shared edge is given once in each direction. Here, the edge from v_0 to v_1 is traversed from v_0 to v_1 by face number one, and from v_1 to v_0 by face number two. Another way to think of this is in terms of face orientation: all faces' vertices should be given consistently in either clockwise or counter-clockwise order, as seen from outside the mesh.

In order to simplify navigation of the `SDFace` data structure, we'll provide macros that make it easy to determine the vertex and face indices before or after a particular index. These macros add appropriate offsets and compute the result modulus three to handle cycling around. To compute the previous index, we add 2 instead of subtracting 1, which avoids taking the modulus of a negative number, the result of which is implementation-dependent in C++.

100 LoopSubdiv
102 SDFace

⟨*LoopSubdiv Macros*⟩≡

```
#define NEXT(i) (((i)+1)%3)
#define PREV(i) (((i)+2)%3)
```

In addition to requiring a manifold mesh, the `LoopSubdiv` class expects that the control mesh specified by the user will be *consistently ordered*–each *directed edge* in the mesh can be present only once. An edge that is shared by two faces should be specified in a different direction by each face. Consider two vertices, v_0 and v_1, with an edge between them. We expect that one of the triangular faces that has this edge will specify its three vertices so that v_0 is before v_1, and that the other face will specify its vertices so that v_1 is before v_0 (Figure 3.10). A Möbius strip is one example of a surface that cannot be consistently ordered, but such surfaces come up rarely in rendering so in practice this restriction is not troublesome.

Given this assumption about the input data, we will initialize this mesh's topological data structures. We first loop over all of the faces and set their `v` pointers to point to their three vertices. We also set each vertex's `SDVertex::startFace` pointer to point to one of the vertex's neighboring faces. It doesn't matter which of its adjacent faces we choose, so we just keep resetting it each time we come across another face that it is incident to, ensuring that all vertices have some non-NULL face pointer by the time we're done.

⟨*Set face to vertex pointers*⟩≡

```
const int *vp = vertexIndices;
for (i = 0; i < nfaces; ++i) {
    SDFace *f = faces[i];
    for (int j = 0; j < 3; ++j) {
        SDVertex *v = vertices[vp[j]];
        f->v[j] = v;
        v->startFace = f;
    }
    vp += 3;
}
```

Now we need to set each face's `f` pointer to point to its neighboring faces. This is a bit trickier, since face adjacency information isn't directly specified by the user. We'll loop over the faces and store an `SDEdge` object for each of their three edges; when we come to another face that shares the same edge, we can update both faces' neighbor pointers.

⟨*LoopSubdiv Local Structures*⟩+≡

```
struct SDEdge {
    ⟨SDEdge Constructor⟩
    ⟨SDEdge Comparison Function⟩
    SDVertex *v[2];
    SDFace *f[2];
    SDFace **fptr;
};
```

SDFace	102
SDFace::v	102
SDVertex	101
SDVertex::startFace	101

The constructor takes pointers to the two vertices at each end of the edge. It orders them so that `v[0]` holds the one that is first in memory; This code may seem strange, but we're simply relying on the fact that pointers in C++ are really just 32-bit numbers that can be manipulated like integers, and that the ordering of vertices on an edge is arbitrary. By sorting vertices on the address of the pointer, we guarantee that we properly recognize that the edge (v_a, v_b) is the same as the edge (v_b, v_a), regardless of what order the vertices are given in.

okay, so "fptr" is never used for anything meaningful. Can we get rid of it please? Same for f[1].

⟨*SDEdge Constructor*⟩≡

```
SDEdge(SDVertex *v0 = NULL, SDVertex *v1 = NULL) {
    v[0] = min(v0, v1);
    v[1] = max(v0, v1);
    f[0] = f[1] = NULL;
    fptr = NULL;
}
```

We also define an ordering operation for `SDEdge` objects so that they used by other data structures that rely on ordering being well-defined.

⟨*SDEdge Comparison Function*⟩≡

```
bool operator<(const SDEdge &e2) const {
    if (v[0] == e2.v[0]) return v[1] < e2.v[1];
    return v[0] < e2.v[0];
}
```

Now we can get to work, looping over the edges in all of the faces and updating the neighbor pointers as we go. We use an STL `set<>` to store the edges that have only one adjacent face so far. The `set<>` allows us to search for a particular edge in $O(\log n)$, using the comparison function above.

⟨*Set neighbor pointers in* `faces`⟩≡

```
set<SDEdge> edges;
for (i = 0; i < nfaces; ++i) {
    SDFace *f = faces[i];
    for (int edge = 0; edge < 3; ++edge) {
        ⟨Update neighbor pointer for edge⟩
    }
}
```

For each edge in each face, we create an edge object and see if the same edge was seen previously. If so, we initialize both faces' neighbor pointers across the edge. If not, we add the edge to the set of edges.

103 `NEXT`
104 `SDEdge`
104 `SDEdge::f`
104 `SDEdge::fptr`
104 `SDEdge::v`
102 `SDFace`
102 `SDFace::f`

this variable naming is very confusing – you shouldn't be seetting a vertex (v0) to an edge (edge). I realize what's going on, but the code is quite hard to read. Also, NEXT(edge) sounds like the next edge. This needs some fixing.

⟨*Update neighbor pointer for* `edge`⟩≡

```
int v0 = edge, v1 = NEXT(edge);
SDEdge e(f->v[v0], f->v[v1]);
if (edges.find(e) == edges.end()) {
    ⟨Handle new edge⟩
}
else {
    ⟨Handle previously-seen edge⟩
}
```

Given an edge that we haven't seen before, we store the current face's pointer in the edge object's `f[0]` member. When we come across the other face that shares this edge (if any), we can thus know what the neighboring face is. We also store a pointer to the location in the current `SDFace` that will point to the neighboring face once we find it. **this refers to fptr, right? It's not used anywhere! Search for it, you'll see I'm right. Let's delete all this crap.**

⟨*Handle new edge*⟩≡

```
e.f[0] = f;
e.fptr = &(f->f[edge]);
edges.insert(e);
```

When we find the second face on an edge, we can set the neighbor pointers for each of the two faces. We then remove the edge from the edge set, since no edge can be shared by more than two faces.

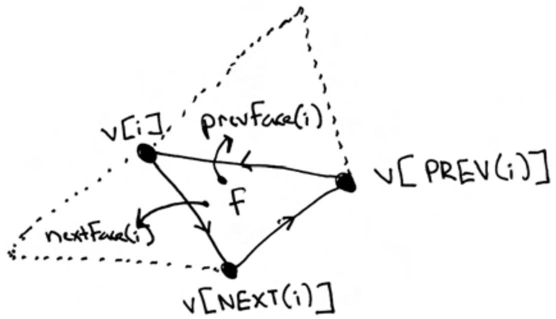


Figure 3.11: Given a vertex `v[i]` and a face that it is incident to, `f`, we define the *next face* as the face adjacent to `f` across the edge from `v[i]` to `v[NEXT(i)]`. The previous face is defined analogously.

⟨*Handle previously-seen edge*⟩≡

```
e = *edges.find(e);
*e.fptr = f;
f->f[edge] = e.f[0];
edges.erase(e);
```

SDEdge::f	104
SDEdge::fptr	104
SDFace	102
SDFace::f	102
SDVertex	101
SDVertex::boundary	101
SDVertex::startFace	101

What happens to the edges left in the edges set at the end? Are they deleted? Could we use those to set the boundary flag? – Jessica wants to know.

Now that all faces have proper neighbor pointers, we can set the `boundary` and `regular` flags in each of the vertices. In order to deterime if a vertex is a boundary vertex, we'll define an ordering of faces around a vertex (Figure 3.11). For a vertex `v[i]` on a face `f`, we define the vertex's *next face* as the face across the edge from `v[i]` to `v[NEXT(i)]` and the *previous face* as the face across the edge from `v[PREV(i)]` to `v[i]`.

We will frequently need to know the valence of a vertex, so we provide the method `SDVertex::valence()`.

⟨*LoopSubdiv Inline Functions*⟩≡

```
inline int SDVertex::valence() {
    SDFace *f = startFace;
    if (!boundary) {
        ⟨Compute valence of interior vertex⟩
    }
    else {
        ⟨Compute valence of boundary vertex⟩
    }
}
```

To compute the valence of a non-boundary vertex, we count the number of the adjacent faces starting by following each face's neighbor pointers around the vertex until we reach the starting face. The valence is equal to the number of faces visited.

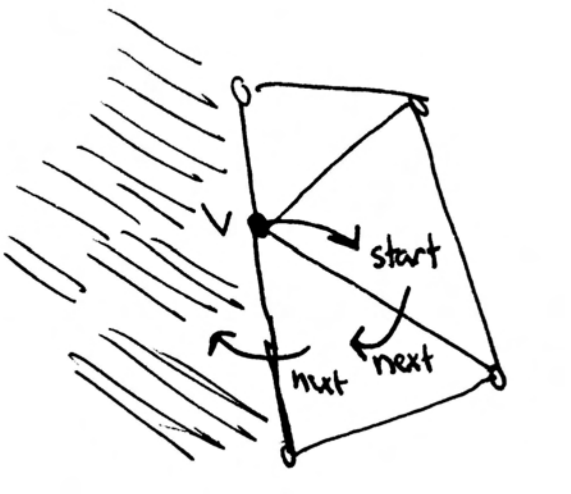


Figure 3.12: We can determine if a vertex is a boundry vertex by starting from the adjacent face `startFace` and following next face pointers around the vertex. If we come to a face that has no next neighbor face, then the vertex is on a boundary. If we return to `startFace`, it's an interior vertex.

108 SDFace::nextFace()
108 SDFace::prevFace()

⟨*Compute valence of interior vertex*⟩≡

```
int nf = 1;
while ((f = f->nextFace(this)) != startFace)
    ++nf;
return nf;
```

For boundary vertices we use the same approach, though in this case, the valence is one more than the number of adjacent faces. The loop over adjacent faces is slightly more complicated here: we follow pointers to the next face around the vertex until we reach the boundary, counting the number of faces seen. We then start again at `startFace` and follow previous face pointers until we encounter the boundary in the other direction.

⟨*Compute valence of boundary vertex*⟩≡

```
int nf = 1;
while ((f = f->nextFace(this)) != NULL)
    ++nf;
f = startFace;
while ((f = f->prevFace(this)) != NULL)
    ++nf;
return nf+1;
```

By successively going to the next face around `v`, we can iterate over the faces adjacent to it. If we eventually return to the face we started at, then we are at an interior vertex; if we come to an edge with a `NULL` neighbor pointer, then we're at a boundary vertex–see Figure 3.12. Once we've determined if we have a boundary vertex, we compute to valence of the vertex and set the `regular` flag if the valence is 6 for an interior vertex or 4 for a boundary vertex.

⟨*Finish vertex initialization*⟩≡

```
for (i = 0; i < nvertices; ++i) {
    SDVertex *v = vertices[i];
    SDFace *f = v->startFace;
    do {
        f = f->nextFace(v);
    } while (f && f != v->startFace);
    v->boundary = (f == NULL);
    if (!v->boundary && v->valence() == 6)
        v->regular = true;
    else if (v->boundary && v->valence() == 4)
        v->regular = true;
    else
        v->regular = false;
}
```

Here is the utility function that finds the index of a given vertex for one of the faces adjacent to it. It's a fatal error to pass a pointer to a vertex that isn't part of the current face—this case would represent a bug elsewhere in the subdivision code.

⟨*SDFace Methods*⟩≡

```
int vnum(SDVertex *vert) const {
    for (int i = 0; i < 3; ++i)
        if (v[i] == vert) return i;
    Severe("Basic logic error in SDFace::vnum()");
    return -1;
}
```

NEXT	103
PREV	103
SDFace	102
SDFace::f	102
SDFace::v	102
SDVertex	101
SDVertex::boundary	101
SDVertex::regular	101
SDVertex::startFace	101

Since the next face for a vertex v[i] on a face f is over the *i*th edge (recall the mapping of edge neighbor pointers from Figure 3.9), we can find the appropriate face neighbor pointer easily given the index i for the vertex, which the vnum() utility function provides. The previous face is across the edge from PREV(i) to i, so we return f[PREV(i)] for the previous face.

⟨*SDFace Methods*⟩+≡

```
SDFace *nextFace(SDVertex *vert) {
    return f[vnum(vert)];
}
```

⟨*SDFace Methods*⟩+≡

```
SDFace *prevFace(SDVertex *vert) {
    return f[PREV(vnum(vert))];
}
```

It will be very useful to be able to get the next and previous vertices around a face starting at any vertex. The SDFace::nextVert() and SDFace::prevVert() methods do just that (Figure 3.13).

⟨*SDFace Methods*⟩+≡

```
SDVertex *nextVert(SDVertex *vert) {
    return v[NEXT(vnum(vert))];
}
```

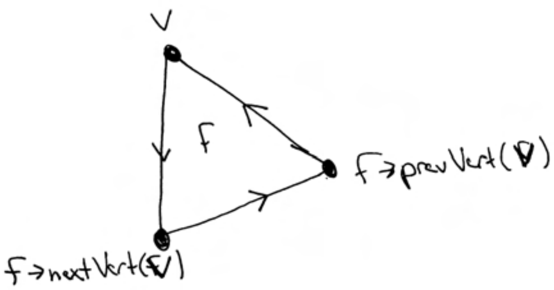


Figure 3.13: Given a vertex `v` on a face `f`, the method `f->prevVert(v)` returns the previous vertex around the face from `v` and `f->nextVert(v)` returns the next vertex.

⟨*SDFace Methods*⟩+≡
```
SDVertex *prevVert(SDVertex *vert) {
    return v[PREV(vnum(vert))];
}
```

3.7.2 Bounds

Loop subdivision surfaces have the *convex hull property*: the limit surface is guaranteed to be inside the convex hull of the original control mesh. Thus, for the bounding methods, we can just bound the original control vertices. The bounding methods are essentially equivalent to those in `TriangleMesh`, so we won't include them here.

38	`BBox`
100	`LoopSubdiv`
103	`PREV`
102	`SDFace::v`
108	`SDFace::vnum()`
101	`SDVertex`
87	`TriangleMesh`

⟨*LoopSubdiv Public Methods*⟩+≡
```
BBox ObjectBound() const;
BBox WorldBound() const;
```

3.7.3 Subdivison

Now we can show how subdivision proceeds with the Loop rules. The `LoopSubdiv` shape doesn't support intersection directly, but will apply subdivision a fixed number of times to generate a `TriangleMesh` for rendering. An exercise at the end of the chapter discusses adaptive subdivision, where that each original face is subdivided just enough so that the result looks smooth from a particular viewpoint.

⟨*LoopSubdiv Method Definitions*⟩+≡
```
bool LoopSubdiv::CanIntersect() const {
    return false;
}
```

The `Refine()` method handles all of the subdivision. We repeatedly apply the subdivision rules to the mesh, each time generating a new mesh to be used as the input to the next step. After each subdivision step, the `f` and `v` arrays in the `Refine()` method are updated to point to the faces and vertices from the level of subdivision just computed. When we are done subdividing, a `TriangleMesh` representation of the surface is created and returned to the caller.

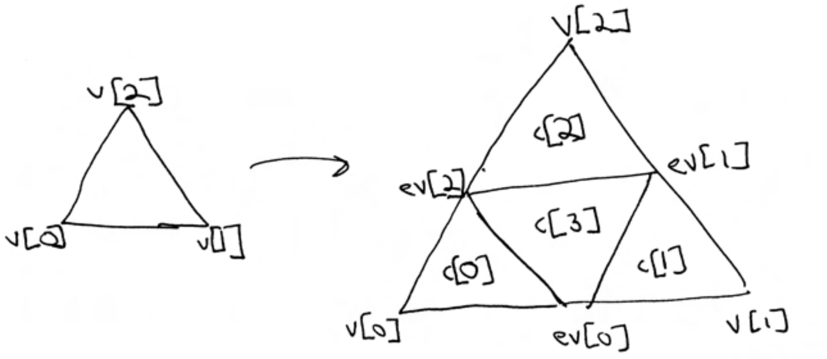


Figure 3.14: Basic Loop subdivision of a single face: four child faces are created, ordered such that the *i*th child face is adjacent to the *i*th vertex of the original face and the fourth child face is in the center of the subdivided face. Three edge vertices need to be computed; they are numbered so that the *i*th edge vertex is along the *i*th edge of the original face. **This diagram could be clearer, in particular it should show what the ”child” pointers do.**

LoopSubdiv	100
LoopSubdiv::faces	101
LoopSubdiv::nLevels	101
LoopSubdiv::vertices	101
ObjectArena	668
Reference	664
SDFace	102
SDVertex	101
Shape	63
vector	658

What is an ObjectArena? We should say something about this before using it.

⟨*LoopSubdiv Method Definitions*⟩+≡

```
void LoopSubdiv::Refine(vector<Reference<Shape> > &refined) const {
    vector<SDFace *> f = faces;
    vector<SDVertex *> v = vertices;
    ObjectArena<SDVertex> vertexArena;
    ObjectArena<SDFace> faceArena;
    for (int i = 0; i < nLevels; ++i) {
        ⟨Update f and v for next level of subdivision⟩
    }
    ⟨Push vertices to limit surface⟩
    ⟨Compute vertex tangents on limit surface⟩
    ⟨Create TriangleMesh from subdivision mesh⟩
}
```

The main loop of a subdivision step proceeds as follows: We create `vectors` for all of the vertices and faces at this level of subdivision and then proceed to compute new vertex positions and update the topological representation for the refined mesh. Figure 3.14 shows the basic refinement rules for faces in the mesh. Each face is split into four child faces, such that the *i*th child face is next to the *i*th vertex of the input face and the final face is in the center. Three new vertices are then computed along the split edges of the original face.

⟨*Update* `f` *and* `v` *for next level of subdivision*⟩≡

```
  vector<SDFace *> newFaces;
  vector<SDVertex *> newVertices;
  ⟨Allocate next level of children in mesh tree⟩
  ⟨Update vertex positions and create new edge vertices⟩
  ⟨Update new mesh topology⟩
  ⟨Prepare for next level of subdivision⟩
```

First, we allocate storage for the updated values of the vertices in the input mesh. We also allocate storage for the child faces. We don't yet do any initialization of the new vertices and faces other than setting the `regular` and `boundary` flags for the vertices. Subdivision leaves boundary vertices on the boundary and interior vertices in the interior. Furthermore, it doesn't change the valence of vertices in the mesh.

⟨*Allocate next level of children in mesh tree*⟩≡

```
  for (u_int j = 0; j < v.size(); ++j) {
      v[j]->child = new (vertexArena) SDVertex;
      v[j]->child->regular = v[j]->regular;
      v[j]->child->boundary = v[j]->boundary;
      newVertices.push_back(v[j]->child);
  }
  for (u_int j = 0; j < f.size(); ++j)
      for (int k = 0; k < 4; ++k) {
          f[j]->children[k] = new (faceArena) SDFace;
          newFaces.push_back(f[j]->children[k]);
      }
```

102	`SDFace`
102	`SDFace::children`
101	`SDVertex`
101	`SDVertex::boundary`
101	`SDVertex::regular`
658	`vector`

Computing new vertex positions

Before we worry about the topology of the subdivided mesh, we compute positions for all of the vertices in the mesh. First, we will consider the problem of computing updated positions for all of the vertices that were already present in the mesh; these vertices are called *even vertices*. We will then compute the new vertices on the split edges–these are called *odd vertices*.

⟨*Update vertex positions and create new edge vertices*⟩≡

```
  ⟨Update vertex positions for even vertices⟩
  ⟨Compute new odd edge vertices⟩
```

Different techniques are used to compute the updated positions for each of the different types of even vertices–regular and extraordinary, boundary and interior. This gives four cases to handle.

⟨*Update vertex positions for even vertices*⟩≡

```
  for (u_int j = 0; j < v.size(); ++j) {
      if (!v[j]->boundary) {
          ⟨Apply one-ring rule for even vertex⟩
      }
      else {
          ⟨Apply boundary rule for even vertex⟩
      }
  }
```

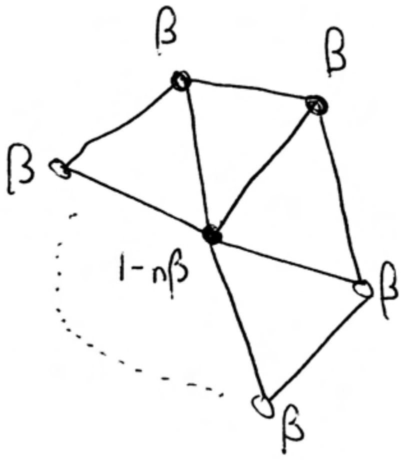


Figure 3.15: The new position v' for a vertex v is computed by weighting the adjacent vertices v_i by a weight β and weighting v by $(1-n\beta)$, where n is the valence of v. The adjacent vertices v_i are collectively referred to as the *one ring* around v.

`pSubdiv::weightOneRing()`	113
`SDVertex::boundary`	101
`SDVertex::child`	101
`SDVertex::P`	101
`SDVertex::regular`	101
`SDVertex::valence()`	106

For both types of interior vertices, we take the set of vertices adjacent to each vertex (called the *one-ring* around it, reflecting the fact that it's a ring of neighbors) and weight each of the neighbor vertices by a weight β (Figure 3.15.). The vertex we are updating, in the center, is weighted by $1-n\beta$, where n is the valence of the vertex. Thus, the new position v' for a vertex v is:

$$v' = (1-n\beta)v + \sum_{i=1}^{N} \beta v_i.$$

This formulation ensures that the sum of weights is one, which guarantees the convex hull property we used above for bounding the surface. The position of the vertex being updated is only affected by vertices that are nearby; this is known as *local support*. Loop subdivision is particularly efficient to implement because its subdivision rules all have this property.

The particular weight β used for this step is a key component of the subdivision method, and must be chosen carefully in order to ensure smoothness of the limit surface among other desirable properties. The `LoopSubdiv::beta()` method below computes a β value based on the vertex's valence that ensures smoothness. For regular interior vertices, `LoopSubdiv::beta()` returns $\frac{1}{16}$. Since this is a common case, we use the number $\frac{1}{16}$ directly instead of calling `LoopSubdiv::beta()` every time.

either show why or direct the reader to a proof/derivation.

⟨*Apply one-ring rule for even vertex*⟩≡

```
if (v[j]->regular)
    v[j]->child->P = weightOneRing(v[j], 1.f/16.f);
else
    v[j]->child->P = weightOneRing(v[j], beta(v[j]->valence()));
```

What the heck is 3/16 here? Explain shit like this.

⟨*LoopSubdiv Private Methods*⟩≡

```
static Float beta(int valence) {
    if (valence == 3) return 3.f/16.f;
    else return 3.f / (8.f * valence);
}
```

The `LoopSubdiv::weightOneRing()` function loops over the one-ring of adjacent vertices and applies the given weight to compute a new vertex position. It uses the `SDVertex::oneRing()` function, defined below, which returns the positions of the vertices around the vertex `vert`.

⟨*LoopSubdiv Method Definitions*⟩+≡

```
Point LoopSubdiv::weightOneRing(SDVertex *vert, Float beta) {
    ⟨Put vert one-ring in Pring⟩
    Point P = (1 - valence * beta) * vert->P;
    for (int i = 0; i < valence; ++i)
        P += beta * Pring[i];
    return P;
}
```

Jesus, we re-compute valence a lot. Could we either make this a variable or thunk the damn function so we don't walk all around the mesh like 20 times per vertex?

100	`LoopSubdiv`
33	`Point`
102	`SDFace`
108	`SDFace::nextFace()`
108	`SDFace::nextVert()`
101	`SDVertex`
101	`SDVertex::boundary`
101	`SDVertex::P`
101	`SDVertex::startFace`
106	`SDVertex::valence()`

⟨*Put* `vert` *one-ring in* `Pring`⟩≡

```
int valence = vert->valence();
Point *Pring = (Point *)alloca(valence * sizeof(Point));
vert->oneRing(Pring);
```

⟨*LoopSubdiv Method Definitions*⟩+≡

```
void SDVertex::oneRing(Point *P) {
    if (!boundary) {
        ⟨Get one ring vertices for interior vertex⟩
    }
    else {
        ⟨Get one ring vertices for boundary vertex⟩
    }
}
```

It's relatively easy to get the one-ring around an interior vertex: we loop over the faces adjacent to the vertex, and for each face grab the next vertex the center vertex.

⟨*Get one ring vertices for interior vertex*⟩≡

```
SDFace *face = startFace;
do {
    *P++ = face->nextVert(this)->P;
    face = face->nextFace(this);
} while (face != startFace);
```

The one-ring around a boundary vertex is a bit more tricky. We will carefully store the one ring in the given `Point` array so that the first and last entries in the array are the two adjacent vertices along the boundary. This requires that we first

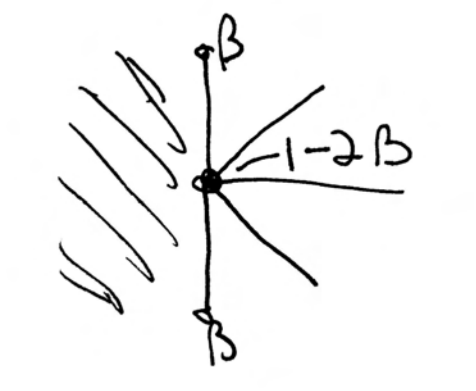


Figure 3.16: Subdivision on a boundary edge: the new position for the vertex in the center is computed by weighting it and its two neighbor vertices by the weights shown.

loop around neighbor faces until we reach a face on the boundary and then loop around the other way, storing vertices one by one. **if we're just going to multiply everything by β, why does the order matter? Say something here.**

SDFace	102
SDFace::nextFace()	108
SDFace::nextVert()	108
SDFace::prevFace()	108
SDFace::prevVert()	109
SDVertex::child	101
SDVertex::P	101
SDVertex::startFace	101

⟨*Get one ring vertices for boundary vertex*⟩≡

```
SDFace *face = startFace, *f2;
while ((f2 = face->nextFace(this)) != NULL)
    face = f2;
*P++ = face->nextVert(this)->P;
do {
    *P++ = face->prevVert(this)->P;
    face = face->prevFace(this);
} while (face != NULL);
```

For vertices on the boundary, the new vertex's position is only based on the two neighboring boundary vertices (Figure 3.16). By not depending on interior vertices, we ensure that two abutting surfaces that share the same vertices on the boundary will have abutting limit surfaces. The `weightBoundary()` utility function applies the given weighting on the two neighbor vertices v_1 and v_2 to compute the new position v' as:

$$v' = (1-2\beta)v + \beta v_1 + \beta v_2.$$

The same weight of $\frac{1}{8}$ is used for both regular and extraordinary vertices.

⟨*Apply boundary rule for even vertex*⟩≡

```
v[j]->child->P = weightBoundary(v[j], 1.f/8.f);
```

The `weightBoundary()` function applies the given weights at a boundary vertex. Because the `oneRing()` function orders the boundary vertex's one ring such that the first and last entries are the boundary neighbors, the implementation here is particularly straightforward.

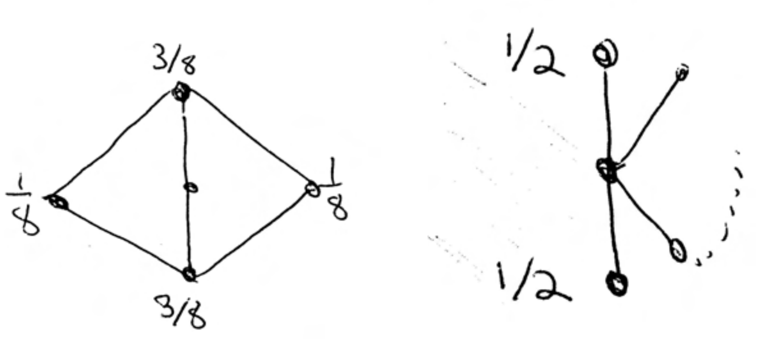


Figure 3.17: Subdivision rule for edge split: the position of the new odd vertex, **marked with an “x” (what?)**, is found by weighting the two vertices at the ends of the edge and the two vertices opposite it on the adjacent triangles. On the left are the weights for an interior vertex; on the right are the weights for a boundary vertex.

⟨*LoopSubdiv Method Definitions*⟩+≡

```
Point LoopSubdiv::weightBoundary(SDVertex *vert, Float beta) {
    ⟨Put vert one-ring in Pring⟩
    Point P = (1-2*beta) * vert->P;
    P += beta * Pring[0];
    P += beta * Pring[valence-1];
    return P;
}
```

100	LoopSubdiv
33	Point
104	SDEdge
102	SDFace
101	SDVertex
101	SDVertex::P

Now we’ll compute the positions of the odd vertices, the new vertices along the split edges of the mesh. We loop over each edge of each face in the mesh, computing the new vertex that splits the edge (Figure 3.17). For interior edges, the new vertex is found by weighting the two vertices at the ends of the edge (v_0 and v_1) and the two vertices across from the edge on the adjacent faces (v_2 and v_3). We loop through all three edges of each face, and each time we see an edge that hasn’t been seen before, we compute and store the new odd vertex in the splitEdges associative array.

⟨*Compute new odd edge vertices*⟩≡

```
map<SDEdge, SDVertex *> splitEdges;
for (u_int j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Compute odd vertex on kth edge⟩
    }
}
```

As we did when setting the face neighbor pointers in the original mesh, we create an SDEdge object for the edge and see if it is in the set of edges we’ve already visited. If it isn’t, we compute the new vertex on this edge and add it to the map. The map is an associative array structure that performs efficient lookups. **you know this is O(log n) time, not constant, right? map is implemented as a tree.**

⟨*Compute odd vertex on kth edge*⟩≡

```
SDEdge edge(face->v[k], face->v[NEXT(k)]);
SDVertex *vert = splitEdges[edge];
if (!vert) {
    ⟨Create and initialize new odd vertex⟩
    ⟨Apply edge rules to compute new vertex position⟩
    splitEdges[edge] = vert;
}
```

In Loop subdivision, the new vertices added by subdivision are always regular. This means that the proportion of extraordinary vertices to regular vertices will decrease with each level of subdivision. We can therefore immediately initialize the `regular` member of the new vertex. The `boundary` member can also be easily initialized, by checking to see if there is a neighbor face across the edge that we're splitting. Finally, we'll go ahead and set the vertex's `startFace` pointer here. For all odd vertices on the edges of a face, the center child (child face number three) is guaranteed to be adjacent to the new vertex.

⟨*Create and initialize new odd vertex*⟩≡

```
vert = new (vertexArena) SDVertex;
newVertices.push_back(vert);
vert->regular = true;
vert->boundary = (face->f[k] == NULL);
vert->startFace = face->children[3];
```

NEXT	103
SDEdge	104
SDEdge::v	104
SDFace::children	102
SDFace::f	102
SDFace::otherVert()	117
SDFace::v	102
SDVertex	101
SDVertex::boundary	101
SDVertex::P	101
SDVertex::regular	101
SDVertex::startFace	101

For odd boundary vertices, the new vertex is just the average of the two adjacent vertices. For odd interior vertices, the two vertices at the ends of the edge are given weight $\frac{3}{8}$, and the two vertices opposite the edge are given weight $\frac{1}{8}$ (Figure 3.17). These last two vertices can be found using the `SDFace::otherVert()` utility, which returns the vertex opposite a given edge of a face.

⟨*Apply edge rules to compute new vertex position*⟩≡

```
if (vert->boundary) {
    vert->P =  0.5f * edge.v[0]->P;
    vert->P += 0.5f * edge.v[1]->P;
}
else {
    vert->P =  3.f/8.f * edge.v[0]->P;
    vert->P += 3.f/8.f * edge.v[1]->P;
    vert->P += 1.f/8.f *
        face->otherVert(edge.v[0], edge.v[1])->P;
    vert->P += 1.f/8.f *
        face->f[k]->otherVert(edge.v[0], edge.v[1])->P;
}
```

The `SDFace::otherVert()` method is self-explanatory:

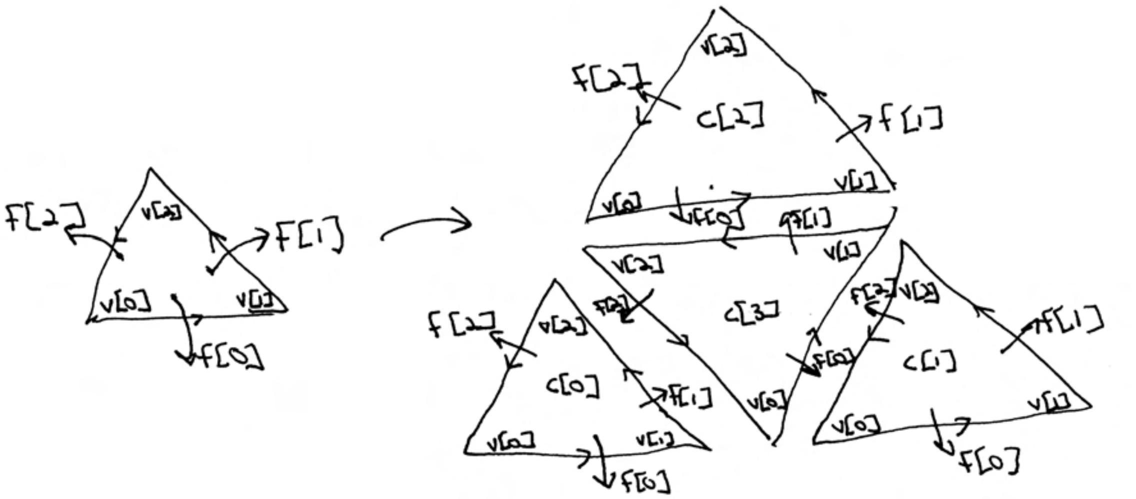


Figure 3.18: Each face is split into four child faces, such that the *i*th child is adjacent to the *i*th vertex of the original face, and such that the *i*th child face's *i*th vertex is the child of the *i*th vertex of the original face. The vertices of the center child are oriented such that the *i*th vertex is the odd vertex along the *i*th edge of the parent face.

102 SDFace::v
101 SDVertex

⟨*SDFace Methods*⟩+≡

```
SDVertex *otherVert(SDVertex *v0, SDVertex *v1) {
    for (int i = 0; i < 3; ++i)
        if (v[i] != v0 && v[i] != v1)
            return v[i];
    Severe("Basic logic error in SDVertex::otherVert()");
    return NULL;
}
```

Updating mesh topology

In order to keep the details of the topology update as straightforward as possible, the numbering scheme for the subdivided faces and their vertices has been chosen carefully–see Figure 3.18 for a summary. Review the figure carefully; these conventions are key to the next few pages.

There are four main tasks required to update the topological pointers of the refined mesh:

1. The odd vertices' `SDVertex::startFace` pointers need to store a pointer to one of their adjacent faces.

2. Similarly, the even vertices' `SDVertex::startFace` pointers must be set.

3. The new faces' neighbor `f[i]` pointers need to be set to point to the neighboring faces.

4. The new faces' `v[i]` pointers need to point to the incident vertices.

We already initialized the `startFace` pointers of the odd vertices when we first created them; we'll handle the other three tasks in order here.

⟨*Update new mesh topology*⟩≡

```
⟨Update even vertex face pointers⟩
⟨Update face neighbor pointers⟩
⟨Update face vertex pointers⟩
```

We will first set the `startFace` pointer for the children of the even vertices. If a vertex is the *i*th vertex of its `startFace`, then it is guaranteed that it will be adjacent to the *i*th child face of `startFace`. Therefore we just need to loop through all the parent vertices in the mesh, and for each one find its vertex index in its `startFace`. This index can then be used to find the child face adjacent to the new even vertex.

⟨*Update even vertex face pointers*⟩≡

```
for (u_int j = 0; j < v.size(); ++j) {
    SDVertex *vert = v[j];
    int vertNum = vert->startFace->vnum(vert);
    vert->child->startFace = vert->startFace->children[vertNum];
}
```

NEXT	103
SDFace	102
SDFace::children	102
SDFace::f	102
SDFace::vnum()	108
SDVertex	101
SDVertex::startFace	101

Next we update the face neighbor pointers for the newly-created faces. We break this into two steps: one to update neighbors among children of the same parent, and one to do neighbors across children of different parents. This involves some tricky pointer manipulation.

⟨*Update face neighbor pointers*⟩≡

```
for (u_int j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Update children f pointers for siblings⟩
        ⟨Update children f pointers for neighbor children⟩
    }
}
```

For the first step, recall that the interior child face is always stored in `children[3]`. Furthermore, the $k+1$st child face (for $k = 0,1,2$) is across the *k*th edge of the interior face, and the interior face is across the $k+1$st edge of the *k*th face.

⟨*Update children* `f` *pointers for siblings*⟩≡

```
face->children[3]->f[k] = face->children[NEXT(k)];
face->children[k]->f[NEXT(k)] = face->children[3];
```

We'll now update the childrens' face neighbor pointers that point to children of other parents. Only the first three children need to be addressed here; the interior child's neighbor pointers have already been fully initialized. Inspection of Figure 3.18 reveals that the *k*th and `PREV`(*k*)th edges of the *i*th child need to be set. To set the *k*th edge of the *k*th child, we first find the *k*th edge of the parent face, then the neighbor parent `f2` across that edge. If `f2` exists (meaning we aren't on a boundary), we find the neighbor paren't index for the vertex `v[k]`. That index is equal to the index of the neighbor child we are searching for. We then repeat this process to find the child across the `PREV`(*k*)th edge.

⟨*Update children* `f` *pointers for neighbor children*⟩≡

```
SDFace *f2 = face->f[k];
face->children[k]->f[k] =
    f2 ? f2->children[f2->vnum(face->v[k])] : NULL;
f2 = face->f[PREV(k)];
face->children[k]->f[PREV(k)] =
    f2 ? f2->children[f2->vnum(face->v[k])] : NULL;
```

Finally, we handle the fourth step in the topological updates: setting the childrens' `v[i]` vertex pointers.

⟨*Update face vertex pointers*⟩≡

```
for (u_int j = 0; j < f.size(); ++j) {
    SDFace *face = f[j];
    for (int k = 0; k < 3; ++k) {
        ⟨Update child vertex pointer to new even vertex⟩
        ⟨Update child vertex pointer to new odd vertex⟩
    }
}
```

For the kth child face (for $k = 0,1,2$), the kth vertex corresponds to the even vertex that is adjacent to it. (For the non-interior children faces, there is one even vertex and two odd vertices; for the interior child face, there are three odd vertices). We can find this vertex by following the child pointer of the parent vertex, available from the parent face.

103	`NEXT`
103	`PREV`
104	`SDEdge`
102	`SDFace`
102	`SDFace::children`
102	`SDFace::f`
102	`SDFace::v`
108	`SDFace::vnum()`
101	`SDVertex`
101	`SDVertex::child`

⟨*Update child vertex pointer to new even vertex*⟩≡

```
face->children[k]->v[k] = face->v[k]->child;
```

To update the rest of the vertex pointers, we re-use the `splitEdges` associative array to find the odd vertex for each split edge of the parent face. Three child faces have that vertex as an incident vertex. Fortunately, the vertex indices for the three faces are easily found, again based on the numbering scheme established in Figure 3.18.

⟨*Update child vertex pointer to new odd vertex*⟩≡

```
SDVertex *vert =
    splitEdges[SDEdge(face->v[k], face->v[NEXT(k)])];
face->children[k]->v[NEXT(k)] = vert;
face->children[NEXT(k)]->v[k] = vert;
face->children[3]->v[k] = vert;
```

After the geometric and topological work has been done for a subdivision step, we move the newly-created vertices and faces into the `v` and `f` arrays, deleting the old ones, since we no longer need them. We only do these deletions after the first time through the loop, however; the original faces and vertices of the control mesh are left intact.

What is going on here

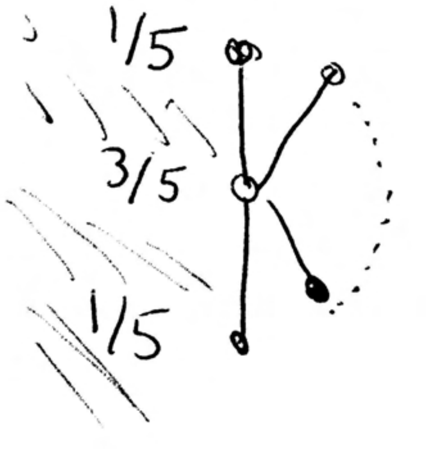


Figure 3.19: To push a boundary vertex onto the limit surface, we apply the weights shown to the vertex and its neighbors along the edge.

⟨*Prepare for next level of subdivision*⟩≡

```
#if 0
if (i != 0) {
    for (u_int j = 0; j < f.size(); ++j)
        delete f[j];
    for (u_int j = 0; j < v.size(); ++j)
        delete v[j];
}
#endif
f = newFaces;
v = newVertices;
```

To the limit surface and output

One of the remarkable properties of subdivision surfaces is that there are special subdivision rules that let us compute the positions that the vertices of the mesh would have if we continued subdividing infinitely. We apply these rules here to initialize an array of limit surface positions, `Plimit`. Note that it's important to temporarily store the limit surface positions somewhere other than in the vertices while the computation is taking place. Because the limit surface position of each vertex depends on the original positions of its surrounding vertices, the original positions of all vertices must remain unchanged until the computation is done.

The limit rule for a boundary vertex weights the two neighbor vertices by $\frac{1}{5}$ and the center vertex by $\frac{3}{5}$ (Figure 3.19 **this figure doesn't add very much**); the rule for interior vertices is based on a function `gamma()`, which computes appropriate vertex weights based on the valence of the vertex.

⟨*Push vertices to limit surface*⟩≡

```
Point *Plimit = new Point[v.size()];
for (u_int i = 0; i < v.size(); ++i) {
    if (v[i]->boundary)
        Plimit[i] = weightBoundary(v[i], 1.f/5.f);
    else
        Plimit[i] = weightOneRing(v[i], gamma(v[i]->valence()));
}
for (u_int i = 0; i < v.size(); ++i)
    v[i]->P = Plimit[i];
```

⟨*LoopSubdiv Private Methods*⟩+≡

```
static Float gamma(int valence) {
    return 1.f / (valence + 3.f / (8.f * beta(valence)));
}
```

In order to generate a smooth-looking triangle mesh with per-vertex surface normals, we'll also compute a pair of non-parallel tangent vectors at each vertex. As with the limit rule for positions, this is an analytic computation that gives the precise tangents on the actual limit surface.

⟨*Compute vertex tangents on limit surface*⟩≡

```
vector<Normal> Ns;
Ns.reserve(v.size());
for (u_int i = 0; i < v.size(); ++i) {
    SDVertex *vert = v[i];
    Vector S(0,0,0), T(0,0,0);
    ⟨Put vert one-ring in Pring⟩
    if (!vert->boundary) {
        ⟨Compute tangents of interior face⟩
    }
    else {
        ⟨Compute tangents of boundary face⟩
    }
    Ns.push_back(Normal(Cross(S, T)));
}
```

Figure 3.20 shows the setting for computing tangents in the mesh interior. The center vertex is given a weight of zero and the neighbors are given weights w_i. To compute the first tangent vector S, the weights are

$$w_i = \cos\left(\frac{2\pi i}{n}\right),$$

where n is the valence of the vertex. The second tangent T, is computed with weights

$$w_i = \sin\left(\frac{2\pi i}{n}\right).$$

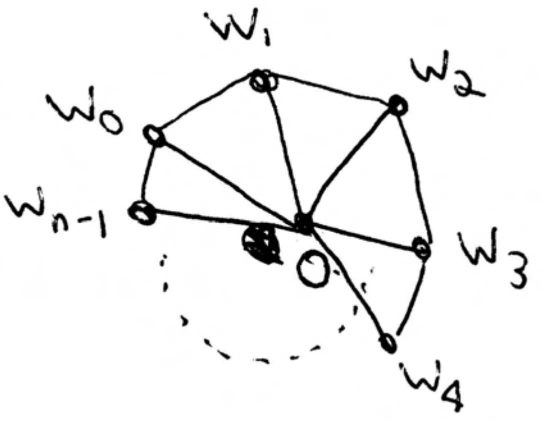


Figure 3.20: To compute tangents for interior vertices, the one-ring vertices are weighted with weights w_i. The center vertex, where the tangent is being computed, always has a weight of 0.

M_PI 678
Vector 27

⟨*Compute tangents of interior face*⟩≡

```
for (int k = 0; k < valence; ++k) {
    S += cosf(2.f*M_PI*k/valence) * Vector(Pring[k]);
    T += sinf(2.f*M_PI*k/valence) * Vector(Pring[k]);
}
```

Tangents on boundary vertices are a bit trickier; Figure 3.21 shows the ordering of vertices in the one ring expected in the discussion below.

The first tangent, known as the *across tangent*, is given by the vector between the two neighboring boundary vertices:

$$S = v_{n-1} - v_0.$$

The second tangent, known as the *transverse tangent* is computed based on the vertex's valence. The center vertex is given a weight w_c which can be zero. The one-ring vertices are given weights specified by a vector $(w_0, w_1, \ldots, w_{n-1})$. The transverse tangent rules we will use are:

valence	w_c	w_i
2	-2	(1, 1)
3	-1	(0,1,0)
4 (regular)	-2	(-1, 2, 2, -1)

For valences of 5 and higher, $w_c = 0$ and

$$\begin{aligned} w_0 &= w_{n-1} = \sin\theta \\ w_i &= (2\cos\theta - 2)\,\sin(\theta i) \end{aligned}$$

where

$$\theta = \frac{\pi}{n-1}.$$

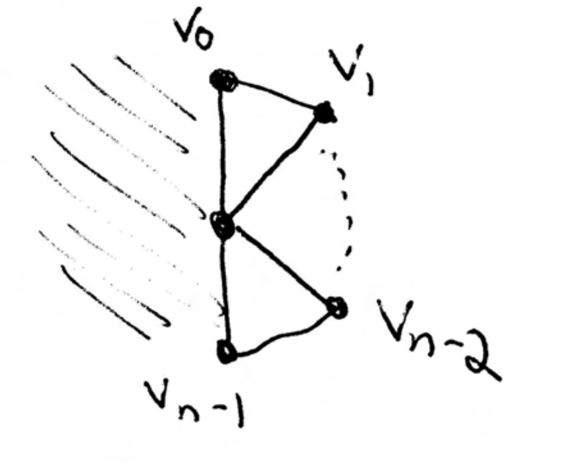


Figure 3.21: Tangents at boundary vertices are also computed as weighted averages of the adjacent vertices. However, some of the boundary tangent rules incorporate the value of the center vertex.

⟨*Compute tangents of boundary face*⟩≡

```
S = Pring[valence-1] - Pring[0];
if (valence == 2)
    T = Vector(Pring[0] + Pring[1] - 2 * vert->P);
else if (valence == 3)
    T = Pring[1] - vert->P;
else if (valence == 4) // regular
    T = Vector(-1*Pring[0] + 2*Pring[1] + 2*Pring[2] +
        -1*Pring[3] + -2*vert->P);
else {
    Float theta = M_PI / float(valence-1);
    T = Vector(sinf(theta) * (Pring[0] + Pring[valence-1]));
    for (int k = 1; k < valence-1; ++k) {
        Float wt = (2 * cosf(theta) - 2) * sinf((k) * theta);
        T += Vector(wt * Pring[k]);
    }
    T = -T;
}
```

678 M_PI
27 Vector

Finally, the fragment ⟨*Create* `TriangleMesh` *from subdivision mesh*⟩ creates the triangle mesh object and adds it to the `refined` vector passed to the `LoopSubdiv::Refine()` method. We won't include it here, since it's just a straightforward transformation of the subdivided mesh into an indexed triangle mesh.

Further Reading

Introduction to Ray Tracing has an extensive survey of algorithms for ray–shape intersection (**?**). Heckbert has written a technical report that discusses the mathematics of quadrics for graphics applications in detail, with many citations to literature in mathematics and other fields (Heckbert 1984). Hanrahan describes a system that automates the process of deriving a ray intersection routine for surfaces defined by implicit polynomials; his system emits C source code to perform the intersection test and normal computation for a surface described by a given equation (Hanrahan 1983). Other notable early papers include Kajiya's work on computing intersections with surfaces of revolution and procedurally-generated fractal terrains (Kajiya 1983) and his technique for computing intersections with parametric patches (Kajiya 1982). More recently, Stürzlinger and others have done work on more efficient techniques for direct ray intersection with patches (Stürzlinger 1998). The ray–triangle intersection test in Section 3.6 was developed by Möller and Trumbore (Möller and Trumbore 1997).

The notion of shapes that repeatedly refine themselves into collections of other shapes until ready for rendering was first introduced in the REYES renderer (Cook, Carpenter, and Catmull 1987). Pharr et al applied a similar approach to a ray tracer (Pharr, Kolb, Gershbein, and Hanrahan 1997).

An excellent introduction to differential geometry is Gray's book (Gray 1993); Section 14.3 of it presents the Weingarten equations. Turkowski's technical report has expressions for first and second derivatives of a handful of parametric primitives (Turkowski 1990a).

The Loop subdivision method was originally developed by Charles Loop (Loop 1987). Our implementation uses the improved rules for subdivision and tangents along boundary edges developed by Hoppe et al (Hoppe, DeRose, Duchamp, Halstead, Jin, McDonald, Schweitzer, and Stuetzle 1994). There has been extensive work in subdivision recently; the SIGGRAPH course notes give a good summary of the state-of-the-art and also have extensive references (Zorin, Schröder, DeRose, Kobbelt, Levin, and Sweldins 2000).

Procedural stochastic models Fournier et al (Fournier, Fussel, and Carpenter 1982).

Exercises

3.1 One nice property of mesh-based shapes like triangle meshes and subdivision surfaces is that we can transform the shape's vertices into world space, so that it isn't necessary to transform rays into object space before performing ray intersection tests. Interestingly enough, it is possible to do the same thing for ray–quadric intersections.

The implicit forms of the quadrics in this chapter were all of the form

$$Ax^2 + Bxy + Cxz + Dy^2 + Eyz + Fz^2 + G = 0,$$

where some of the constants $A \ldots G$ were zero. More generally, we can define quadric surfaces by the equation

$$Ax^2 + By^2 + Cz^2 + 2Dxy + 2Eyz + 2Fxz + 2Gz + 2Hy + 2Iz + J = 0,$$

where most of the parameters $A \ldots J$ don't directly correspond to the $A \ldots G$ above. In this form, the quadric can be represented by a 4×4 symmetric matrix Q:

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} A & D & F & G \\ D & B & E & H \\ F & E & C & I \\ G & H & I & J \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = P^T \cdot Q \cdot P = 0$$

Given this representation, first show that the matrix Q' representing a quadric transformed by the matrix M is:

$$Q' = (M^T)^{-1} Q M^{-1}.$$

To do so, show that for any point p where $p^T Q p = 0$, if we apply a transformation M to p and compute $p' = Mp$, we'd like to find Q' so that $(p')^T Q' p' = 0'$.

Next, substitute the ray equation into the more general quadric equation above to compute a, b, and c values for the quadratic equation in terms of entries of the matrix Q to pass to the `Quadratic()` function.

Now implement this approach in `lrt` and use it instead of the original quadric intersection routines. Note that you will still need to transform the resulting world-space hit points into object space to test against θ_{max}, if it is not 2π, etc. How does performance compare to the original scheme?

3.2 Improve the object-space bounding box routines for the quadrics to properly account for $\theta_{\mathrm{max}} \neq 2\pi$.

3.3 There is room to optimize the implementations of the various quadric primitives in `lrt` in a number of ways. For example, for complete spheres (i.e., not partial spheres with limited z and ϕ ranges), some of the tests in the intersection routine are unnecessary. Furthermore, many of the quadrics have excess calls to trigonometric functions that could be turned into simpler expressions using insight about the geometry of the particular primitives. Investigate ways to speed up these methods. How much does this improve the overall runtime of `lrt`?

3.4 Currently `lrt` recomputes the partial derivatives $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ for triangles every time they are needed, even though they are constant for each triangle. Precompute these vectors and analyze the speed/storage tradeoff, especially for large triangle meshes. How does the depth complexity of the scene affect this tradeoff?

3.5 Implement a general polygon primitive. `lrt` currently transforms polygons with more than three vertices into a collection of triangles by XXX. This is actually only correct for convex polygons without holes. Support all kinds of polygons as as first-class primitive. How to compute plane equation from a normal and a point on the plane.... Then intersect ray with the plane the polygon sits in. Project that point and the polygon vertices to 2D. Then apply a 2D point in polygon test; easy one is to essentially ray trace in 2D–intersect the ray with each of the edge segments, count how many it goes through. If odd number, are inside the polygon and have an intersection. Figure 3.22.

Haines has written an article that surveys a number of approaches for efficient point in polygon tests (Haines 1994); some of the techniques described there may be helpful for optimizing this test.

Schneider and Eberly discuss strategies getting all the corner cases right, e.g. for when the 2D ray is aligned precisely with an edge of the polygon (Schneider and Eberly 2003, Section XX).

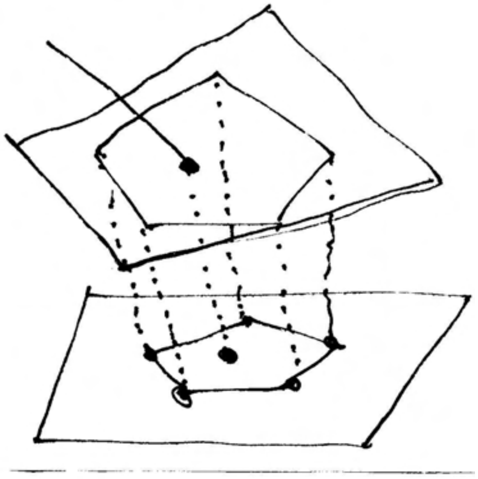


Figure 3.22: Polygon projection onto plane for intersection.

3.6 subdiv extensions: ”crease”, n integer vertices to specify chain of edges, one float, infinity, giving sharpness. for crease, use boundary subdivision rules along the edges, giving a sharp feature there.

LoopSubdiv 100
Shape 63

”hole” face property, inherit to children, just don’t output at end

3.7 Implement adaptive subdivision for the subdivision surface `Shape`. A weakness of the basic implementation is that each face is always refined a fixed number of times: this may mean that some faces are under-refined, leading to visible faceting in the triangle mesh, and some faces are over-refined, leading to excessive memory use and rendering time. Instead, stop subdividing faces once a particular error threshold has been reached.

An easy error threshold to implement computes the face normals of each face and its directly adjacent faces. If they are sufficiently close to each other (e.g. as tested via dot products), then the limit surface for that face will be reasonably flat.

The trickiest part of this exercise is that some faces that don’t need subdivision due to the flatness test will still need to be subdivided in order to provide vertices so that neighboring faces that do need to subdivide can get their vertex one-rings. In particular, adjacent faces can differ by no more than one level of subdivision.

3.8 Use the triangular face refinement infrastructure from the `LoopSubdiv` shape to implement displacement mapping. Displacement mapping is a technique related to bump mapping, where an offset function is defined over the entire surface. Rather than just adjusting the surface normal as in bump mapping, the actual surface shape is modified by displacement mapping. The usual approach to displacement mapping is to finely tessellate the geometric shape and to then evaluate the displacement function at its vertices, moving each vertex the given distance along its normal.

Because displacement mapping may make the extent of the shape larger, the bounding box of the un-displaced shape will need to be expanded by the maximum displacement distance that a particular displacement function will ever generate.

Refine each face of the mesh until, when projected onto the image, it is roughly the size of the separation between pixels. To do this, you will need to be able to estimate the image pixel-based length of an edge in the scene when it is projected onto the screen. After you have done this, use the texturing infrastructure in Chapter 11 to evaluate displacement functions.

3.9 CSG!

3.10 Ray tracing point-sampled geometry: extending methods for rendering complex models represented as a collection of point samples (Levoy and Whitted 1995; Pfister, Zwicker, van Baar, and Gross 2000; Rusinkiewicz and Levoy 2000), Schaufler and Jensen recently described a method for intersecting rays with collections of oriented point samples in space (Schaufler and Jensen 2000). They probabilisticly determine that an intersection has occurred when a ray approaches a sufficient local density of point samples and compute a surface normal with a weighted average of the nearby samples. Read their paper and extend `lrt` to supoprt a point-sampled geometry shape. Do any of `lrt`'s basic interfaces need to be extended or generalized to support a shape like this?

63 Shape
87 TriangleMesh

3.11 Ray tracing ribbons: Hair is often modeled as a collection of *generalized cylinders*, which are defined as the cylinder that results from sweeping a disk along a given curve. Because there are often a large number of individual hairs, an efficient method for intersecting rays with generalized cylinders is needed for ray tracing hair. A number of methods have been developed to compute ray intersections with generalized cylinders (Bronsvoort and Klok 1985; de Voogt, van der Helm, and Bronsvoort 2000); investigate these algorithms and extend `lrt` to support a fast hair primitive with one of them. Alternatively, investigate the generalization of Schaufler and Jensen's approach for probabilistic point intersection (Schaufler and Jensen 2000) to probabilistic line intersection and apply this to fast ray tracing of hair.

3.12 Implicit functions. More general functions, sums of them to define complex surface. Good for molecules, water drops, etc. Introduced by Blinn (Blinn 1982a). Wyvill and Wyvill give new falloff function with a number of advantages (Wyvill and Wyvill 1989). Kalra and Barr (Kalra and Barr 1989) and Hart (Hart 1996) give methods for ray tracing them.

3.13 Procedurally-described parametric surfaces: write a `Shape` that takes an expression of the form $f(u,v) \to (x,y,z)$ that describes a parametric surface as a function of (u,v) position. Evaluate the given function at a grid of (u,v) positions to create a `TriangleMesh` that approximates the given surface when the `Shape::Refine()` method is called.

3.14 Generative modeling: Snyder and Kajiya have described an elegant mathematical framework for procedurally-described geometric shapes (Snyder and

Kajiya 1992; Snyder 1992); investigate this approach and apply it to procedural shape description in `lrt`.

3.15 L-systems: A very successful technique for procedurally describing plants was first introduced to graphics by Alvy Ray Smith (Smith 1984), who applied *Lindenmayer systems* (l-systems) to describing branching plant structures. L-systems describe the branching structure of these types of shapes via a grammar. Prusinkiewicz and collaborators have generalized this approach to encompass **XXX (Prusinkiewicz, Mündermann, Karwowski, and Lane 2001; Deussen, Hanrahan, Lintermann, Mech, Pharr, and Prusinkiewicz 1998; Prusinkiewicz, James, and Mech 1994; Prusinkiewicz 1986).**

4. Primitives and Intersection Acceleration

135	Aggregate
132	GeometricPrimitive
134	InstancePrimitive
375	Material
130	Primitive
63	Shape
43	Transform

The classes described in the last chapter focus exclusively on representing geometric properties of 3D objects. Although the `Shape` class is a convenient abstraction for geometric operations such as intersection and bounding, it is insufficient for direct use in a rendering system. To construct a scene, we must be able to place individual primitives at specific locations in world coordinates. In addition, we need to bind material properties to each primitive to we can specify their appearance. To accomplish these goals, we introduce the `Primitive` class, and provide three separate implementations.

Shapes to be rendered directly are represented by the `GeometricPrimitive` class. This class, in addition to placing the shape within the scene, also contains a description of the shape's appearance properties. So that the geometric and shading portions of `lrt` can be cleanly separated, these appearance properties are encapsulated in the `Material` class, which is described in chapter 10.

Some scenes contain many instances of the same geometry at different locations. Direct support for instancing can greatly reduce the memory requirements for such scenes, since we only need to store a pointer to the geometry for each primitive. `lrt` provides the `InstancePrimitive` class for this task; each `InstancePrimitive` has a separate `Transform` to place it in the scene, but can share geometry with other `InstancePrimitives`. This allows us to render extremely complex scenes such as the one in figure **ecosystem**.

Finally, we provide the `Aggregate` class, which can hold many `Primitives`. Although this can just be a convenient way to group geometry, `lrt` uses this class to implement *acceleration structures*, which are techniques for avoiding the $O(n)$

linear complexity of testing a ray against all *n* objects in a scene. Since a ray through a scene will typically only intersect a handful of the primitives and will be nowhere near most of the others, there is substantial room for improvement compared to naively performing a ray intersection test with each primitive. Another benefit to re-using the `Primitive` interface for these acceleration structures is that `lrt` can support hybrid approaches where an accelerator of one type holds accelerators of another types. This chapter will describe the implementation of two accelerators, one (`GridAccel`) based on overlaying a uniform grid over the scene, and the other (`KdTreeAccel`) based on recursive spatial subdivision.

4.1 Geometric Primitives

The abstract `Primitive` base class is really the bridge between the geometry processing and shading subsystems of `lrt`. In order to avoid complex logic about when `Primitive`s can be destroyed, it inherits from the `ReferenceCounted` base class, which automatically tracks how many references there are to an object, freeing its storage when the last reference goes out of scope. Rather than storing pointers to these primitives, holding a `Reference<Primitive>` ensures that the reference counts are computed correctly. The `Reference` class otherwise behaves as if it was a pointer to a `Primitive`.

⟨*Primitive Declarations*⟩≡

```
class  Primitive : public ReferenceCounted {
public:
    ⟨Primitive Interface⟩
};
```

BBox	38
DifferentialGeometry	58
GridAccel	139
Intersection	131
KdTreeAccel	154
Reference	664
ReferenceCounted	663
Shape	63

Because the `Primitive` class connects geometry and shading, its interface contains methods related to both. There are five geometric routines, of which all are similar to a corresponding `Shape` method. The first, `Primitive::WorldBound()`, returns a box that encloses the primitive's geometry in world space. There are many uses for such a bound; we use it to place the `Primitive` in the acceleration data structures.

⟨*Primitive Interface*⟩+≡

```
virtual BBox WorldBound() const = 0;
```

Similarly to the `Shape` class, all primitives must be able to either determine if a given ray intersects their geometry, or else refine themselves into one or more new primitives. Like the `Shape` interface, we provide the `Primitive::CanIntersect()` method so `lrt` can determine whether the underlying geometry is intersectable or not.

One difference from the `Shape` interface is that the `Primitive` intersection methods return `Intersection` structures rather than `DifferentialGeometry`. These `Intersection` structures hold mor information about the intersection than just the local coordinate frame, such as a pointer to the material properties at the hit point.

Another difference is that `Shape::Intersect()` returns the parametric distance along the ray to the intersection in a `Float *` output variable, while `Primitive::Intersect()` is responsible for updating `Ray::maxt` with this value if an intersection is found.

In this way, the geometric routines from the last chapter do not need to know how the parametric distance will be used by the rest of the system.

⟨*Primitive Interface*⟩+≡

```
virtual bool CanIntersect() const;
virtual bool Intersect(const Ray &r, Intersection *in) const = 0;
virtual bool IntersectP(const Ray &r) const = 0;
virtual void Refine(vector<Reference<Primitive> > &refined) const;
```

The `Intersection` structure holds information about a ray–primitive intersection, including information about the differential geometry of the point on the surface, and a pointer to the `Primitive` that the ray hit, and its world to object space transformation.

⟨*Primitive Declarations*⟩+≡

```
struct Intersection {
    ⟨Intersection Public Methods⟩
    DifferentialGeometry dg;
    const Primitive *primitive;
    Transform WorldToObject;
};
```

It may be necessary to repeatedly refine a primitive until all of the primitives it has returned are themselves intersectable. The `Primitive::FullyRefine()` utility method handles this task. Its implementation is straightforward; we maintain a queue of primitives to be refined (called `todo` in the code below), and invoke the `Primitive::Refine()` method repeatedly on entries in that queue. Intersectable `Primitive`s returned by `Primitive::Refine()` are placed on the output queue, while non-intersectable ones are placed on the `todo` list.

491	`AreaLight`
58	`DifferentialGeometry`
130	`Primitive`
36	`Ray`
664	`Reference`
43	`Transform`
658	`vector`

⟨*Primitive Interface*⟩+≡

```
void FullyRefine(vector<Reference<Primitive> > &refined) const;
```

⟨*Primitive Method Definitions*⟩+≡

```
void Primitive::FullyRefine(
        vector<Reference<Primitive> > &refined) const {
    vector<Reference<Primitive> > todo;
    todo.push_back(const_cast<Primitive *>(this));
    while (todo.size()) {
        ⟨Refine last primitive in todo list⟩
    }
}
```

⟨*Refine last primitive in todo list*⟩≡

```
Reference<Primitive> prim = todo.back();
todo.pop_back();
if (prim->CanIntersect())
    refined.push_back(prim);
else
    prim->Refine(todo);
```

In addition to the geometric methods, a `Primitive` object has two methods related to their material properties. The first, `Primitive::GetAreaLight()`, returns a pointer to the `AreaLight` that describes the primitive emission distribution, if the primitive is itself a light source. If the primitive is *not* emissive, this method returns NULL.

The second method, `Primitive::GetBSDF()`, returns a representation of the light scattering properties of the material at the given point on the surface in a `BSDF` object. In addition to the differential geometry at the hit point, it takes the world to object space transformation as a parameter. This information will be required by the `InstancePrimitive` class, described later in this chapter.

⟨*Primitive Interface*⟩+≡

```
virtual const AreaLight *GetAreaLight() const = 0;
virtual BSDF *GetBSDF(const DifferentialGeometry &dg,
    const Transform &WorldToObject) const = 0;
```

Whoa – this should be somewhere else, like in the lighting chapter

Given the `Primitive::GetAreaLight()` method, we will add a method to the `Intersection` class that makes it easy to compute the emitted radiance at a surface point.

⟨*Intersection Method Definitions*⟩≡

```
Spectrum Intersection::Le(const Vector &w) const {
    const AreaLight *area = primitive->GetAreaLight();
    return area ? area->L(dg.p, dg.nn, w) : Spectrum(0.);
}
```

AreaLight	491
AreaLight::L()	492
BSDF	370
DifferentialGeometry	58
DifferentialGeometry::p	58
InstancePrimitive	134
Intersection	131
Intersection::dg	131
Material	375
Primitive	130
Reference	664
Shape	63
Spectrum	181
Transform	43
Vector	27

4.1.1 Geometric Primitive

The `GeometricPrimitive` class represents a single shape (e.g. a sphere) in the scene. One `GeometricPrimitive` is allocated for each shape in the scene description provided by the user.

⟨*Primitive Declarations*⟩+≡

```
class  GeometricPrimitive : public Primitive {
public:
    ⟨GeometricPrimitive Public Methods⟩
private:
    ⟨GeometricPrimitive Private Data⟩
};
```

Each `GeometricPrimitive` holds a reference to a `Shape` and its `Material`. In addition, because primitives in `lrt` may be area light sources, we store a pointer to an `AreaLight` object that describes its emission characteristics (this pointer is set to `NULL` if the primitive does not emit light).

⟨*GeometricPrimitive Private Data*⟩≡

```
Reference<Shape> shape;
Reference<Material> material;
AreaLight *areaLight;
```

The `GeometricPrimitive` constructor just initializes these variables from the parameters passed to it; its implementation is omitted.

⟨*GeometricPrimitive Public Methods*⟩+≡

```
GeometricPrimitive(const Reference<Shape> &s, const Reference<Material> &m,
    AreaLight *a);
```

Most of the methods of the `Primitive` interface related to geometric processing are simply forwarded the corresponding `Shape` method. For example, `GeometricPrimitive::Intersect()` calls the `Shape::Intersect()` method of its enclosed `Shape` to do the actual geometric intersection, and initializes an `Intersection` object to describe the hit found, if any. We also use the returned parametric hit distance to update the `Ray::maxt` member. The primary advantage of storing the distance to the closest hit in `Ray::maxt` is that we may be able to quickly reject any intersections that lie farther along the ray than any already found.

⟨*GeometricPrimitive Method Definitions*⟩+≡

```
bool GeometricPrimitive::Intersect(const Ray &r,
        Intersection *isect) const {
    Float thit;
    if (shape->Intersect(r, &thit, &isect->dg)) {
        isect->primitive = this;
        isect->WorldToObject = shape->WorldToObject;
        r.maxt = thit;
        return true;
    }
    return false;
}
```

We won't include the implementations of `GeometricPrimitive::WorldBound()`, `GeometricPrimitive::IntersectP()`, `GeometricPrimitive::CanIntersect()`, or `GeometricPrimitive::Refine()` here; they just forward these requests on to the `Shape` in a similar manner.

`GeometricPrimitive::GetAreaLight()` just returns the `GeometricPrimitive::areaLight` member. `GeoemtricPrimitive::GetBSDF()` is implemented in Section 10.2, after the `Texture` `BSDF` classes have been described.

135	`Aggregate`
491	`AreaLight`
370	`BSDF`
132	`GeometricPrimitive`
134	`InstancePrimitive`
131	`Intersection`
131	`Intersection::dg`
131	`Intersection::primitive`
131	`Intersection::WorldToObject`
375	`Material`
130	`Primitive`
36	`Ray`
36	`Ray::maxt`
664	`Reference`
63	`Shape`
64	`Shape::WorldToObject`
394	`Texture`

4.1.2 Object Instancing

Object instancing is a classic technique in rendering that re-uses multiple transformed copies of a single collection of geometry at multiple positions in a scene. For example, in a model of a concert hall with thousands of identical seats, the scene description can be effectively compressed by a large amount if all of the seats refer to a shared geometric representation of a single seat. The ecosystem scene in Figure **??** has over four thousand individual plants of various types, though only sixty one unique plant models. Because each plant model is instanced multiple times, the complete scene has 19.5 million triangles total, though only 1.1 million triangles are stored in memory. Thanks to primitive reuse though object instancing, `lrt` uses only approximately 300 MB for rendering this scene.

Object instancing is handled by the `InstancePrimitive` class. It takes a reference to the shared `Primitive` that represents the instanced model, and the instance-to-world-space transformation that places it in the scene. If the geometry to be instanced is contained in multiple `Primitives`, the calling code responsible for placing them in an `Aggregate` class.

The `InstancePrimitive` similarly requries that the primitive be intersectable; it would be a waste of time and memory for all of the instances to individually refine the primitive. **Seems like there should be some mechanism for hiding this cleanly, but I'm not sure. It's annoying that we can't do lazy refinement just because there are multiple instances of something. What if they're all hidden?** (See the `lrtObjectInstance()` function in Appendix B.3.5 for the code that creates instances based on the scene description file, refining and creating aggregates as described here.)

⟨*Primitive Declarations*⟩+≡

```
class  InstancePrimitive : public Primitive {
public:
    ⟨InstancePrimitive Public Methods⟩
private:
    ⟨InstancePrimitive Private Data⟩
};
```

⟨*InstancePrimitive Public Methods*⟩≡

```
InstancePrimitive(Reference<Primitive> &i, const Transform &i2w) {
    instance = i;
    InstanceToWorld = i2w;
    WorldToInstance = i2w.GetInverse();
}
```

⟨*InstancePrimitive Private Data*⟩≡

```
Reference<Primitive> instance;
Transform InstanceToWorld, WorldToInstance;
```

ePrimitive::IntersectP()	134
Intersection	131
ersection::WorldToObject	131
Primitive	130
Primitive::Intersect()	131
Ray	36
Ray::maxt	36
Reference	664
Transform	43

need a clearer description of "instance space". I think this paragraph could use more work. The `InstancePrimitive::Intersect()` and `InstancePrimitive::IntersectP()` methods transform the ray from world space to instance space before passing it along to the primitive. If an intersection is found, the routines transform the returned differential geometry back out into "true" world space and updates the `Intersection::WorldToObject` transformation to be the correct full transformation from world space to object space. This way, the instanced primitive in unaware that its concept of "world space" is actually not the real scene world space; the `InstancePrimitive` does the necessary work so that instances behave as expected.

⟨*InstancePrimitive Method Definitions*⟩≡

```
bool InstancePrimitive::Intersect(const Ray &r, Intersection *isect) const {
    Ray ray = WorldToInstance(r);
    if (instance->Intersect(ray, isect)) {
        r.maxt = ray.maxt;
        isect->WorldToObject = isect->WorldToObject *
            WorldToInstance;
        ⟨Transform instance's differential geometry to world space⟩
        return true;
    }
    return false;
}
```

⟨*InstancePrimitive Public Methods*⟩+≡

```
BBox WorldBound() const { return InstanceToWorld(instance->WorldBound()); }
```

Finally, the InstancePrimitive::GetAreaLight() and InstancePrimitive::GetBSDF() methods should never be called; these methods in the primitive that the ray actually hit will be called instead. Their implementations (not shown here) simply result in a runtime error.

4.2 Aggregates

Only the most trivial ray tracing systems do not contain some sort of acceleration structure. Without one, tracing a ray through a scene would take $O(n)$ time, since the ray would need to be tested against each primitive in turn, looking for the closest intersection. However, it most scenes, this is extremely wasteful, since the ray passes nowhere near the vast majority of primitives. The goal of acceleration structures is to allow the quick, simultaneous rejection of *groups* of primitives, and also to order the search process so that nearby intersections are likely to be found first.

The Aggregate class provides an interface for grouping multiple Primitive objects together. Because Aggregates themselves support the Primitive interface, no special support is required elsewhere in lrt for acceleration. In fact, by implementing acceleration in this way, it is easy to experiment with new acceleration techniques by simply adding a new Aggregate primitive to lrt.

Like InstancePrimitives, the Aggregate intersection routines set the Intersection::primitive pointer to the primitive that the ray actually hit, not the aggregate that holds the primitive. Because lrt will use this pointer to obtain information about the primitive being hit (its reflection and emission properties), the Aggregate::GetAreaLight() and Aggregate::GetBSDF() methods should never be called, so those methods (not shown here) will simply cause a runtime error.

38	BBox
139	GridAccel
134	InstancePrimitive
135	InstancePrimitive::GetAreaLight(
135	InstancePrimitive::GetBSDF()
134	InstancePrimitive::instance
134	InstancePrimitive::InstanceToWor
154	KdTreeAccel
130	Primitive
130	Primitive::WorldBound()

have some brief discussion of object subdivision (e.g. HBV) versus spatial subdivision approaches

Also mention general trade-off of time spent builting the hierarchy to improve its quality versus number of ray intersection tests.

⟨*Primitive Declarations*⟩+≡

```
class  Aggregate : public Primitive {
public:
    ⟨Aggregate Public Methods⟩
};
```

4.2.1 Ray–Box Intersections

Both the GridAccel and the KdTreeAccel in the next two sections store a BBox that surrounds all of their primitives. This box can be used to quickly determine if a ray doesn't intersect any of the primitives; if the ray misses the box, it also must miss all of the primitives inside it. Furthermore, both of these accelerators use the point at which the ray enters the bounding box and the point at which it exits as part of the input to their traversal algorithms.

Therefore, we will add a BBox method, BBox::IntersectP(), that checks for a ray–box intersection and returns the two parametric t values of the intersection if there is one. Note that BBox is not a Primitive, which is a benefit here, since we want *two t* values from its IntersectP() method instead of none.

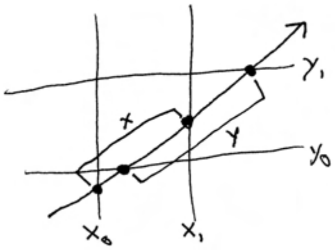


Figure 4.1: Intersecting a ray with an axis-aligned bounding box: we compute intersection points with each pair of slabs in turn, progressively narrowing the parametric interval. Here in 2D, the intersection of the *x* and *y* extents along the ray gives the extent where the ray is inside the box.

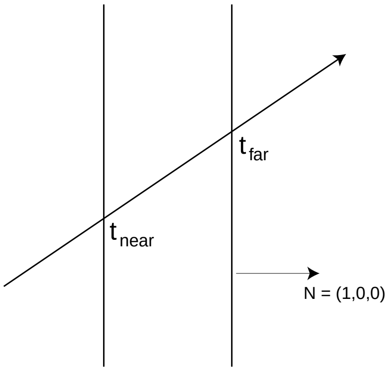


Figure 4.2: Intersecting a ray with a pair of axis-aligned slabs: the two slabs shown here are planes described by $x = c$, for some constant value c. The normal of each slab is $(1, 0, 0)$.

BBox 38

Finding these intersections is fairly simple. One way to think of bounding boxes is as the intersection of three slabs, where a slab is simply the region of space between two parallel planes. To intersect a ray against a box, we intersect the ray against each of the box's three slabs in turn. Because the slabs are aligned with the three coordinate axes, a number of optimizations can be made in the ray–slab tests.

The basic ray–bounding box intersection algorithm works as follows: we start with a parametric interval that covers that range of positions *t* along the ray where we're interested in finding intersections; typically, this is $[0, \infty)$. We will then successively compute the two parametric *t* positions where the ray intersects each pair of axis-aligned slabs. We compute the set-intersection of the per-slab intersection interval with our `BBox` intersection interval, returning failure if we find that the resulting interval is degenerate. If, after checking all three slabs, the interval is non-degenerate, we have the parametric range of the ray that is inside the box. Figure 4.1 illustrates this process.

Finding the intersection of a ray with an axis-aligned plane only requires a few computations; see the discussion of ray–disk intersections in Section 3.4.3 for a review of this process. Figure 4.2 shows the basic geometry of a ray and a pair of slabs.

If the `BBox::IntersectP()` method returns true, the intersection's parametric range is returned in the optional arguments `hitt0` and `hitt1`. Intersections outside of the `Ray::mint`/`Ray::maxt` range of the ray are ignored.

⟨*BBox Method Definitions*⟩+≡

```
bool BBox::IntersectP(const Ray &ray, Float *hitt0,
        Float *hitt1) const {
    Float t0 = ray.mint, t1 = ray.maxt;
    for (int i = 0; i < 3; ++i) {
        ⟨Update interval for ith bounding box slab⟩
    }
    if (hitt0) *hitt0 = t0;
    if (hitt1) *hitt1 = t1;
    return true;
}
```

For each pair of slabs, this routine needs to compute two ray–plane intersections, giving the parametric t values where the intersections occur. Consider the pair of slabs along the x axis: they are can be described by the two planes through the points $(x_1, 0, 0)$ and $(x_2, 0, 0)$, each with normal $(1, 0, 0)$. There are two t values to compute, one for each plane. Consider the first one, t_1. From the ray–plane intersection equation, we have:

$$t_1 = -\frac{((\mathrm{o}(\mathrm{r}) - (x_1, 0, 0)) \cdot (1, 0, 0))}{(\mathbf{d}(\mathrm{r}) \cdot (1, 0, 0))}$$

Because the y and z components of the normal are zero, we can use the definition of the dot product to simplify this substantially:

$$t_1 = -\frac{\mathrm{o}(\mathrm{r})_x - x_1}{\mathbf{d}(\mathrm{r})_x} = \frac{x_1 - \mathrm{o}(\mathrm{r})_x}{\mathbf{d}(\mathrm{r})_x}$$

38	`BBox`
39	`BBox::pMax`
39	`BBox::pMin`
36	`Ray`
35	`Ray::d`
36	`Ray::maxt`
36	`Ray::mint`
35	`Ray::o`

The code to compute these values starts by computing the reciprocal of the corresponding component of the ray direction so that it can multiply by this factor instead of performing multiple expensive divisions. Note that although we are dividing by this component, it is not necessary to verify that it is non-zero! If it is, then `invRayDir` will hold an infinite value, either $-\infty$ or ∞, and the rest of the algorithm still works correctly.[1]

⟨*Update interval for* `i`*th bounding box slab*⟩≡

```
Float invRayDir = 1.f / ray.d[i];
Float tNear = (pMin[i] - ray.o[i]) * invRayDir;
Float tFar  = (pMax[i] - ray.o[i]) * invRayDir;
⟨Update parametric interval from slab intersection ts⟩
```

The two distances are reordered so that t_{near} holds the closer intersection and t_{far} the farther one. This gives a parametric range $[t_{\mathrm{near}}, t_{\mathrm{far}}]$ which is used to compute the set intersection with the current range $[t_0, t_1]$ to compute a new range. If this new

[1] This assumes that the architecture being used supports IEEE floating-point arithmetic, which is universal on modern systems. The relevant properties of IEEE floating-point arithmetic are that for all $v > 0$, $v/0 = \infty$ and for all $w < 0$, $w/0 = -\infty$, where ∞ is a special value such that any positive number multiplied by ∞ gives ∞, any negative number multiplied by ∞ gives $-\infty$, etc. See **XXX** for information about IEEE floating point.

We seem to be missing this figure.

Figure 4.3: The grid setting caption.

range is empty (i.e. $t_0 > t_1$), then the code can immediately return failure. There is another floating-point related subtlety here, pointed out to us by Evan Parker: in the case where the ray origin is in the plane of one of the bounding box slabs and the ray lies in the plane of the slab, it is possible that `tNear` or `tFar` will be computed by an expression of the form $0/0$, which results in a IEEE floating-point "not a number" (NaN) value. Like infinity values, NaNs are have well-specified semantics: for example, any logical comparison involving a nan always evaluates to false. Therefore, the code that updates the values of `t0` and `t1` was carefully written so that if `tNear` or `tFar` is NaN, then `t0` or `t1` won't ever take on a NaN value but will always remain unchanged. When we first wrote this code, we wrote `t0 = max(t0, tNear)`, which might assign NaN to `t0` depending on how `max()` was implemented.

⟨*Update parametric interval from slab intersection ts*⟩≡

```
  if (tNear > tFar) swap(tNear, tFar);
  t0 = tNear > t0 ? tNear : t0;
  t1 = tFar  < t1 ? tFar  : t1;
  if (t0 > t1) return false;
```

GridAccel 139
KdTreeAccel 154

4.3 Grid Accelerator

The `GridAccel` class is an accelerator that divides an axis-aligned region of space into equal-sized chunks (called "voxels"). Each voxel stores references to the primitives that overlap it (see Figure 4.3). Given a ray, it steps through each of the voxels that the ray passes through in order, checking for intersections with only the primitives in each voxel Useless ray intersection tests are reduced substantially because primitives far away from the ray aren't considered at all. Furthermore, because the voxels are considered from near to far along the ray, it is possible to stop performing intersection tests once we have found an intersection and know that it is not possible for any closer intersections to exist.

The `GridAccel` structure can be initialized quickly, and it takes only a simple computation to determine the sequence of voxels through which a given ray will pass. However, this simplicity is a doubled-edged sword; `GridAccel` can suffer from poor performance when the data in the scene aren't distributed evenly throughout space. If there's a small region of space with a lot of geometry in it, all that geometry might fall in a single voxel, and performance will suffer when a ray passes through that voxel, as many intersection tests will be performed. This is sometimes referred to as the "teapot in a stadium" problem. The basic problem is that the data structure cannot adapt well to the distribution of the data: if we use a very fine grid we spend too much time stepping through empty space, and if our grid is too coarse we gain little benefit from the grid at all. The `KdTreeAccel` in the next section adapts to the distribution of geometry such that it doesn't suffer from this problem.

we're not consistent about saying stuff like this. `lrt`'s grid accelerator is defined in `accelerators/grid.cpp`.

```
⟨GridAccel Declarations⟩≡
  class  GridAccel : public Aggregate {
  public:
      ⟨GridAccel Public Methods⟩
  private:
      ⟨GridAccel Private Public Methods⟩
      ⟨GridAccel Private Data⟩
  };
```

4.3.1 Creation

The `GridAccel` constructor takes a vector of `Primitive`s to be stored in the grid. It automatically determines the number of voxels to store in the grid based on the number of primitives.

One factor that adds to the complexity of the grid's implementation is the fact that some of these primitives may not be directly intersectable (they may return `false` from `Primitive::CanIntersect()`), and need to refine themselves into sub-primitives before intersection tests can be performed. This is a problem because when we building the grid, we might have a scene with a single primitive in it and choose to build a coarse grid with few voxels. However, if the primitive is later refined for intersection tests, it might turn into millions of primitives and the original grid resolution would be far too small to efficiently find intersections. `lrt` addresses this problem in one of two ways:

```
135 Aggregate
130 Primitive
664 Reference
658 vector
```

- If the `refineImmediately` flag to the constructor is `true`, all of the `Primitive`s are refined until they have turned into intersectable primitives. This may waste time and memory for scenes where some of the primitives wouldn't have ever been refined since no rays approached them.
- Otherwise, primitives are refined only when a ray enters one of the voxels they are stored in. If they create multiple `Primitive`s when refined, the new primitives are stored in a new instance of a `GridAccel` that replaces the original `Primitive` in the top-level grid. This allows us to handle primitive refinement without needing to re-build the entire grid each time another primitive is refined. We keep track of whether a grid was constructed explicitly by `lrt` or implicitly by a `Refine()` method for bookkeeping purposes.

```
⟨GridAccel Method Definitions⟩≡
  GridAccel::GridAccel(const vector<Reference<Primitive> > &p,
          bool forRefined, bool refineImmediately)
      : gridForRefined(forRefined) {
      ⟨Initialize prims with primitives for grid⟩
      ⟨Initialize mailboxes for grid⟩
      ⟨Compute bounds and choose grid resolution⟩
      ⟨Compute voxel widths and allocate voxels⟩
      ⟨Add primitives to grid voxels⟩
  }
```

We seem to be missing this figure.

Figure 4.4: This is why we need mailboxing.

⟨*GridAccel Private Data*⟩≡

```
bool gridForRefined;
```

First, the constructor determines the final set of `Primitive`s to store in the grid, either directly using the primitives passed in or refining all of them until they are intersectable.

⟨*Initialize* `prims` *with primitives for grid*⟩≡

```
vector<Reference<Primitive> > prims;
if (refineImmediately)
    for (u_int i = 0; i < p.size(); ++i)
        p[i]->FullyRefine(prims);
else
    prims = p;
```

Because primitives may overlap multiple grid voxels, there is the possibility that a ray will be tested multiple times against the same primitive as it passes through those voxels (Figure 4.4). A technique called *mailboxing* makes it possible to quickly determine if a ray has already been tested against a particular primitive, so these extra tests can be avoided. In this technique, each ray is assigned a unique integer id. The id of the most recent ray that was tested against that primitive is stored along with the primitive itself. As the ray passes through voxels in the grid, the ray's id is compared with the primitives' ids–if they are different, the ray–primitive intersection test is performed and the primitive's id is updated to match the ray's. If the ray encounters the same primitive in later voxels, the ids will match and the test is trivially skipped.[2]

`AllocAligned()`	667
`GridAccel`	139
`MailboxPrim`	141
`Primitive`	130
`Primitive::FullyRefine()`	131
`Reference`	664
`vector`	658

The `GridAccel` constructor creates a `MailboxPrim` structure for each primitive. Grid voxels store pointers to the `MailboxPrim`s of the primitives that overlap them. The `MailboxPrim` stores both a reference to the primitive as well as the integer tag that identifies the last ray that was tested against it. All of the mailboxes are allocated in a single contiguous cache-aligned block for improved memory performance.

⟨*Initialize mailboxes for grid*⟩≡

```
nMailboxes = prims.size();
mailboxes = (MailboxPrim *)AllocAligned(nMailboxes *
    sizeof(MailboxPrim));
for (u_int i = 0; i < nMailboxes; ++i)
    new (&mailboxes[i]) MailboxPrim(prims[i]);
```

[2] This approach depends on the fact that the grid finds the intersection for a ray and returns before any other rays are passed to `GridAccel::Intersect()`; if this was not the case, the grid would still find the right ray–primitive intersections, though unecessary tests might be performed as multiple rays overwrote the mailbox ids in primitives that they passed by. In particular, if `lrt` was multi-threaded, the mailboxing scheme would need to be revisited as rays from different threads would sometimes be passing through the grid simultaneously. In general, parallel raytracing makes mailboxing much more complicated.

⟨*MailboxPrim Declarations*⟩≡

```
struct MailboxPrim {
    MailboxPrim(const Reference<Primitive> &p) {
        primitive = p;
        lastMailboxId = -1;
    }
    Reference<Primitive> primitive;
    int lastMailboxId;
};
```

⟨*GridAccel Private Data*⟩+≡

```
u_int nMailboxes;
MailboxPrim *mailboxes;
```

After the overall bounds have been computed, the grid needs to determine how many voxels to create along each of the *x*, *y*, and *z* axes. The `voxelsPerUnitDist` variable is set in the fragment below, giving the average number of voxels that should be created per unit distance in each of the three directions. Given that value, multiplication by the grid's extent in each direction gives the number of voxels to make. We cap the number of voxels in any direction to 64, to avoid creating enormous data structures for complex scenes.

⟨*Compute bounds and choose grid resolution*⟩≡

```
for (u_int i = 0; i < prims.size(); ++i)
    bounds = Union(bounds, prims[i]->WorldBound());
Vector delta = bounds.pMax - bounds.pMin;
⟨Find voxelsPerUnitDist for grid⟩
for (int axis = 0; axis < 3; ++axis) {
    NVoxels[axis] = Round2Int(delta[axis] * voxelsPerUnitDist);
    NVoxels[axis] = Clamp(NVoxels[axis], 1, 64);
}
```

38	`BBox`
39	`BBox::pMax`
39	`BBox::pMin`
677	`Clamp()`
130	`Primitive`
130	`Primitive::WorldBound()`
664	`Reference`
40	`Union()`
27	`Vector`

⟨*GridAccel Private Data*⟩+≡

```
int NVoxels[3];
BBox bounds;
```

As a first approximation to choosing a grid size, the total number of voxels should be be roughly proportional to the total number of primitives; if the primitives were uniformly distributed, this would mean that a constant number of primitives were in each voxel. Though the primitives won't be uniformly distributed in general, this is a reasonable approximation. While increasing the number of voxels improves efficiency by reducing the average number of primitives per voxel (and thus reducing the number of ray–object intersection tests that need to be performed), doing so also increases memory use, hurts cache performance, and increases the time spent tracing the ray's path through the greater number of voxels it overlaps. On the other hand, too few voxels obviously leads to poor performance, due to an increased number of ray–primitive intersections tests to be performed.

Given the goal of having the number of voxels be proportional to the number of primitives, the cube root of the number of objects is an appropriate starting point for the grid resolution in each direction. In practice, this value is typically scaled by an empirically-chosen factor; in `lrt` we use a scale of three. Whichever of the x, y or z dimensions has the largest extent will have exactly $3\sqrt[3]{N}$ voxels for a scene with N primitives. The number of voxels in the other two directions are set in an effort to create voxels that are as close to regular cubes as possible. The `voxelsPerUnitDist` variable is the foundation of these computations; it gives the is the number of voxels to create per unit distance. Its value is set such that `cubeRoot` voxels will be created along the axis with the largest extent.

⟨*Find* `voxelsPerUnitDist` *for grid*⟩≡

```
int maxAxis = bounds.MaximumExtent();
Float invMaxWidth = 1.f / delta[maxAxis];
Float cubeRoot = 3.f * powf(Float(prims.size()), 1.f/3.f);
Float voxelsPerUnitDist = cubeRoot * invMaxWidth;
```

Given the number of voxels in each dimension, the constructor next sets the `GridAccel::Width` vector, which holds the world-space widths of the voxels in each direction. It also precomputes the `GridAccel::InvWidth` values, so that routines that would otherwise divide by the `Width` value can perform a multiplication rather than dividing. Finally, it allocates an array of pointers to `Voxel` structures for each of the voxels in the grid. These pointers are set to `NULL` initially and will only be allocated for any voxel with one or more overlapping primitives.[3]

`AllocAligned()`	667
`BBox::MaximumExtent()`	41
`GridAccel`	139
`GridAccel::NVoxels`	141
`MailboxPrim`	141
`Vector`	27
`Voxel`	144

⟨*Compute voxel widths and allocate voxels*⟩≡

```
for (int axis = 0; axis < 3; ++axis) {
    Width[axis] = delta[axis] / NVoxels[axis];
    InvWidth[axis] = (Width[axis] == 0.f) ? 0.f : 1.f / Width[axis];
}
int nVoxels = NVoxels[0] * NVoxels[1] * NVoxels[2];
voxels = (Voxel **)AllocAligned(nVoxels * sizeof(Voxel *));
memset(voxels, 0, nVoxels * sizeof(Voxel *));
```

⟨*GridAccel Private Data*⟩+≡

```
Vector Width, InvWidth;
Voxel **voxels;
```

Once the voxels themselves have been allocated, primitives can be added to the voxels that they overlap. The `GridAccel` constructor adds each primitive's corresponding `MailboxPrim` to the voxels that its bounding box overlaps.

⟨*Add primitives to grid voxels*⟩≡

```
for (u_int i = 0; i < prims.size(); ++i) {
    ⟨Find voxel extent of primitive⟩
    ⟨Add primitive to overlapping voxels⟩
}
```

First, the world-space bounds of the primitive are converted to the integer voxel coordinates that contain that its two opposite corners. This is done by the utility function `GridAccel::PosToVoxel()`, which turns a world space (x,y,z) position into the voxel that contains that point.

[3] Some grid implementations try to save even more memory by using a hash table from (x,y,z) voxel number to voxel structures. This saves the memory for the `voxels` array, which may be substantial if the grid has very small voxels, and the vast majority of them are empty. However, this approach increases the computational expense of finding the `Voxel` structure for each voxel that a ray passes through.

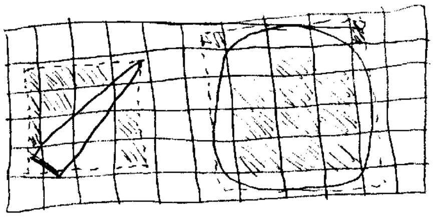


Figure 4.5: Two examples of cases where using the bounding box of a primitive to determine which grid voxels it should be stored in will cause it to be stored in a number of voxels unnecessarily: on the left, a long skinny triangle has a lot of empty space inside its axis-aligned bounding box and it is inaccurately added to the shaded voxels. On the right, the surface of the sphere doesn't intersect many of the voxels inside its bound, and they are also inaccurately included in the sphere's extent. While this error slightly degrades the grid's performance, it doesn't lead to incorrect ray-intersection results.

⟨*Find voxel extent of primitive*⟩≡

```
BBox pb = prims[i]->WorldBound();
int vmin[3], vmax[3];
for (int axis = 0; axis < 3; ++axis) {
    vmin[axis] = PosToVoxel(pb.pMin, axis);
    vmax[axis] = PosToVoxel(pb.pMax, axis);
}
```

⟨*GridAccel Private Public Methods*⟩≡

```
int PosToVoxel(const Point &P, int axis) const {
    int v = Float2Int((P[axis] - bounds.pMin[axis]) * InvWidth[axis]);
    return Clamp(v, 0, NVoxels[axis]-1);
}
```

38	BBox
39	BBox::pMax
39	BBox::pMin
677	Clamp()
142	GridAccel::InvWidth
141	GridAccel::NVoxels
144	GridAccel::Offset()
142	GridAccel::voxels
33	Point

The primitive is now added to all of the voxels that its bounds overlap. This is a conservative test for voxel overlap–at worst it will overestimate the voxels that the primitive overlaps. Figure 4.5 shows an example of two cases where this method leads to primitives being stored in more voxels than necessary. An exercise at the end of this chapter describes a more accurate method for associating primitives with voxels.

⟨*Add primitive to overlapping voxels*⟩≡

```
for (int z = vmin[2]; z <= vmax[2]; ++z)
    for (int y = vmin[1]; y <= vmax[1]; ++y)
        for (int x = vmin[0]; x <= vmax[0]; ++x) {
            int offset = Offset(x, y, z);
            if (!voxels[offset]) {
                ⟨Allocate new voxel and store primitive in it⟩
            }
            else {
                ⟨Add primitive to already-allocated voxel⟩
            }
        }
```

The `GridAccel::Offset()` utility functions give the offset into the `voxels` array for a particular (x, y, z) voxel. This is a standard technique for encoding a multi-dimensional array in a 1D array.

⟨*GridAccel Private Public Methods*⟩+≡

```
inline int Offset(int x, int y, int z) const {
    return z*NVoxels[0]*NVoxels[1] + y*NVoxels[0] + x;
}
```

To further reduce memory used for dynamically-allocated voxels and to improve their memory locality, the grid constructor uses an `ObjectArena` to hand out memory for voxels. **This is the first time we've talked about arenas, isn't it? Obviously this needs to be beefed up.**

⟨*Allocate new voxel and store primitive in it*⟩≡

```
voxels[offset] = new (voxelArena) Voxel(&mailboxes[i]);
```

⟨*GridAccel Private Data*⟩+≡

```
ObjectArena<Voxel> voxelArena;
```

If this isn't the first primitive to overlap this voxel, the `Voxel` has already been allocated and the primitive is handed off to the `Voxel::AddPrimitive()` method.

⟨*Add primitive to already-allocated voxel*⟩≡

```
voxels[offset]->AddPrimitive(&mailboxes[i]);
```

`GridAccel::NVoxels`	141
`GridAccel::voxels`	142
`MailboxPrim`	141
`ObjectArena`	668
`Voxel::AddPrimitive()`	145

Now we will define the `Voxel` structure, which records the primitives that overlap its extent. Because many `Voxels` may be allocated for a grid, we use a few simple techniques to keep the size of a `Voxel` small: variables that record its basic properties are packed into a single 32 bit word, and we use a `union` to overlap a few pointers of various types, only one of which will actually be used depending on the number of overlapping primitives.

⟨*Voxel Declarations*⟩≡

```
struct Voxel {
    ⟨Voxel Public Methods⟩
    union {
        MailboxPrim *onePrimitive;
        MailboxPrim **primitives;
    };
    u_int allCanIntersect:1;
    u_int nPrimitives:31;
};
```

When a `Voxel` is first allocated, only a single primitive has been found that overlaps it, so `Voxel::nPrimitives` is one, and `Voxel::onePrimitive` is used to store a pointer to its `MailboxPrim`. As more primitives are found to overlap, `Voxel::nPrimitives` will be greater than one, and `Voxel::primitives` is set to point to a dynamically-allocated array of pointers to `MailboxPrim` structures. Because these conditions are mutually-exclusive, the pointer to the single primitive and pointer to the array of pointers to primitives can share the same memory by being stored in a `union`. `Voxel::allCanIntersect` is used to record if all of the primitives in the voxel are intersectable or if some need refinement. For starters, it is conservatively set to false.

⟨*Voxel Public Methods*⟩≡

```
Voxel(MailboxPrim *op) {
    allCanIntersect = false;
    nPrimitives = 1;
    onePrimitive = op;
}
```

When `Voxel::AddPrimitive()` is called, this must mean that two or more primitives overlap the voxel, so the primitives' `MailboxPrim` pointers will be stored in its `Voxel::primitives` array. Memory for this array must be allocated in two cases: if the voxel currently holds a single primitive and we need to store a second, or if the allocated array is full. Rather than using more space in the voxel structure to store the current size of the array, the code here follows the convention that the array size will always be a power of two. Thus, whenever the `Voxel::nPrimitives` count is a power of two, the array has been filled and more memory is needed.

⟨*Voxel Public Methods*⟩+≡

```
void AddPrimitive(MailboxPrim *prim) {
    if (nPrimitives == 1) {
        ⟨Allocate initial primitives array in voxel⟩
    }
    else if (IsPowerOf2(nPrimitives)) {
        ⟨Increase size of primitives array in voxel⟩
    }
    primitives[nPrimitives] = prim;
    ++nPrimitives;
}
```

667	`AllocAligned()`
678	`IsPowerOf2()`
141	`MailboxPrim`
144	`Voxel`
144	`Voxel::allCanIntersect`
144	`Voxel::nPrimitives`
144	`Voxel::onePrimitive`
144	`Voxel::primitives`

Recall that `Voxel::onePrimitive` and `Voxel::primitives` are stored in a `union`. Therefore, it is important to store the memory for the array of pointers in a local variable on the stack and initialize its first entry from `Voxel::onePrimitive` before `Voxel::primitives` is initialized with the array pointer. Otherwise, the value of `Voxel::onePrimitive` would be clobbered before it was added to the new array, since `Voxel::onePrimitive` and `Voxel::primitives` share the same memory.

⟨*Allocate initial* `primitives` *array in voxel*⟩≡

```
MailboxPrim **p = (MailboxPrim **)AllocAligned(
    2 * sizeof(MailboxPrim *));
p[0] = onePrimitive;
primitives = p;
```

Similarly, it's necessary to be careful with setting `Voxel::primitives` to the pointer to the expanded array of `MailboxPrim` pointers.

⟨*Increase size of* `primitives` *array in voxel*⟩≡

```
int nAlloc = 2 * nPrimitives;
MailboxPrim **p = (MailboxPrim **)AllocAligned(nAlloc *
    sizeof(MailboxPrim *));
for (u_int i = 0; i < nPrimitives; ++i)
    p[i] = primitives[i];
FreeAligned(primitives);
primitives = p;
```

We won't show the straightforward implementations of the `GridAccel::WorldBound()` or `GridAccel::CanIntersect()` methods or its destructor.

4.3.2 Traversal

The `GridAccel::Intersect()` method handles the task of determining which voxels a ray passes through and calling the appropriate ray–primitive intersection routines.

⟨*GridAccel Method Definitions*⟩+≡

```
bool GridAccel::Intersect(const Ray &ray, Intersection *isect) const {
    ⟨Check ray against overall grid bounds⟩
    ⟨Get ray mailbox id⟩
    ⟨Set up 3D DDA for ray⟩
    ⟨Walk ray through voxel grid⟩
}
```

`BBox::Inside()`	40
`BBox::IntersectP()`	137
`GridAccel`	139
`ridAccel::CanIntersect()`	146
`GridAccel::WorldBound()`	146
`Intersection`	131
`MailboxPrim`	141
`Point`	33
`Ray`	36
`Ray::mint`	36
`Voxel::nPrimitives`	144
`Voxel::primitives`	144

The first task is to determine where the ray enters the grid, which gives the starting point for traversal through the voxels. If the ray's origin is inside the grid's bounding box, then clearly it begins there. Otherwise the `GridAccel::Intersect()` method finds the intersection of the ray with the grid's bounding box. If it hits, the first intersection along the ray is the starting point. If the ray misses the grid's bounding box, there can be no intersection with any of the geometry in the grid so `GridAccel::Intersect()` returns immediately.

⟨*Check ray against overall grid bounds*⟩≡

```
Float rayT;
if (bounds.Inside(ray(ray.mint)))
    rayT = ray.mint;
else if (!bounds.IntersectP(ray, &rayT))
    return false;
Point gridIntersect = ray(rayT);
```

Once we know that there is work to do, the next task is to find a unique ray identifier for mailboxing. We simply use a monotonic sequence of ray identifiers sorted in the `GridAccel::curMailboxId` member.

⟨*Get ray mailbox id*⟩≡

```
int rayId = ++curMailboxId;
```

⟨*GridAccel Private Data*⟩+≡

```
static int curMailboxId;
```

We now compute the initial (x,y,z) integer voxel coordinates for this ray as well as a number of auxiliary values that will make it very efficient to incrementally compute the set of voxels that the ray passes through. The ray–voxel traversal computation is similar in spirit to Bresenhamn's classic line drawing algorithm, where the series of pixels that a line passes through are found incrementally using just addition and comparisons to step from one pixel to the next. The main difference between the ray marching algorithm and Bresenham's are that we would like to find *all* of the voxels that the ray passes through, while Bresenham's algorithm does not provide this guarantee.

The values the ray–voxel stepping algorithm needs to keep track of are:

1. The coordinates of the voxel currently being considered, `Pos`.

2. The parametric *t* position along the ray where it makes its next crossing into another voxel in each of the *x*, *y*, and *z* directions, `NextCrossingT` (Figure 4.6). For example, for a ray with a positive *x* direction component, the parametric value along the ray where it crosses into the next voxel in *x*, `NextCrossingT[0]` is the parametric starting point `rayT` plus the *x* distance to the next voxel divided by the ray's *x* direction component. (This is similar to the ray–plane intersection formula.)

3. The change in the current voxel coordinates after a step in each direction (1 or -1), stored in `Step`.

35 Ray::d

4. The distance along the ray between voxels in each direction, `DeltaT`. These values are found by dividing the width of a voxel in a particular direction by the ray's corresponding direction component, giving the parametric distance along the ray that we have to travel to get from one side of a voxel to the other in the particular direction.

5. The coordinates of the last voxel the ray passes through before it exits the grid, `Out`.

The first two items will be updated as we step through the grid, while the last three are constant per ray.

```
⟨Set up 3D DDA for ray⟩≡
  Float NextCrossingT[3], DeltaT[3];
  int Step[3], Out[3], Pos[3];
  for (int axis = 0; axis < 3; ++axis) {
      ⟨Compute current voxel for axis⟩
      if (ray.d[axis] >= 0) {
          ⟨Handle ray with positive direction for voxel stepping⟩
      }
      else {
          ⟨Handle ray with negative direction for voxel stepping⟩
      }
  }
```

Computing the voxel address that the ray starts out in is easy since this method has already determined the position where the ray enters the grid. We simply use the utility routine `GridAccel::PosToVoxel` defined earlier.

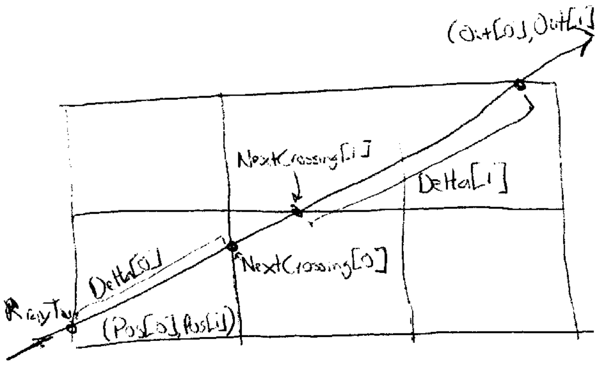


Figure 4.6: Stepping a ray through a voxel grid: `rayT` is the distance along the ray to the first intersection with the grid. The distance along the ray to the next distance we cross into the next voxel in the *x* direction is stored in `NextCrosing[0]`, and similarly for the *y* and *z* (not shown) directions. When we cross into the next *x* voxel, for example, we can immediately update the value of `NextCrossingT[0]` by adding a fixed value, the voxel width in *x* divided by the ray's *x* direction, `DeltaT[0]`.

`BBox::pMin`	39
`GridAccel::NVoxels`	141
`GridAccel::PosToVoxel()`	143
`GridAccel::Width`	142
`Ray::d`	35

⟨*Compute current voxel for axis*⟩≡

```
Pos[axis] = PosToVoxel(gridIntersect, axis);
```

If the ray's direction component is zero for a particular axis, then the `NextCrossingT` value for that axis will be initialized to the IEEE floating point ∞ value by the computation below. The voxel stepping logic later in this section will always decide to step in one of the other directions and we will correctly never step in this direction. This is convenient because we can handle rays that are perpendicular to any axis without any special code to test for division by zero.

⟨*Handle ray with positive direction for voxel stepping*⟩≡

```
NextCrossingT[axis] = rayT +
    (VoxelToPos(Pos[axis]+1, axis) - gridIntersect[axis]) /
        ray.d[axis];
DeltaT[axis] = Width[axis] / ray.d[axis];
Step[axis] = 1;
Out[axis] = NVoxels[axis];
```

This comes out of nowhere; maybe it should be presented alongsided PosToVoxel?

The `GridAccel::VoxelToPos()` method is the opposite of `GridAccel::PosToVoxel()`; it returns the position of a particular voxel's lower corner.

⟨*GridAccel Private Public Methods*⟩+≡

```
Float VoxelToPos(int p, int axis) const {
    return bounds.pMin[axis] + p * Width[axis];
}
```

⟨*GridAccel Private Public Methods*⟩+≡

```
Point VoxelToPos(int x, int y, int z) const {
    return bounds.pMin +
        Vector(x * Width[0], y * Width[1], z * Width[2]);
}
```

Similar computations compute these values for rays with negative direction components:

⟨*Handle ray with negative direction for voxel stepping*⟩≡

```
NextCrossingT[axis] = rayT +
    (VoxelToPos(Pos[axis], axis) - gridIntersect[axis]) /
        ray.d[axis];
DeltaT[axis] = -Width[axis] / ray.d[axis];
Step[axis] = -1;
Out[axis] = -1;
```

Once all the preprocessing is done for the ray, we can step through the grid. Starting with the first voxel that the ray passes through, we check for intersections with the primitives inside that voxel. If we find a hit, the boolean flag `hitSomething` is set to true. We must be careful, however, because the intersection point may be outside the current voxel since primitives may overlap multiple voxels. Therefore, the method doesn't immediately return when done processing a voxel where an intersection was found. Instead, we use the fact that the primitive's intersection routine will update the `Ray::maxt` member. When stepping through voxels, we will return only when we enter a voxel at a point that is beyond the closest found intersection.

39	BBox::pMin
144	GridAccel::Offset()
142	GridAccel::voxels
148	GridAccel::VoxelToPos()
142	GridAccel::Width
131	Intersection
141	MailboxPrim
33	Point
36	Ray
35	Ray::d
27	Vector
144	Voxel

⟨*Walk ray through voxel grid*⟩≡

```
bool hitSomething = false;
for (;;) {
    Voxel *voxel =
    voxels[Offset(Pos[0], Pos[1], Pos[2])];
    if (voxel != NULL)
        hitSomething |= voxel->Intersect(ray, isect, rayId);
    ⟨Advance to next voxel⟩
}
return hitSomething;
```

perhaps a transition saying that we have a Voxel::Intersect routine

⟨*GridAccel Method Definitions*⟩+≡

```
bool Voxel::Intersect(const Ray &ray, Intersection *isect, int rayId) {
    ⟨Refine primitives in voxel if needed⟩
    ⟨Loop over primitives in voxel and find intersections⟩
}
```

The boolean `Voxel::allCanIntersect` member tells us whether all of the primitives in the voxel are known to be intersectable. If this value is `false`, we loop over all primitives, calling their refinement routines as needed until only intersectable geometry remains. The logic for finding the i^{th} `MailboxPrim` in the loop over primitives is slightly complicated by a level of pointer indirection, since a single primitive and multiple primitives are stored differently in voxels. Handling this case in the way done below is worthwhile since it moves the test for whether we should be using the `Voxel::onePrimitive` item for a single primitive or the `Voxel::primitives` array for multiple primitives outside the body of the loop.

⟨*Refine primitives in voxel if needed*⟩≡

```
if (!allCanIntersect) {
    MailboxPrim **mpp;
    if (nPrimitives == 1) mpp = &onePrimitive;
    else mpp = primitives;
    for (u_int i = 0; i < nPrimitives; ++i) {
        MailboxPrim *mp = mpp[i];
        ⟨Refine primitive in mp if it's not intersectable⟩
    }
    allCanIntersect = true;
}
```

Primitives that need refinement are refined until only intersectable primitives remain, and a new `GridAccel` is created to hold the returned primitives if more than one was returned. One reason to always make a `GridAccel` for multiple reifined primitives is that doing so simplifies primitive refinement; a single `Primitive` always turns into a single object that represents all of the new `Primitive`s, so it's never necessary to increase the number of primitives in the voxel. If this primitive overlaps multiple voxels, then because all of them hold a pointer to a single `MailboxPrim` for it, it suffices to just update the primitive reference in the the shared `MailboxPrim` directly, and there's no need to loop over all of the voxels.[4]

GridAccel	139
MailboxPrim	141
MailboxPrim::primitive	141
Primitive	130
rimitive::CanIntersect()	131
Primitive::FullyRefine()	131
Reference	664
vector	658
Voxel::allCanIntersect	144
Voxel::nPrimitives	144
Voxel::onePrimitive	144
Voxel::primitives	144

⟨*Refine primitive in* `mp` *if it's not intersectable*⟩≡

```
if (!mp->primitive->CanIntersect()) {
    vector<Reference<Primitive> > p;
    mp->primitive->FullyRefine(p);
    if (p.size() == 1)
        mp->primitive = p[0];
    else
        mp->primitive = new GridAccel(p, true, false);
}
```

Once we know that we have only intersectable primitives, the loop over `MailboxPrims` for performing intersection tests here again has to deal with the difference between voxels with one primitive and voxels with multiple primitives in the same manner that the primitive refinement code did.

[4] The bounding box of the original unrefined primitive must encompass the refined geometry as well, so there's no danger that the refined geometry will overlap more voxels than before. On the other hand, it also may overlap many fewer voxels, which would lead to unnecessary intersection tests.

⟨*Loop over primitives in voxel and find intersections*⟩≡

```
bool hitSomething = false;
MailboxPrim **mpp;
if (nPrimitives == 1) mpp = &onePrimitive;
else mpp = primitives;
for (u_int i = 0; i < nPrimitives; ++i) {
    MailboxPrim *mp = mpp[i];
    ⟨Do mailbox check between ray and primitive⟩
    ⟨Check for ray–primitive intersection⟩
}
return hitSomething;
```

Here now is the mailbox check; if this ray was previously intersected against this primitive in another voxel, the redundant intersection test can be trivially skipped.

⟨*Do mailbox check between ray and primitive*⟩≡

```
if (mp->lastMailboxId == rayId)
    continue;
```

Finally, if we determine that a ray–primitive intersection test is necessary, the primitive's mailbox needs to be be updated.

⟨*Check for ray–primitive intersection*⟩≡

```
mp->lastMailboxId = rayId;
if (mp->primitive->Intersect(ray, isect)) {
    hitSomething = true;
}
```

141	`MailboxPrim`
141	`MailboxPrim::lastMailboxId`
131	`Primitive::Intersect()`
144	`Voxel::nPrimitives`
144	`Voxel::onePrimitive`
144	`Voxel::primitives`

After doing the intersection tests for the primitives in the current voxel, it is necessary to step to the next voxel in the ray's path. We need to decide whether to step in the *x*, *y*, or *z* direction. Fortunately, the `NextCrossingT` variable tells us the distance to the next crossing for each direction, and we can simply choose the smallest one. Traversal can be terminated if this step goes outside of the voxel grid, or if the selected `NextCrossingT` value is beyond the *t* distance of an already-found intersection. Otherwise, we step to the chosen voxel, and increment the chosen direction's `NextCrossingT` by its `DeltaT` value, so that future traversal steps will know how far to go before stepping in this direction again.

⟨*Advance to next voxel*⟩≡

```
⟨Find stepAxis for stepping to next voxel⟩
if (ray.maxt < NextCrossingT[stepAxis])
    break;
Pos[stepAxis] += Step[stepAxis];
if (Pos[stepAxis] == Out[stepAxis])
    break;
NextCrossingT[stepAxis] += DeltaT[stepAxis];
```

Choosing the axis along which to step basically requires finding the smallest of three numbers, an extremely straightforward task. However, in this case an optimization is possible because we don't care about the *value* of the smallest number, just its index in the `NextCrossingT` array. We can compute this index without any branching, which can lead to substantial performance improvements on a modern CPU.

The tricky bit of code below determines which of the three `NextCrossingT` values is the smallest and sets `stepAxis` accordingly. It encodes this logic by setting each of the three low-order bits in an integer to the results of three comparisons between pairs of `NextCrossingT` values. We then use a table (`cmpToAxis`) to map the resulting integer to the direction with the smallest value.

This kind of optimization is frequently available when trying to find the minimum or maximum of a very small group of numbers. An exercise at the end of the chapter asks you to explore the benefits of this approach.

⟨*Find* `stepAxis` *for stepping to next voxel*⟩≡

```
int bits = ((NextCrossingT[0] < NextCrossingT[1]) << 2) +
    ((NextCrossingT[0] < NextCrossingT[2]) << 1) +
    ((NextCrossingT[1] < NextCrossingT[2]));
const int cmpToAxis[8] = { 2, 1, 2, 1, 2, 2, 0, 0 };
int stepAxis = cmpToAxis[bits];
```

The grid also provides a special `GridAccel::IntersectP()` method that is optimized for checking for intersection along shadow rays, where we are only interested in the presence of an intersection, rather than the details of the intersection itself. It is almost identical to the `GridAccel::Intersect()` routine, except that it calls the `Primitive::IntersectP()` method of the primitives rather than `Primitive::Intersect()`, and it immediately stops traversal when any intersection is found. Because of the small number of differences, we won't include the implementation here.

`KdTreeAccel`	154
`Ray`	36

⟨*GridAccel Public Methods*⟩+≡

```
bool IntersectP(const Ray &ray) const;
```

4.4 Kd-Tree Accelerator

Binary space partitioning (BSP) trees adaptively subdivide space into irregularly-sized regions. The most important consequence of this design is that they can be much more effective than a regular grid for irregular collections of geometry. A BSP tree starts with a bounding box that encompasses the entire scene. If the number of primitives in the box is greater than some threshold, the box is split in half by a plane. Primitives are then assigned to whichever half they overlap. Primitives that lie in both halves are assigned twice. This process continues recursively until either each sub-region contains a sufficiently small number of primitives, or a maximum splitting depth is reached. Because the splitting planes can be placed at arbitrary positions inside the overall bound and because different parts of 3D space can be refined to different degrees, BSP trees can easily handle uneven distributions of geometry.

Two variations of BSP trees are *kd-trees* and *octrees*. A kd-tree simply restricts the splitting plane to be perpendicular to one of the coordinate axes; this makes traversal and construction of the tree more efficient. The octree uses three axis-perpendicular planes simultaneously, splitting the box into eight regions at each step. In this section, we will implement a kd-tree for ray intersection acceleration in the `KdTreeAccel` class. Source code for this class can be found in the file `accelerators/kdtree.cpp`.

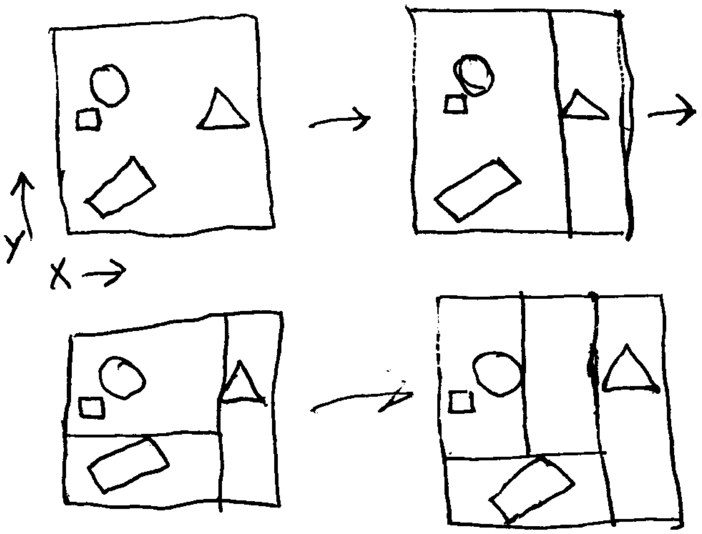


Figure 4.7: The kd-tree is built by recursively splitting the bounding box of the scene geometry along one of the coordinate axes. Here, the first split is along the *x* axis; it is placed so that the triangle is precisely alone in the right region and the rest of the primitives end up on the left. The left region is then refined a few more times with axis-aligned splitting planes. The details of the refinement criteria–which axis is used to split space at each step, at which position along the axis the plane is placed, and at what point refinement terminates–can all substantially affect the performance of the tree in practice.

⟨*KdTreeAccel Declarations*⟩+≡

```
class  KdTreeAccel : public Aggregate {
public:
    ⟨KdTreeAccel Public Methods⟩
private:
    ⟨KdTreeAccel Private Data⟩
};
```

In addition to the primitives to be stored, the `KdTreeAccel` constructor takes a few values that will be used to guilde the decisions that will be made as the tree is built; these parameters are just stored in member variables for later use. For simplicity of implementation, the `KdTreeAccel` requires that all of the primitives it stores are intersectable. We leave as an exercise the task of improving the implementation to do lazy refinement like the `GridAccel` does. Therefore, the constructor starts out by refining all primitives until all are intersectable before bulding the tree; see Figure 4.7 for an overview of how the tree is built.

⟨*KdTreeAccel Method Definitions*⟩≡

```
KdTreeAccel::KdTreeAccel(const vector<Reference<Primitive> > &p,
        int icost, int tcost, Float ebonus, int maxp, int maxDepth)
    : isectCost(icost), traversalCost(tcost),
    emptyBonus(ebonus), maxPrims(maxp)
{
    vector<Reference<Primitive > > prims;
    for (u_int i = 0; i < p.size(); ++i)
        p[i]->FullyRefine(prims);
    ⟨Initialize mailboxes for KdTreeAccel⟩
    ⟨Build kd-tree for accelerator⟩
}
```

`Aggregate`	135
`AllocAligned()`	667
`GridAccel`	139
`MailboxPrim`	141
`Primitive`	130
`Primitive::FullyRefine()`	131
`Reference`	664
`vector`	658

⟨*KdTreeAccel Private Data*⟩≡

```
int isectCost, traversalCost, maxPrims;
Float emptyBonus;
```

As with `GridAccel`, the kd-tree uses mailboxing to avoid repeated intersections with primitives that straddle splitting planes and overlap multiple regions of the tree. In fact, it uses the exact same `MailboxPrim` structure.

⟨*Initialize mailboxes for* `KdTreeAccel`⟩≡

```
curMailboxId = 0;
nMailboxes = prims.size();
mailboxPrims = (MailboxPrim *)AllocAligned(nMailboxes *
    sizeof(MailboxPrim));
for (u_int i = 0; i < nMailboxes; ++i)
    new (&mailboxPrims[i]) MailboxPrim(prims[i]);
```

⟨*KdTreeAccel Private Data*⟩+≡

```
u_int nMailboxes;
MailboxPrim *mailboxPrims;
mutable int curMailboxId;
```

4.4.1 Tree Representation

The kd-tree is a binary tree, where each interior node always has two children and where leaves of the tree store the primitives that overlap them. Each interior node must provide access to three pieces of information:

- Split axis: which of the *x*, *y*, or *z* axes we split along at this node
- Split position: the position of the splitting plane along the axis
- Children: information about how to reach the two child nodes beneath it

Each leaf node needs only to record which primitives overlap it.

It is worth going through a bit of trouble to ensure that all interior leaf nodes and many leaf notes use just 8 bytes of memory (assuming 4 byte `Float`s and pointers), because doing so ensures that four nodes will fit into a 32 byte cache line. Because there are many nodes in the tree and because many nodes are accessed for each ray, minimizing the size of the node representation substantially improves cache performance. Our initial implementation used a 16 byte node representation; when we reduced the size to 8 bytes we obtained an almost 20% speed increase. Both leaves and interior nodes types of node are represented by the `KdAccelNode` structure below; the comments after each `union` member indicate whether a particular field is used for interior nodes, leaf nodes, or both.

139 GridAccel
141 MailboxPrim

⟨*KdAccelNode Declarations*⟩+≡

```
struct KdAccelNode {
    ⟨KdAccelNode Methods⟩
    union {
        u_int flags;   // Both
        Float split;   // Interior
        u_int nPrims;  // Leaf
    };
    union {
        u_int aboveChild;           // Interior
        MailboxPrim *onePrimitive;  // Leaf
        MailboxPrim **primitives;   // Leaf
    };
};
```

The two low order bits of the `KdAccelNode::flags` variable are used to differentiate between interior nodes with *x*, *y*, and *z* splits (where these bits hold the values 0, 1, and 2, respectively), and leaf nodes (where these bits hold the value 3.)

It is relatively easy to store leaf nodes in 8 bytes: since the low two bits of `KdAccelNode::flags` are used to indicate that this is a leaf, the upper 30 bits of `KdAccelNode::nPrims` are available to record how many primitives overlap it. This should be plenty, because if the tree was built properly there should be just a handful of primitives in each leaf. As with `GridAccel`, if just a single primitive overlaps a `KdAccelNode` leaf, its `MailboxPrim` pointer is stored directly in the `KdAccelNode::onePrimitive` field. If more primitives overlap, memory is dynamically allocated for an array of them pointed to by `KdAccelNode::primitives`.

Leaf nodes are easy to initialize; the number of primitives must be shifted two bits to the left before being stored so that the low two bits of KdAccelNode::flags can be set to 11 to indicate that this is a leaf node.

⟨*KdAccelNode Methods*⟩≡

```
void initLeaf(int *primNums, int np,
        MailboxPrim *mailboxPrims, MemoryArena &zone) {
    nPrims = np << 2;
    flags |= 3;
    ⟨Store MailboxPrim *s for leaf node⟩
}
```

For leaf nodes with zero or one overlapping primitives, no dynamic memory allocation is necessary thanks to the KdAccelNode::onePrimitive field. For the case where multiple primitives overlap, the caller passes in a MemoryArena for allocating memory for the arrays of MailboxPrim pointers. This helps to reduce wasted space for these allocations and improves cache efficiency by placing all of these arrays together in memory.

⟨*Store* MailboxPrim **s for leaf node*⟩≡

```
if (np == 0)
    onePrimitive = NULL;
else if (np == 1)
    onePrimitive = &mailboxPrims[primNums[0]];
else {
    primitives = (MailboxPrim **)zone.Alloc(np *
        sizeof(MailboxPrim *));
    for (int i = 0; i < np; ++i)
        primitives[i] = &mailboxPrims[primNums[i]];
}
```

KdAccelNode	155
KdAccelNode::flags	155
KdAccelNode::nPrims	155
KdAccelNode::onePrimitive	155
KdAccelNode::primitives	155
MailboxPrim	141
MemoryArena	670

Getting interior nodes down to 8 bytes takes a bit more work. As explained above, the lowest two bits of KdAccel::flags are used to record which axis the node was split along. Yet the split position along that axis is stored in KdAccelNode::split, a Float value that occupies the same memory as KdAccelNode::flags. This seems impossible—we can't just ask the compiler to use the top 30 bits of KdAccelNode::split as a Float.

It turns out that as long as the lowest two bits of KdAccelNode::flags are set after KdAccelNode::split, this technique works thanks to the layout of Floats in memory. For IEEE floating point, the two bits used by KdAccelNode::flags are the least-significant bits of the floating-point mantissa value, so changing their original value only minimally affects the floating-point value that is stored. Figure 4.8 illustrates the layout in memory.

Although this trick is fairly complicated, it is worth it for the performance benefits. In addition, all of the complexity is hidden behind a small number of KdAccelNode methods, so the rest of the system is insulated from our special representation.

So that we don't need memory to store pointers to the two child nodes of an interior node, all of the nodes are allocated in a single contiguous block of memory, and the child of an interior node that is responsible for space "below" the splitting plane is always stored in the array position immediately after its parent (this also

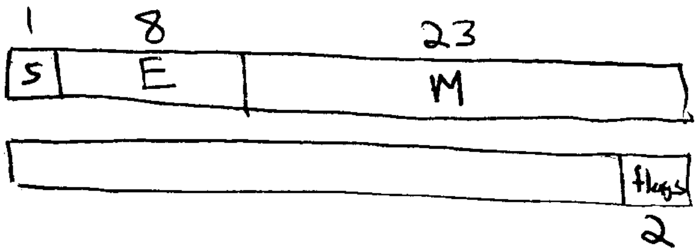


Figure 4.8: Layout of floats and ints in memory...

improves cache performance, by keeping at least one child close to its parent in memory.) The other child, representing space above the splitting plane will end up at somewhere else in the array; KdAccelNode::aboveChild stores its position.

Given all those conventions, the code to initialize an interior node is straightforward. The split position is stored before the split axis is written in KdAccelNode::flags. Rather than directly assigning the axis to KdAccelNode::flags, which would clobber KdAccelNode::split as well, it's necessary to carefully set just the low two bits of the flags with the axis's value.

155 KdAccelNode
155 KdAccelNode::flags
155 KdAccelNode::nPrims
155 KdAccelNode::split

⟨*KdAccelNode Methods*⟩+≡
```
void initInterior(int axis, Float s) {
    split = s;
    flags &= ˜3;
    flags |= axis;
}
```

Finally, we'll provide a few methods to extract various values from the node, so that callers don't have to be aware of the admittedly complex details of its representation.

⟨*KdAccelNode Methods*⟩+≡
```
Float SplitPos() const { return split; }
int nPrimitives() const { return nPrims >> 2; }
int SplitAxis() const { return flags & 3; }
bool IsLeaf() const { return (flags & 3) == 3; }
```

4.4.2 Tree construction

The kd-tree is built with a recursive top-down algorithm. At each step, we have an axis-aligned region of space and a set of primitives that overlap that region. Each region is either split into two sub-regions and turned into an interior node, or a leaf node is created with the overlapping primitives, terminating the recursion.

As mentioned in the discussion of KdAccelNodes, all tree nodes are stored in a contiguous array; KdTreeAccel::nextFreeNode records the next node in this array that is available, and KdTreeAccel::nAllocedNodes records the total number that have been allocated. By setting both of them to zero and not allocating any nodes at startup, we ensure that an allocation will be done immediately when the first node of the tree is initialized.

It is also necessary to determine a maximum tree depth if one wasn't given to the constructor. Though the tree construction process will normally terminate naturally at a reasonable depth, we cap the maximum depth so that the amount of memory used for the tree cannot grow without bound in pathological cases. We have found that the expression $8 + 1.3\log(N)$ gives a good maximum depth for a variety of scenes.

⟨*Build kd-tree for accelerator*⟩≡

```
nextFreeNode = nAllocedNodes = 0;
if (maxDepth <= 0)
    maxDepth = Round2Int(8 + 1.3f * Log2Int(prims.size()));
⟨Compute bounds for kd-tree construction⟩
⟨Allocate working memory for kd-tree construction⟩
⟨Initialize primNums for kd-tree construction⟩
⟨Start recursive construction of kd-tree⟩
⟨Free working memory for kd-tree construction⟩
```

⟨*KdTreeAccel Private Data*⟩+≡

```
KdAccelNode *nodes;
int nAllocedNodes, nextFreeNode;
```

BBox	38
KdAccelNode	155
Primitive::WorldBound()	130
prims0	166
prims1	166
Union()	40
vector	658

Because the construction routine will be repeatedly using the bounding boxes of the primitives along the way, we store them in a vector so that the potentially-slow `Primitive::WorldBound()` methods don't need to be called repeatedly.

⟨*Compute bounds for kd-tree construction*⟩≡

```
vector<BBox> primBounds;
primBounds.reserve(prims.size());
for (u_int i = 0; i < prims.size(); ++i) {
    BBox b = prims[i]->WorldBound();
    bounds = Union(bounds, b);
    primBounds.push_back(b);
}
```

⟨*KdTreeAccel Private Data*⟩+≡

```
BBox bounds;
```

One of the parameters to the tree construction routine is an array of integers indicating which primitives overlap the current node. For the root node, we just need an array with `prims.size()` entries set up such that the i^{th} entry has the value *i*.

⟨*Initialize* `primNums` *for kd-tree construction*⟩≡

```
int *primNums = new int[prims.size()];
for (u_int i = 0; i < prims.size(); ++i)
    primNums[i] = i;
```

`KdTreeAccel::buildTree()` is called for each tree node; it is responsible for deciding if the node should be an interior node or leaf and updating the data structures appropriately. The last three parameters, `edges`, `prims0`, and `prims1`, are pointers to data from the ⟨*Allocate working memory for kd-tree construction*⟩ fragment, which will be defined in a few pages.

⟨Start recursive construction of kd-tree⟩≡

```
  buildTree(0, bounds, primBounds, primNums, prims.size(), maxDepth,
      edges, prims0, prims1);
```

This destructor is out of nowhere...

⟨Free working memory for kd-tree construction⟩≡

```
  delete[] primNums;
```

The main parameters to `KdTreeAccel::buildTree` are the offset into the array of `KdAccelNodes` to use for the node that it creates, the bounding box that gives the region of space that the node covers, and the indices of primitives that overlap it.

⟨KdTreeAccel Method Definitions⟩+≡

```
  void KdTreeAccel::buildTree(int nodeNum, const BBox &nodeBounds,
          const vector<BBox> &allPrimBounds,  int *primNums,
          int nPrims, int depth, BoundEdge *edges[3], int *prims0,
          int *prims1, int badRefines) {
      ⟨Get next free node from nodes array⟩
      ⟨Initialize leaf node if termination criteria met⟩
      ⟨Initialize interior node and continue recursion⟩
  }
```

If all of the allocated nodes have been used, node memory is reallocated with twice as many entries and the old values are copied over. The first time `KdTreeAccel::buildTree()` is called, `KdTreeAccel::nAllocedNodes` will be zero and an initial block of tree nodes will be allocated.

667	AllocAligned()
38	BBox
162	BoundEdge
155	KdAccelNode
156	KdAccelNode::initLeaf()
154	KdTreeAccel
158	KdTreeAccel::nAllocedNodes
158	KdTreeAccel::nextFreeNode
158	KdTreeAccel::nodes
166	prims0
166	prims1
658	vector

⟨Get next free node from `nodes` array⟩≡

```
  if (nextFreeNode == nAllocedNodes) {
      int nAlloc = max(2 * nAllocedNodes, 512);
      KdAccelNode *n = (KdAccelNode *)AllocAligned(nAlloc *
          sizeof(KdAccelNode));
      if (nAllocedNodes > 0) {
          memcpy(n, nodes, nAllocedNodes * sizeof(KdAccelNode));
          FreeAligned(nodes);
      }
      nodes = n;
      nAllocedNodes = nAlloc;
  }
  ++nextFreeNode;
```

A leaf node is created (stopping the recursion) either if there are a sufficently small number of primitives in the region, or if the maximum depth has been reached. The `depth` parameter starts out as the tree's maximum depth and is decremented at each level.

⟨Initialize leaf node if termination criteria met⟩≡

```
  if (nPrims <= maxPrims || depth == 0) {
      nodes[nodeNum].initLeaf(primNums, nPrims, mailboxPrims, zone);
      return;
  }
```

As described above, `KdAccelNode::initLeaf()` uses a memory zone to allocate space for variable-sized arrays of primitives. Because the zone used here is a member variable, the memory it allocates will naturally all be freed when the `KdTreeAccel` is destroyed.

```
⟨KdTreeAccel Private Data⟩+≡
  MemoryArena zone;
```

If we are building an internal node, it is necessary to choose a splitting plane, classify the primitives with respect to that plane, and recurse.

```
⟨Initialize interior node and continue recursion⟩≡
  ⟨Choose split axis position for interior node⟩
  ⟨Create leaf if no good splits were found⟩
  ⟨Classify primitives with respect to split⟩
  ⟨Recursively initialize children nodes⟩
```

Our implementation chooses a split based on a cost model that estimates the computational expense of performing ray intersection tests, including the time spent traversing nodes of the tree and the time spent on ray–primitive intersection tests. Its goal is to minimize the total cost; we implement a greedy algorithm that minimizes the cost for the node individually. The estimated cost is computed for several candidate splitting planes in the node, and the split that gives the lowest cost is chosen.

KdTreeAccel	154
MemoryArena	670
Primitive	130

The idea behind the cost model is straightforward: at any node of the tree we could just create a leaf node for the current region and geometry. In that case, any ray that passes through this region will be tested against all of the overlapping primitives and will incur a cost of

$$\sum_{i=1}^{N} t_i(i),$$

where N is the number of primitives in the region and $t_i(i)$ is the time to compute a ray–object intersection with the ith primitive.

The other option is to split the region. In that case, rays will incur the cost

$$t_t + p_0 \sum_{i=1}^{N_b} t_i(b_i) + p_1 \sum_{i=1}^{N_a} t_i(a_i),$$

where t_t is the time it takes to traverse the interior node and determine which of the children the ray passes through, p_0 and p_1 are the probabilities that the ray passes through each of the two regions, b_i and a_i are the indices of primitives below and above the splitting plane, and and N_b and N_a are the number of primitives that overlap the regions below and above the splitting plane, respectively. The choice of splitting plane affects both the two probabilities as well as the number of primitives on each side of the split.

In our implementation, we will make the simplifying assumption that $t_i(i)$ is the same for all of the primitives; this is probably not too far from reality, and any error that it introduces doesn't seem to affect the performance of this accelerator very much. Another possibility would be to add a method to `Primitive` that returns an estimate of the number of CPU cycles its intersection test requires. The intersection

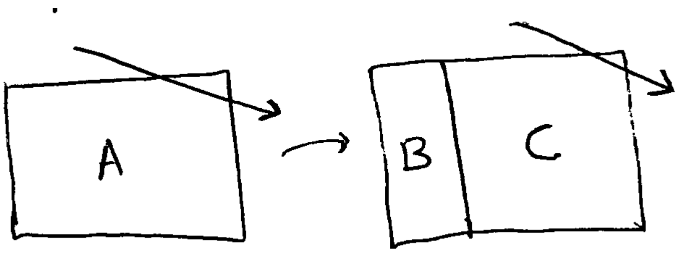


Figure 4.9: Split A into B and C...

cost t_i and the traversal cost t_t can be set by the user; their default values are 80 and 1, respectively. Ultimately, it is the ratio of these two values that primarily determines the behavior of the tree-building algorithm.

Finally, it is worth giving a slight preference to choosing splits where one of the children has no primitives overlapping it, since rays passing through these regions can immediately advance to the next kd-tree node without any ray–primitive intersection tests. Thus, the revised costs for unsplit and split regions are respectively

162 BoundEdge

$$\begin{aligned} &t_i N \\ &t_t + (1 - b_e)(p_b N_b t_i + p_a N_a t_i), \end{aligned}$$

where b_e is a “bonus” value that is zero unless one of the two regions is completely empty, in which case it takes on a value between zero and one.

The probabilities p_0 and p_1 are easily computed using ideas from geometric probability. It can be shown that for a convex volume A contained in another convex volume B, the conditional probability that a random ray passing through B will also pass through A is the ratio of their surface areas, s_A and s_B:

$$p(A|B) = \frac{s_A}{s_B}.$$

Because we are interested in the cost for rays passing through the interior node, we can use this result directly. Thus, given a split of a region A into two sub-regions B and C (see Figure 4.9), the probability that a ray passing through A will also pass through either of the subregions is easily computed.

The last problem to address is how to generate candidate splitting positions and how to efficiently compute the cost for each candidate. It can be shown that the minimum cost with this model will be attained at a split that is coincident with one of the faces of one of the primitives bounding boxes–there’s no need to consider splits at intermediate positions. (To convince yourself of this, consider what happens to the cost function between the edges of the faces). Here, we will consider all bounding box faces inside the region for all three axes.

The cost for checking all of these candidates thus can be kept relatively low with a carefully-structured algorithm. To compute these costs, we will sweep across the projections of the bounding boxes onto each axis and keep track of which gives the lowest cost (Figure 4.10.) Each bounding box has two edges on each axis, each of which is represented by a `BoundEdge` structure. This structure records the position

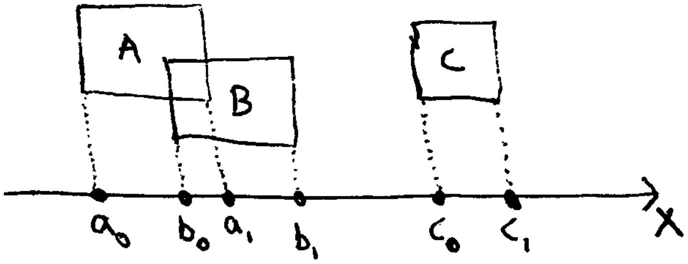


Figure 4.10: Projections of bbox edges onto the axis...

of the edge along the axis, whether it represents the start or end of a bounding box (going from low to high along the axis), and which primitive it is associated with.

⟨*KdAccelNode Declarations*⟩+≡

```
struct BoundEdge {
    ⟨BoundEdge Public Methods⟩
    Float t;
    int primNum;
    enum { START, END } type;
};
```

⟨*BoundEdge Public Methods*⟩+≡

```
BoundEdge(Float tt, int pn, bool starting) {
    t = tt;
    primNum = pn;
    type = starting ? START : END;
}
```

We will need at most `2 * prims.size()` `BoundEdge`s when computing costs for any tree node, so we allocate the memory for the edges for all three axes once and then reuse it for each node that is created. The fragment ⟨*Free working memory for kd-tree construction*⟩, not included here, frees this space after the tree has been built.

⟨*Allocate working memory for kd-tree construction*⟩≡

```
BoundEdge *edges[3];
for (int i = 0; i < 3; ++i)
    edges[i] = new BoundEdge[2*prims.size()];
```

After determining the estimated cost for creating a leaf, `KdTreeAccel::buildTree()` loops over the three axes and computes the cost function for each candidate split. `bestAxis` and `bestOffset` record the axis and bounding box edge index that gave the lowest cost so far, `bestCost`. `invTotalSA` is initialized to the reciprocal of the node's surface area; its value will be used when computing the probabilities of rays passing through each of the candidate children nodes.

⟨*Choose split axis position for interior node*⟩≡

```
int bestAxis = -1, bestOffset = -1;
Float bestCost = INFINITY;
Float oldCost = isectCost * nPrims;
Vector d = nodeBounds.pMax - nodeBounds.pMin;
Float invTotalSA = 1.f / (2.f * (d.x*d.y + d.x*d.z + d.y*d.z));
⟨Choose which axis to split along⟩
⟨Initialize edges for axis⟩
⟨Compute cost of all splits for axis to find best⟩
```

text here

more than just text, we don't loop over the axes any more. Some of the previous discussion is therefore wrong.

Always split along the axis with the largest extent; works well in practice, saves the work of checking all of them.

⟨*Choose which axis to split along*⟩≡

```
int axis;
if (d.x > d.y && d.x > d.z) axis = 0;
else axis = (d.y > d.z) ? 1 : 2;
```

First the `edges` array for the **current axis** is initialized using the bounding boxes of the overlapping primitives. The array is then sorted from low to high along the axis so that we can sweep over the box edges from first to last.

38	BBox
162	BoundEdge
678	INFINITY
27	Vector

⟨*Initialize edges for* `axis`⟩≡

```
for (int i = 0; i < nPrims; ++i) {
    int pn = primNums[i];
    const BBox &bbox = allPrimBounds[pn];
    edges[axis][2*i  ] = BoundEdge(bbox.pMin[axis], pn, true);
    edges[axis][2*i+1] = BoundEdge(bbox.pMax[axis], pn, false);
}
sort(&edges[axis][0], &edges[axis][2*nPrims]);
```

The C++ standard library routine `sort()` requires that the structure being sorted define an ordering; this is easily done with the `BoundEdge::t` values. However, one subtlety is that if the `BoundEdge::t` values match, it is necessary to try to break the tie by comparing the node's types; this is necessary since `sort()` depends on the fact that `a < b` and `b < a` is only true if `a == b`.

⟨*BoundEdge Public Methods*⟩+≡

```
bool operator<(const BoundEdge &e) const {
    if (t == e.t)
        return (int)type < (int)e.type;
    else return t < e.t;
}
```

Given the sorted array of edges, we'd like to quickly compute the cost function for a split at each one of them. The probabilities for a ray passing through each child node are easily computed, and the number of primitives on each side of the split is tracked by `nBelow` and `nAbove`. At the first edge, all primitives must be above that edge by definition, so `nAbove` is initialized to `nPrims` and `nBelow` is zero. When we encounter a starting edge of a bounding box, we know that the enclosed primitive will overlap the volume below the potential split at that edge. When we encounter an ending edge, the enclosed primitive must be above the edge. The tests at the start and end of the loop body update the primitive counts for these cases.

⟨*Compute cost of all splits for* `axis` *to find best*⟩≡

```
int nBelow = 0, nAbove = nPrims;
for (int i = 0; i < 2*nPrims; ++i) {
    if (edges[axis][i].type == BoundEdge::END) --nAbove;
    Float edget = edges[axis][i].t;
    if (edget > nodeBounds.pMin[axis] &&
        edget < nodeBounds.pMax[axis]) {
        ⟨Compute cost for split at ith edge⟩
    }
    if (edges[axis][i].type == BoundEdge::START) ++nBelow;
}
```

Given all of this information, the cost for a particular split is easily computed. `belowSA` and `aboveSA` are hold the surface areas of the two candidate child bounds; they are easily computed by adding up the areas of the six faces. Given an axis number, we can use the `otherAxis` array to quickly compute the indices of the other two axes without branching.

BoundEdge 162

⟨*Compute cost for split at* `i`*th edge*⟩≡

```
int otherAxis[3][2] = { {1,2}, {0,2}, {0,1} };
int otherAxis0 = otherAxis[axis][0], otherAxis1 = otherAxis[axis][1];
Float belowSA = 2 * (d[otherAxis0] * d[otherAxis1] +
    (edget - nodeBounds.pMin[axis]) * (d[otherAxis0] + d[otherAxis1]));
Float aboveSA = 2 * (d[otherAxis0] * d[otherAxis1] +
    (nodeBounds.pMax[axis] - edget) * (d[otherAxis0] + d[otherAxis1]));
Float pBelow = belowSA * invTotalSA, pAbove = aboveSA * invTotalSA;
Float eb = (nAbove == 0 || nBelow == 0) ? emptyBonus : 0.f;
Float cost = traversalCost + isectCost * (1.f - eb) *
    (pBelow * nBelow + pAbove * nAbove);
⟨Update best split if this is lowest cost so far⟩
```

⟨*Update best split if this is lowest cost so far*⟩≡

```
if (cost < bestCost)  {
    bestCost = cost;
    bestAxis = axis;
    bestOffset = i;
}
```

It may happen that there are no possible splits found in the tests above (Figure 4.11 illustrates a case where this may happen). In this case, there isn't a single candidate position at which to split the node. Refining such a node doesn't do any good, since both children will still have the same number of overlapping primitives. When we detect this condition, we give up and make a leaf node.

It is also possible that the best split will have a cost that is still higher than the cost for not splitting the node at all. If it is substantially worse and there aren't too many primitives, a leaf node is made immeditately. Otherwise, `badRefines` keeps

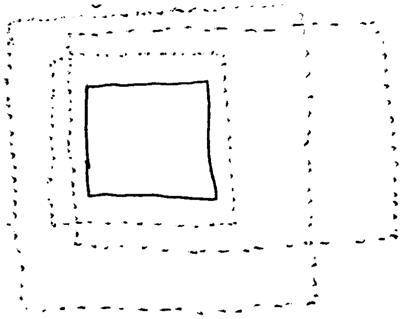


Figure 4.11: No useful splits possible due to overlap. Three bounding boxes overlap the node, yet none of their edges are inside it.

track of how many bad splits have been made so far above the current node of the tree. It's worth allowing a few slighly poor refinements since later splits may be able to find much better ones given a smaller subset of primitives to consider.

⟨*Create leaf if no good splits were found*⟩≡

```
if (bestCost > oldCost) ++badRefines;
if ((bestCost > 4.f * oldCost && nPrims < 16) ||
    bestAxis == -1 || badRefines == 3) {
    nodes[nodeNum].initLeaf(primNums, nPrims, mailboxPrims, zones);
    return;
}
```

162 BoundEdge
156 KdAccelNode::initLeaf()
166 prims0
166 prims1

Having chosen a split position, the edges can be used to quickly classify the primitives as being above, below, or on both sides of the split in the same way as was done to keep track of `nBelow` and `nAbove` in the code above.

⟨*Classify primitives with respect to split*⟩≡

```
int n0 = 0, n1 = 0;
for (int i = 0; i < bestOffset; ++i)
    if (edges[bestAxis][i].type == BoundEdge::START)
        prims0[n0++] = edges[bestAxis][i].primNum;
for (int i = bestOffset+1; i < 2*nPrims; ++i)
    if (edges[bestAxis][i].type == BoundEdge::END)
        prims1[n1++] = edges[bestAxis][i].primNum;
```

Recall that the node number of the "below" child of this node is the current node number plus one. After the recursion has returned from that side of the tree, the `nextFreeNode` offset is used for the "above" child. The only other important detail here is that the `prims0` memory is passed directly for re-use by both children, while the `prims1` pointer is advanced forward first. This is necessary since the current invocation of `KdTreeAccel::buildTree()` depends on its `prims1` values being preserved over the first recursive call to `KdTreeAccel::buildTree()` below, since it must be passed as a parameter to the second call. However, there is no corresponding need to preserve the `edges` values or to preserve `prims0` beyond its immediate used in the first recursive call.

⟨*Recursively initialize children nodes*⟩≡

```
Float tsplit = edges[bestAxis][bestOffset].t;
nodes[nodeNum].initInterior(bestAxis, tsplit);
BBox bounds0 = nodeBounds, bounds1 = nodeBounds;
bounds0.pMax[bestAxis] = bounds1.pMin[bestAxis] = tsplit;
buildTree(nodeNum+1, bounds0,
    allPrimBounds, prims0, n0, depth-1, edges, prims0, prims1 + nPrims, badRefi
nodes[nodeNum].aboveChild = nextFreeNode;
buildTree(nodes[nodeNum].aboveChild, bounds1, allPrimBounds,
    prims1, n1, depth-1, edges, prims0, prims1 + nPrims, badRefines);
```

Thus, much more space for the `prims1` array of integers for storing the overlapping primitive numbers is needed than for the `prims0` array, which only needs to handle the primitives at a single level at a time.

⟨*Allocate working memory for kd-tree construction*⟩+≡

```
int *prims0 = new int[prims.size()];
int *prims1 = new int[(maxDepth+1) * prims.size()];
```

4.4.3 Traversal

Figure 4.12 shows the basic process of ray traversal through the tree. Intersecting the ray with the tree's overall bounds gives initial `tmin` and `tmax` values, marked with "x"s in the figure. As with the grid accelerator, if the ray misses the scene bounds, we can quickly return `false`. Otherwise, we begin to descend into the tree, starting at the root. At each interior node, we determine which of the two children the ray enters first, and process both children in order. Traversal ends either when the ray exits the tree or when the closest intersection is found.

`BBox`	38
`BBox::IntersectP()`	137
`Intersection`	131
`KdAccelNode::aboveChild`	155
`ccelNode::initInterior()`	157
`KdTreeAccel`	154
`KdTreeAccel::buildTree()`	159
`dTreeAccel::nextFreeNode`	158
`KdTreeAccel::nodes`	158
`Ray`	36

⟨*KdTreeAccel Method Definitions*⟩+≡

```
bool KdTreeAccel::Intersect(const Ray &ray,
        Intersection *isect) const {
    ⟨Compute initial parametric range of ray inside kd-tree extent⟩
    ⟨Prepare to traverse kd-tree for ray⟩
    ⟨Traverse kd-tree nodes in order for ray⟩
}
```

The algorithm starts by finding the overall parametric range $[t_{\min}, t_{\max}]$ of the ray's overlap with the tree, exiting immediately if there is no overlap.

⟨*Compute initial parametric range of ray inside kd-tree extent*⟩≡

```
Float tmin, tmax;
if (!bounds.IntersectP(ray, &tmin, &tmax))
    return false;
```

Before tree traversal starts, a new mailbox id is found for the ray and the the reciprocals of the components of the direction vector are precomputed so that it is possible to order to replace divides with multiplies in the main traversal loop. The array of `KdToDo` structures is used to record the nodes yet to be processed for the ray; it is ordered so that the last active entry in the array is the next node that should be considered. The maximum number of entries needed in this array is

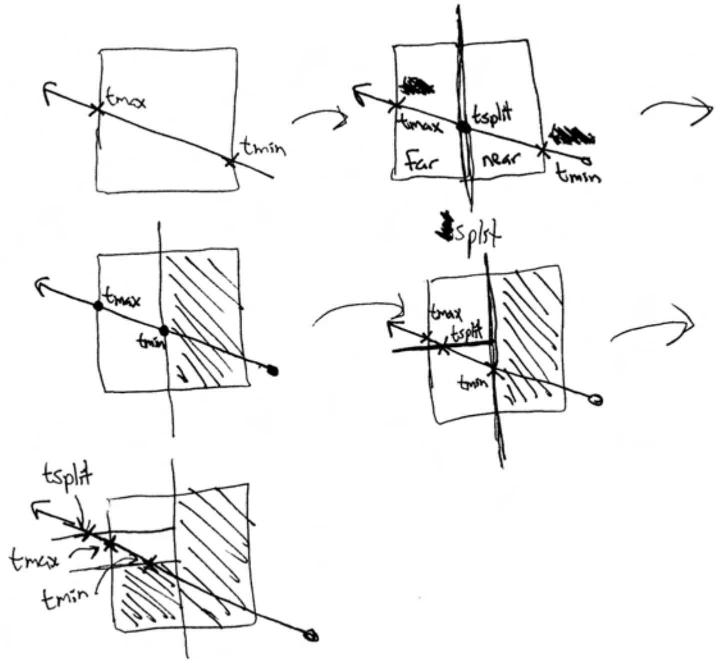


Figure 4.12: Traversal of a ray through the kd-tree: the ray is intersected with the bounds of the tree, giving an initial parametric $[t_{\text{min}}, t_{\text{max}}]$ range to consider. Because this range is non-empty, we need to consider the two children of the root node, here. The ray first enters the child on the right, labeled "near", where it has a parametric range $[t_{\text{min}}, t_{\text{split}}]$. If the near node is a leaf with primitives in it,we intersect the ray with the primitives; otherwise we process its children nodes. If no hit is found, or if a hit is found beyond $[t_{\text{min}}, t_{\text{split}}]$, then the far node, on the left, is processed. This sequence continues–processing tree nodes in a depth-first, front-to-back traversal–until the closest intersection is found or the ray exits the tree.

the maximum depth of the kd-tree; the array size used below should be more than enough in practice.

XXX load ray.o into a Point here so compiler knows won't be modified? XXX

⟨*Prepare to traverse kd-tree for ray*⟩≡

```
int rayId = curMailboxId++;
Vector invDir(1.f/ray.d.x, 1.f/ray.d.y, 1.f/ray.d.z);
#define MAX_TODO 64
KdToDo todo[MAX_TODO];
int todoPos = 0;
```

⟨*KdTreeAccel Declarations*⟩+≡

```
struct KdToDo {
    const KdAccelNode *node;
    Float tmin, tmax;
};
```

The traversal continues through the nodes, processing a single leaf or interior node each time through the loop.

XXX uncomment and do no book stats stuff XXX

⟨*Traverse kd-tree nodes in order for ray*⟩≡

```
bool hit = false;
const KdAccelNode *node = &nodes[0];
while (node != NULL) {
    ⟨Bail out if we found a hit closer than the current node⟩
    if (!node->IsLeaf()) {
        ⟨Process kd-tree interior node⟩
    }
    else {
        ⟨Check for intersections inside leaf node⟩
        ⟨Grab next node to process from todo list⟩
    }
}
return hit;
```

KdAccelNode	155
KdAccelNode::IsLeaf()	157
dTreeAccel::curMailboxId	154
KdTreeAccel::nodes	158
Ray::d	35
Ray::maxt	36
Vector	27

An intersection may have been previously found in a primitive that overlaps multiple nodes. If the intersection was outside the current node when first detected, it is necessary to keep traversing the tree until we come to a node where $t_{\min}$ is beyond the intersection; only then is it certain that there is no closer intersection with some other primitive.

⟨*Bail out if we found a hit closer than the current node*⟩≡

```
if (ray.maxt < tmin) break;
```

For interior tree nodes the first thing to do is to intersect the ray with the node's splitting plane and determine if one or both of the children nodes needs to be processed and in what order the ray passes through them.

⟨*Process kd-tree interior node*⟩≡

```
⟨Compute distance along ray to split plane⟩
⟨Get node children pointers for ray⟩
⟨Advance to next child node, possibly enqueue other child⟩
```

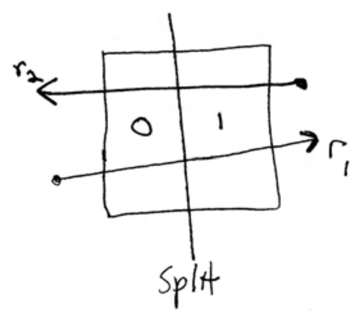


Figure 4.13: The position of the origin of the ray with respect to the splitting plane can be used to determine which of the node's children should be processed first. If a ray like r_1 is "below" side of the splitting plane, we should process the "below" chilld before the "above" child, and vice versa.

The parametric distance to the split plane is computed in the same manner as was done in computing the intersection of a ray and an axis-aligned plane for the ray–bounding box test.

⟨*Compute distance along ray to split plane*⟩≡

```
int axis = node->SplitAxis();
Float tplane = (node->SplitPos() - ray.o[axis]) * invDir[axis];
```

155 KdAccelNode
157 KdAccelNode::SplitAxis()
157 KdAccelNode::SplitPos()
35 Ray::o

Now it is necessary to determine the order the ray encounters the children nodes, so that the tree is traversed in front-to-back order along the ray. Figure 4.13 shows the geometry of this computation. The position of the ray's origin with respect to the splitting plane is enough to distinguish between the two cases, ignoring for now the case where the ray doesn't actually pass through one of the two nodes.

⟨*Get node children pointers for ray*⟩≡

```
const KdAccelNode *firstChild, *secondChild;
int belowFirst = ray.o[axis] <= node->SplitPos();
if (belowFirst) {
    firstChild = node + 1;
    secondChild = &nodes[node->aboveChild];
}
else {
    firstChild = &nodes[node->aboveChild];
    secondChild = node + 1;
}
```

It may not be necessary to process both children of this node. Figure 4.14 shows some configurations where the ray only passes through one of the children that need to be handled. The ray will never miss both children, since otherwise the current interior node should never have been traversed.

The first `if` test in the code below corresponds to the left side of the figure: only the near node needs to be processed if it can be shown that the ray doesn't overlap the far node because it faces away from the far node or doesn't overlap it because $t_{\text{split}} > t_{\text{max}}$. The right side of the figure shows the similar case tested in the second `if` test: the near node may not need processing if the ray doesn't overlap it.

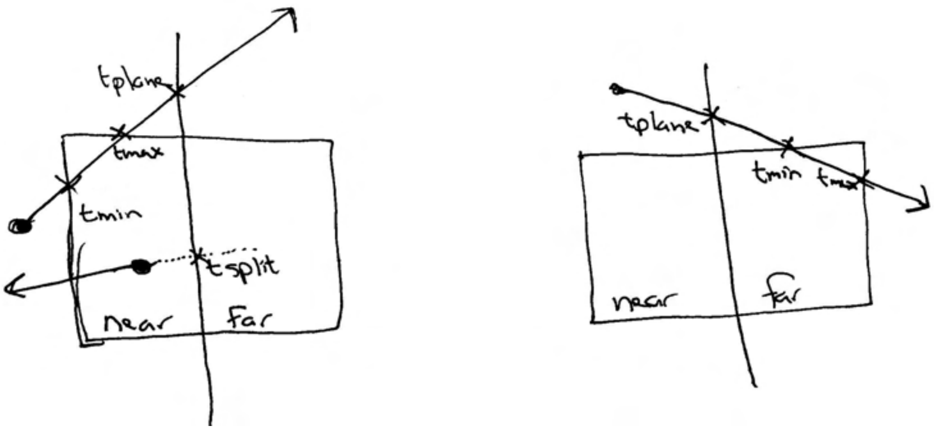


Figure 4.14: Two cases where both children of a node don't need to be processed because the ray doesn't overlap them. On the left, the top ray intersects the splitting plane beyond the ray's t_{max} position and thus doesn't enter the far child. The bottom ray is facing away from the splitting plane, indicated by a negative t_{split} value. On the right, the ray intersects the plane before the ray's t_{min} value, indicating that the near plane doesn't need processing.

KdToDo::node 168
KdToDo::tmax 168
KdToDo::tmin 168

Otherwise, the `else` clause handles the case of both children needing processing; the near node will be processed next and the far node goes on the `todo` list.

⟨*Advance to next child node, possibly enqueue other child*⟩≡

```
if (tplane > tmax || tplane < 0)
    node = firstChild;
else if (tplane < tmin)
    node = secondChild;
else {
    ⟨Enqueue secondChild in todo list⟩
    node = firstChild;
    tmax = tplane;
}
```

⟨*Enqueue* `secondChild` *in todo list*⟩≡

```
todo[todoPos].node = secondChild;
todo[todoPos].tmin = tplane;
todo[todoPos].tmax = tmax;
++todoPos;
```

If the current node is a leaf, intersection tests are performed against the primitives in the leaf, though the mailbox test makes it possible to avoid re-testing primitives that have already been considered for this ray.

⟨*Check for intersections inside leaf node*⟩≡

```
u_int nPrimitives = node->nPrimitives();
if (nPrimitives == 1) {
    MailboxPrim *mp = node->onePrimitive;
    ⟨Check one primitive inside leaf node⟩
}
else {
    MailboxPrim **prims = node->primitives;
    for (u_int i = 0; i < nPrimitives; ++i) {
        MailboxPrim *mp = prims[i];
        ⟨Check one primitive inside leaf node⟩
    }
}
```

Finally, we check the mailbox id of the ray, and call the `Primitive::Intersect()` routine.

⟨*Check one primitive inside leaf node*⟩≡

```
if (mp->lastMailboxId != rayId) {
    mp->lastMailboxId = rayId;
    if (mp->primitive->Intersect(ray, isect))
        hit = true;
}
```

After doing the intersection tests at the leaf node, the next node to process is loaded from the `todo` array. If no more nodes remain, then we know that the ray passed through the tree without hitting anything.

139	`GridAccel`
157	`KdAccelNode::nPrimitives()`
155	`KdAccelNode::onePrimitive`
155	`KdAccelNode::primitives`
168	`KdToDo::node`
168	`KdToDo::tmax`
168	`KdToDo::tmin`
154	`KdTreeAccel`
141	`MailboxPrim`
141	`MailboxPrim::lastMailboxId`
131	`Primitive::Intersect()`
36	`Ray`

⟨*Grab next node to process from todo list*⟩≡

```
if (todoPos > 0) {
    --todoPos;
    node = todo[todoPos].node;
    tmin = todo[todoPos].tmin;
    tmax = todo[todoPos].tmax;
}
else
    break;
```

Like the `GridAccel`, the `KdTreeAccel` has a specialized intersection method for shadow rays which is not shown here. It is largely similar to the `KdTreeAccel::Intersect()` method, just calling `Primitive::IntersectP()` method and returning `true` as soon as it finds any intersection without worrying about finding the closest one.

⟨*KdTreeAccel Public Methods*⟩+≡

```
bool IntersectP(const Ray &ray) const;
```

Further Reading

After the introduction of the ray tracing algorithm, an enormous amount of research was done to try to find effective ways to speed it up, primarily by developing improved ray tracing acceleration structures. Arvo and Kirk's chapter in *An Introduction to Ray Tracing* summarizes the state of the art as of 1989. *Ray Tracing News*, `www.acm.org/tog/resources/RTNews/`, is an excellent resource for general ray tracing information and has particularly useful discussion about implementation issues and tricks of the trade.

Clark first suggested using bounding volumes to cull collections of objects for standard visible-surface determination algorithms (Clark 1976). Building on this work, Rubin and Whitted developed the first hierarchical data structures for scene representation for fast ray tracing (Rubin and Whitted 1980). Weghorst et al's paper discussed the trade-offs of using various shapes for bounding volumes and suggested projecting objects to the screen and using a z-buffer rendering to accelerate eye rays (Weghorst, Hooper, and Greenberg 1984).

Fujimoto et al were the first to intorduce uniform voxel grids, similar to what we describe in this chapter (Fujimoto, Tanaka, and Iwata 1986). Snyder and Barr described a number of key improvements to this approach, and showed their use for rendering extremely complex scenes (Snyder and Barr 1987). Hierarchical grids were first described by Jevans and Wyvill (Jevans and Wyvill 1989). More recent techniques for hierarchical grids were developed by Cazals et al and Klimaszewski and Sederberg (Cazals, Drettakis, and Puech 1995; Klimaszewski and Sederberg 1997).

`KdTreeAccel 154`

Glassner introduced the use of octrees for ray intersection acceleration (Glassner 1984); this approach was more robust to scenes with non-uniform distributions of geometry. The kd-tree was first described by Kaplan (**?**). Kaplan's tree construction algorithm always split nodes down their middle. A better approach for building trees and the basis for the method used in the `KdTreeAccel` was intruced by MacDonald and Booth (MacDonald and Booth 1990), who estimated ray–node traversal probabilities using relative surface areas. Naylor has also written on general issues of constructing good kd-trees (Naylor 1993). Havran and Bittner (Havran and Bittner 2002) have recently revisited many of these issues and introduced some useful improvements. Adding a bonus factor for tree nodes that are completely empty was suggested by Hurley et al (Hurley, Kapustin, Reshetov, and Soupikov 2002).

Jansen first described the efficient ray traversal algorithm for kd-trees (Jansen 1986); Arvo has also investigated these issues (Arvo 1988). Sung and Shirley describe a ray traversal algorithm's implementation for a BSP-tree accelerator (Sung and Shirley 1992); our `KdTreeAccel` traversal code is loosely based on theirs.

An early object subdivision approach was the hierarchial bounding volumes of Goldsmith and Salmon (Goldsmith and Salmon 1987). They also were the first to introduce techniques for estimating the probability of a ray intersecting a bounding volume based on the volume's surface area.

Arnaldi et al and Amanatides and Woo came up with mailboxing (Arnaldi, Priol, and Bouatouch 1987; Amanatides and Woo 1987).

Kay Kajiya (Kay and Kajiya 1986).

Arvo and Kirk 5D position direction subdivision (Arvo and Kirk 1987).

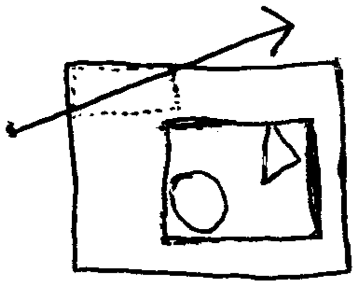


Figure 4.15: If a bounding box of the overlapping geometry is stored in each voxel for fast rejection of unnecessary ray–primitive intersection tests, an alternative to checking for ray–bounding box intersection is to find the bounding box of the ray inside the voxel (shown here with a dashed line) and test to see if that overlaps the geometry bound.

Kirk and Arvo introduced the unifying principle of *meta-hierarchies* (Kirk and Arvo 1988); they showed that by implementing acceleration data structures to conform to the same interface as is used for primitives in the scene, it's easy to mix and match multiple intersection schemes in a scene without needing to have particular knowledge of it.

Smits on fast ray–box intersection, general issues of efficient ray tracing (Smits 1998).

Papers by Woo, Pearce, etc. with additional clever tricks

Exercises

4.1 Try using bounding box tests to improve the grid's performace: inside each grid voxel, store the bounding box of the geometry that overlaps the voxel. Use this bounding box to quickly skip intersection tests with geometry if the ray doesn't intersect the bound. Develop criteria based on the number of primitives in a voxel and the size of their bound with respect to the voxel's bound to only do the bounding box tests for voxels where doing so is likely to improve performance. When is this extra work worthwhile?

4.2 Rather than computing a ray–bounding box intersection for the technique described in the previous exercise, it can be more efficient to check to see if the bounding box of a ray's implement ray bound in each voxel; then check for overlap of ray bound with world bound of the objects first–very cheap test...

4.3 Rewrite the ⟨*Find* `stepAxis` *for stepping to next voxel*⟩ fragment to compute the stepping axis in the obvious way (with a few comparisons and branches). Evaluate the performance benefit of `lrt`'s table-based approach on several CPU's. What do the results tell you about the architecture of each CPU? How do your results vary if you compare 4 or 5 numbers instead of just 3?

4.4 Generalize the grid implementation in this chapter to be hierarchical: refine voxels that have an excessive number of primitives overlapping them to instead hold a finer sub-grid to store its geometry. (See for example Jevans and Wyvill's paper for a basic approach to this problem (Jevans and Wyvill 1989).)

4.5 Develop a more complex hierarchial grid implementation, following the approach of either Cazals et al (Cazals, Drettakis, and Puech 1995) or Klimaszewski and Sederberg (Klimaszewski and Sederberg 1997). How does it compare to hierarchical grids based on Jevans and Wyvill's approach?

4.6 Implement a primitive list "accelerator" that just stores an array that holds all of the primitives and loops over all of them for every intersection test. How much does using this accelerator make the system slow down? Is this accelerator ever faster than the `GridAccel` or `KdTreeAccel`? Describe a contrived example where the primitive list would be faster than a grid or kd-tree even for a complex scene.

GridAccel 139
KdTreeAccel 154

4.7 Implement smarter overlap tests for building accelerators. Using objects' bounding boxes to determine which grid cells and which sides of a kd-tree split they overlap can hurt the performance of the accelerators and cause unnecessary intersection tests. (Recall Figure 4.5.) Add a `bool Shape::Overlaps(const BBox &) const` method to the shape interface that takes a world-space bounding box and determines if the shape truly overlaps the given bound. A default implementation could get the world bound from the shape and use that for the test and specialized versions could be written for frequently-used shapes. Implement this method for `Sphere`s and `Triangle`s and modify the accelerators to call it. Measure the change in `lrt`'s performance.

4.8 Fix the `KdTreeAccel` so that it doesn't always immediately refine all primitives before building the tree. For example, one approach is to build additional kd-trees as needed, storing these sub-trees in the hierarchy where the original unrefined primitive was. Implement this approach, or come up with a better technique to address this problem and measure the change in running time and memory use for a variety of scenes.

4.9 Investigate alternative cost functions for building kd-trees for the `KdTreeAccel`. How much can a poorly cost function hurt its performance? How much improvement can be had compared to the current one?

4.10 Geometry caching: hold limited amount in memory, discard as needed and call `Primitive::Refine()` later if geometry is needed again. LRU scheme...

4.11 The grid and kd-tree accelerators both take a 3D region of space, subdivite it into cells, and record which primitives overlap each cell. Hierarchical bounding volumes (HBVs) appraoch the problem in a different way, starting with all of the primitives and progressively partitioning them into smaller spatially-nearby subsets. This process gives a hierarchy of primitives... XXX

The top node of the hierarchy holds a bound that encompasses all of the primitives in the scene (see Figure 4.16). It has two or more children nodes,

We seem to be missing this figure.

Figure 4.16: The Hierarchical bounding figure?

each of which bounds a subset of the scene. This continues recursively until the bottom of the tree, at which point the bound around a single primitive is stored. Read Goldsmith and Salmon's paper about building HBV hierarchies and implement their approach as an `Accelerator` in `lrt`. Compare its performance against the grid and kd-tree accelerators.

4.12 Meta hierarchies: The idea of using spatial data structures can be generalized to include spatial data structures that themselves hold other spatial data structures, rather than just primitives. Not only could we have a grid that has sub-grids inside the grid cells that have many primitives in them (thus partially solving the adaptive refinement problem), but we could also have the scene organized into a HBV where the leaf nodes are grids that hold smaller collections of spatially-nearby primitives. Such hybrid techniques can bring the best of a variety of spatial data structure-based ray intersection acceleration methods. In `lrt`, because both geometric primitives and intersection accelerators inherit from the `Primitive` base class and thus provide the same interface, it's easy to mix and match in this way.

`130 Primitive`

4.13 Disable the mailbox test in the grid or kd-tree accelerator and measure how much `lrt` slows down when rendering various scenes. How effective is mailboxing? How many redundant intersection tests are performed without it? One alternative to mailboxing is to update the rays $[t_{\min}, t_{\max}]$ range for the accelerator cell that it is currently in, so that the primitives will ignore intersections outside that range and may be able to avoid performing a complete intersection test if no intersection is possible in the current range. How does the performance of that approach compare to mailboxing?

4.14 There is a subtle bug in the mailboxing schemes for both the grid and the kd-tree that may cause intersections to be missed after a few billion rays have been traced. Describe a scenario where this might happen and suggest how this bug could be fixed. How likely is this bug to cause an incorrect result to be returned by the accelerator?

5. Color and Radiometry

181 Spectrum

In order to describe how light is represented and sampled to compute images, we will first establish some background in *radiometry*. Radiometry is the area of study of the propagation of electromagnetic radiation in environments. The wavelengths of electromagnetic radiation between (approximately) 370nm and 730nm account for light visible to the human visual system and are of particular interest in rendering. The lower wavelengths, $\lambda \approx 400$nm are the blue-ish colors, the middle wavelengths $\lambda \approx 550$nm are the greens, and the upper wavelengths $\lambda \approx 650$nm are the reds.

We will introduce four key radiometric quantities–flux, intensity, irradiance, and radiance–that describe electromagnetic radiation. By evaluating the amount of radiation arriving on the camera's image plane, we can accurately model the process of image formation. These radiometric quantities generally vary according to wavelength, and are described by a *spectral power distribution* (SPD), which is a function of wavelength, λ. This chapter starts by describing the `Spectrum` class that `lrt` uses to represent SPDs. We will then introduce basic concepts of radiometry and some theory behind light scattering from surfaces.

For now, we will ignore the effects of smoke, fog, and all other atmospheric phenomena and assume that the scene is a collection of surfaces in a vacuum. These restrictions will be relaxed in Chapter 12.

5.1 Spectral Representation

```
⟨color.h*⟩≡
  #include "lrt.h"
  ⟨Spectrum Declarations⟩
```

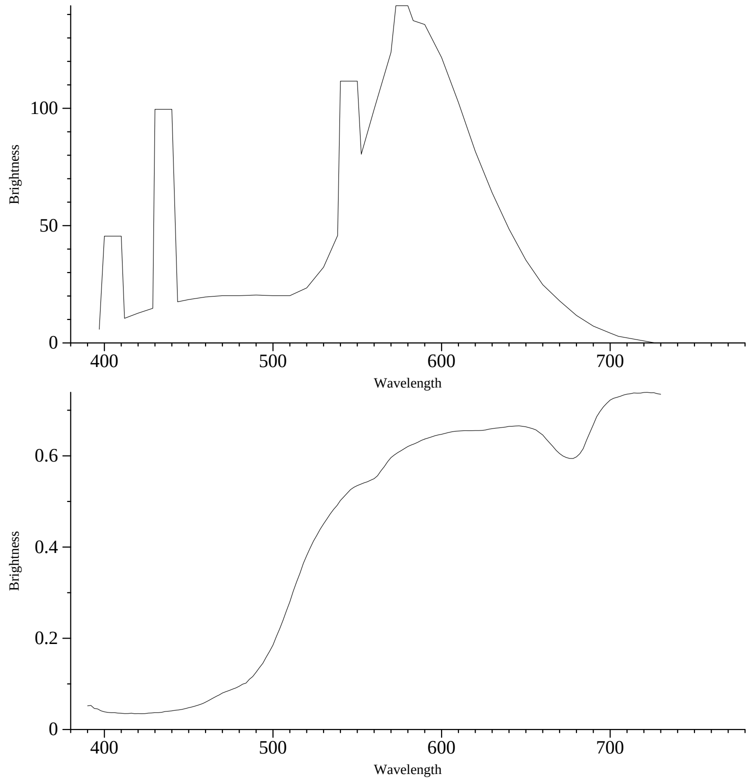


Figure 5.1: Spectral power distributions of a fluorescent light (top) and the reflectance of lemon skin (bottom). Wavelengths around 400nm are blue-ish colors, greens and yellows are in the middle range of wavelengths, and reds have wavelengths around 700nm. The fluorescent light's SPD is even spikier than shown here, where the SPDs have been binned into 10nm ranges; it emits much of its illumination at single frequencies.**The y-axis of the lemon graph is labeled wrong, and the text is REALLY small.**

```
⟨color.cpp*⟩≡
  #include "color.h"
  ⟨Spectrum Method Definitions⟩
```

The SPDs of real-world objects can be quite complicated; Figure 5.1 shows a graph of the spectral distribution of emission from a fluorescent light and the spectral distribution of the reflectance of lemon skin. Given such functions, we would like a compact, efficient, and accurate way to represent them. A number of approaches have been developed that are based on finding good *basis functions* to represent SPDs. The idea behind basis functions is to map the infinite-dimensional space of possible SPD functions to a low-dimensional space of coefficients $c_i \in \mathbb{R}$. For example, a trivial basis function is the constant function $B(\lambda) = 1$. An arbitrary SPD would be represented by a single coefficient c equal to its average value, so that its basis function approximation would be $cB(\lambda) = c$. This is obviously a poor approximation, since it has no chance to account for the SPD's possible complexity.

It is often convenient to limit ourselves to *linear basis functions*. This means that the basis functions are pre-determined functions of wavelength and aren't themselves parameterized. For example, if we were using Gaussians as basis functions and wanted to have a linear basis, we need to set their respective widths and central wavelengths ahead of time. If we allowed the widths and center positions to vary based on the SPD we were trying to fit, we would be performing non-linear approximation. Though non-linear basis functions can naturally adapt to the complexity of SPDs, they tend to be less computationally efficient. Also, the theory of non-linear approximation is very difficult, and even an introduction would be beyond the scope of this book. Because it is not a primary goal of `lrt` to provide the most comprehensive spectral representations, we will only implement infrastructure for linear basis functions.

Given a set of linear basis functions B_i, coefficients c_i for a SPD $S(\lambda)$ can be computed by

$$c_i = \int_\lambda B_i(\lambda) S(\lambda) \,\mathrm{d}\lambda, \tag{5.1.1}$$

so that

$$S(\lambda) \approx \sum_i c_i B_i(\lambda).$$

Measured SPDs of real-world objects are often given in 10nm increments; this corresponds to a step-function basis:

$$B(\lambda)_{a,b} = \begin{cases} 1 & : \quad a \le \lambda < b \\ 0 & : \quad \text{otherwise} \end{cases}$$

Another common basis function is the delta function that evaluates the SPD at single wavelengths. Others that have been investigated include polynomials and Gaussians.

Given an SPD and its associated set of linear basis function coefficients, a number of operations on the spectral distributions can be easily expressed directly in terms of the coefficients. For example, to compute the coefficients c_i' for the SPD given by multiplying a scalar k with a SPD $S(\lambda)$, where the coefficients for $S(\lambda)$ are c_i, we have:

$$\begin{aligned} c_i' &= \int_\lambda B_i(\lambda)(kS(\lambda)) \,\mathrm{d}\lambda \\ c_i' &= k \int_\lambda B_i(\lambda) S(\lambda) \,\mathrm{d}\lambda \\ c_i' &= k c_i \end{aligned}$$

Such a multiplication might be used to adjust the brightness of a light source. Similarly, for two SPDs $S_1(\lambda)$ and $S_2(\lambda)$ represented by coefficients c_i^1 and c_i^2 **I don't like numerical superscripts; they're too confusing and look like powers**, the sum $S_1(\lambda) + S_2(\lambda)$ can be shown to be

$$c_i' = \sum c_i^1 + c_i^2.$$

Thus, by converting to a basis function representation, a number of otherwise potentially-tricky operations with SPDs are made straightforward.

We will often need to multiply two SPDs together. For example, the product of the SPD of light arriving at a surface with the SPD of the surface's reflectance gives the SPD of light reflected from the surface. In general, the coefficients for the SPD representing the product of two SPDs doesn't work out so cleanly, even with linear basis functions:

$$\begin{aligned} c_i &= \int_\lambda B_i(\lambda)\,(S_1(\lambda)S_2(\lambda))\,\mathrm{d}\lambda \\ &\approx \int_\lambda B_i(\lambda)\left(\sum_j c^1_j B_j(\lambda)\right)\left(\sum_k c^2_k B_k(\lambda)\right)\mathrm{d}\lambda \\ &= \sum_j\sum_k c^1_j c^2_k \int_\lambda B_i(\lambda)B_j(\lambda)B_k(\lambda)\mathrm{d}\lambda \end{aligned}$$

The integrals of the product of the three basis functions can be precomputed and stored in n matrices of size n^2 each, where n is the number of basis functions. Thus, n^3 multiplications are necessary to compute the new coefficients. Alternatively, If one of the colors is known ahead of time (e.g. a surface's reflectance), we can precompute an matrix S defined so that the $S_{i,j}$ element is

$$S_{i,j} = \int_\lambda S_1(\lambda)B_i(\lambda)B_j(\lambda).$$

Then, multiplication with another SPD is just a matrix-vector multiply with S and the vector c^2_i, requiring n^2 multiplications.

`Spectrum 181`

In `lrt`, we will choose computational efficiency over generality and further limit the supported basis functions to be orthonormal. This means that for $i \neq j$,

$$\int_\lambda B_i(\lambda)B_j(\lambda)\mathrm{d}\lambda = 0$$

and

$$\int_\lambda B_i(\lambda)B_i(\lambda)\mathrm{d}\lambda = 1.$$

Under these assumptions, the coefficients for the product of two SPDs is just the product of their coefficients

$$c_i = c^1_i c^2_i,$$

requiring only n multiplications.

XXX need to note, though, that the coefficients for the product of two SPDs will not in general have the same values as the products of their coefficients:

Other than requiring that the basis functions used be linear and orthonormal, `lrt` places no further restriction on them. In fact, `lrt` operates purely on basis function coefficients: colors are specified in input files and texture maps as coefficients and `lrt` can write out images of coefficients–almost no knowledge of the particular basis functions being used is needed.

5.1.1 Spectrum Class

The `Spectrum` class holds a compile-time fixed number of basis function coefficients, given by `COLOR_SAMPLES`.

⟨*Global Constants*⟩+≡

```
#define COLOR_SAMPLES 3
```

⟨*Spectrum Declarations*⟩≡

```
class Spectrum {
public:
    ⟨Spectrum Public Methods⟩
    ⟨Spectrum Public Data⟩
private:
    ⟨Spectrum Private Data⟩
};
```

⟨*Spectrum Private Data*⟩≡

```
Float c[COLOR_SAMPLES];
```

Two `Spectrum` constructors are provided, one initializing a spectrum with the same value for all coefficients, and one initializing it from an array of coefficients.

⟨*Spectrum Public Methods*⟩≡

```
Spectrum(Float intens = 0.) {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        c[i] = intens;
}
```

⟨*Spectrum Public Methods*⟩+≡

180 COLOR_SAMPLES

```
Spectrum(Float cs[COLOR_SAMPLES]) {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        c[i] = cs[i];
}
```

A variety of arithmetic operations on `Spectrum` objects are supported; the implementations are all quite straightforward. First are operations to add pairs of spectral distributions.

⟨*Spectrum Public Methods*⟩+≡

```
Spectrum &operator+=(const Spectrum &s2) {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        c[i] += s2.c[i];
    return *this;
}
```

⟨*Spectrum Public Methods*⟩+≡

```
Spectrum operator+(const Spectrum &s2) const {
    Spectrum ret = *this;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        ret.c[i] += s2.c[i];
    return ret;
}
```

Similarly, subtraction, multiplication and division of spectra is defined component-wise. We won't include all of the code for those cases, or for multiplying or dividing them by scalar values, since there's little additional value to seeing it all.

this text needs work

While this method is redundant given the operators defined so far, for performance critical sections of code where one would like to update a `Spectrum` with a weighted value of another `Spectrum`, (`s = w*s2;`), the `AddWeighted()` method can do the same computation more efficiently. Many compilers are not able to optimize the computation as well if it's written using the operators above, since they lead to the creation of a temporary `Spectrum` to hold the product, which is then assigned to the result.

⟨*Spectrum Public Methods*⟩+≡

```
void AddWeighted(Float w, const Spectrum &s) {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        c[i] += w * s.c[i];
}
```

We also provide the obvious equality test.

⟨*Spectrum Public Methods*⟩+≡

```
bool operator==(const Spectrum &sp) const {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        if (c[i] != sp.c[i]) return false;
    return true;
}
```

We frequently want to know if a spectrum is "black". If, for example, a surface has zero reflectance, we can avoid casting reflection rays that will eventually be multiplied by zeroes.

`COLOR_SAMPLES`	180
`Spectrum`	181
`Spectrum::c`	181

⟨*Spectrum Public Methods*⟩+≡

```
bool Black() const {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        if (c[i] != 0.) return false;
    return true;
}
```

Also useful are functions that take the square root of a spectrum or raise the components of a `Spectrum` to a given power (note that the power is also given as a `Spectrum`, to allow component-wise powers **Pow() is not used, do we need it?**). Because the product of two spectra is computed with products of their coefficients, taking the square root of the coefficients gives the square root of the SPD. The square root of a spectrum is used to approximate Fresnel phenomena in Chapter 9.

⟨*Spectrum Public Methods*⟩+≡

```
Spectrum Sqrt() const {
    Spectrum ret;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        ret.c[i] = sqrtf(c[i]);
    return ret;
}
```

⟨*Spectrum Public Methods*⟩+≡

```
Spectrum Pow(const Spectrum &e) const {
    Spectrum ret;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        ret.c[i] = c[i] > 0 ? powf(c[i], e.c[i]) : 0.f;
    return ret;
}
```

And for volume rendering...

⟨*Spectrum Public Methods*⟩+≡
```
Spectrum operator-() const;
friend Spectrum Exp(const Spectrum &s);
```

Some portions of the image-processing pipeline will want to clamp a spectrum to ensure that its coefficients are within some allowable range.

⟨*Spectrum Public Methods*⟩+≡
```
Spectrum Clamp(Float low, Float high) const {
    Spectrum ret;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        ret.c[i] = ::Clamp(c[i], low, high);
    return ret;
}
```

Finally, we provide a useful debugging routine to check if any of the coefficients of an SPD is NaN. This frequently happens when code accidentaly divides by zero.

⟨*Spectrum Public Methods*⟩+≡
```
bool IsNaN() const {
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        if (isnan(c[i])) return true;
    return false;
}
```

677	Clamp()
180	COLOR_SAMPLES
181	Spectrum
181	Spectrum::c

5.1.2 XYZ Color

A remarkable property of the human visual system makes it possible to represent colors with just three floating-point numbers. The *tristimulus theory* of color perception says that all visible SPDs can be accurately represented for human observers with three values, x_λ, y_λ, and z_λ. Given a SPD $S(\lambda)$, these values are computed by convolving it with the *spectral matching curves*, $X(\lambda)$, $Y(\lambda)$ and $Z(\lambda)$:

$$\begin{aligned} x_\lambda &= \int_\lambda S(\lambda)\, X(\lambda) \mathrm{d}\lambda \\ y_\lambda &= \int_\lambda S(\lambda)\, Y(\lambda) \mathrm{d}\lambda \\ z_\lambda &= \int_\lambda S(\lambda)\, Z(\lambda) \mathrm{d}\lambda. \end{aligned}$$

These curves were determined by the Commission Internationale de l'Éclairge (CIE) standards body after a series of experiments with human test subjects. and are graphed in Figure 8.5. It is believed that these matching curves are generally similar to the responses of the three types of color-sensitive cones in the human retina. Remarkably, SPDs with substantially different distributions may have very similar x_λ, y_λ, and z_λ values. To the human observer, such SPDs actually appear the same visually. Pairs of such spectra are called *metamers*.

This brings us to a subtle point about color spaces and spectral power distributions. Most color spaces attempt to model colors that are visible to humans, and therefore use only three coefficients, exploiting the tristimulus theory of color perception. Although XYZ works well to represent a given SPD to be displayed for a human observer, it is *not* a particularly good set of basis functions for spectral computation. For example, though XYZ values would work well to describe the perceived color of lemon-skin or a fluorescent light individually (recall Figure 5.1, which graphs these two SPDs), the product of their respective XYZ values is likely to give a noticeably different XYZ color than the XYZ value computed by multiplying more accurate representations of their SPDs and *then* computing the XYZ value.

With that in mind, we will add a method to the `Spectrum` class that returns the XYZ values for its SPD. It turns out that when converting a spectrum described by basis function coefficients in one basis to another basis, the new basis function coefficients can be written as weighted sums of the old basis function coefficients. For example, for x_λ,

$$\begin{aligned} x_\lambda &= \int S(\lambda) X(\lambda) \mathrm{d}\lambda \\ &\approx \int_\lambda \left(\sum_i c_i B_i(\lambda) \right) X(\lambda) \mathrm{d}\lambda \\ &= \sum_i c_i \left(\int_\lambda B_i(\lambda) X(\lambda) \mathrm{d}\lambda \right) \\ &= \sum_i c_i w_{x,i}. \end{aligned}$$

`COLOR_SAMPLES` 180
`Spectrum` 181

Thus, the weight values $w_{x,i}$, $w_{y,i}$ and $w_{z,i}$ can be precomputed and stored in an array for whatever particular basis functions are being used. The `Spectrum::XYZ()` method uses these arrays to return the spectrum's XYZ representation.

⟨*Spectrum Public Methods*⟩+≡
```
void XYZ(Float xyz[3]) const {
    xyz[0] = xyz[1] = xyz[2] = 0.;
    for (int i = 0; i < COLOR_SAMPLES; ++i) {
        xyz[0] += XWeight[i] * c[i];
        xyz[1] += YWeight[i] * c[i];
        xyz[2] += ZWeight[i] * c[i];
    }
}
```

Therefore, we now finally need to settle on the default set of SPD basis functions for `lrt`. Though not sufficient for high-quality spectral computations, an expedient choice is to use the spectra of standard red, green, and blue phosphors for televisions and CRT display tubes. A standard set of these RGB spectra has been defined for high-definition television; the weights to convert from these RGBs to XYZ values are below:

this sucks. If the user changes COLOR_SAMPLES, it should just work. Should we actually go ahead and do spectral rendering? How much work would that be? This is a pretty non-physically based part of LRT right here. Also the LRT input should be able to convert any given input color data into its own internal representation. Can the LRT input take RGB, XYZ, and/or sampled spectral data and use it out of the box?

⟨*Spectrum Method Definitions*⟩+≡

```
Float Spectrum::XWeight[COLOR_SAMPLES] = {
    0.412453f, 0.357580f, 0.180423f
};
Float Spectrum::YWeight[COLOR_SAMPLES] = {
    0.212671f, 0.715160f, 0.072169f
};
Float Spectrum::ZWeight[COLOR_SAMPLES] = {
    0.019334f, 0.119193f, 0.950227f
};
```

For convenience in computing values for `XWeight`, `YWeight` and `ZWeight` for other spectral basis functions, we will also provide the values of the standard $X(\lambda)$, $Y(\lambda)$, and $Z(\lambda)$ response curves sampled at 1nm increments from 360nm to 830nm.

⟨*Spectrum Public Data*⟩≡

```
static const int CIEstart = 360;
static const int CIEend = 830;
static const int nCIE = CIEend-CIEstart+1;
static const Float CIE_X[nCIE];
static const Float CIE_Y[nCIE];
static const Float CIE_Z[nCIE];
```

180 COLOR_SAMPLES
181 Spectrum
181 Spectrum::c

The *y* coordinate of the XYZ color is closely related to *luminance*, which measures the percieved brightness of a color. (Luminance is discussed in more detail in Section 8.3.1.) For the convenience of methods there, we will provide a method to compute it alone in a separate utility method.

⟨*Spectrum Public Methods*⟩+≡

```
Float y() const {
    Float v = 0.;
    for (int i = 0; i < COLOR_SAMPLES; ++i)
        v += YWeight[i] * c[i];
    return v;
}
```

The *y* coordinate also gives a convenient way to order `Spectrum` instances from dark to bright.

⟨*Spectrum Public Methods*⟩+≡

```
bool operator<(const Spectrum &s2) const {
    return y() < s2.y();
}
```

5.2 Basic Radiometry

Radiometry gives us a set of ideas and mathematical tools to describe light propagation and reflection in environments; it forms the basis of the derivation of the rendering algorithms that will be used throughout the rest of this book. Interestingly enough, radiometry wasn't originally derived from first principles using the basic physics of light, but was based on an abstraction of light based on particle flows. As such, effects like polarization of light do not naturally fit into radiometry, though connections have since been made between radiometry and Maxwell's equations, giving it a solid basis in physics.

Radiative transfer is the phenomenological study of the transfer of radiant energy. It is based on radiometric principles and operates at the *geometrical optics* level, where macroscopic properties of light suffice to describe how light interacts with objects much larger than the light's wavelength. It is not uncommon to incorporate results from wave optics models, but these results need to be expressed in the language of radiative transfer's basic abstractions.[1] In this manner, it is possible to describe interactions of light with objects whose size is close to the wavelength of the light and thereby model effects like dispersion and interference. At an even finer level of detail, quantum mechanics is needed to describe light's interaction with atoms. Fortunately, direct simulation of quantum mechanical principles is unnecessary for solving rendering problems in computer graphics, so the intractability of such an approach is avoided.

In `lrt`, we will assume that geometric optics is an adequate model for the description of light and light scattering. This leads to a few assumptions about the behavior of light:

- *Linearity*: the combined effect of two inputs to an optical system is always equal to the sum of the effects of each of the inputs individually.

- *Energy conservation*: more energy is never produced by a scattering event than there was to start with.

- *No polarization*: we will ignore polarization of the electromagnetic field; as such, the only relevant property of light particles is their wavelength (or frequency). While the radiative transfer framework has been extended to include the effects of polarization, we will ignore this effect for simplicity.

- *No fluorescence or phosphorescence*: the behavior of light at one wavelength is completely independent of light's behavior at other wavelengths. As with polarization, it is not too difficult to include these effects, but they would add little practical value to our system.

- *Steady state*: light in the environment is assumed to have reached equlibrium, so its radiance distribution isn't changing over time. This happens nearly instantaneously with light in realistic scenes.

The most significant loss from assuming geometric optics is that diffraction and interference effects cannot easily be accounted for. As noted by Preisendorfer, this is hard to fix given these assumptions because, for example, the total flux over two areas isn't necessarily equal to sum of flux over each individually (Preisendorfer 1965, p. 24).

[1] Preisendorfer has connected radiative transfer theory to Maxwell's classical equations describing electromagnetic fi elds (Preisendorfer 1965, Chapter 14); his framework both demonstrates their equivalence and makes it easier to apply results from one world-view to the other. More recent work was done in this area by Fante (Fante 1981).

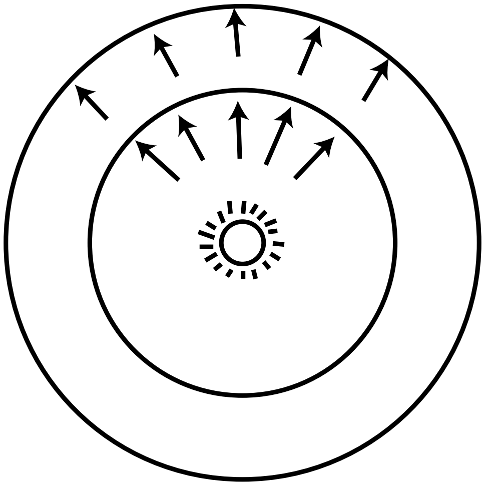


Figure 5.2: Radiant flux, Φ, measures energy passing through a surface or region of space. Here, flux from a point light source is being measured at a sphere that surrounds the light.

5.2.1 Basic quantities

There are four radiometric quantities that are central to rendering:

- Flux
- Irradiance
- Intensity
- Radiance

All of these quantities are generally functions of wavelength. For the remainder of this chapter, we will not make this dependence explicit, but it is important to keep in mind.

Radiant flux, also known as *power*, is the total amount of energy passing through a surface or region of space per unit time. Its units are $\frac{J}{s}$ (more commonly "Watts") and it is normally signified by the symbol Φ. Total emission from light sources is generally described in terms of flux; Figure 5.2 shows flux from a point light measured by the total amount of energy passing through the imaginary sphere around the light. Note that the amount of flux measured on either of the two spheres in Figure 5.2 is the same–although less energy is passing through any local part of the large sphere than the small sphere, the greater area of the large sphere accounts for this.

Irradiance (E) is the area density of flux, $\frac{W}{m^2}$. For the point light example in Figure 5.2, irradiance on the outer sphere is less than the irradiance on the inner sphere, since the area on the outer sphere is larger. In particular, for a sphere in this configuration that has radius r,

$$E = \frac{\Phi}{4\pi r^2}.$$

This explains why received energy from a light falls off with the squared distance from the light.

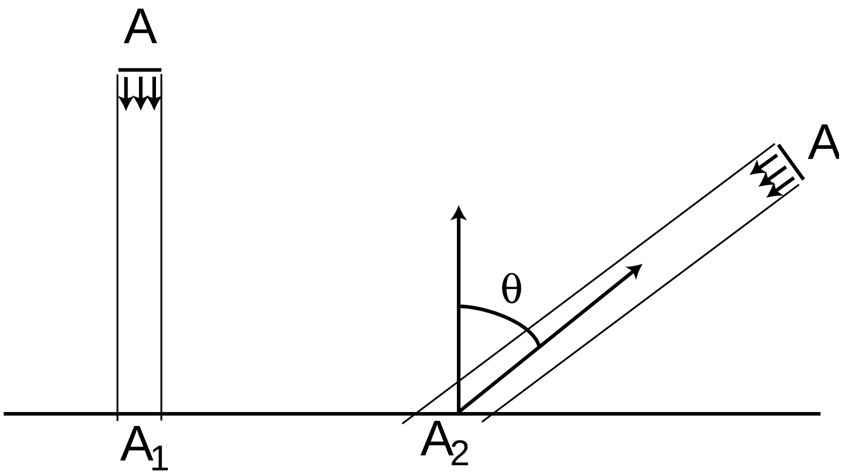


Figure 5.3: Irradiance (E) arriving at a surface varies according to the cosine of the angle of incidence of illumination, since illumination is over a larger area at lower incident directions. This effect was first described by Lambert; it is known as Lambert's Law.

The irradiance equation can also help us understand the origin of *Lambert's Law*, which says that the amount of light arriving at a surface is related to the cosine of the angle between the light direction and the surface normal–see Figure 5.3. Consider a light source with area A and flux Φ that is shining on a surface. If the light is shining directly down on the surface (left), then the area on the surface receiving light A_1 is equal to A and irradiance at any point inside A_1 is

$$E_1 = \frac{\Phi}{A}.$$

However, if the light is at an angle to the surface (right), the total area on the surface receiving light is larger. If the area of the light source is small, then the area receiving flux, A_2, is roughly $A/\cos\theta$. For points inside A_2, the irradiance is therefore

$$E_2 = \frac{\Phi \cos\theta}{A}.$$

This is the origin of the cosine law for radiance.

More formally, to cover the cases like when the emitted flux distribution isn't constant, irradiance at a point is actually defined as

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A}, \tag{5.2.2}$$

where the differential flux from the light is computed over a differential area receiving flux.

In order to define the radiometric quantity intensity, we first need to define the notion of the *solid angle*. Solid angles are just the extension of two-dimensional angles in a plane to angle on a sphere. The *plane angle* is the total angle subtended by some object with respect to some position; see Figure 5.4. Consider the unit circle around the point p; if we project the shaded object on to that circle, some

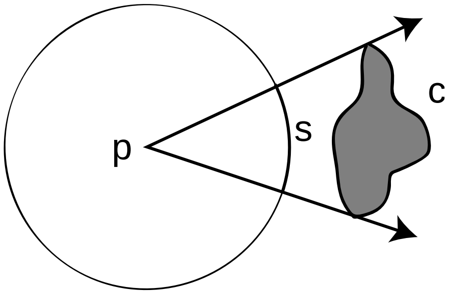


Figure 5.4: The plane angle of an object *c* as seen from a point *p* is equal to the angle it subtends as seen from p, or equivalently as the length of the arc *s* on the unit sphere.

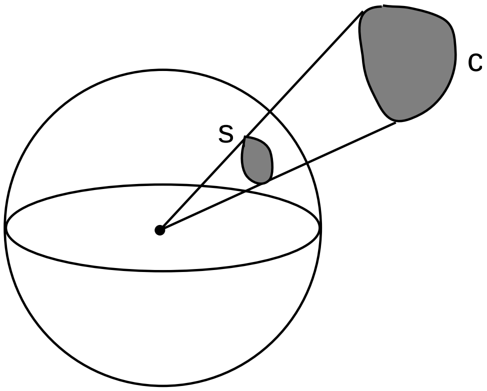


Figure 5.5: The solid angle *s* subtended by an object *c* in three dimensions is similarly computed by projecting *c* onto the unit sphere and measuring its area there.

length of the circle *s* will be covered by its projection. The arc-length of *s* (which is the same as the angle θ) is the angle *subtended* by the object. Plane angle is given the unit *radians*.

The solid angle extends the 2D unit circle to a 3D unit sphere (Figure 5.5). The total area *s* is the solid angle subtended by the object. Solid angle is given the unit *steradians*. The entire sphere subtends a solid angle of 4π and a hemisphere subtends 2π.

We will use the symbol ω to describe directions on the unit sphere centered around some point. (These directions can also be thought of as point on the unit sphere around p. We will therefore use the convention that ω is always a normalized vector). We can now define intensity, which is flux density per solid angle,

$$I = \frac{\mathrm{d}\Phi}{\mathrm{d}\omega}. \tag{5.2.3}$$

Intensity is generally only used when describing the distribution of light by direction from point light sources.

Finally, radiance (L) is the flux density per unit area, per unit solid angle. In terms of flux, it is

$$L = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\omega \, dA^{\perp}} \tag{5.2.4}$$

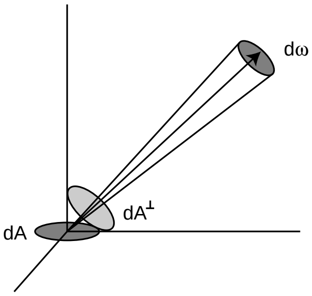


Figure 5.6: Radiance L is defined at a point by the ratio of the differential flux incident along a direction ω to the differential solid angle dω times the differential projected area of the receiving point.

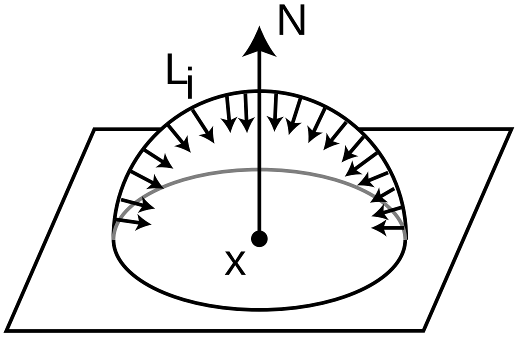


Figure 5.7: Irradiance at a point p is given by the integral of radiance times the cosine of the incident direction over the entire upper hemisphere above the point.

where $dA^{\perp}$ is the projected area of dA on a hypothetical surface perpendicular to ω–see Figure 5.6. All those differential terms don't need to be as confusing as they initially appear–just think of radiance as the limit of the measurement of incident light at the surface as a small cone of incident directions of interest dω becomes very small, and as the local area of interest on the surface dA also becomes very small.

Now that we have defined these various units, it's easy to derive relations between them. For instance, irradiance at a point p due to radiance over a set of directions Ω is

$$E(\mathrm{p}) = \int_{\Omega} L(\mathrm{p}, \omega) \cos\theta \, d\omega, \tag{5.2.5}$$

where $L(\mathrm{p}, \omega)$ denotes the arriving radiance at position p as seen along direction ω (see Figure 5.7). (The $\cos\theta$ term in this integral is due to the $dA^{\perp}$ term in the definition of radiance.) We are often interested in irradiance over the hemisphere of directions about a given surface normal $\mathbf{n}$, $\mathcal{H}^2(\mathbf{n})$ or the entire sphere of directions $\mathcal{S}^2$.

5.3 Working with Radiometric Integrals

One of the main tasks in rendering is integrating information about the values of particular radiometric quantities to compute information about other radiometric

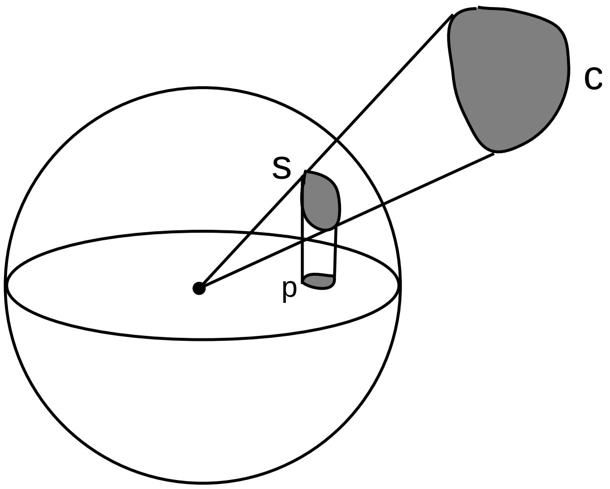


Figure 5.8: The projected solid angle subtended by an object *c* is the cosine-weighted solid angle that it subtends. It can be computed by finding the object's solid angle *s*, projecting it down to the plane, and measuring its area there. Thus, the projected solid angle depends on the surface normal where it is being measured, since the normal orients the plane of projection.

quantities. There are a few important tricks that can be used to make this task easier.

5.3.1 Integrals over projected solid angle

The various cosine terms in integrals for radiometric quantities can clutter things up and distract from what is being expressed in the integral. There is an different way that the integrals can be written that removes this distraction. The *projected solid angle* subtended by an object is determined by projecting the object on to the unit sphere, as is done for solid angle, but then projecting the resulting shape down on to the unit disk–see Figure 5.8. Integrals over hemispheres of directions with respect to solid angle can equivalently be written as integrals over projected solid angles.

The projected solid angle measure is related to the solid angle measure by

$$\mathrm{d}\omega^{\perp} = \cos\theta\,\mathrm{d}\omega,$$

so the irradiance-from-radiance integral can be written more simply as

$$E(\mathrm{p}, \mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L(\omega)\,\mathrm{d}\omega^{\perp}.$$

For the rest of this book, we will write integrals over directions in terms of solid angle, rather than projected solid angle. When reading rendering integrals in other contexts, however, be sure to be aware of the measure of domain of integration.

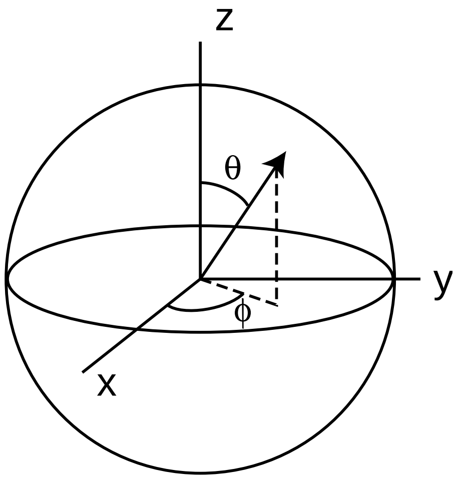


Figure 5.9: A given direction vector can be written in terms of spherical coordinates (θ, ϕ) if the x, y, and z basis vectors are given as well. The spherical angle formulae make it easy to convert between the two representations.

5.3.2 Integrals over spherical coordinates

Vector 27

It is often convenient to transform integrals over solid angle into integrals over spherical coordinates (θ, ϕ). Recall that an (x, y, z) direction vector can be alternatively written in terms of spherical angles (see Figure 5.9):

$$\begin{aligned} x &= \sin\theta\cos\phi \\ y &= \sin\theta\sin\phi \\ z &= \cos\theta \end{aligned}$$

For convenience, we'll define two functions that turn θ and ϕ values into (x, y, z) direction vectors. The first applies the equations above directly. Notice that these functions are passed the sine and cosine of θ, but the angle ϕ. This is because the sine and cosine of θ are frequently available directly to the calling function (through a vector dot product, for example).

$\langle$*Geometry Inline Functions*$\rangle{+}{\equiv}$

```
inline Vector SphericalDirection(Float sintheta, Float costheta,
        Float phi) {
    return Vector(sintheta * cosf(phi),
        sintheta * sinf(phi), costheta);
}
```

The second function takes three basis vectors to replace the x, y and z axes and returns the appropriate direction vector with respect to the coordinate frame that they define.

⟨*Geometry Inline Functions*⟩+≡

```
inline Vector SphericalDirection(Float sintheta, Float costheta,
        Float phi, const Vector &x, const Vector &y,
        const Vector &z) {
    return sintheta * cosf(phi) * x +
        sintheta * sinf(phi) * y + costheta * z;
}
```

The spherical angles for a direction can be found by:

$$\begin{aligned}\theta &= \arccos z \\ \phi &= \arctan \frac{y}{z}\end{aligned}$$

Corresponding functions are below. Note that `SphericalTheta()` assumes that the vector `v` has been normalized before being passed in.

⟨*Geometry Inline Functions*⟩+≡

```
inline Float SphericalTheta(const Vector &v) {
    return acosf(v.z);
}
```

⟨*Geometry Inline Functions*⟩+≡

```
inline Float SphericalPhi(const Vector &v) {
    return atan2f(v.y, v.x) + M_PI;
}
```

678 M_PI
27 Vector

In order to write an integral over solid angle in terms of an integral over (θ, ϕ), we need to be able to express the relationship between the differential area of a set of directions $d\omega$ and the differential area of a (θ, ϕ) pair–see Figure 5.10. The differential area $d\omega$ is the product of the differential lengths of the sides of $d\omega$, $\sin\theta d\phi$ and $d\theta$. Therefore,

$$d\omega = \sin\theta \, d\theta \, d\phi.$$

We can thus see that the irradiance integral over the hemisphere (Equation 5.2.5 with $\Omega = \mathcal{H}^2(\mathbf{n})$) can equivalently be written

$$E(\mathrm{p}, \mathbf{n}) = \int_0^{2\pi} \int_0^{\pi/2} L(\mathrm{p}, \theta, \phi) \cos\theta \sin\theta \, d\theta \, d\phi$$

So if the radiance is the same from all directions, this simplifies to $E = \pi L$.

Just as we found irradiance in terms of incident radiance, we can also compute the total flux emitted from some object over the hemisphere about the normal by integrating over the object's surface area A:

$$\Phi = \int_A \int_{\mathcal{H}^2(\mathbf{n})} L(\mathrm{p}, \omega) \cos\theta \, d\omega \, dA$$

5.3.3 Integrals over area

One last transformation of integrals that can simplify computation is to turn integrals over directions into integrals over area. Consider the irradiance integral again (Equation 5.2.5), where there is a quadrilateral with constant outgoing radiance and

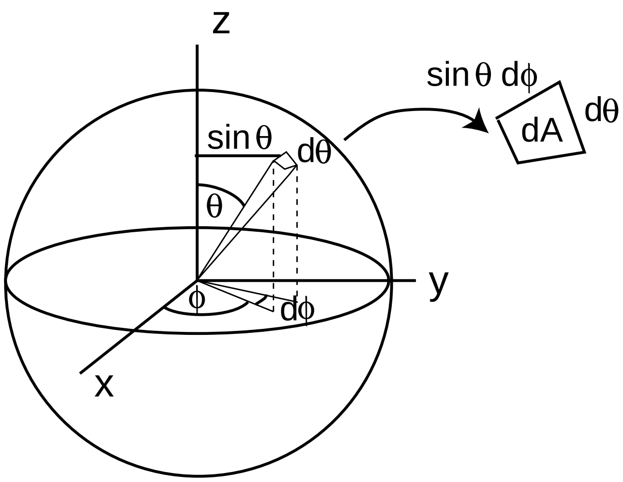


Figure 5.10: The differential area dA subtended by a differential solid angle is the product of the differential lengths of the two edges $\sin\theta \mathrm{d}\phi$ and $\mathrm{d}\theta$. The resulting relationship, $\mathrm{d}\omega = \sin\theta \mathrm{d}\theta \mathrm{d}\phi$, is the key to converting between integrals over solid angles and integrals over spherical angles.

where we'd like to compute the resulting irradiance at a point p. The easiest way to write this integral is over the area of the quadrilateral; writing it as an integral over directions is less straightforward, since given a particular direction, the computation to determine if the quadrilateral is visible in that direction is non-trivial.

Differential area is related to differential solid angle by

$$\mathrm{d}\omega = \frac{\mathrm{d}A \cos\theta}{r^2} \tag{5.3.6}$$

where θ is the angle between the surface normal of dA and r is the distance from p to dA (see Figure 5.11).

We will not derive this result here, but it can be understood intuitively: if dA is at distance 1 from p and is aligned exactly so that it is facing down dω, then $\mathrm{d}\omega = \mathrm{d}A$, $\theta = 0$, and Equation 5.3.6 holds. As dA moves farther away from p, or as it rotates so that it's not aligned with the direction of dω, the r^2 and $\cos\theta$ terms compensate accordingly to reduce dω.

Therefore, we can write the irradiance integral for the quadrilateral source as

$$E(\mathrm{p}) = \int_A L \cos\theta_i \frac{\cos\theta_o \,\mathrm{d}A}{r^2}$$

where θ_i is the angle between the surface normal at p and the direction from p to the point p′ on the light, and θ_o is the angle between the surface normal at p′ on the light and the direction from p′ to p (see Figure 5.12.)

5.4 Surface Reflection

When light in an environment is incident on a surface, the surface scatters the light, re-reflecting some of it back into the environment. For example, the skin of a lemon mostly absorbs light in the blue wavelengths, but reflects most of light in the

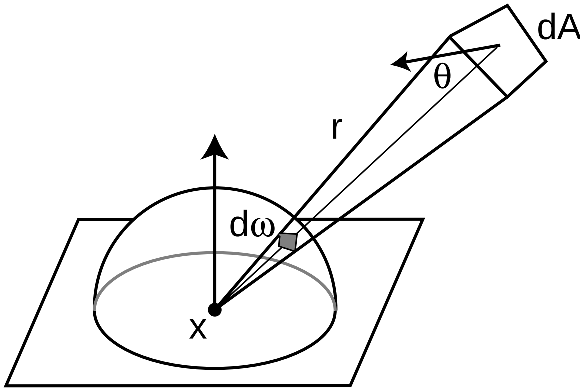


Figure 5.11: The differential solid angle subtended by a differential area dA is equal to d$A\cos\theta/r^2$, where θ is the angle between dA's surface normal and the vector to the point p and r is the distance from p to dA.

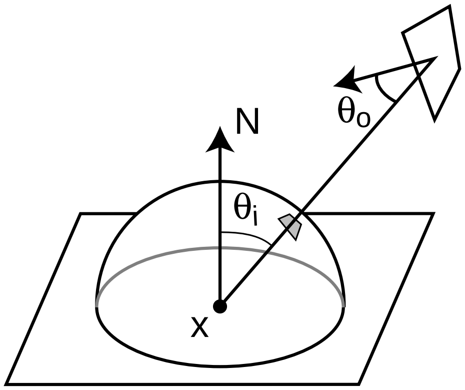


Figure 5.12: To compute irradiance at a point p from a quadrilateral source, it's easier to integrate over the surface area of the source than to integrate over the irregular set of directions that it subtends. The relationship between solid angles and areas given by Equation 5.3.6 lets us go back and forth between the two approaches.

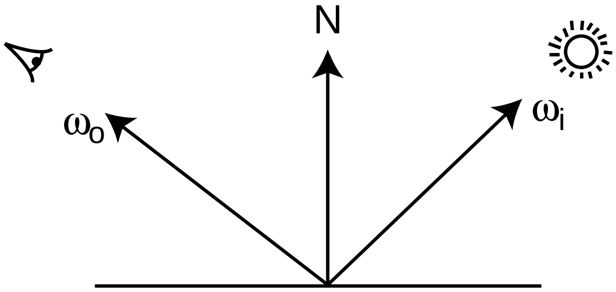


Figure 5.13: The bidirectional reflectance distribution function (BRDF) is a four-dimensional function over pairs of directions ω_i and ω_o that describes how much incident light along ω_i is scattered from the surface in the direction ω_o.

red and green wavelengths (recall the lemon skin reflectance SPD in Figure 5.1.) Therefore, when it is illuminated with white light, its color is yellow. The skin has pretty much the same color no matter what direction it's being observed from, although for some directions a highlight is visible, where it is more white than yellow.

In contrast, the color seen in a mirror depends almost entirely on the viewing direction. At a fixed point on the mirror, as the viewing angle changes, the object that is reflected in the mirror changes accordingly. Furthermore, mirrors generally don't change the color of the object they are reflecting.

5.4.1 The BRDF

There are a few concepts in radiometry that give formalisms for describing these types of reflection. One of the most important is the *bidirectional reflectance distribution function*, (BRDF). Consider the setting in Figure 5.13: we'd like to know how much radiance is leaving the surface in the direction ω_o toward the viewer, $L_{\mathrm{o}}(\mathrm{p},\omega_o)$ as a result of incident radiance along the direction ω_i, $L_{\mathrm{i}}(\mathrm{p},\omega_i)$. The reader is warned not to be misled by diagrams like Figure 5.13, however. These kinds of diagrams frequently show the scattering situation from a side view, but we always need to be aware that the vectors ω_o and ω_i are not always co-planar with the surface normal N.

If the direction ω_i is considered a differential cone of directions, we can compute the resulting differential irradiance at p by

$$dE(\mathrm{p},\omega_i) = L(\mathrm{p},\omega_i)\cos\theta_i\,\mathrm{d}\omega_i. \tag{5.4.7}$$

A differential amount of radiance will be reflected in the direction ω_o. An important assumption made in radiometry is that the system is linear: doubling the amount of energy going into it will lead to a doubling of the amount going out of it. This is a reasonable assumption as long energy levels are not extreme.

Therefore, the reflected differential radiance is

$$dL_{\mathrm{o}}(\mathrm{p},\omega_o) \propto dE(\mathrm{p},\omega_i).$$

The constant proportionality for the particular pair of directions ω_i and ω_o is defined to be the surface's BRDF:

$$f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i) = \frac{dL(\mathrm{p},\omega_o)}{dE(\mathrm{p},\omega_i)} = \frac{dL(\mathrm{p},\omega_o)}{L(\mathrm{p},\omega_i)\cos\theta_i d\omega_i} \tag{5.4.8}$$

Physically-based BRDFs have two important qualities:

1. *Reciprocity*: for all pairs of directions ω_i and ω_o, $f_{\mathrm{r}}(\mathrm{p},\omega_i,\omega_o) = f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i)$.

2. *Energy conservation*: the total energy of light reflected is less than or equal to the energy of incident light. For all directions ω,

$$\int_{\mathcal{S}^2} f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega)\cos\theta d\omega \leq 1.$$

The surface's *bidirectional transmittance distribution function* (BTDF) can be defined in a similar manner to the BRDF. The BTDF is generally denoted by $f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i)$, where ω_i and ω_o are in opposite hemispheres around p. Interestingly, the BTDF does not obey reciprocity; we will discuss this in detail in Section 9.2.

For convenience in equations, we will denote the BRDF and BTDF considered together as $f(\mathrm{p},\omega_o,\omega_i)$; we will call this the *bidirectional scattering distribution function* (BSDF). Chapter 9 is entirely devoted to describing BSDFs that are used in graphics.

Using the definition of the BSDF, we have

$$dL_{\mathrm{o}}(\mathrm{p},\omega_o) = L_{\mathrm{i}}(\mathrm{p},\omega_i)\, f(\mathrm{p},\omega_o,\omega_i)\cos\theta_i d\omega_i.$$

We can integrate this over the sphere of incident directions around p to compute the outgoing radiance in direction ω_o due to the incident illumination at p: **explain where the absolute value signs come from in this equation; they're not in the previous one...**

$$L_{\mathrm{o}}(\mathrm{p},\omega_o) = \int_{\mathcal{S}^2} L_{\mathrm{i}}(\mathrm{p},\omega_i)\, f(\mathrm{p},\omega_o,\omega_i)\,|\cos\theta_i| d\omega_i \tag{5.4.9}$$

This is a fundamental equation in rendering; it describes how an incident distribution of light at a point is transformed into an outgoing distribution, based on the scattering properties of the surface. It is often called the *scattering equation* when the sphere $\mathcal{S}^2$ is the domain (as it is here), or the *reflection equation*, when just the upper hemisphere $\mathcal{H}^2(\mathbf{n})$ is being integrated over.

Further Reading

Hall's book summarizes the state-of-the-art in spectral representations through 1989 (Hall 1989) and Glassner's *Principles of Digital Image Synthesis* covers the topic through the mid-90s (**?**). Meyer was the one of the first researchers to closely investigate spectral representations in graphics; XXX. Later, Raso and Fournier proposed a polynomial representation for spectra (Raso and Fournier 1991).

Our discussion of SPD representation with basis functions is based on Peercy's 1993 SIGGRAPH paper (Peercy 1993). In that paper, Peercy chose particular basis functions in a scene-dependent manner: by looking at the SPDs of the lights and reflecting objects in the scene, a small number of basis functions that could accurately represent the scene's SPDs were found using characteristic vector analysis.

Another approach to spectral representation was investigated by Sun et al; they partitioned SPDs into a smooth base SPD and a set of spikes (Sun, Fracchia, Drew, and Calvert 2001). Each part was represented differently, using basis functions that worked well for each particular type of function.

He and Stam have use wave optics stuff in graphics (He, Torrance, Sillion, and Greenberg 1991; Stam 1999). Also cite appropriate part of Preisendorfer and Chandrasekhar.

Non-linear approximation paper (cited in Ren's paper...)XXX

`Spectrum 181`

Arvo has investigated the connection between rendering algorithms in graphics and previous work in *transport theory*, which applies classical physics to particles and their interactions to predict their overall behavior and global illumination algorithms (Arvo 1993; Arvo 1995).

XXX where to get real-world SPD data

McCluney's book on radiometry (McCluney 1994) is an excellent introduction to the topic. Preisendorfer also covers radiometry in an accessible manner and delves into the relationship between radiometry and the physics of light (Preisendorfer 1965). Moon and Spencer's books (Moon and Spencer 1936; Moon and Spencer 1948) and Gershun's article (Gershun 1939) are classic early introductions to radiometry. Lambert's seminal early writings about photometry from the mid-18th century were recently translated by DiLaura (Lambert 2001).

Exercises

5.1 Experiment with different basis functions for spectral representation. How many coefficients are needed for accurate rendering of tricky situations like fluorescent lighting? How much does the particular choice of basis affect the number of coefficients needed?

5.2 Generalize the `Spectrum` class so that it's not limited to orthonormal basis functions. Implement Peercy's approach of choosing basis functions based on the main SPDs in the scene. Does the improvement in accuracy make up for the additional computational expense of computing the products of spectra.

5.3 Generalize the `Spectrum` class further to support non-linear basis functions. Compare the results to more straightforward spectral representations.

5.4 Compute the irradiance at a point due to a unit-radius disk *h* units directly above its normal with constant outgoing radiance of 10 J/m^2 sr. Do the computation twice, once as an integral over solid angle and once as an integral over area. (Hint: if the results don't match and you write the integral over the disks' area as an integral over radius *r* and an integral over angle θ, see Section XXX in the Monte Carlo chapter for a hint about XXXXXX.)

5.5 Similarly, compute the irradiance at a point due to a square quadrilateral with outgoing radiance of 10 J/m^2 sr that has sides of length 1 and is 1 unit directly above it along its surface normal.

6. Camera Models

202 Camera

In addition to describing the objects that make up the scene, we also need to describe how the scene is viewed and how its three-dimensional representation is mapped to a two-dimensional image. This chapter describes the `Camera` class and its implementations, which generate primary to sample the scene and generate the image. By generating these rays in different ways, `lrt` can create many types of images of the same 3D scene. We will show a few implementations of the `Camera` interface, each of which generates rays in a different way.

6.1 Camera Model

```
⟨camera.h*⟩≡
  #include "lrt.h"
  #include "color.h"
  #include "sampling.h"
  #include "geometry.h"
  #include "transform.h"
  ⟨Camera Declarations⟩
```

```
⟨camera.cpp*⟩≡
  #include "lrt.h"
  #include "camera.h"
  #include "film.h"
  #include "mc.h"
  ⟨Camera Method Definitions⟩
```

We will define an abstract `Camera` base class that holds generic camera options and defines an interface for all camera implementations to provide.

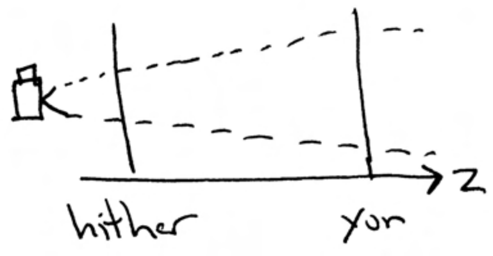


Figure 6.1: The camera's clipping planes give the range of space along the *z* axis that will be images; objects in front of the hither plane or beyond the yon plane will not be visible in the image. Setting the clipping planes to tightly encompass the objects in the scene is important for many scanline algorithms, but is less important for ray-tracing.

```
⟨Camera Declarations⟩≡
  class Camera {
  public:
       ⟨Camera Interface⟩
       ⟨Camera Public Data⟩
  protected:
       ⟨Camera Protected Data⟩
  };
```

Ray 36

The main method that camera subclasses need to implement is `Camera::GenerateRay()`, which generates a ray for a given image sample. It is important that the camera normalize the direction component of the returned ray—many other parts of the system will depend on this behavior.

This method also returns a floating-point value that gives a weight for the effect that light arriving at the film plane along the generated ray will have on the final image. Most cameras will always set this to one, although cameras that simulate real physical lens systems might need to set this value based on the optics and geometry of the lens system being simulated.

```
⟨Camera Interface⟩≡
  virtual Float GenerateRay(const Sample &sample,
       Ray *ray) const = 0;
```

The base `Camera` constructor takes a number of parameters that are appropriate for all camera types. They include the transformation that places the camera in the scene, and the near and far *clipping planes*, which give distances along the camera space *z* axis that delineate the scene being rendered[1]. Any geometric primitives in front of the near plane or beyond the far plane will not be rendered; see Figure 6.1.

Real-world cameras have a shutter that opens for a short period of time to expose the film to light; one result of this non-zero exposure time is that objects that move during the film exposure time are blurred; this effect is called *motion blur*. To

[1] Although the names “near” and “far” make clear intuitive sense for these planes, graphics systems frequently refer to them as “hither” and “yon”, respectively. Although there is probably a historic reason for this **WHAT MIGHT THAT BE?**, a practiacal reason is that `near` and `far` are reserved keywords in Microsoft's C and C++ compilers.

model this effect in `lrt`, each ray has a time value associated with it–by sampling the scene over a range of times, motion can be captured. Thus, all `Camera`s store a shutter open and shutter close time. **Note, however, that `lrt` does not currently support motion blur. We provide a properly sampled time value to allow for this future expansion, however.**

Finally, `Camera`s contain an instance of the `Film` class to represent the final image to be computed. `Film` will be described in Chapter 8.

⟨*Camera Method Definitions*⟩+≡

```
Camera::Camera(const Transform &world2cam, Float hither, Float yon,
        Float sopen, Float sclose, Film *f) {
    WorldToCamera = world2cam;
    CameraToWorld = WorldToCamera.GetInverse();
    ClipHither = hither;
    ClipYon = yon;
    ShutterOpen = sopen;
    ShutterClose = sclose;
    film = f;
}
```

⟨*Camera Protected Data*⟩≡

```
Transform WorldToCamera, CameraToWorld;
Float ClipHither, ClipYon;
Float ShutterOpen, ShutterClose;
```

202	`Camera`
294	`Film`
43	`Transform`
55	`Transform::GetInverse()`

⟨*Camera Public Data*⟩≡

```
Film *film;
```

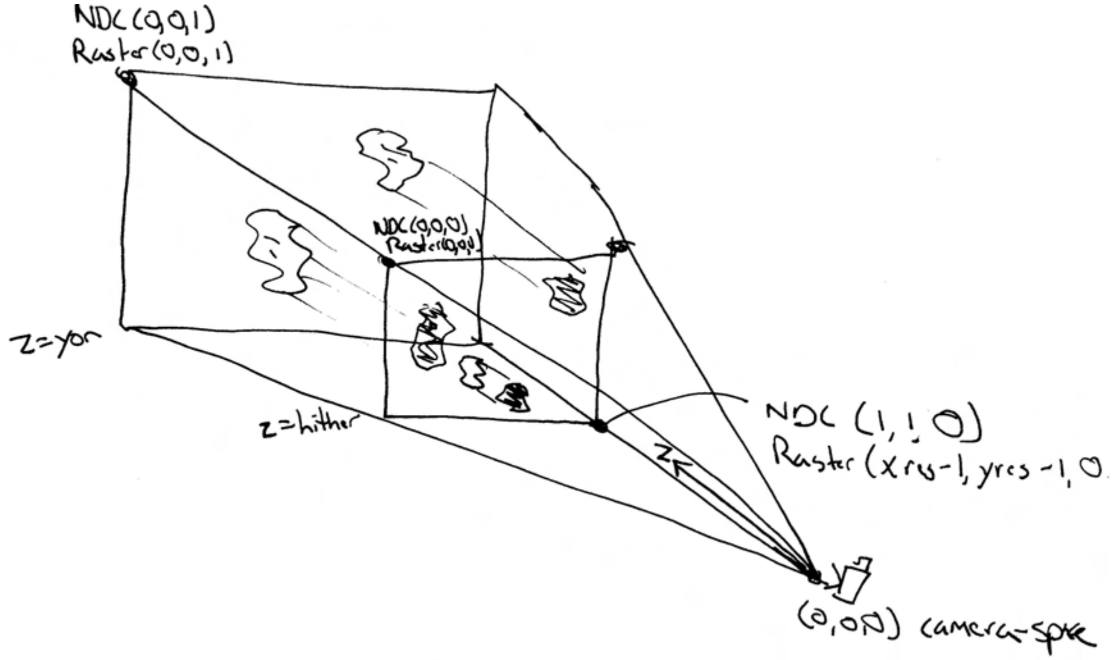
6.1.1 Camera Coordinate Spaces

We have already made use of two important modeling coordinate spaces, object space and world space. We will now introduce three more useful coordinate spaces that have to do with the camera and imaging. Including object and world space, we now have the following. (See Figure 6.2.)

- *Object space*: This is the coordinate system in which geometric primitives are defined. For example, spheres in `lrt` are defined to be centered at the origin of object space.

- *World space*: While each primitive may have its own object space, there is a single world space that the objects in the scene are placed in relation to. Each primitive has an object to world transformation that determines how it is located in world space. World space is the standard frame that all spaces are defined in terms of.

- *Camera space*: A virtual camera is placed in the scene at some world-space point with a particular viewing direction and “up” vector. This defines a new coordinate system around that point with the origin at the camera’s location. The z axis is mapped to the viewing direction, and the y axis mapped to the up direction.**(see Section ?? on page ??.)** This is a handy space for reasoning



Camera 202

Figure 6.2: A handful of camera-related coordinate spaces help to simplify the implementation of `Camera`s. The camera class holds transformations between them. Scene objects in world space are viewed by the camera, which sits at the origin of camera space and looks down the $+z$ axis. Objects between the hither and yon planes are projected onto the image plane at $z =$ hither in camera space. The image plane is at $z = 0$ in raster space, where x and y range from $(0,0)$ to $(\text{xResolution} - 1, \text{yResolution} - 1)$. Normalized device coordinate (NDC) space normalizes raster space so that x and y range from $(0,0)$ to $(1,1)$.

about which objects are potentially visible to the camera. For example, if an object's camera-space bounding box is entirely behind the $z = 0$ plane (and the camera doesn't have a field of view wider than 180 degrees), the object will not be visible to the camera.

- *Screen space*: Screen space is defined on the image plane. The camera projects objects in camera space onto the image plane; the parts inside the *screen window* are visible in the image that is generated. **What are the x and y extents of this space? This is confusing later I think.** Depth z values in screen space range from zero to one, corresponding to points at the near and far clipping planes, respectively. Note that although this is called "screen" space, it is still a 3D coordinate system, since z values are meaningful.
- *NDC Normalized device coordinate* space: This is the coordinate system for the actual image being rendered. In x and y, this space ranges from $(0,0)$ to $(1,1)$, with $(0,0)$ being the upper left corner of the image. Depth values are the same as in screen space and a linear transformation converts from screen to NDC space.
- *Raster space*: This is almost the same as NDC space, except the x and y coordinate range from $(0,0)$ to $(\mathrm{xResolution}-1, \mathrm{yResolution}-1)$.

202 Camera
294 Film

All cameras store a world space to camera space transformation; this can be used to transform primitives in the scene into camera space. The origin of camera space is the camera's position, and the camera looks down the camera space z axis. The projective cameras in the next section will use matrices to transform between all of these spaces as needed, but cameras with unusual imaging characteristics can't necessarily represent these transformations with 4x4 matrices.

6.2 Projective Camera Models

One of the fundamental parts of 3D computer graphics is the *3D viewing problem*: how a three-dimensional scene is projected onto a two-dimensional image for display. Most of the classic approaches can be expressed by a 4x4 projective transformation matrix. Therefore, we will introduce a projection matrix camera class and then define two simple camera models. The first implements an orthographic projection, and the other implements a perspective projection–these are two classic and widely-used projections.

⟨Camera Declarations⟩+≡

```
class ProjectiveCamera : public Camera {
public:
    ⟨ProjectiveCamera Public Methods⟩
protected:
    ⟨ProjectiveCamera Protected Data⟩
};
```

In addition to the world to camera transformation and the projective transformation matrix, the `ProjectiveCamera` takes the screen-space extent of the image, clipping plane distances, a pointer to the `Film` class for the camera, and additional parameters for motion blur and depth of field. Depth of field, the implementation of which will be shown at the end of this section, simulates blurriness of out-of-focus objects in real lens systems.

⟨*Camera Method Definitions*⟩+≡

```
ProjectiveCamera::ProjectiveCamera(const Transform &w2c,
        const Transform &proj, const Float Screen[4],
        Float hither, Float yon, Float sopen,
        Float sclose, Float lensr,
        Float focald, Film *f)
    : Camera(w2c, hither, yon, sopen, sclose, f) {
    ⟨Initialize depth of field parameters⟩
    ⟨Compute projective camera transformations⟩
}
```

The ProjectiveCamera implementations pass the projective transformation up to the base class constructor here. This transformation gives us the camera to screen projection; from that we can compute most of the others that we need.

⟨*Compute projective camera transformations*⟩≡

```
CameraToScreen = proj;
WorldToScreen = CameraToScreen * WorldToCamera;
⟨Compute projective camera screen transformations⟩
RasterToCamera = CameraToScreen.GetInverse() * RasterToScreen;
```

⟨*ProjectiveCamera Protected Data*⟩≡

```
Transform CameraToScreen, WorldToScreen, RasterToCamera;
```

The only non-trivial one of the precomputed transformations is ProjectiveCamera::ScreenToRast
Note the composition of transformations where (reading from bottom to top), we start with a point in screen space, translate so that the upper left corner of the screen is at the origin, and then scale by the reciprocal of the screen width and height, giving us a point with *x* and *y* coordinates between zero and one (these are NDC coordinates). Finally, we scale by the raster resolution, so that we end up covering the raster range from $(0,0)$ up to the overall raster resolution.

⟨*Compute projective camera screen transformations*⟩≡

```
ScreenToRaster = Scale(film->xResolution, film->yResolution, 1.f) *
    Scale(1.f / (Screen[1] - Screen[0]),
          1.f / (Screen[2] - Screen[3]), 1.f) *
    Translate(Vector(-Screen[0], -Screen[3], 0.f));
RasterToScreen = ScreenToRaster.GetInverse();
```

⟨*ProjectiveCamera Protected Data*⟩+≡

```
Transform ScreenToRaster, RasterToScreen;
```

6.2.1 Orthographic Camera

Camera	202
Film	294
Film::xResolution	294
Film::yResolution	294
ProjectiveCamera	205
Scale()	47
Transform	43
Transform::GetInverse()	55
Translate()	46
Vector	27

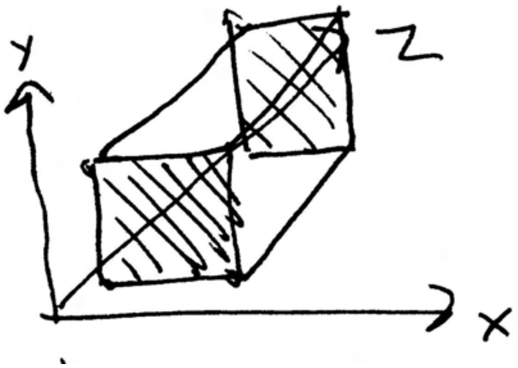


Figure 6.3: The orthographic view volume is an axis-aligned box in camera space, defined such that objects inside the region are projected onto the $z =$ hither face of the box.

⟨*orthographic.cpp**⟩≡

```
#include "camera.h"
#include "film.h"
#include "paramset.h"
⟨OrthographicCamera Declarations⟩
⟨OrthographicCamera Definitions⟩
```

294	Film
208	Orthographic()
205	ProjectiveCamera
43	Transform

⟨*OrthographicCamera Declarations*⟩≡

```
class OrthoCamera : public ProjectiveCamera {
public:
    ⟨OrthoCamera Public Methods⟩
};
```

The orthographic transformation takes a rectangular region of the scene and projects it onto the front face of the box that defines the region. It doesn't give the effect of foreshortening–objects becoming smaller on the image plane as they get farther away–but it does leave parallel lines parallel and preserves relative distance between objects. Figure 6.3 shows how this rectangular volume gives the visible region of the scene.

The orthographic camera constructor generates the orthographic transformation matrix with the `Orthographic()` transformation function, which will be defined shortly.

⟨*OrthographicCamera Definitions*⟩≡

```
OrthoCamera::OrthoCamera(const Transform &world2cam,
        const Float Screen[4], Float hither, Float yon,
        Float sopen, Float sclose, Float lensr,
        Float focald, Film *f)
    : ProjectiveCamera(world2cam, Orthographic(hither, yon),
         Screen, hither, yon, sopen, sclose,
         lensr, focald, f) {
}
```

The orthographic viewing transformation leaves x and y coordinates unchanged, but maps z values at the hither plane to 0 and z values at the yon plane to 1. (See

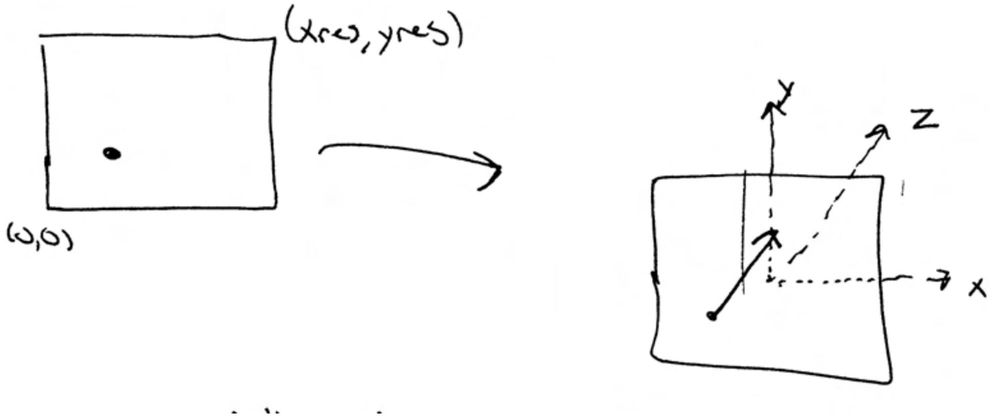


Figure 6.4: orthographic ray generation: raster space to ray...

Figure 6.3.) It is easy to derive: first, the scene is translated along the z axis so that the near clipping plane is aligned with $z = 0$. Then, the scene is scaled in z so that the far clipping plane maps to $z = 1$. The composition of these two transformations gives the overall transformation.

`Scale()`	47
`Transform`	43
`Translate()`	46
`Vector`	27

⟨*Transform Method Definitions*⟩+≡

```
Transform Orthographic(Float znear, Float zfar) {
    return Scale(1.f, 1.f, 1.f / (zfar-znear)) *
        Translate(Vector(0.f, 0.f, -znear));
}
```

We can now write the code to take a sample point in raster space and turn it into a camera ray. The `Sample::imageX` and `Sample::imageY` components of the camera sample are raster-space x and y coordinates on the image plane (the contents of the `Sample` structure are described in detail in chapter 7. We use following process: first, we transform the raster-space sample position into a point in camera space; this gives us the origin of the camera ray–a point located on the near clipping plane. Because the camera-space viewing direction points down the z axis, the camera space ray direction is $(0,0,1)$. The ray's `maxt` value is set so that intersections beyond the far clipping plane will be ignored; this is easily computed since the ray's direction is normalized. Finally, the ray is transformed into world space before this method returns.

If depth of field has been enabled for this scene, the fragment ⟨*Modify ray for depth of field*⟩ takes care of modifying the ray so that depth of field is simulated. Depth of field will be explained later in this section.

Need to ensure no scaling in camera to world...

⟨*OrthographicCamera Definitions*⟩+≡

```
  Float OrthoCamera::GenerateRay(const Sample &sample, Ray *ray) const {
      ⟨Generate raster and camera samples⟩
      ray->o = Pcamera;
      ray->d = Vector(0,0,1);
      ⟨Set ray time value⟩
      ⟨Modify ray for depth of field⟩
      ray->mint = 0.;
      ray->maxt = ClipYon - ClipHither;
      ray->d = ray->d.Hat();
      CameraToWorld(*ray, ray);
      return 1.f;
  }
```

The `Sample` structure tells us what "time" this ray should be traced **(again, this is for a future motion blur expansion)**. The `Sample`'s `time` value ranges between 0 and 1, so we simply use it to linearly interpolate between the provided shutter open and close times.

⟨*Set ray time value*⟩≡

```
  ray->time = Lerp(sample.time, ShutterOpen, ShutterClose);
```

Once all of the transformation matrices have been set up, we just set up the raster space sample point and transform it to camera space.

⟨*Generate raster and camera samples*⟩≡

```
  Point Pras(sample.imageX, sample.imageY, 0);
  Point Pcamera;
  RasterToCamera(Pras, &Pcamera);
```

203	Camera::CameraToWorld
203	Camera::ClipHither
203	Camera::ClipYon
203	Camera::ShutterClose
203	Camera::ShutterOpen
677	Lerp()
207	OrthoCamera
33	Point
206	ProjectiveCamera::RasterToCamera
36	Ray
35	Ray::d
36	Ray::maxt
36	Ray::mint
35	Ray::o
36	Ray::time
239	Sample::imageX
239	Sample::imageY
239	Sample::time
27	Vector

6.2.2 Perspective Camera

⟨*perspective.cpp**⟩≡

```
  #include "camera.h"
  #include "film.h"
  #include "paramset.h"
  ⟨PerspectiveCamera Declarations⟩
  ⟨PerspectiveCamera Method Definitions⟩
```

The perspective projection is similar to the orthographic projection in that it projects a volume of space onto a 2D image plane. However, it includes the effect of *foreshortening*: objects that are far away are projected to be smaller than objects of the same size that are closer. Furthermore, unlike the orthographic projection, the perspective projection also doesn't preserve distances or angles in general, and parallel lines no longer remain parallel. The perspective projection is a reasonably close match for how the eye and camera lenses generate images of the three-dimensional world.

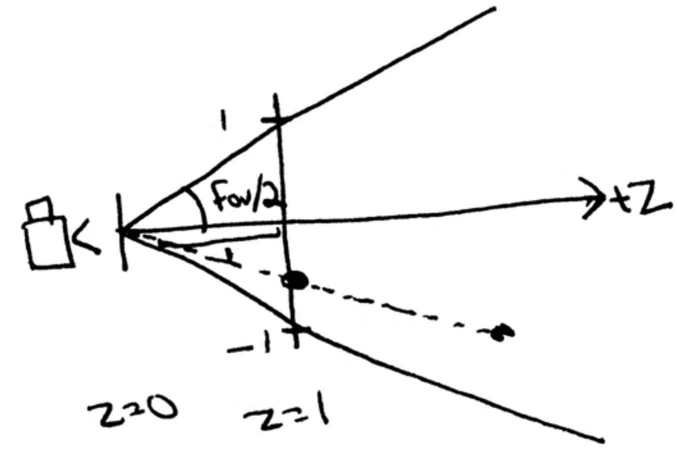


Figure 6.5: The perspective transformation matrix projects points in camera space onto the image plane. The x' and y' coordinates of the projected points are equal to the unprojected x and y coordinates divided by the z coordinate. The projected z' coordinate is computed so that z points on the hither plane map to $z' = 0$ and points on the yon plane map to $z' = 1$.

Film	294
ProjectiveCamera	205
Transform	43

⟨*PerspectiveCamera Declarations*⟩≡

```
class PerspectiveCamera : public ProjectiveCamera {
public:
    ⟨PerspectiveCamera Public Methods⟩
};
```

⟨*PerspectiveCamera Method Definitions*⟩≡

```
PerspectiveCamera::PerspectiveCamera(const Transform &world2cam,
        const Float Screen[4], Float hither, Float yon,
        Float sopen, Float sclose, Float lensr, Float focald,
        Float fov, Film *f)
    : ProjectiveCamera(world2cam, Perspective(fov, hither, yon),
        Screen, hither, yon, sopen, sclose,
        lensr, focald, f) {
}
```

The perspective projection describes perspective viewing of the scene. Points in the scene are projected onto a viewing plane at $z = 1$; this is one unit away from the virtual camera at $z = 0$)–see Figure 6.5.

⟨*Transform Method Definitions*⟩+≡

```
Transform Perspective(Float fov, Float n, Float f) {
    ⟨Perform projective divide⟩
    ⟨Scale to canonical viewing volume⟩
}
```

The process is most easily understood in two steps:

- First, points p in camera space are projected onto the viewing plane. A little algebra shows that the projected x' and y' coordinates on the viewing plane can be computed by dividing x and y by the point's z coordinate value. The projected z depth is remapped so that z values at the hither plane go to 0 and z values at the yon plane go to 1. The computation we'd like to do is:

$$\begin{aligned} x' &= x/z \\ y' &= y/z \\ z' &= \frac{f(z-n)}{z(f-n)}. \end{aligned}$$

Fortunately, all of this can easily be encoded in a four-by-four matrix using homogeneous coordinates:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & -\frac{fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

⟨*Perform projective divide*⟩≡

```
  Matrix4x4 *persp =
      new Matrix4x4(1, 0,       0,           0,
            0, 1,       0,           0,
            0, 0, f/(f-n), -f*n/(f-n),
            0, 0,       1,           0);
```

675	Matrix4x4
677	Radians()
47	Scale()
43	Transform

- Second, we account for the angular field of view specified by the user and scale the (x,y) values on the projection plane so that points inside the field of view project to coordinates between $[-1,1]$ on the view plane. (For square images, both x and y will lie between $[-1,1]$ in screen space. Otherwise, the direction in which the image is narrower will map to $[-1,1]$ and the wider direction will map to an appropriately larger range of screen-space values.) The scale that is applied after the projective transformation takes care of this. (Recall that the tangent is equal to the ratio of the opposite side of a right triangle to the adjacent side. Here the adjacent side is defined to have a length of 1, so the opposite side has the length $\tan(\mathrm{fov}/2)$. Scaling by the reciprocal of this length this maps the field of view to range from $[-1,1]$.

⟨*Scale to canonical viewing volume*⟩≡

```
  Float invTanAng = 1.f / tanf(Radians(fov) / 2.f);
  return Scale(invTanAng, invTanAng, 1) *
         Transform(persp);
```

this is confusing. Doesn't ortho have the same rastertoscreen transformation as perspective? Where is the -1 → 1 transformation happening in the ortho camera? Has anyone tested the ortho camera in a while?

For a perspective projection, rays originate from the sample position on the hither plane and have the direction given by the vector from $(0,0,0)$ through the sample position. Therefore, we compute the ray's direction by subtracting $(0,0,0)$ from the sample's camera-space position. In other words, the ray's vector direction is component-wise equal to its point position. Rather than doing a useless subtraction to convert the point to a direction, we just component-wise initialize the vector `ray->d` from the point `Pcamera`. Because the generated ray's direction may be quite short, we scale it up by the inverse of the near clip plane location; although this isn't strictly necessary (there's no particular need for the ray direction to be normalized), it can be more intuitive when debugging if the ray's direction has a magnitude somewhat close to one.

As with the `OrthoCamera`, the ray's `maxt` value is set to lie on the far clipping plane.

⟨*PerspectiveCamera Method Definitions*⟩+≡

```
Float PerspectiveCamera::GenerateRay(const Sample &sample,
        Ray *ray) const {
    ⟨Generate raster and camera samples⟩
    ray->o = Pcamera;
    ray->d = Vector(Pcamera.x, Pcamera.y, Pcamera.z);
    ⟨Set ray time value⟩
    ⟨Modify ray for depth of field⟩
    ray->d = ray->d.Hat();
    ray->mint = 0.;
    ray->maxt = (ClipYon - ClipHither) / ray->d.z;
    CameraToWorld(*ray, ray);
    return 1.f;
}
```

`Camera::CameraToWorld`	203
`Camera::ClipHither`	203
`Camera::ClipYon`	203
`PerspectiveCamera`	210
`Ray`	36
`Ray::d`	35
`Ray::maxt`	36
`Ray::mint`	36
`Ray::o`	35
`Vector`	27

6.2.3 Depth of Field

Real cameras have lens systems that focus light through a finite-sized aperture onto the film plane. Because the aperture has finite area, a single point in the scene may be projected onto an area on the film plane. (And correspondingly, a single point on the film plane may see different parts of the scene, depending on which part of the lens it's receiving light from.) Figure 6.6 shows this effect. The point p_1 doesn't lie on the plane of focus, so is projected through the lens onto an area p_1' on the film plane. The point p_2 does lie on the plane of focus, so it projects to a single point p_2' on the image plane. Therefore, p_1 will be blurred on the image plane while p_2 will be in sharp focus.

To understand how to compute the proper ray through an arbitrary point on the lens, we make the simplifying assumption that we are using a single spherical lens. Figure 6.7 shows such a lens, as well as many quantities that we will refer to in the following derivation. These quantities are summarized in the following table:

U	Angle of the ray with respect to the lens axis
U'	Angle of refracted ray with respect to the lens axis
Φ	Angle of sphere normal at hit point with respect to the lens axis
I	Angle of incidence of primary ray
I'	Angle of incidence of refracted ray
Z	distance from the ray's sphere intersection to the ray's axis intersection
Z'	distance from the ray's sphere intersection to the refracted ray's axis intersection
h	height of intersection
ε	depth of intersection into sphere

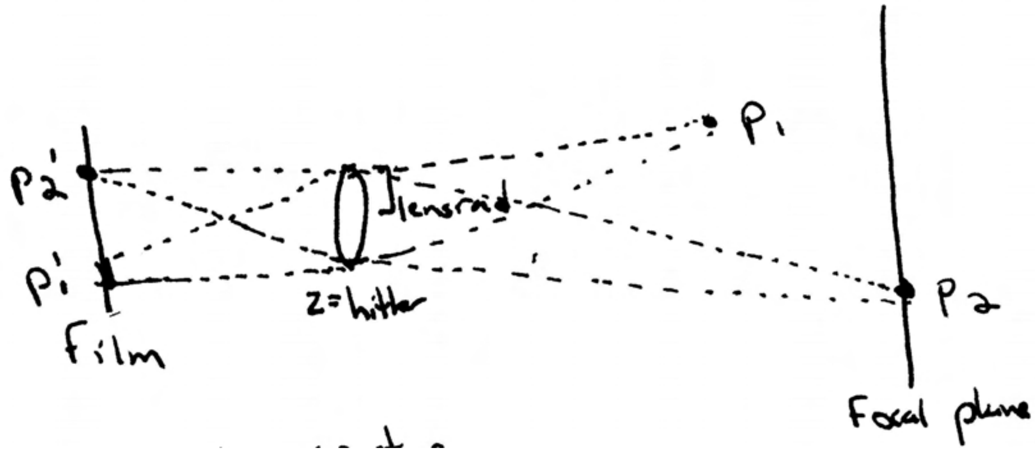


Figure 6.6: Real-world cameras have a lens with finite aperture and lens controls that adjust the lens position with respect to the film plane. Because the aperture is finite, objects in the scene aren't all imaged onto the film in perfect focus. Here, the point p_1 doesn't lie on the plane of points in perfect focus, so it images to an area p_1' on the film and is blurred. The point p_2 does lie on the focal plane, so it images to a point p_2' and is in focus. Both increasing aperture size and increasing an object's distance from the focal plane increase its blurriness.

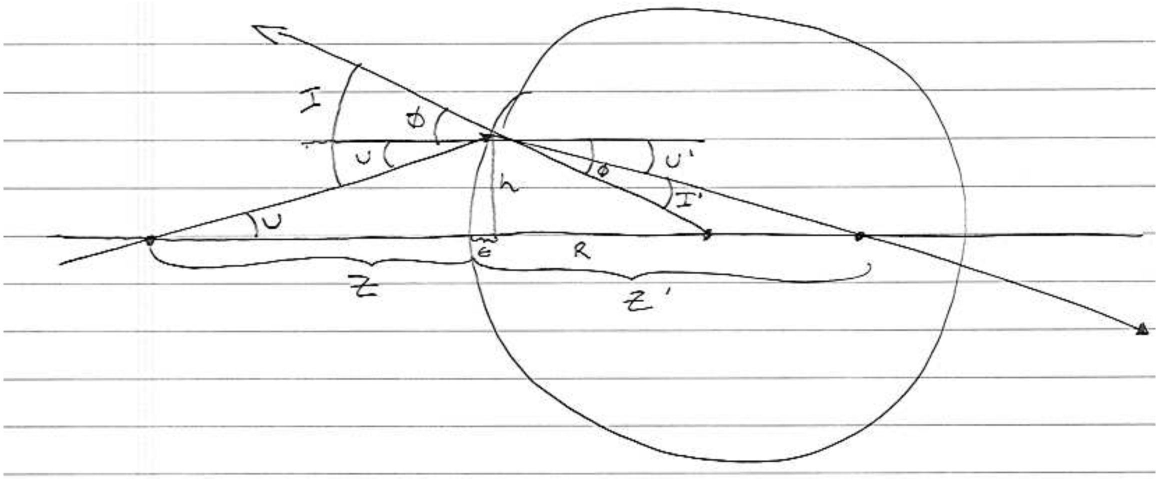


Figure 6.7: Cross-section of a spherical lens.

Note that $I = \Phi + U$ and $I' = \Phi - U'$. Now, in order to determine the relationships between these variables, we first make two simplifying assumptions: R is big, and the incoming ray is nearly parallel to the lens axis. These assumptions are called the *paraxial assumptions*. Two immediate consequences of these assumptions are that $\varepsilon = 0$ and for most small angular quantities α, $\sin\alpha = \tan\alpha = \alpha$.

Now, we simply apply Snell's law and simplify with our approximations. Snell's law gives:

$$\eta' \sin I' = \eta \sin I$$

But by the paraxial approximation, this is

$$\eta' I' = \eta$$
$$\eta' \left(\Phi - U'\right) = \eta \left(\Phi + U\right)$$

Now, we make use of some angular approximations:

$$\begin{aligned} U &\approx \tan U = \frac{h}{z} \\ U' &\approx \tan U' = -\frac{h}{z'} \\ \Phi &\approx \sin\Phi = -\frac{h}{R} \end{aligned}$$

Substituting these approximations, we have

$$\eta' \left(-\frac{h}{R} + \frac{h}{z'}\right) = \eta \left(-\frac{h}{R} + \frac{h}{z}\right)$$

Cancelling h and rearranging terms gives

$$\frac{\eta'}{z'} = \frac{\eta}{z} + \frac{\eta' - \eta}{R}$$

Notice that the form of this equation looks like $\frac{1}{z'} = \frac{1}{z} + C$, which leads directly to the perspective transformation. Also note that the relationship between z and z' does not depend on the angle of incidence; *all* rays through z refract to z'. This is how lenses are able to focus!

Of course, a real (thin) lens will have two spherical surfaces through which rays refract. Each surface contributes a $\frac{\eta - \eta'}{R}$ term, so the *refractive power* of a thin lens is given by

$$\left(\eta - \eta'\right) \left(\frac{1}{R_1} - \frac{1}{R_2}\right) = \frac{1}{f}$$

f is called the *focal length* of the lens, and is measured in units of $\frac{1}{m}$, which are sometimes called *diopters*.

Note that the focal length is not the same as the *focal distance*. Focal length is an inherent property of a lens and does not change[2].

[2] Readers familiar with traditional photography have probably used a *zoom lens*; these are special kinds of lenses that do in fact allow the focal length to change. This is accomplished by using multiple glass lenses and moving them relative to each other. The full lens *system* then has a focal

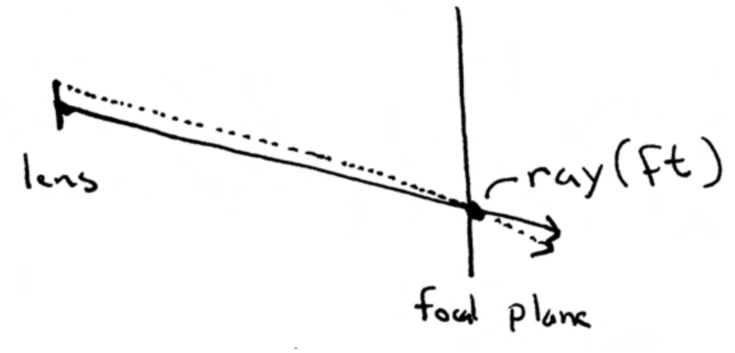


Figure 6.8: To adjust a camera ray for depth of field, we first compute the distance along the ray, `ft`, where it intersects the focal plane. We then shift the ray's origin from the center of the lens to the sampled lens position and construct a new ray (dashed line) from the new origin that still goes through the same point on the focal plane. This ensures that points on the focal plane remain in focus but that other points are blurred appropriately.

Focal distance, however, is not fixed and can almost always be changed in any camera system. A point in space will image through the lens to a finite area on the film plane, as shown in figure **??**. This area is typically circular, and is called the *circle of confusion*. The size of the circle of confusion determines how out-of-focus the point is. Note that this size depends on the size of the aperture; larger apertures will yield larger circles of confusion. This is why pinhole cameras render the entire scene perfectly in focus; the infinitessimally small aperture results in extremely small circles of confusion.

The size of the circle of confusion is also affected by the distance between the object and the lens. The focal distance is the distance from the lens to the plane where objects project to a circle of confusion with zero radius. These points will appear to be perfectly in focus. It is crucial to understand, however, that all types of film (analog or digital) will tolerate a certain amount of blur. This means that objects do not have to be exactly on the focal plane to appear in sharp focus. In the case of computer graphics, this corresponds (roughly) to the circle of confusion being smaller than a pixel. There will be some minimum and maximum distances from the lens at which objects will appear in focus; this range is called the lenses *depth of field*.

Projective cameras take two extra parameters for depth of field: one sets the size of the lens aperture and the other sets the focal distance.

⟨*Initialize depth of field parameters*⟩≡

```
  LensRadius = lensr;
  FocalDistance = focald;
```

⟨*ProjectiveCamera Protected Data*⟩+≡

```
  Float LensRadius, FocalDistance;
```

length that can be adjusted, even though the focal lengths of the individual glass elements remains fixed.

The math behind computing circles of confusion and depth of field boundaries is not difficult; it mostly involves repeated application of similar triangles. Even so, we can simulate focus in a ray tracer without understanding any of these constructions, and in just a few lines of code:

```
⟨Modify ray for depth of field⟩≡
  if (LensRadius > 0.) {
      ⟨Sample point on lens⟩
      ⟨Compute point on plane of focus⟩
      ⟨Update ray for effect of lens⟩
  }
```

To see why this is so simple, consider how the projective cameras simulate a pinhole camera: The rays generated for a pinhole camera must all pass through the pinhole (i.e., the center of the lens. However, for a lens of non-zero radius, we would like the ray to be able to pass through an arbitrary point on the lens. Since the camera is pointing down the z axis, we only need to modify the x and y coordinates of the ray origin to accomplish this.

The `ConcentricSampleDisk()` function, defined in Chapter 14, takes a (u,v) sample position in $[0,1]^2$ and maps it to the 2D disk with radius 1. To get a point on the lens, we scale these coordinates by the lens radius. The `Sample` provides the (u,v) lens-sampling parameters in the `Sample::lensX` and `Sample::lensY` fields. **can we rename these to lensU and lensV?**

Camera::ClipHither	203
ConcentricSampleDisk()	515
Point	33
iveCamera::FocalDistance	215
ectiveCamera::LensRadius	215
Ray::d	35
Sample::lensX	239
Sample::lensY	239

```
⟨Sample point on lens⟩≡
  Float lens_x, lens_y;
  ConcentricSampleDisk(sample.lensX, sample.lensY, &lens_x, &lens_y);
  lens_x *= LensRadius;
  lens_y *= LensRadius;
```

Once we have adjusted the origin of the ray away from the center of the lens, we need to determine the proper direction for the new ray. We could compute this using Snell's law, but the paraxial approximation and our knowledge of focus makes this much simpler. We know that *all* rays from our given image sample through the lens must converge somewhere on the focal plane. Finding this point of convergence is extremely simple; we just compute it directly for the ray through the center of the lens. Since rays through the lens center remain straight, no refraction is required!

Since we know that the focal plane is perpendicular to the z axis and the ray originates on the near clipping plane, intersecting the lens through the ray center with the plane is particularly simple. The t value of the intersection is given by:

$$t = \frac{\text{focalDistance} - \text{hither}}{\mathbf{d}(\mathrm{r})_z}$$

```
⟨Compute point on plane of focus⟩≡
  Float ft = (FocalDistance - ClipHither) / ray->d.z;
  Point Pfocus = (*ray)(ft);
```

Now we can compute the ray. The origin is shifted to the sampled point on the lens and the direction is set so that the ray still passes through the point on the plane of focus, `Pfocus`.

```
⟨Update ray for effect of lens⟩≡
  ray->o.x += lens_x;
  ray->o.y += lens_y;
  ray->d = Pfocus - ray->o;
```

6.3 Environment Camera

```
⟨environment.cpp*⟩≡
  #include "camera.h"
  #include "film.h"
  #include "paramset.h"
  ⟨EnvironmentCamera Declarations⟩
  ⟨EnvironmentCamera Definitions⟩
```

```
⟨EnvironmentCamera Declarations⟩≡
  class EnvironmentCamera : public Camera {
  public:
      ⟨EnvironmentCamera Public Methods⟩
  private:
      ⟨EnvironmentCamera Private Data⟩
  };
```

202	Camera
205	ProjectiveCamera
35	Ray::d
35	Ray::o

One advantage of ray tracing compared to scanline or rasterization rendering methods is that it's easy to have unusual image projections; we have great freedom in how the image sample positions are mapped into ray directions, since the rendering algorithm doesn't depend on properties such as straight lines in the scene always projecting to straight lines in the image.

In this section, we will describe a camera model that traces rays in all directions around a point in the scene, giving a two-dimensional view of everything that is visible from that point. Consider a sphere around the camera position in the scene; choosing points on that sphere gives directions to trace rays in. If we parameterize the sphere with spherical coordinates, each point on the sphere is associated with a (θ, ϕ) pair, where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. (See Section 5.3.2 on page 192 for more details on spherical coordinates.) This type of image is particularly useful because it compactly captures a representation of all of the incident light at a point on the scene. It will be useful later when we discuss environment mapping and environment lighting: two rendering techniques that are based on image-based representations of light in a scene.

Notice that the `EnvironmentCamera` derives directly from the `Camera` class, not the `ProjectiveCamera` class. This is because the environmental projection is non-linear and cannot be captured by a single 4×4 matrix. An image generated with this kind of projection is shown in Figure 6.9. θ values range from 0 at the top of the image to π at the bottom of the image, and ϕ values range from 0 to 2π, moving from left to right across the image.

All rays generated by this camera have the same origin; for efficiency we compute the world-space position of the camera once in the constructor.

Figure 6.9: An image rendered with the `EnvironmentCamera`, which traces rays in all directions from the camera position. The resulting image gives a representation of all light arriving at that point in the scene, and can be used for image-based lighting techniques that will be described in Chapters 13 and 16.

⟨*EnvironmentCamera Definitions*⟩≡

```
EnvironmentCamera::EnvironmentCamera(const Transform &world2cam,
        Float hither, Float yon, Float sopen, Float sclose,
        Film *film)
    : Camera(world2cam, hither, yon, sopen, sclose, film) {
    rayOrigin = CameraToWorld(Point(0,0,0));
}
```

Camera	202
Camera::CameraToWorld	203
Camera::ClipHither	203
Camera::ClipYon	203
EnvironmentCamera	217
Film	294
Point	33
Ray	36
Ray::maxt	36
Ray::mint	36
Ray::o	35
Transform	43

⟨*EnvironmentCamera Private Data*⟩≡

```
Point rayOrigin;
```

Note that the `EnvironmentCamera` still uses the near and far clipping planes to restrict the value of the ray's parameter t. In this case, however, these are really *clipping spheres*, since all rays originate at the same point and radiate outward.

given the above, don't we really want to ignore cliphither and clipyon if we're trying to sample the environment for lighting?

⟨*EnvironmentCamera Definitions*⟩+≡

```
Float EnvironmentCamera::GenerateRay(const Sample &sample,
        Ray *ray) const {
    ray->o = rayOrigin;
    ⟨Generate environment camera ray direction⟩
    ⟨Set ray time value⟩
    ray->mint = ClipHither;
    ray->maxt = ClipYon;
    return 1.f;
}
```

To compute the (θ,ϕ) coordinates for this ray, we first compute NDC coodinates from the raster image sample position. These are then scaled up to cover the (θ,ϕ) range and then the spherical coordinate formula is used to comupte the ray direction.

⟨*Generate environment camera ray direction*⟩≡

```
Float theta = M_PI * sample.imageY / film->yResolution;
Float phi = 2 * M_PI * sample.imageX / film->xResolution;
Vector dir(sinf(theta) * cosf(phi), cosf(theta),
    sinf(theta) * sinf(phi));
CameraToWorld(dir, &ray->d);
```

Our readers familiar with cartography will recognize this is the classic *Mercator projection*.

Further Reading

Akenine–Möller and Haines have a particularly well-written derivation of the orthographic and perspective projection matrices in *Real Time Rendering* (Akenine-Möller and Haines 2002). Other good references for projections are Rogers and Adams' *Mathematical Elements for Computer Graphics* (Rogers and Adams 1990), Watt and Watt (Watt and Watt 1992), Foley et al (Foley, van Dam, Feiner, and Hughes 1990) and Eberly's book on game engine design (Eberly 2001). (Originally Sutherland sketchpad stuff?)

Potmesil and Chakravarty did early work on depth of field and motion blur in computer graphics (Potmesil and Chakravarty 1981; Potmesil and Chakravarty 1982; Potmesil and Chakravarty 1983). Cook and collaborators developed a more accurate model for these effects based on *distribution ray tracing*; this is the approach we have implemented in this chapter (Cook, Porter, and Carpenter 1984; Cook 1986).

Kolb et al investigated simulating complex camera lens systems with ray tracing in order to model the imaging effects of real cameras (Kolb, Hanrahan, and Mitchell 1995). Another unusual projection method was used by Greene and Heckbert for generating images for Omnimax theaters (Greene and Heckbert 1986a). The `EnvironmentCamera` in this chapter is similar to the camera model described by Musgrave (Musgrave 1992).

203	`Camera::CameraToWorld`
294	`Film::xResolution`
294	`Film::yResolution`
678	`M_PI`
35	`Ray::d`
239	`Sample::imageX`
239	`Sample::imageY`
27	`Vector`

More about map projections... XXX

Exercises

6.1 Moving camera

6.2 Kolb, Mitchell, and Hanrahan have described a camera model for ray tracing based on simulating the lens system of a real camera, which is comprised of a set of glass lenses arranged to form an image on the film plane (Kolb, Hanrahan, and Mitchell 1995). Read their paper and implement a camera model in `lrt` that implements theyr algorithm for following rays through lens systems. Test your implementation with some of the lens description data from their paper.

7.Sampling and Reconstruction

237 Sampler

We will now describe how the Sampler chooses the points at which the image should be sampled, and how the pixels in the output image are computed from the radiance values computed for the samples. We saw a glimpse of how these sample points were used by lrt's camera model in the previous chapter. The mathematical background for this process is given by *sampling theory*: the theory of taking discrete sample values from continuous signals and then reconstructing new signals from those samples. Most of the previous development of sampling theory has been for encoding and compressing audio (e.g. over the telephone), and for television signal encoding and transmission. In rendering, we face the two-dimensional instance of this problem, where we're sampling an image at particular positions by tracing rays into the scene and then using the reconstructed approximation of the image function to compute values for the output pixels that form an image when displayed. It is important to carefully address the sampling problem in a renderer; a relatively small amount of work in improving sampling can substantially improve the images that the system generates.

A closely related problem is *reconstruction*: how to use the samples and the values that were computed for them to compute values for the pixels in the final image. Many samples around each pixel may contribute to its final value; the way in which they are blended together to compute the pixel's value can also noticeably affect the quality of the final image.

Include that famous Seurat painting with a blowup. Do we need permission?

Figure 7.1: Seurat's painting *blah blah frenchy something*. Notice how the overall painting appears to be a smoothly varying image, while the magnified inset reveals the pointillistic style.

7.1 Fourier Theory

What is an image? Although this might seem like a simple question, it belies a rich field of theory called *signal processing*. Consider first the case of human vision. The signal leaving the retina is a bunch of dots, one for each cone or rod[1]. But we do not perceive the world as an up-close Seurat painting (see Figure **??**; we see a continuous signal! The rods and cones in the eye *sample* the world, and our brain *reconstructs* the world from those samples.

Digital image synthesis is not so different. A renderer also produces a collection of individual colored dots. These dots are a discrete approximation to a continuous signal. Of course, this approximation can introduce errors, and the field of signal processing helps us understand, quantify, and lessen this error.

Everyone would probably agree that we would like as much spatial resolution as possible. However, in practice, our ability to generate ultra-high resolution images is limited by things like screen resolution, computational expense, or in the case of photography, film grain. At first glance, this situation seems pretty hopeless. How can we hope to capture tiny phenomena in a coarse image?

Of course, we know that the situation is *not* hopeless; we look at images every day without trouble. The answer lies in certain properties of the human visual system, such as area-averaging and insensitivity to noise. We will not give an overview of human vision in this book; see Glassner (Glassner 1995) for an introduction **better reference for human vision**.

One simple way to make pictures look their best is just not to display anything that is wrong. While this might seem obvious, it is not simple to define "wrong" in this context. In practice, this goal is unattainable because of the aforementioned discrete approximation. The sampling and reconstruction process introduces error known as aliasing, which can manifest itself in a variety of ways including jagged edges, strobing, flashing, flickering, or popping. These errors come up because the sampling process discards information from the continuous domain. To make matters worse, the visual system will tend to fill in data where there is none, so we also need to worry about how the missing data will be interpreted by the viewer.

In the one dimensional case, consider a signal given by a function $f(x)$ where we can evaluate f at any x' value we choose. Each such x' is a *sample position*, and the value of $f(x')$ is the *sample value*. The left half of Figure 7.2 shows a set of samples taken with uniform spacing (indicated by black dots) of a smooth 1D function. From a set of such samples, $(x', f(x'))$, we'd like to *reconstruct* a new signal $\tilde{f}$ that approximates the original function f. On the right side of Figure 7.2 is a piecewise-linear reconstructed function that approximates $f(x)$ by linearly interpolating neighboring sample values (readers already familiar with sampling theory will recognize this as reconstruction with a hat function). Because the only information we have about f comes from the sample values at the positions x', $\tilde{f}$ is likely

[1] Of course the human visual system is substantially more complex than this.

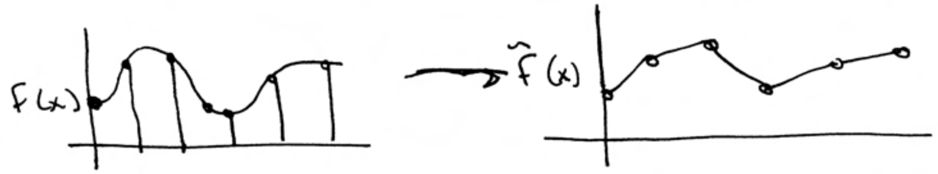


Figure 7.2: By taking a set of *point samples* of $f(x)$, we determine its value at those positions (left). From the sample values, we can *reconstruct* a function $\tilde{f}(x)$ which is an approximation to $f(x)$ (right). The sampling theorem, introduced in Section **??**, makes a precise statement about the conditions on $f(x)$ and the number of samples taken under which $\tilde{f}(x)$ is exactly the same as $f(x)$. That the exact function can sometimes be reconstructed exactly from mere point samples is remarkable.

to not match f perfectly since we have no knowledge of f's behavior between the sample values that we have.

We would like the reconstructed function to match the original function as closely as possible. The tool we will use to understand the quality of the match is *Fourier analysis*. Although we will review the basic concepts here, we hope that most readers have had some exposure to Fourier analysis already, and we will not present a full treatise on the subject here. Indeed, that would be a book in itself; Glassner provides a few useful chapters with a computer graphics angle (Glassner 1995), and Bracewell has a more complete treatment (Bracewell 2000). Students already very familiar with Fourier analysis are still encouraged to read this summary, since each presentation (unfortunately) seems to introduce its own notation.

Fourier analysis is based around the *Fourier transform*, which represents a signal in "frequency space". This representation helps us underestand what happens when a signal is turned into a bunch of samples. Basically, the Fourier transform decomposes a signal into a weighted sum of sinusoids. Just as a vector in $\mathbb{R}^n$ can be projected onto an arbitrary basis, so too can functions be projected onto new bases.

To understand what this means, first consider the example of vectors in $\mathbb{R}^3$. The typical reference frame is some origin O, and three perpendicular unit vectors $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$. Vectors in this space are written as $\mathbf{V} = (V_x, V_y, V_z)$. If we think of these as weights, we can write $\mathbf{V} = V_x\mathbf{X} + V_y\mathbf{Y} + V_z\mathbf{Z}$. Notice, however, that V_x is just the inner (dot) product of $\mathbf{V}$ and $\mathbf{X}$, and so we can alternately write $\mathbf{V} = (\mathbf{V}\cdot\mathbf{X})\mathbf{X} + (\mathbf{V}\cdot\mathbf{Y})\mathbf{Y} + (\mathbf{V}\cdot\mathbf{Z})\mathbf{Z}$.

This idea of taking the inner product of your object and each basis member in turn generalizes nicely to other spaces, such as the space of functions. For example, assume we have a 1D function $y = f(x)$ and we would like to represent it as a weighted sum of basis functions $\phi_i(x)$: $f(x) = \sum_i c_i \phi_i(x)$. We would also like our basis functions to have the same "perpendicular" property as our vector basis; for functions the corresponding notion is called orthogonality.

Two functions $\phi_i(x)$ and $\phi_j(x)$ are orthogonal over some interval $\Gamma = [a, b]$ if:

$$\int_a^b \overline{\phi_i(x)}\phi_j(x)dx = \begin{cases} 0 & i \neq j \\ \neq 0 & \text{otherwise} \end{cases}$$

Notice that we have taken the compliment of $\phi_i(x)$, in case it is a complex function. **Should explain this – see PODIS for good explanation.**

Now, suppose we want to project a function $f(x)$ onto some given orthogonal basis $\{\phi_i\}$. That is, we would like to write $f(x) = \sum_i c_i \phi_i(x)$. By minimizing the mean squared error of the projection over the interval $[a,b]$, it can be shown that the desired c_i are:

$$c_i = \frac{\int_a^b \overline{f(x)} \phi_i(x)}{\int_a^b \phi_i(x)\phi_i(x)}$$

7.1.1 Complex Exponentials

In Fourier analysis, the basis set we will chose are the *complex exponentials* $e^{i\omega t}$, where $i = \sqrt{-1}$. There are two important things to understand about the complex exponentials. First, these functions are periodic! Euler's formula is:

$$e^{i\omega t} = \cos(\omega t) + i\sin(\omega t)$$

From this formulation it is easy to see that the complex exponentials are periodic, with period $T = \frac{2\pi}{\omega}$.

Second, complex exponentials form an orthogonal basis for the space of all 1D functions! We take an infinite family of complex exponentials $\Psi_n(t) = e^{in\omega t}$, where $n \in \mathbb{Z}$. These functions are orthogonal over a full period $\Gamma = [t_0, t_0 + \frac{2\pi}{\omega}]$, which can be seen from the definition of orthogonality:

$$\begin{aligned} \int_\Gamma \overline{\Psi_n(t)} \Psi_m(t) dt &= int_\Gamma e^{-in\omega t} e^{im\omega t} dt \\ &= int_\Gamma e^{i\omega t(n-m)} dt \end{aligned}$$

If $n = m$, then this is just $\frac{2\pi}{\omega}$. If not,

$$\begin{aligned} &= \frac{1}{i(m-n)\omega} e^{i(m-n)\omega t} \Big|_{t_0}^{t_0+\frac{2\pi}{\omega}} \\ &= \frac{1}{i(m-n)\omega} e^{i(m-n)\omega t_0} \left(e^{i2\pi(m-n)} - 1 \right) \end{aligned}$$

But the right hand term is zero because of the periodicity of the complex exponentials, so we have orthogonality.

At this point we will skip some details regarding the projection of periodic vs. aperiodic functions and simply assert that we can project any function $x(t)$ onto the complex exponentials with the formula:

$$X(\omega) = \frac{1}{2\pi} \int_{-\infty}^{\infty} x(t) e^{-i\omega t} dt \tag{7.1.1}$$

This new function X is a function of *frequency*. It tells us how much of each frequency ω is present in the original signal. For example, a sinusoid $x(t) = \sin 2\pi t$ contains a single frequency $\omega = 1$, and the fourier transform of this $x(t)$ is indeed a delta function $X(\omega) = \delta(\omega - 1)$ **sanity check on this**.

BOX	SINC
GAUSSIAN	GAUSSIAN
CONSTANT	DELTA
SINUSOID	TRANSLATED DELTA
SHAH	SHAH

Table 7.1: Fourier pairs – need better captions and graphs and equations

Equation 7.1.1 is called the *Fourier analysis* equation, or sometimes just the *Fourier transform*. We can also transform from the frequency domain back to the spatial domain using the *Fourier synthesis* equation, or the *inverse Fourier transform*:

$$x(t) = \int_{-\infty}^{\infty} X(\omega) e^{i\omega t} dt \tag{7.1.2}$$

The reader should be warned that the constants in front of these integrals are not always the same in different derivations. Some authors (notably Glassner (Glassner 1995) and many in the physics community) prefer to multiply each integral by $\frac{1}{\sqrt{2\pi}}$ to emphasize the symmetry between the two equations.

Table 7.1.1 shows some of the more important functions in rendering, and their Fourier transforms.

7.2 Sampling Theory

Representing general continuous functions in a computer is impossible. When faced with this task, the typical solution is to carve the function domain into small regions and associate a number with each region. Informally, this is the sampling process.

In graphics, we could just choose a value for x, ω, t, and any other parameters we need to trace a ray, and call that a sample. But this is not how the real world works; sensors such as a CCD cell *integrate* over some finite area, they don't sample. This approximation can lead to many different kinds of errors.

Of course, this is only half the story. In order to produce something we can see, we have to eventually recreate some continuous intensity function. This is typically the job of the display. For example, in a CRT, each phosphor glows in some distribution not unlike a Gaussian function in area. Pixel intensity has some angular distribution as well; this is mostly uniform for CRT's, but as anyone who has tried to view a laptop from an angle knows, it can be quite directional on LCD displays. Informally, the process of taking a collection of numbers and converting them back to a continuous signal is the reconstruction process.

If you aren't careful about each of these two processes, you can get all kinds of artifacts. It is sometimes useful to distinguish between artifacts due to sampling and those due to reconstruction; when we wish to be precise we will call sampling artifacts "prealiasing", and reconstruction artifacts "postaliasing". Any attempt to fix these errors is broadly classified as "antialiasing", although the distinction of "antiprealiasing" and "antipostaliasing" is typically not made. The best way to understand, analyze, and eliminate these errors is through Fourier analysis.

Figure, maybe like the one from the 2000 Subdivision notes?

Figure 7.3: The convolution operator.

7.2.1 Convolution

Formally, the convolution operation $\otimes$ is defined as:

$$f(t) \otimes h(t) = \int_{\infty}^{\infty} f(\tau)h(t-\tau)d\tau$$

The convolution of two functions is a new function. Intuitively, to evaluate this new function at some value t, we center the function h (typically referred to as the “kernel” or “filter”) at t, and integrate the product of this shifted version of h and the function f. Figure 7.3 shows this operation in action.

This operation is crucial in sampling theory, mainly due to two important facts. The first is that the convolution of a function f with a delta function is simply the original function f (this is easy to prove — see exercise **??**).

The other is the “Convolution Theorem”. This theorem gives us an easy way to compute the Fourier transform of a convolved signal. In particular, the Convolution Theorem answers the question: given two functions f and h with associated Fourier transforms F and H, respectively, what is the Fourier transform $Y(\omega)$ of $f \otimes h$? From the definitions, we have:

$$\begin{aligned} Y(\omega) &= \int_{-\infty}^{\infty} (f \otimes h)e^{-i\omega t} dt \\ &= \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} f(\tau)h(t-\tau)d\tau \right) e^{-i\omega t} dt \end{aligned}$$

Changing the order of integration, we get:

$$= \int_{-\infty}^{\infty} f(\tau) \left(\int_{-\infty}^{\infty} h(t-\tau)e^{-i\omega t} dt \right) d\tau$$

A property of the Fourier transform is that $\mathcal{F}(g(t-t_0)) = e^{-i\omega t_0}\mathcal{F}(g(t))$, so:

$$\begin{aligned} &= \int_{-\infty}^{\infty} f(\tau)e^{-i\omega\tau}H(\omega)\, d\tau \\ &= F(\omega)H(\omega) \end{aligned}$$

This shows that convolution in the spatial domain is equivalent to *multiplication* in the frequency domain. We leave it to the reader to prove the property of the Fourier transform mentioned above (see exercise **??**). We can similarly show that multiplication in the spatial domain is equivalent to convolution in the frequency domain.

7.2.2 Back to Sampling

What does this all have to do with sampling and reconstruction? Recall the informal definition of the sampling process is that we just evaluate a function at some

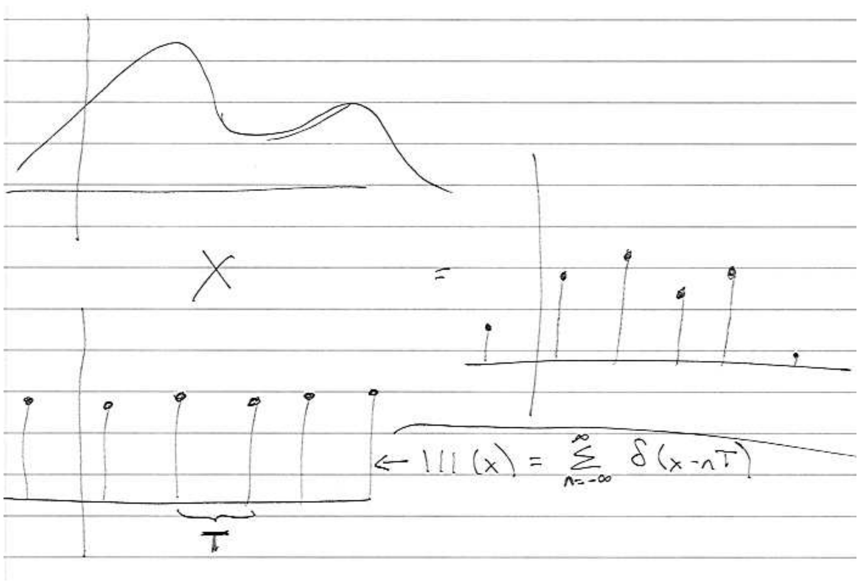


Figure 7.4: Formalizing the sampling process. The function $f(x)$ is multiplied by the shah function $\text{III}_T(x)$, leaving an infinite sequence of scaled delta functions.

regular interval and store the values. Formally, this corresponds to multiplying the function by a “shah”, or “impulse train” function. The shah $\text{III}_T(x)$ is defined as:

$$\text{III}_T(x) = \sum_{-\infty}^{\infty} \delta(x - nT)$$

where T defines the period, or *sampling rate*. This formal way of thinking about sampling is shown in figure 7.4.

Of course, this multiplication is happening in the spatial domain. Let’s consider what’s going on in the frequency domain. Furthermore, we will assume that the function $f(x)$ is bandlimited[2], so its Fourier transform will have compact support. A representative spectrum for $F(\omega)$ is shown in figure 7.5.

We also know the spectrum of the shah function $\text{III}_T(x)$, from table 7.1.1. The Fourier transform of a shah function with period T is another shah function with period $\frac{2\pi}{T}$. This reciprocal period is crucial — it means that if the samples are farther apart in the spatial domain, they are closer together in the frequency domain.

By the convolution theorem, we know that the fourier transform of our sampled signal is just the convolution of $F(\omega)$ and this new shah function. But remember that convolving a function with a delta function just yields a copy of that function. Therefore, convolving with a shah function yeilds an infinite series of copies of the original function, with spacing equal to the period of the shah. This is shown in figure 7.6. This is the spectrum of our series of samples.

7.2.3 Reconstruction

So how do we get back to our original function? Looking at figure 7.6, the answer is obvious: just throw away all but one of the copies of our spectrum, obtaining the

[2] A function $f(x)$ is bandlimited if there exists some frequency ω_0 such that $f(x)$ contains no frequencies greater than ω_0.

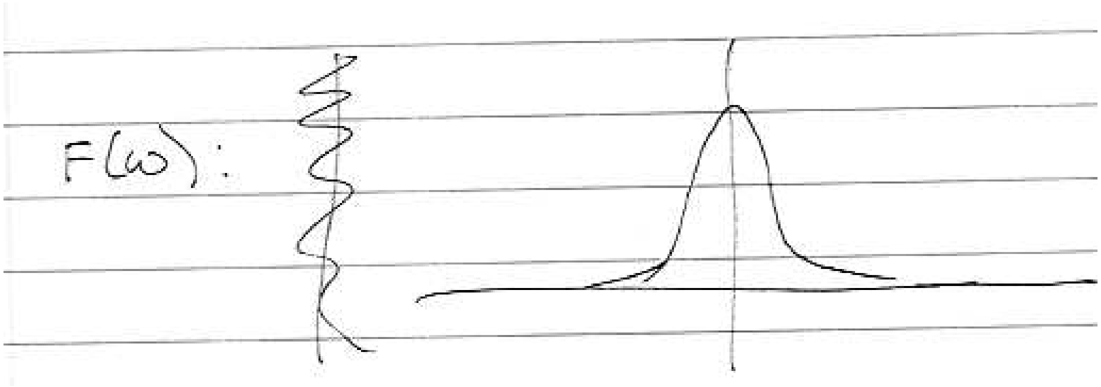
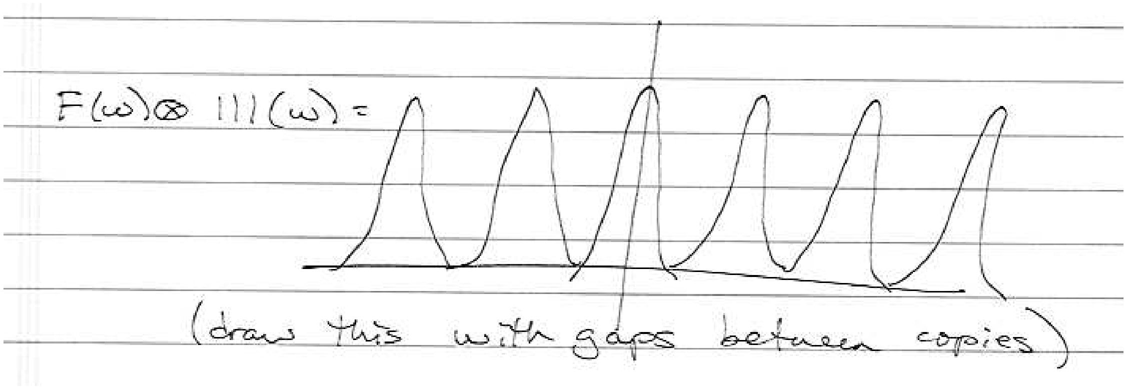


Figure 7.5: A representative bandlimited spectrum $F(\omega)$. Notice that all bandlimited functions must have spectra with compact support.



(draw this with gaps between copies)

Figure 7.6: The convolution of $F(\omega)$ and our shah function, resulting in infinitely many copies of the function F.

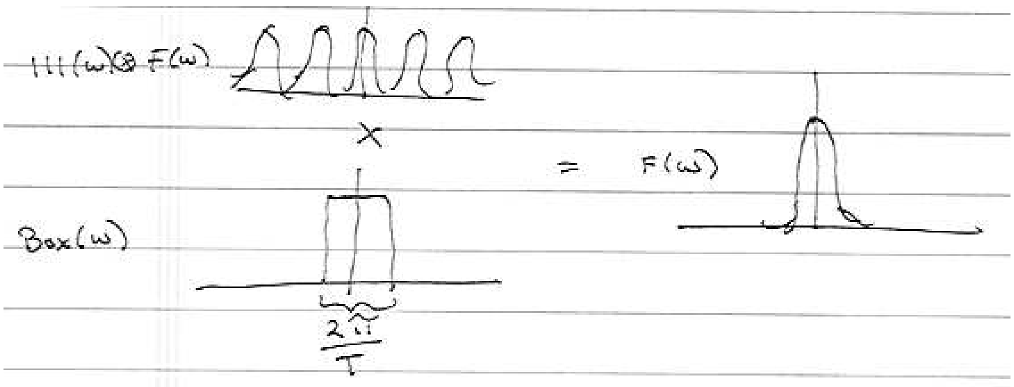


Figure 7.7: Multiplying a series of copies of $F(\omega)$ by the appropriate box function yields the original spectrum.

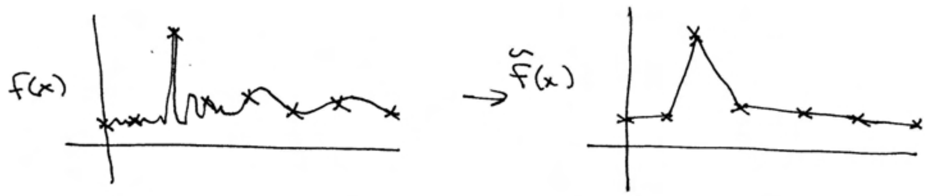


Figure 7.8: Undersampled 1D function: when the original function has undulations at a higher frequency than half the sampling frequency, it's not possible to reconstruct the original function accurately. The result of under-sampling a high-frequency function is *aliasing*, where low-frequency errors that aren't present in the original function appear. Here, the reconstructed function on the right has a much larger value over much of the left side of the graph of $\tilde{f}(x)$ than the original function $f(x)$ did.**had to take this out of ifdraft for a reference; might not make sense anymore.**

original curve $F(\omega)$. Then we have the original spectrum, and can easily compute the original function by means of the inverse Fourier transform.

In order to throw away all but the center copy of the spectrum, we just multiply (in the frequency domain, remember) our copies by a box function of the appropriate width, as shown in figure 7.7. The box function acts as a perfect low-pass filter.

This seems great! We started with $F(\omega)$, but the sampling process yeilded an infinite set of copies, so we throw all but one away with a low-pass filter, and we're back to the original function. Does this mean that we can always sample any function and perfectly reconstruct it? Surely there must be a catch.

7.2.4 Aliasing

The key to succesful reconstruction is the ability to exactly recover the original spectrum $F(\omega)$ by simply multiplying the sampled spectrum with a box function of the appropriate width. Notice that in figure 7.6, the spectra are separated by empty space, so perfect reconstruction is possible. Consider what happens, however, if

Figure 7.9: Aliasing from point sampling the function $\cos(x^2 + y^2)$; at the left side of the image, the function has a low frequency–tens of pixels per cycle–so it is represented accurately. Moving to the right, however, aliasing artifacts appear in the top image since the sampling rate doesn't keep up with the function's highest frequency. If high frequency elements of the signal are removed with filtering before sampling, as was done in the bottom image, the right side of the image takes on a constant grey color. (Example due to Don Mitchell.) Some aliasing errors remain in both images, due to the book printing process.

the original function was sampled with a lower sampling rate.

Recall that the fourier transform of a shah function III_T with period T is a new shah function with period $\frac{2\pi}{T}$. This means that if the spacing between samples increases in the spatial domain, the sample spacing decreases in the frequency domain, pushing the copies of our original spectrum $F(\omega)$ closer together. If the copies get too close together, they actually overlap! Because we are actually adding the copies together, the new spectrum no longer looks like many copies of the original, as shown in figure **??**. When we multiply this new spectrum by a box function, we obtain a spectrum that is similar but not equal to our original $F(\omega)$. In particular high-frequency details in the original signal have manifested themselves as low-frequency details in the new signal. These new low-frequency artifacts are called *aliases* (because high frequencies are "masquerading" as low frequencies), and the resulting signal is said to be *aliased*.

Figure 7.9 shows the effect of undersampling the two-dimensional function $f(x,y) = \cos(x^2 + y^2)$; the origin $(0,0)$ is at the center of the left edge of the image. At the left side of the top image, the reconstructed image accurately represents the original signal, though as we move farther to the right and f has higher and higher frequency content, aliasing starts. The circular patterns that appear in the center and right of the image are severe aliasing artifacts.

A naive solution to this problem would be to simply increase the sampling rate until the copies of the spectrum are sufficiently far apart as to not overlap, thereby eliminating aliasing completely. In fact, the *sampling theorem* tells us exactly what rate is required. This theorem says that as long as the frequency of uniform sample points ω_s is greater than *twice* the maximum frequency present in the signal ω_m, it is possible to reconstruct the original signal *perfectly* from the samples. This minimum sampling frequency is called the *Nyquist frequency*.

Unfortunately, this assumes that ω_m is finite, and therefore is only relevant for bandlimited signals. Non-bandlimited signals have spectra with *infinite* support, so no matter how far apart the copies of their spectra are (e.g., how high a sampling rate we use) there will always be overlap. This naturally leads to the question: "what signals are bandlimited"? Unfortunately, it turns out that in computer graphics, there are almost no bandlimited signals that we would want to sample. In particular, any function containing a discontinuity cannot be bandlimited, and we can therefore not perfectly sample and reconstruct it. This makes sense, because the function's discontinuity will always fall between two samples, and the samples provide no information about the location of the discontinuity.

If, in spite of all this, we still want to compute an un-aliased set of samples that represent the function, but our sampling rate isn't high enough to eliminate aliasing, sampling theory offers two options:

- Sample the function at a higher-frequency (*super-sample* it). If we can achieve a high enough sampling rate that the Nyquist limit is respected, the resulting sample values can be used to perfectly reconstruct the original function. Even if this is not possible and aliasing is still present, the error is always less.

- Filter (e.g., blur) the function so that no high frequencies remain that can't be captured accuately by the original sampling rate.

While the filter process modifies the original function by blurring it, it is generally preferable to have an alias-free sampled representation of the blurred function than an aliased representation of the original function. On the bottom of Figure 7.9, high frequencies have been removed from the function before sampling. The result is that the image takes on the average value of the function where previously there was aliasing error.

For the functions we need to sample in rendering, it's often either impossible or very difficult to know the frequency content of the signal being sampled. Nevertheless, the sampling theorem is still useful. First, it tells us the effect of increasing the sampling frequency: the point at which aliasing starts is pushed out to a higher frequency. Second, given some particular sampling frequency, it tells us the frequency beyond which we should try to remove high frequency data from the signal; if the function can filter itself directly according to the rate at which it is being sampled, aliasing can also be reduced. (This idea will be revisited in Chapter 11 when we introduce texture filtering.)

7.2.5 Application to image synthesis

The basic application of these ideas to the two-dimensional case of sampling and reconstructing images of rendered scenes is straightforward; we have an image, which we can think of as a function of two-dimensional (x,y) image locations to radiance values L:

$$f(x,y) \to L.$$

The good news is that, with our ray tracer, we can evaluate this function at any (x,y) point that we choose. The bad news is that we can only point sample the image function f: it's not generally possible to pre-filter f to the high frequencies from it before sampling[3].

It is useful to generalize the definition of the scene function to be a higher-dimensional function that also depends on the time t and (u,v) lens sample position at which it is sampled. Because the rays from the Camera are based on these five quantities, varying any of them gives a different potential ray and thus a potentially-different value of f. For a particular image position, the radiance at that point will generally vary across time and position on the lens (if there are moving objects in the scene and a finite-aperture camera, respectively.)

Even more generally, because many of the integrators defined in Chapter 16 use statistical techniques to estimate the radiance along a given ray, they may return a different radiance value when repeatedly given the same ray. If we further extend the scene radiance function to include sample values used by the integrator (e.g. to choose points on area light sources for illumination computation), we have an even-higher dimensional image function

Sampler 237

$$f(x,y,t,u,v,i_1,i_2,\ldots) \to L.$$

Sampling all of these dimensions well is an important part of generating high-quality imagery efficiently; for example, if we ensure that nearby (x,y) positions on the image don't tend to have similar (u,v) positions on the lens, we have better coverage of the sample space and better images for a given number of samples. The Sampler classes in the next few sections will address the issue of sampling all of these dimensions as well as possible.

7.2.6 Sources of Aliasing in Rendering

Geometry is one of the biggest causes of aliasing in rendered images. When projected onto the image plane, an object's boundary introduces a *step function*, where the image function's value discontinuously jumps from one value to another. A one-dimensional example of a step function is shown in Figure 7.10. Unfortunately, step functions have infinite frequency content, which means that no sampling density is sufficiently high to correctly capture them. Even worse, the perfect reconstruction filter causes artifacts when applied to aliased samples–ringing artifacts appear in the reconstructed image, an effect known as *Gibbs phenomenon*. Figure 7.11 shows an example of this effect for a 1D function. Choosing an effective reconstruction filter in the face of aliasing requires a mix of science, artistry

[3] The pre-filtering approach to antialiasing in image synthesis has been tried. Given a filter F and a signal S, the final pixel value can be written as $\int_A \delta(x,y)\,(S \otimes F)\,dx\,dy$. But we can rewrite this as $\int_A (\delta(x,y) \otimes F)\,S\,dx\,dy$! Here we are pre-filtering the sampler's delta function instead of the signal. This leads to the idea of generalized rays (e.g. cone tracing or pyramid tracing), which was explored in the mid 1980's. Eventually this approach was abandoned due to the complexity of intersecting a generalized ray with a primitive, and also due to the errors of repeated approximations introduced by secondary rays.

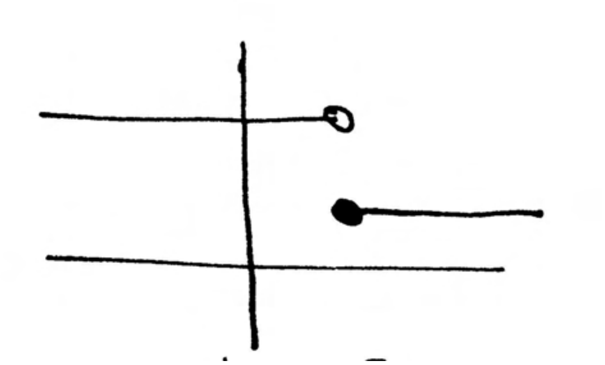


Figure 7.10: 1D step function: the function discontinuously jumps from one value to another. Such functions have infinitely-high frequency content. As such, a finite number of point samples can never adequately capture their behavior well enough so that we can apply perfect reconstruction.

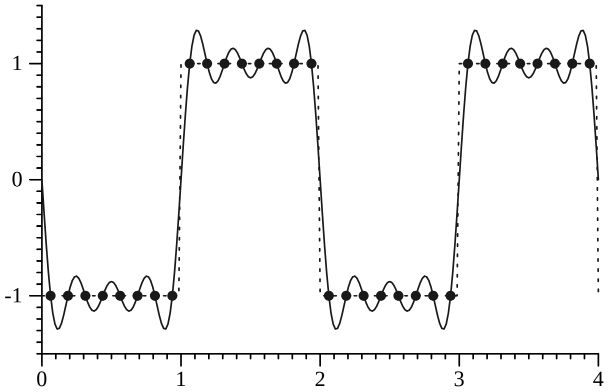


Figure 7.11: Illustration of Gibbs phenomenon. When a set of aliased samples of a function that hasn't been sampled at the Nyquist rate is reconstructed with the sinc reconstruction filter, the reconstructed function will have "ringing" artifacts, where it oscillates around the true function. Here a 1D step function (dashed line) has been sampled with a sample spacing of 0.125. When reconstructed with the sinc, the ringing appears (solid line).

and personal taste, as we will see later in this chapter. Another source of geometric aliasing is very small objects in the scene: if geometry is small enough that it falls in between samples on the image plane, it can unpredictably disappear and reappear over multiple frames of an animation.

Another source of aliasing can come from the texture and materials on an object. *Shading aliasing* can come from texture maps on objects that haven't been filtered correctly (addressing this problem is the topic of much of Chapter 11), or from small highlights on shiny surfaces; if the sampling rate is not high enough to sample these features adequately, aliasing will result. Furthermore, a sharp shadow cast by an object introduces another step function in the final image; while it is possible to identify the position of step functions from geometric edges on the image plane, detecting step functions from shadow boundaries is much more difficult. The key insight is that we can never remove all sources of aliasing from our images, but we must also develop techniques for mitigating their impact.

7.2.7 Non-uniform sampling

Although the image functions that we'll be sampling are known to have infinite-frequency components and thus can't be perfectly reconstructed, not all is lost. It turns out that varying the spacing between samples in a non-uniform way can reduce the visual impact of aliasing. For a fixed sampling rate that isn't sufficient to capture the function, both uniform and non-uniform sampling produce incorrect reconstructed signals. However, non-uniform sampling tends to turn the regular aliasing artifacts into noise, which is less objectionable to the human visual system.

Figure 7.12 shows this effect with the same cosine function example as was used as an example previously. On top, we have the function sampled at a fixed rate using uniform samples. Below, we have *jittered* each sample location, adding a small random number to its position in x and y before evaluating the cosine function. The aliasing patterns have been broken up and transformed into high-frequency noise artifacts.

This is an interesting result, since it shows that the best sampling patterns according to the signal processing view (which only argues for increasing the uniform sample frequency) don't always give the best results perceptually. In particular, some image artifacts are more visually acceptable than others. This observation will guide our development of good image sampling patterns through the rest of this chapter.

7.2.8 Adaptive sampling

One approach that has been suggested to combat aliasing is *adaptive super-sampling*: if we can identify the regions of the signal with frequencies higher than the Nyquist limit, we can take additional samples in those regions without needing to incur the computational expense of increasing the sampling frequency everywhere. It is hard to get this approach to work well in practice, however, since it's hard to find all of the places where super-sampling is needed. Most schemes for doing so are based on examining adjacent sample values and finding ones where there is a significant change in sample value between the two; the hypothesis is that the signal may have high frequencies in that region.

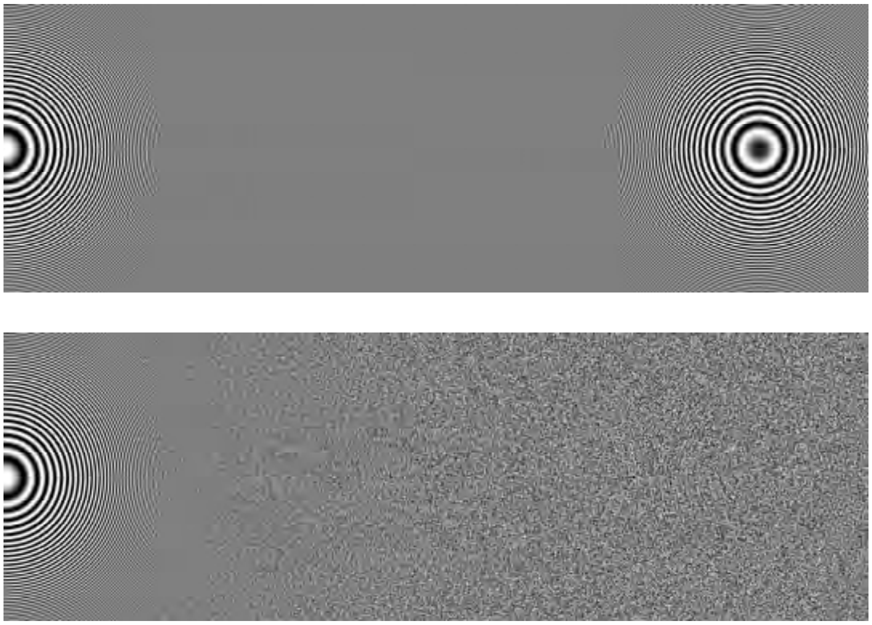


Figure 7.12: Jittered sampling of the aliased cosine function (top) changes the regular, low-frequency aliasing artifacts from under-sampling the signal into high-frequency noise (bottom). **I still see some ring-like structure near the right. Why is that?**

In general, adjacent sample values cannot tell us with certainty what is really happening in between them: the function may have huge variation between the two of them, but just happen to return to the same value at each. Or adjacent samples may have substantially different values without aliasing being present. For example, the texture filtering algorithms in Chapter 11 work hard to eliminate aliasing due to image maps and procedural textures; we would not want an adaptive sampling routine to take extra samples in an area where texture values are changing quickly but not with excessively high frequencies present.

In general, some areas that need super-sampling will always be missed by adaptive approaches, leaving the only recourse to be increasing the basic sampling rate anyway. Adaptive anti-aliasing works well to turn a very aliased image to a less aliased image, but is usually not able to make a visually flawless image more efficiently.

7.2.9 Understanding Pixels

There are two important subtleties related to the pixels that constitute a discrete image that are important to keep in mind throughout the remainder of this chapter. First, it is crucial to remember that pixels are point samples of the image function at discrete points on the image plane; there is no "area" associated with a pixel. As Alvy Ray Smith has emphatically pointed out, thinking of pixels as small squares with finite area is an incorrect mental model that leads to a series of errors (Smith 1995). By introducing the topics of this chapter with a signal processing approach, we have tried to lay the groundwork for the right mental model.

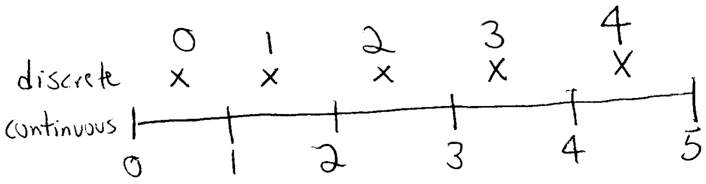


Figure 7.13: Discrete versus continuious representation of pixels...

The second issues is that there is a subtlety in how the pixel coordinates are computed. The pixels in the final image are naturally defined at discrete integer (x,y) coordinates on a pixel grid but the `Sampler`s in this chapter will be generating image samples at continuous floating-point (x,y) positions. Heckbert has written a short note that explains possible pitfalls in the interaction between these two domains. The natural way to map between them is to round continuous coordinates to the nearest discrete coordinate; this is appealing since it maps continuous coordinates that happen to have the same value as discrete coordinates to that discrete coordinate. However, the result is that given a range of discrete coordinates spanning a range $[x_0,x_1]$, then the set of continuous coordinates that covers that range is $[x_0-.5,x_1+.5)$. Thus, code that generates continuous sample positions for a given discrete pixel range is littered with 0.5 offsets. It easy to forget some of these, leading to subtle errors.

Sampler 237

If instead we truncate continuous coordinates c to discrete coordinates d by

$$d = \lfloor c \rfloor,$$

and convert from discrete to continuous by

$$c = d + .5,$$

then the range of continuous coordinates for the discrete range $[x_0,x_1]$ is naturally $[x_0,x_1+1)$ and our code is much simpler. This convention, which we will adopt in `lrt`, is shown graphically in Figure 7.13.

7.3 Image Sampling Interface

The core sampling declarations and functions are in the files `sampling.h` and `sampling.cpp`. Each of the various specific sample generation plug-ins is in its own source file in the `samplers/` directory.

```
⟨sampling.h*⟩≡
  #include "lrt.h"
  #include "geometry.h"
  ⟨Sampling Declarations⟩
  ⟨Sampling Inline Functions⟩
```

```
⟨sampling.cpp*⟩≡
  #include "lrt.h"
  #include "sampling.h"
  #include "transport.h"
  #include "volume.h"
  ⟨Sampler Method Definitions⟩
  ⟨Sample Method Definitions⟩
  ⟨Sampling Function Definitions⟩
```

We can now start to describe the operation of a few classes that generate good image sampling patterns. It may be surprising to see that some of them are have a significant amount of complexity behind them. In practice, creating high-quality sample patterns can substantially improve a ray tracer's efficiency, allowing it to create a high quality image with fewer rays than if a lower-quality pattern was used. The run-time expense for using the best sampling patterns is approximately the same as for lower-quality patterns, and because evaluating the value for each image sample is not inexpensive, doing this work is pays dividends.

All of the sampler implementations inherit from an abstract `Sampler` class that defines their interface. `Sampler`s have two main tasks:

1. Generating a sequence of multi-dimensional sample positions. Two dimensions give the raster-space image sample position and another gives the time at which the sample should be taken; this ranges from zero to one, and is scaled by the camera to cover the time period that the shutter is open. Two more sample values give a (u, v) lens position to sample for depth of field; these also vary from zero to one.

   Just as we can conquer the complexity of the 2D image function, most of the light transport algorithms in Chapter 16 use sample points to choose positions on area light sources when estimating illumination from them. For this and other tasks, they are most efficient then their sample points are also well-chosen. We also make choosing these points the job of the `Sampler`, since the best ways to select these points take into account the sample points chosen for adjacent image samples.

2. Taking the radiance values computed for particular image samples, reconstructing and filtering them, and computing the final values for the output pixels, which are usually located at different positions than any of the samples taken. We will describe this part of their operation in Section 7.7.

```
⟨Sampling Declarations⟩≡
  class Sampler {
  public:
      ⟨Sampler Interface⟩
      ⟨Sampler Public Data⟩
  };
```

All `Sampler`s take a few common parameters in their constructors that must be passed on to the base class's constructor. They are the overall image resolution in the *x* and *y* dimensions, the number of samples per pixel to take in each direction, the image crop window in normalized device coordinates $[0, 1]^2$, and a pointer to a Filter to be used to filter the image samples to compute the final pixel values. We store these values in member variables for later use.

The samples-per-pixel stuff is kind of annoying; it doesn't always correspond naturally to the sampler at hand. Thoughts? What's up with the different number per pixel in x and y?

⟨*Sampler Method Definitions*⟩+≡

```
Sampler::Sampler(int xstart, int xend,
                 int ystart, int yend,
                 int xs, int ys) {
    xPixelSamples = xs;
    yPixelSamples = ys;
    xPixelStart = xstart;
    xPixelEnd = xend;
    yPixelStart = ystart;
    yPixelEnd = yend;
}
```

The constructor just initializes variables that give the range of pixels in *x* and *y* for which we need to generate samples. Samples for pixels ranging from xPixelStart to xPixelEnd-1, inclusive, in *x* (and analogously in *y*) need to be generated by the Sampler.

Filter 281
Sampler 237

⟨*Sampler Public Data*⟩≡

```
int xPixelSamples, yPixelSamples;
int xPixelStart, xPixelEnd, yPixelStart, yPixelEnd;
```

Samplers need to implement the Sampler::GetNextSample() method, which is here declared as a pure virtual function. The Scene::Render() method calls this function until it returns false; as long as it keeps returning true, it should fill in the sample that is passed in with sample values for the next sample to be taken. All of the dimensions of the sample values it generates have values in the range $[0,1]$, except for imageX and imageY, which should be given with respect to the image size in raster coordinates.

⟨*Sampler Interface*⟩+≡

```
virtual bool GetNextSample(Sample *sample) = 0;
```

So that it's easy for the main rendering loop to figure out what percentage of the scene has been rendered after some number of samples have been processed, the Sampler::TotalSamples() method returns the total expected number of samples that the Sampler will be returning.[4]

⟨*Sampler Interface*⟩+≡

```
int TotalSamples() const {
    return xPixelSamples * yPixelSamples *
        (xPixelEnd - xPixelStart) * (yPixelEnd - yPixelStart);
}
```

[4] The low discrepancy and best candidate samplers, described later in the chapter, may actually return a few more or less samples than TotalSamples() reports. However, since computing the actual number that they will generate can't be done quickly, and since an exact number doesn't need to be known for this purpose, we just return the expected number.

7.3.1 Sample representation and allocation

The Sample structure is used by Samplers to store a single sample. A single Sample is allocated in the Scene::Render() method. For each camera ray to be generated, this Sample pointer is passed to the Sampler to have its values initialized. It is then passed to the camera and integrators, which read values from it to use to construct the camera ray and to do lighting calculations, respectively.

Cameras use the fixed fields in the Sample (imageX, imageY, etc.) to generate an appropriate camera ray, but various integrators have different needs from the camera, depending on the details of the light transport algorithm they implement. For example, the basic Whitted integrator doesn't do any random sampling, so doesn't need any sample values, but the direct lighting integrator uses sample values to randomly choose which light source to sample as well as to randomly choose positions on area light sources. Therefore, the integrators will be given an opportunity to request various sample types. Information about what they ask for is stored in the Sample object; when it is later passed to the particular Sampler implementation, it is the Sampler's responsibility to generate all of the requested types of samples.

```
⟨Sampling Declarations⟩+≡
  struct Sample {
      ⟨Sample Public Methods⟩
      ⟨Camera Sample Data⟩
      ⟨Integrator Sample Data⟩
  };
```

237 Sampler

```
⟨Camera Sample Data⟩≡
  Float imageX, imageY;
  Float lensX, lensY;
  Float time;
```

The Sample constructor immediately calls the Integrator::RequestSamples() methods of the surface and volume integrators asking them what samples they will need. The integrators can ask for 1D and/or 2D patterns, each with an arbitrary number of entries. For example, in a scene with two area light sources, where the integrator wanted to trace four shadow rays to the first source and eight to the second, the integrator would ask for two 2D sample patterns for each image sample, with four and eight samples each. (It needs a 2D pattern since two dimensions are needed to parameterize the surface of a light.)

If the integrator wanted to randomly select a single light source of out many, it could request a 1D sample pattern for this purpose. By informing the Sampler of as much of the random sampling that will be done during integration as possible, the Integrator makes it possible for the Sampler to carefully construct sample points that cover the high-dimensional sample space well. For example, if nearby image samples tend to use different parts of the area lights for their illumination ocmputations, the resulting images are generally better, since more information has been discovered.

In lrt, we don't allow the integrator to request three or higher dimensional sample patterns; these are much less commonly needed for rendering than one and two dimensional patterns. If necessary, the integrator can combine points from lower-dimensional patterns to get higher-dimensional sample points (e.g. a 1D and a 2D sample pattern to form a 3D pattern.) This may not be quite as good a set of sample points as we could generate with a direct 3D sample generation process, but the shortcomings aren't too bad in practice. This is the reason we provide 2D patterns instead of expecting integrators to request two 1D patterns.

The integrators' implementations of `Integrator::RequestSamples()` will in turn call the `Sample::Add1D()` and `Sample::Add2D()` methods below, which request another sample sequence in one or two dimensions, respectively, with a given number of sample values. After they are done calling these methods, the `Sample` constructor can continue, allocating storage for the requested sample values.

⟨*Sample Method Definitions*⟩≡

```
Sample::Sample(SurfaceIntegrator *surf,
        VolumeIntegrator *vol, const Scene *scene) {
    surf->RequestSamples(this, scene);
    vol->RequestSamples(this, scene);
    ⟨Allocate storage for sample pointers⟩
    ⟨Compute total number of sample values needed⟩
    ⟨Allocate storage for sample values⟩
}
```

Sample::n1D	241
Sample::n2D	241
Scene	8
SurfaceIntegrator	563
VolumeIntegrator	630

The `Sample::Add1D()` and `Sample::Add2D()` methods let the integrators ask for 1D and 2D sets of samples; the implementations of these methods just record the number of samples asked for and return an integer tag that the integrator can later use to access the sample values in the `Sample`.

⟨*Sample Public Methods*⟩+≡

```
u_int Add1D(u_int num) {
    n1D.push_back(num);
    return n1D.size()-1;
}
```

⟨*Sample Public Methods*⟩+≡

```
u_int Add2D(u_int num) {
    n2D.push_back(num);
    return n2D.size()-1;
}
```

It is the `Sampler`'s responsibility to store the samples it generates for the integrators in the `Sample::oneD` and `Sample::twoD` arrays. For 1D sample patterns, it needs to generate `n1D.size()` independent patterns, where the `i`th pattern has `n1D[i]` sample values. These values are stored in `oneD[i][0]` through `oneD[i][n1D[i]-1]`.

To access the samples, the integrator stores the sample tag returned by `Add1D()` in a member variable (for example, `sampleOffset`), and can then access the sample values in a loop like:

```
for (i = 0; i < sample->n1D[sampleOffset]; ++i) {
    Float s = sample->oneD[sampleOffset][i];
    ...
}
```

In 2D, the process is equivalent, but where the `i`th sample is given by the two values `sample->twoD[offset][2*i]` and `sample->twoD[offset][2*i+1]`.

⟨*Integrator* `Sample` *Data*⟩≡

```
vector<u_int> n1D, n2D;
Float **oneD, **twoD;
```

The `Sample` constructor first allocates memory to store the pointers. Rather than allocating memory twice, we do a single allocation that gives enough memory for both 1D and 2D pointers to the `oneD` and `twoD` sample arrays. `twoD` is then set to point at an appropriate offset into this memory, after the last pointer for `oneD`. Splitting up a single allocation like this is useful here because it ensures that `oneD` and `twoD` point to nearby locations in memory, which is likely to reduce cache misses.

⟨*Allocate storage for sample pointers*⟩≡

```
int nPtrs = n1D.size() + n2D.size();
if (!nPtrs) {
    oneD = twoD = NULL;
    return;
}
oneD = (Float **)AllocAligned(nPtrs * sizeof(Float *));
twoD = oneD + n1D.size();
```

667 AllocAligned()
658 vector

We then use the same trick for allocating memory for the actual sample values. First we find the total number of `Float` values needed.

⟨*Compute total number of sample values needed*⟩≡

```
int totSamples = 0;
for (u_int i = 0; i < n1D.size(); ++i)
    totSamples += n1D[i];
for (u_int i = 0; i < n2D.size(); ++i)
    totSamples += 2 * n2D[i];
```

And then we do a single allocation, handing it out in pieces to the various collections of samples.

⟨*Allocate storage for sample values*⟩≡

```
Float *mem = (Float *)AllocAligned(totSamples *
    sizeof(Float));
for (u_int i = 0; i < n1D.size(); ++i) {
    oneD[i] = mem;
    mem += n1D[i];
}
for (u_int i = 0; i < n2D.size(); ++i) {
    twoD[i] = mem;
    mem += 2 * n2D[i];
}
```

The `Sample` destructor, not shown here, just frees the dynamically allocated memory.

7.4 Stratified Sampling

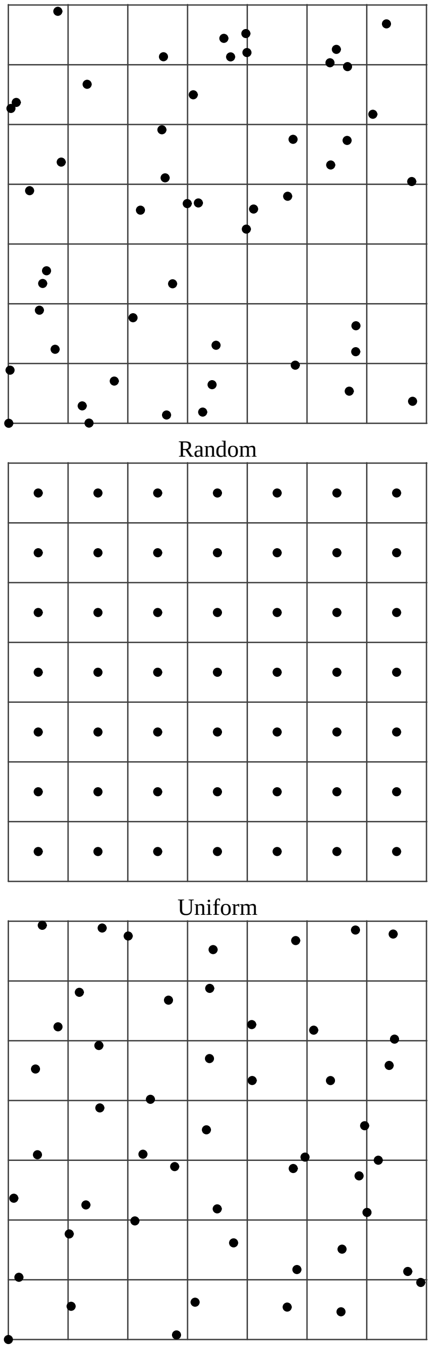
The first sample generator that we will introduce divides the image plane into rectangular regions and generates a single sample inside each region. These regions are commonly called *strata*, and this sampler is called the `StratifiedSampler`. The key idea behind stratification is that by subdividing the sampling domain into non-overlapping regions and taking a single sample from each one, we are less likely to miss important features of the image entirely, since the samples are guaranteed to not be all bunched together. Put another way, for a given number of samples, it does us no good if multiple samples are taken from nearby points in the sample space, since they don't give us much new information about the behavior of the image function that we didn't have already. Better is to take samples far away from the ones we've already taken; stratification improves this substantially. From a signal processing viewpoint, we are implicitly defining an overall sampling rate, where the smaller the strata are, the more of them we have, and thus the higher the sampling rate. In chapter 15 we will develop the necessary mathematics to properly analyze the benefit of stratified sampling; for now we will simply assert that it is better.

The stratified sampler places each sample by choosing a random point inside each stratum; this is done by *jittering* the center point of the stratum by a random amount up to half its width and height. The non-uniformity that results from this jittering helps turn aliasing into noise, as described earlier in the chapter. This sampler also offers a mode where this jittering is not done, giving uniform sampling in the strata; this unjittered mode is mostly useful for comparisons between different sampling techniques rather than rendering final images.

`StratifiedSampler 244`

Figure 7.14 shows a comparison of a few sampling patterns. On the top is a completely random sampling pattern: we have chosen a number of image samples to take and have computed that many random image locations–not using the strata at all. The result is a terrible sampling pattern; some regions of the image have few samples and other areas have clumps of many samples. In the middle is a stratified pattern without jittering (i.e. uniform super-sampling). On the bottom, we have jittered the uniform pattern, adding a random offset to each sample's location but keeping it inside its cell. This gives a better overall distribution than the purely random pattern while preserving the benefits of stratification, though there are still some clumps of samples and some regions that are under-sampled. We will present a more sophisticated image sampling methods in the next two sections that ameliorate some of these remaining problems. Figure 7.15 shows images of the `StratifiedSampler` in action and shows how jittered sample positions turn aliasing artifacts into less-objectionable noise.

```
⟨stratified.cpp*⟩≡
  #include "sampling.h"
  #include "paramset.h"
  #include "film.h"
  ⟨StratifiedSampler Declarations⟩
  ⟨StratifiedSampler Method Definitions⟩
```



Random

Uniform

Jittered

Figure 7.14: Three 2D sampling patterns. The random pattern on the top is a poor pattern, with many clumps of samples that leave large sections of the image poorly sampled. In the middle is a uniform stratified pattern which is better distributed but can exacerbate aliasing artifacts. On the bottom is a stratified jittered pattern, which turns aliasing from the uniform pattern into high-frequency noise while still maintaining the benefits of stratification.

```
⟨StratifiedSampler Declarations⟩≡
  class StratifiedSampler : public Sampler {
  public:
      ⟨StratifiedSampler Public Methods⟩
  private:
      ⟨StratifiedSampler Private Data⟩
  };
```

The `StratifiedSampler` generates samples by scanning over the pixels from left-to-right and top-to-bottom, generating all of the samples for the strata in each pixel before advancing to the next . The sampler holds the offset of the current pixel in the `xPos` and `yPos` member variables, which are initialized to point at the first pixel in the upper left of the image's pixel extent to start out. (Both the crop window and the sample filtering process can cause this corner to not necessarily be $(0, 0)$.)

```
⟨StratifiedSampler Method Definitions⟩≡
  StratifiedSampler::StratifiedSampler(int xstart, int xend, int ystart,
          int yend, int xs, int ys, bool jitter)
      : Sampler(xstart, xend, ystart, yend, xs, ys) {
      jitterSamples = jitter;
      xPos = xPixelStart;
      yPos = yPixelStart;
      ⟨Allocate storage for a pixel's worth of stratified samples⟩
      ⟨Generate stratified camera samples for (xPos,yPos)⟩
  }
```

AllocAligned()	667
Sampler	237
Sampler::xPixelSamples	238
Sampler::xPixelStart	238
Sampler::yPixelSamples	238
Sampler::yPixelStart	238

```
⟨StratifiedSampler Private Data⟩≡
  bool jitterSamples;
  int xPos, yPos;
```

The `StratifiedSampler` computes image, time, and lens samples for an entire pixel's worth of image samples all at once; this allows us to compute better-distributed patterns for the time and lens samples than we could if we compute each sample's values independently. Here we allocate enough memory to store all of the sample values for a pixel.

```
⟨Allocate storage for a pixel's worth of stratified samples⟩≡
  imageSamples = (Float *)AllocAligned(5 * xPixelSamples *
      yPixelSamples * sizeof(Float));
  lensSamples = imageSamples + 2 * xPixelSamples * yPixelSamples;
  timeSamples = lensSamples + 2 * xPixelSamples * yPixelSamples;
```

```
⟨StratifiedSampler Private Data⟩+≡
  Float *imageSamples, *lensSamples, *timeSamples;
```

Naive application of stratification to high-dimensional sampling quickly leads to an intractable number of samples. For example, if we divided the five-dimensional image, lens, and time sample space into four strata in each dimension, the total number of samples per pixel would be $4^5 = 1024$. We could reduce this impact by taking fewer samples in some dimensions (or not stratifying some dimensions, corresponding to a single stratum), but we would then lose the benefit of having

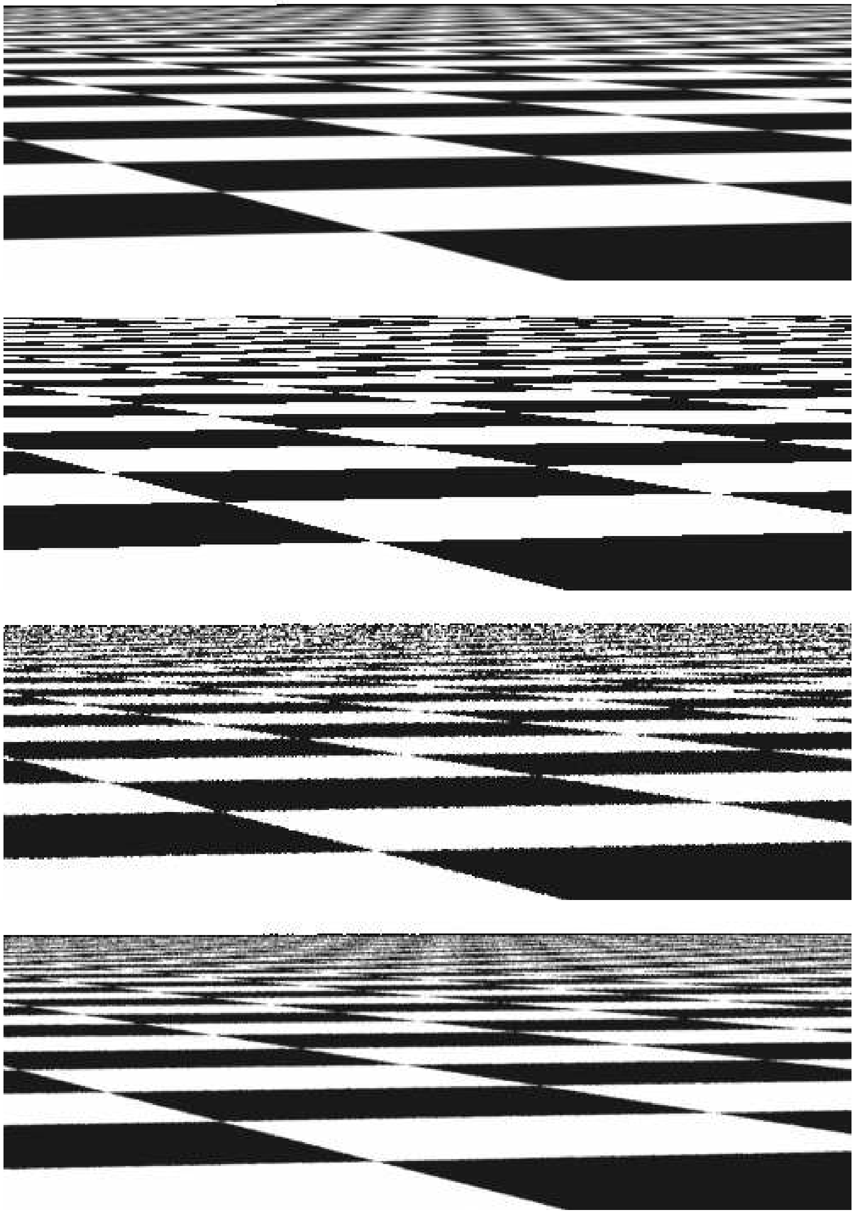


Figure 7.15: Comparisons of image sampling methods with a checkerboard texture: this is a difficult image to render well, since the checkerboard's frequency with respect to the pixel sample spacing increases toward infinity as we approach the horizon. On the top is a reference image, rendered with 256 samples per pixel, showing something close to the best result that we could hope to achieve. Below it is an image rendered with one sample per pixel, with no jittering; note the jaggy artifacts at the edges of checkers in the foreground and the artifacts in the distance where the checker function goes through many cycles between samples. As expected from the signal processing theory, that detail reappears incorrectly as lower-frequency aliasing. The third image shows the result of jittering the image samples, still just taking one sample per pixel; the regular aliasing of the second image has been replaced by less-objectionable noise artifacts. Finally, the bottom image shows the result of four jittered samples per pixel; the result is still inferior to the reference image, but is substantially better than a single sample per pixel.

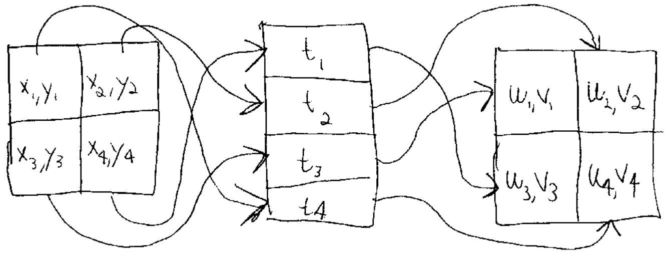


Figure 7.16: We can generate a good sample pattern for a pixel that reaps the benefits of stratification without requiring that all of the sampling dimensions be stratified simultaneously. Here, we have split (x,y) image position, time t, and (u,v) lens position into independent strata with roug regions each. Each is sampled independently. Then, a time sample and a lense sample is randomly associated with each image sample. We retain the benefits of stratification in each of the individual dimensions, without having to exponentially increase the total number of samples as if we had stratified all five dimensions simultaneously.

BestCandidateSampler 275
LDSampler 258

well-stratified samples in those dimensions. This problem with stratification is known as the *curse of dimensionality*.

We can reap most of the benefits of stratification without paying the price in excessive total sampling by computing lower-dimensional stratified patterns for subsets of the domain's dimensions and then randomly associating samples from each set of dimensions. Figure 7.16 shows the basic idea: we'd like to take just four samples for a pixel, while still having them be stratified over all dimensions. We independently generate four 2D stratified image samples, four stratified 1D time samples, and four stratified 2D lens samples. Then, we randomly associate a time and lens sample value with each image sample. The result of this is that each pixel has samples that together have good coverage of the sample space.

Ensuring this good distribution property at the pixel level is a reasonable level of granularity: we would like the samples that are close together on the image plane to have dissimilar time and lens samples—that would mean that the sample positions are not clumped together in the high-dimensional sampling space. If the samples are more widely distributed, the images we generate will generally be better, since they discover more about the values of the image function across the domain. If a sample that is a few (or many) pixels away has similar time or lens sample values, it doesn't negatively affect the overall image quality, since the need for other pixels to be computed from samples that cover the entire sample space well is far more important than extending stratification over large areas.[5]

This fragement generates the camera samples for the current pixel using this technique. As we hand out samples from `GetNextSample()`, the `samplePos` variable tracks how far through these arrays we are.

[5] Of course, with this method, the time and lens sample positions of the image samples for any particular output pixel are well-distributed, but the time and lens sample positions of image samples for neighboring pixels are not known, and thus may be similar. The `BestCandidateSampler` and the `LDSampler` in the next two sections are less succeptible to this problem.

⟨*Generate stratified camera samples for (*`xPos`*,*`yPos`*)*⟩≡

```
StratifiedSample2D(imageSamples, xPixelSamples, yPixelSamples,
    jitterSamples);
StratifiedSample2D(lensSamples, xPixelSamples, yPixelSamples,
    jitterSamples);
StratifiedSample1D(timeSamples, xPixelSamples*yPixelSamples,
    jitterSamples);
⟨Shift stratified image samples to pixel coordinates⟩
⟨Decorrelate sample dimensions⟩
samplePos = 0;
```

⟨*StratifiedSampler Private Data*⟩+≡

```
int samplePos;
```

We will implement 1D and 2D stratified sampling routines as utility functions, since they will be useful elsewhere in `lrt`. Both of them just loop over the given number of strata and place a sample point in each one.

⟨*Sampling Function Definitions*⟩≡

```
void StratifiedSample1D(Float *samp, int nSamples,
        bool jitter) {
    Float invTot = 1.f / nSamples;
    for (int i = 0;  i < nSamples; ++i) {
        Float delta = jitter ? RandomFloat() : 0.5f;
        *samp++ = (i + delta) * invTot;
    }
}
```

679 `RandomFloat()`
238 `Sampler::xPixelSamples`
238 `Sampler::yPixelSamples`
244 `StratifiedSampler::imageSamples`
244 `StratifiedSampler::jitterSamples`
244 `StratifiedSampler::lensSamples`
244 `StratifiedSampler::timeSamples`

⟨*Sampling Function Definitions*⟩+≡

```
void StratifiedSample2D(Float *samp, int nx, int ny,
        bool jitter) {
    Float dx = 1.f / nx, dy = 1.f / ny;
    for (int y = 0; y < ny; ++y)
        for (int x = 0; x < nx; ++x) {
            Float jx = jitter ? RandomFloat() : 0.5f;
            Float jy = jitter ? RandomFloat() : 0.5f;
            *samp++ = (x + jx) * dx;
            *samp++ = (y + jy) * dy;
        }
}
```

The `StratifiedSample2D()` utility function generates samples in the range $[0,1]^2$. Image samples, however, need to be expressed in terms of continuous pixel coordinates. Here we loop over all of the new stratified samples and add the (x,y) pixel number, such that the samples for the discrete pixel (x,y) range from continuous coordinates $[x,x+1)\times[y,y+1)$, following the convention for continuous pixel coordinates described in Section 7.2.9. Figure 7.17 reviews the relationship between discrete and continuous coordinates as relates to samples for a pixel.

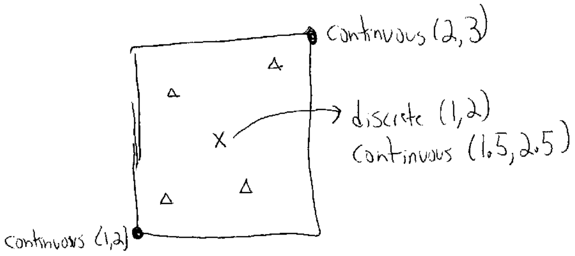


Figure 7.17: Review of sample placement–discrete and continuous...

⟨*Shift stratified image samples to pixel coordinates*⟩≡
```
for (int o = 0; o < 2 * xPixelSamples * yPixelSamples; o += 2) {
    imageSamples[o]   += xPos;
    imageSamples[o+1] += yPos;
}
```

In order to randomly associate a time and lens sample with each image sample, we shuffle the order of the time and lens arrays so that when we are initializing a `Sample` with the `i`th precomputed sample value for this pixel, we can just return the `i`th time and lens sample.

`Sampler::xPixelSamples`	238
`Sampler::yPixelSamples`	238
`StratifiedSampler`	244
`iedSampler::imageSamples`	244
`fiedSampler::lensSamples`	244
`fiedSampler::timeSamples`	244
`StratifiedSampler::xPos`	244
`StratifiedSampler::yPos`	244

⟨*Decorrelate sample dimensions*⟩≡
```
Shuffle(lensSamples, xPixelSamples*yPixelSamples, 2);
Shuffle(timeSamples, xPixelSamples*yPixelSamples, 1);
```

The `Shuffle()` utility function randomly permutes a sample pattern of `count` samples in `dims` dimensions. **More about this, how does Shuffle work? This comes up all the time in lots of programs and I think people are confused about how to randomly permute an array.**

⟨*Sampling Function Definitions*⟩+≡
```
void Shuffle(Float *samp, int count, int dims) {
    for (int i = 0; i < count; ++i) {
        u_int other = RandomUInt() % count;
        for (int j = 0; j < dims; ++j)
            swap(samp[dims*i + j], samp[dims*other + j]);
    }
}
```

We can now implement the `GetNextSample()` method of the `StratifiedSampler`. It starts by checking to see if it needs to generate a new pixel's worth of samples, or if all the samples have already been generated. It generates new samples if needed and then initializes the `Sample` pointer from the stored samples.

⟨*StratifiedSampler Method Definitions*⟩+≡
```
bool StratifiedSampler::GetNextSample(Sample *sample) {
    ⟨Compute new set of samples if needed for next pixel⟩
    ⟨Return next StratifiedSampler sample point⟩
    return true;
}
```

When the samplePos variable that keeps track of the current offset into the pixel-sized table of precomputed samples reaches the end of the table, we move ahead to the next pixel. First we try moving one pixel in *x*, and only move in *y* when we've reached the end of a scanline. Because the *y* strata counter yPos is only advanced when we reach the end of a row of samples in the *x* direction, once the *y* position counter has advanced past the bottom of the image, we're done and therefore return false.

⟨*Compute new set of samples if needed for next pixel*⟩≡

```
if (samplePos == xPixelSamples * yPixelSamples) {
    ⟨Advance to next pixel for stratified sampling⟩
    if (yPos == yPixelEnd)
        return false;
    ⟨Generate stratified camera samples for (xPos,yPos)⟩
}
```

To advance to the next pixel, we first try to move one pixel over in the *x* direction. If that takes us off of the end of the image, we reset the *x* position to the first pixel in the next *x* row of pixels and advance the *y* position.

⟨*Advance to next pixel for stratified sampling*⟩≡

```
if (++xPos == xPixelEnd) {
    xPos = xPixelStart;
    ++yPos;
}
```

```
239 Sample::imageX
239 Sample::imageY
239 Sample::lensX
239 Sample::lensY
239 Sample::time
238 Sampler::xPixelEnd
238 Sampler::xPixelStart
238 Sampler::yPixelEnd
247 StratifiedSampler::samplePos
244 StratifiedSampler::xPos
244 StratifiedSampler::yPos
```

Since the camera samples have been computed at this point, initializing the corresponding Sample fields is easy; we just copy the appropriate values from the sample tables. Generating samples for the integrators introduces some new subtleties, that we will discuss below.

⟨*Return next* StratifiedSampler *sample point*⟩≡

```
sample->imageX = imageSamples[2*samplePos];
sample->imageY = imageSamples[2*samplePos+1];
sample->lensX = lensSamples[2*samplePos];
sample->lensY = lensSamples[2*samplePos+1];
sample->time = timeSamples[samplePos];
⟨Generate stratified samples for integrators⟩
++samplePos;
```

Integrators introduce a new complication since they often use multiple samples per image sample in some dimensions rather than a single sample value, as is the case for lens position and time for cameras. As we have developed the topic of sampling so far, this leaves us with a quandary: if an integrator asks for a set of 64 two-dimensional sample values for each image sample, we have two different goals to try to fulfill:

- First, we would like each image sample's 64 integrator samples to themselves be well-stratified in 2D (e.g. with an 8 by 8 stratified grid). Stratification here will improve the quality of the integrator's results by ensuring that the computation that it does for each individual sample enjoys the benefits of stratified samples for the quantity that the integrator is computing.

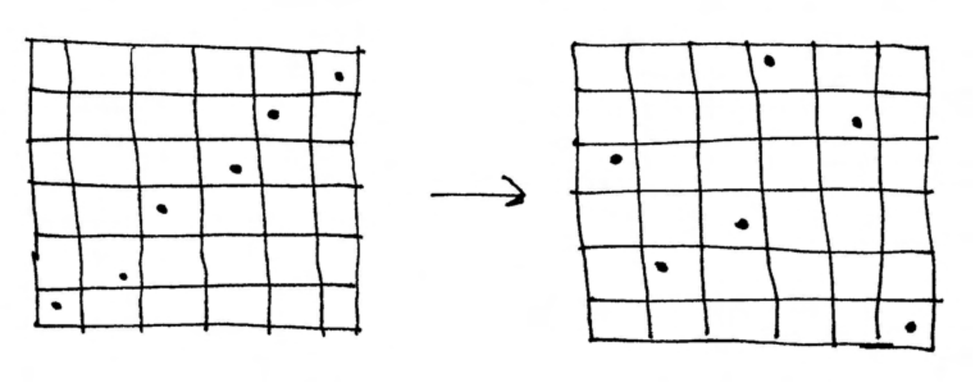


Figure 7.18: Latin hypercube sampling (sometimes called n-rooks sampling) chooses samples such that only a single sample is present in each row and each column of a grid. Here, we do this by generating samples randomly in the cells along the diagonal and then randomly permuting their coordinates. One advantage of LHS is that it can generate any number of samples with this good distribution property, not just a product of $m \times n$ samples, as with stratified patterns.

- Second, we would like to ensure that the set of integrator samples for one image sample aren't too similar to the samples for its neighboring samples. As with time and lens samples, we'd like the points to well-distributed with respect to their neighbors, so that over the region around a single pixel, we have good overall coverage of the entire sample space.

Rather than trying to solve both of these problems simultaneously here, we will only address the first one. The other samplers later in this chapter will revisit this issue with more sophisticated techniques.

A second complication from integrators comes from the possibility that they may ask for an arbitrary number of samples n per image sample, where stratification may not be easily applied. (For example, how to easily generate a stratified 2D pattern of 7 samples?) We could just generate an n by 1 or 1 by n stratified pattern, though this only gives us the benefit of stratification in one dimension, leading to no guarantee of a good poor pattern in the other dimension. Instead of trying to stratify these samples in the way that we did for image and lens samples, we will use an approach called *Latin hypercube sampling* (LHS), which can generate any number of samples in any number of dimensions with reasonably good distribution.

LHS uniformly divides each dimension's axis into n regions and generates a jittered sample in each of the n regions along the diagonal, as shown on the left side of Figure 7.18. These samples are then randomly shuffled in each dimension, given a pattern like the one shown on the right side of Figure 7.18. An advantage of LHS in comparison to stratified sampling is that it minimizes clumping of the samples when projected to any of the axes of the sampling dimensions. This is in contrast to stratified sampling, where $2n$ of the n by n samples in a 2D pattern may project to essentially the same point on each of the axes. Figure 7.19 shows this worst-case situation for a stratified sampling pattern, where the samples in the eight strata in the right half of the pattern are all at essentially the same location on the x axis.

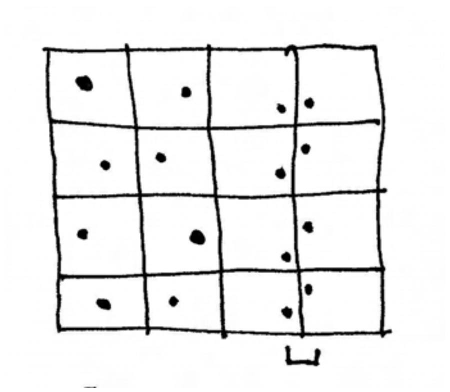


Figure 7.19: A worst-case situation for stratified sampling: for a 2D $n \times n$ pattern, up to $2n$ of the points may project down to essentially the same point on one of the two axes. When "unlucky" patterns like this are generated, the quality of results computed with them can suffer substantially.

LHS isn't necessarily an improvement to stratified sampling, however; it's easy to construct cases where the sample positions are essentially colinear and large areas of $[0,1]^2$ have no samples near them (e.g. when the permutation of the original samples is the identity, leaving them all where they started.) We will revisit this issue in the next section, where we will discuss sample patterns that are simultaneously stratified and distributed in a Latin Hypercube pattern.

679 RandomFloat()
241 Sample::n1D
241 Sample::n2D
241 Sample::oneD
241 Sample::twoD

⟨*Generate stratified samples for integrators*⟩≡

```
for (u_int i = 0; i < sample->n1D.size(); ++i)
    LatinHypercube(sample->oneD[i], sample->n1D[i], 1);
for (u_int i = 0; i < sample->n2D.size(); ++i)
    LatinHypercube(sample->twoD[i], sample->n2D[i], 2);
```

The general-purpose `LatinHypercube()` function generates an arbitrary number of LHS samples in an arbitrary dimension.

⟨*Sampling Function Definitions*⟩+≡

```
void LatinHypercube(Float *samples, int nSamples, int nDim) {
    ⟨Generate LHS samples along diagonal⟩
    ⟨Permute LHS samples in each dimension⟩
}
```

⟨*Generate LHS samples along diagonal*⟩≡

```
Float delta = 1.f / nSamples;
for (int i = 0; i < nSamples; ++i)
    for (int j = 0; j < nDim; ++j)
        samples[nDim * i + j] = (i + RandomFloat()) * delta;
```

To do the permutation, we loop over the samples, processing one dimension at a time and randomly permuting the sample points in each of them. Note that this is a different permutation than the `Shuffle()` routine above—that routine does one permutation, keeping all `nDim` sample points in each sample together, while here we do `nDim-1` separate permutations, separately permuting each dimension in turn. (It's not necessary to permute the first dimension.)

⟨*Permute LHS samples in each dimension*⟩≡

```
for (int i = 1; i < nDim; ++i) {
    for (int j = 0; j < nSamples; ++j) {
        u_int other = RandomUInt() % nSamples;
        swap(samples[nDim * j + i], samples[nDim * other + i]);
    }
}
```

7.5 ***ADV***: Low-Discrepancy Sequences

The underlying goal of the `StratifiedSampler` is to generate a well-distributed but not uniform set of sample points, where no two sample points are too close together and where there aren't any excessively large regions of the sample space with no samples in them. As Figure 7.14 showed, the jittered pattern does this much better than a random pattern does, though its quality can suffer when samples in adjacent strata happen to be close to the shared boundary of their two strata.

Mathematicians have developed a concept called *discrepancy* that can be used to evaluate the quality of a pattern of sample positions in a rigorous manner. Patterns that are well-distributed (in a manner to be formalized shortly) have low discrepancy values. One can thus consider the sample pattern generation problem to be one of finding a suitable *low-discrepancy* pattern of points. A number of deterministic techniques have been developed that generate low-discrepancy point sets, even in high-dimensional spaces. This section will use a few of them as the basis for a low-discrepancy sample generator.

`LDSampler`	258
`RandomUInt()`	248
`StratifiedSampler`	244

7.5.1 Definition of Discrepancy

Before defining the `LDSampler`, we will first introduce a formal definition for discrepancy. The basic idea behind it is that the "quality" of a set of points in an n dimensional space $[0,1]^n$ can be evaluated by looking at regions of the domain $[0,1]^n$, counting the number of points inside the region, and comparing the volume of these regions to the number of sample points inside them. In general, a given fraction of the volume should have roughly the same fraction of the sample points inside of it. While it's not possible for this to always be the case, we can still try to use patterns that minimize the difference between the volume estimated by the points and the actual volume (the *discrepancy*.)

To compute the discrepancy of a set of points, we first pick a family of shapes B that are subsets of $[0,1]^n$. For example, boxes with one corner at the origin are often used. This corresponds to:

$$B = \{[0,v_1] \times [0,v_2] \times \cdots \times [0,v_s]\}$$

where $0 \le v_i \le 1$. Given a sequence of sample points $P = x_1,\ldots,x_N$, the discrepancy of P with respect to B is[6]:

$$D_N(B,P) = \sup_{b \in B} \left| \frac{\sharp\{x_i \in b\}}{N} - \lambda(b) \right|$$

[6] The sup operator is the continuous analog of max. That is, sup $f(x)$ is a constant-valued function of x that passes through the maximum value taken on by $f(x)$.

where $\sharp\{x_i \in b\}$ is the number of points in b and $\lambda(b)$ is the volume of b.

The intuition for why this is a reasonable measure of quality is that $\frac{\sharp\{x_i \in b\}}{N}$ is an approximation of the volume of the box b. Therefore, the discrepancy is the worst error over all possible boxes introduced by this way of approximating volume. When the set of shapes B is the set of boxes with a corner at the origin (described above), this is called the *star discrepancy* $D_N^*(P)$. (Other popular sets of shapes to use to compute discrepancy include axis aligned boxes, where the restriction that one corner be at the origin has been removed).

For a few particular point sets, the discrepancy can be computed analytically. For example, consider the set of points in one dimension

$$x_i = \frac{i}{N}.$$

We can see that the star discrepancy of x_i is

$$D_N^*(x_1, \ldots, x_n) = \frac{1}{N}.$$

For example, take the interval $B = [0, \frac{1}{N})$. Then $\lambda(B) \approx \frac{1}{N}$, but $\sharp\{x_i \in B\} = 0$. This interval (and the intervals $[0, \frac{2}{N})$, etc.) is the interval where the largest differences between volume and fraction of points inside the volume are seen.

We can improve on the star discrepancy of this sequence by modifying it slightly:

$$x_i = \frac{i - \frac{1}{2}}{N}.$$

Then

$$D_N^*(x_i) = \frac{1}{2N}.$$

The bounds for the star discrepancy of a sequence of points in 1D has been shown to be

$$D_N^*(x_i) = \frac{1}{2N} + \max_{1 \le i \le N} \left| x_i - \frac{2i-1}{2N} \right|.$$

Thus, the modified sequence above has the lowest possible discrepancy for a sequence in 1D. In general, it is much easier to analyze and compute bounds for the discrepancy of sequences in 1D than in higher dimensions. For less simple point sequences, and for sequences in higher dimensions, the discrepancy often must be estimated numerically, by constructing a large number of shapes B, computing their discrepancy, and reporting the maximum.

The astute reader will notice that according to the low-discrepancy measure, this uniform sequence in 1D is optimal, though earlier in this chapter, we had determined that irregular jittered patterns were perceptually superior to uniform patterns for image sampling in 2D since they replaced aliasing error with noise. In that framework, uniform samples are clearly not optimal. Fortunately, low-discrepancy patterns in higher dimensions are much less uniform than they are in one dimension and thus usually work reasonably well as sample patterns in practice. Nevertheless, their underlying uniformity is probably the reason why low-discrepancy patterns can be more prone to aliasing than patterns with true pseudo-random variation.

7.5.2 Constructing low-discrepancy sequences

Given the goal of constructing a low-discrepancy sequence, we will now introduce techniques that have been developed specifically to generate sequences of points that have low discrepancy. The techniques that we will describe are built on top of a construction called the *radical inverse*. It is based on the fact that a positive integer value n can be expressed in base b with a sequence of digits $d_m \ldots d_2 d_1$ uniquely determined by:

$$n = \sum_{i=1}^{\infty} d_i b^{i-1}.$$

The radical inverse function Φ_b in base b takes an non-negative integer and converts it to a floating-point value in $[0,1)$, by reflecting these digits about the decimal point:

$$\Phi_b(n) = 0.d_1 d_2 \ldots a_m.$$

Thus, the contribution of the digit d_i to the radical inverse is

$$d_i \cdot \frac{1}{b^i},$$

The function `RadicalInverse()` computes the radical inverse for a given number `n` in the base `base`. It first computes the value of d_1 by taking the remainder of the number n when divided by the base and adds $d_1 b^{-1}$ to the radical inverse value. It then divides n by the base, effectively chopping off the last digit so that the next time through the loop, it can compute d_2 by finding the remainder base `base` and adding $d_2 b^{-2}$ to the sum, etc. This process continues until n is zero, at which point we have found the last non-zero d_i value.

⟨*Sampling Declarations*⟩+≡

```
inline Float RadicalInverse(int n, int base) {
    Float val = 0;
    Float invBase = 1.f / base, invBi = invBase;
    while (n > 0) {
        ⟨Compute next digit of radical inverse⟩
    }
    return val;
}
```

⟨*Compute next digit of radical inverse*⟩≡

```
int d_i = (n % base);
val += d_i * invBi;
n /= base;
invBi *= invBase;
```

Given the `RadicalInverse()` function, we can start constructing low discrepancy sequences. One of the simplest low discrepancy sequences is the Van Der Corput Sequence, which is a one-dimensional sequence given by the radical inverse function in base two.

$$x_i = \Phi_2(i)$$

Figure 7.20 shows the first few values of the Van Der Corput sequence; notice how it recursively splits the intervals of the 1D line in half, generating a sample point at

n	base 2	Φ_2 (n)
1	1	.1 = 1/2
2	10	.01 = 1/4
3	11	.11 = 3/4
4	100	.001 = 1/8
5	101	.101 = 5/8
⋮	⋮	⋮

Figure 7.20: The radical inverse Φ_2 (n) of the first few positive integers, computed in base 2. Notice how successive values of Φ_2 (n) are far from the previous values of Φ_2 (n).

the center of each interval. The discrepancy of this sequence is

$$D_N^*(P) = O\left(\frac{\log N}{N}\right),$$

which matches the best discrepancy that has been attained for infinite sequences of *d* dimensions,

$$D_N^*(P) = O\left(\frac{(\log N)^d}{N}\right).$$

Two well-known low-discrepency sequences that are defined in an arbitrary number of dimensions are the *Halton* and *Hammersley* sequences. Both use the radical inverse function as well. To generate an *n* dimensional Halton sequence, we use the radical inverse base *b*, with a different base for each dimension of the square and where the bases used are all relatively prime to each other. (A natural choice is to use the first *n* prime numbers $(p_1, \ldots, p_n)$.)

$$x_i = (\Phi_2(i), \Phi_3(i), \Phi_5(i), \ldots, \Phi_{p_n}(i)).$$

One of the most useful characteristics of the Halton sequence is that it can be used even if the total number of samples needed isn't known in advance; all prefixes of a given sequence are well-distributed, so thus as additional samples are added to the sequence, the low-discrepancy property will be maintained. The discrepancy of a *d*-dimensional Halton sequence is

$$D_N^*(x_i) = O\left(\frac{(\log N)^d}{N}\right),$$

which is asymptotically optimal.

If the number of samples to be taken (N) is known in advance, the discrepancy can be improved slightly. Hammersley point sets are defined by:

$$x_i = \left(\frac{i - \frac{1}{2}}{N}, \Phi_{b_1}(i), \Phi_{b_2}(i), \ldots, \Phi_{b_n}(i)\right),$$

where N is the total number of samples to be taken and as before all of the bases b_i are relatively prime. The top half of Figure 7.21 shows plots of the first hundred points of the 2D Hammersley and Halton sequences.

The *folded radical inverse* function can be used in place of the original radical inverse function to reduce the discrepancy of Hammersley and Halton sequences. The folded radiacal inverse is defined by adding the offset i to the ith digit d_i and taking the result modulus b before adding the result to the next digit to the right of the decimal point.

$$\Psi_b(n) = \sum_i ((d_i + i - 1) \mod b) \cdot \frac{1}{b^i},$$

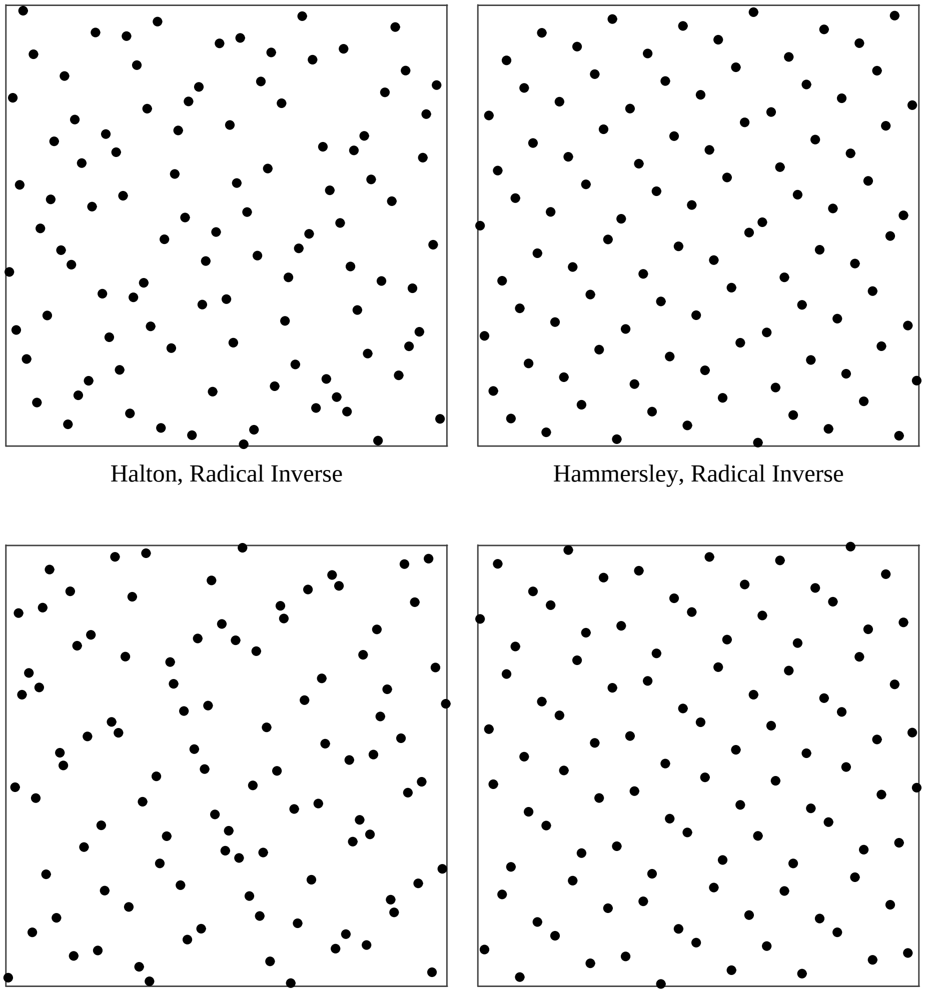
The `FoldedRadicalInverse()` function computes Ψ_b. It is generally similar to the original `RadicalInverse()` function, with two modifications. First, it needs to track which digit is currently being processed, so that the appropriate offset can be added before the modulus computation; this is done with the `modOffset` variable. Second, it needs to handle the fact that Ψ_b is actually an *infinite* sum. Even though the digits d_i are zero after a finite number of terms, the offset that is added ensures that most terms beyond the point where $d_i = 0$ will be non-zero. Fortunately, the finite precision of computer floating-point numbers solves this problem: we can conservatively stop adding digits to the folded radical inverse as soon as we detect that `invBi` is small enough such that adding its contribution to `val` is certain to leave `val` unchanged. The test in the `while` loop watches for this to happen.

⟨*Sampling Declarations*⟩+≡
```
inline Float FoldedRadicalInverse(int n, int base) {
    Float val = 0;
    Float invBase = 1.f/base, invBi = invBase;
    int modOffset = 0;
    while (val + base * invBi != val) {
        ⟨Compute next digit of folded radical inverse⟩
    }
    return val;
}
```

⟨*Compute next digit of folded radical inverse*⟩≡
```
int digit = ((n+modOffset) % base);
val += digit * invBi;
n /= base;
invBi *= invBase;
++modOffset;
```

When the folded radical inverse is used to generate the Hammersley and Halton point sets, they are known as the Hammersley-Zaremba and Halton-Zaramba point sets, after the inventor of the folded radical inverse function. Graphs of the first 100 Hammersley-Zaremba and Halton-Zaremba points are shown in the bottom half of Figure 7.21. It's possible to see visually that the Hammersley sequence has lower discrepancy than the Halton sequence–there are far fewer clumps of nearby sample points. Furthermore, one can see that the folded radical inverse function reduces the discrepancy of the Hammersley sequence; its effect on the Halton sequence is less visually obvious, however.

7.5.3 The Low-Discrepancy Sampler



Halton, Radical Inverse

Hammersley, Radical Inverse

Halton, Folded Radical Inverse

Hammersley, Folded Radical Inverse

Figure 7.21: Graphs of the first 100 points in the Halton (left) and Hammersley (right) low-discrepancy point sequences, using the radical inverse Φ_b in the top row, and the folded radical inverse Ψ_b in the bottom row. The Hammersley sequence has lower discrepancy than the Halton sequence, at the cost of requiring that the number of samples to be taken be known in advance. The folded radical inverse function improves the discrepancy of both sequences.

```
⟨lowdiscrepancy.cpp*⟩≡
  #include "sampling.h"
  #include "paramset.h"
  #include "film.h"
  ⟨LDSampler Declarations⟩
  ⟨LDSampler Method Definitions⟩
```

The `LDSampler` uses one of the two radical inverse functions to generate a Hammersley point set for sampling. It works by mapping the first two dimensions of the Hammersley points from $[0,1]^2$ to a square region on the image plane, starting at (`xPixelStart`, `yPixelStart`) and scaled by a constant amount in both directions so that it covers the pixels up to (`xPixelEnd`, `yPixelEnd`), and possibly beyond in one of the two dimensions (if the image is not square). Any generated samples that are past (`xPixelEnd`, `yPixelEnd`) are discarded. The total number of samples to be generated is determined by computing the total number of pixels in the extent that is being sampled times the number of samples to be taken per pixel. Lens and time sample value positions are found by taking additional dimensions from the Hammersley point set, while the integrator samples again will require special handling.

For non-square images, it's important to use the approach described above for image sampling, generating extra samples and rejecting those that are outside of the image region, rather than scaling the Hammersley point set by different amounts in the *x* and *y* directions. Scaling by different amounts would effectively cause the samples to be more closely spaced in one direction than the other, which is certainly not what one expects when rendering a non-square image. Figure 7.22 compares results of using the low discrepancy sampler to using the stratified sampler of the previous section to sample the checkerboard texture.

`Sampler` 237

```
⟨LDSampler Declarations⟩≡
  class LDSampler : public Sampler {
  public:
      ⟨LDSampler Public Methods⟩
  private:
      ⟨LDSampler Private Data⟩
  };
```

The `LDSampler` constructor computes the length of a side of the square region samples are generated inside of, `extent`, the total number of samples to generate (and its inverse), `nSamples` and `invNSamples`, and the sample number of the next Hammersley point x_i, to be computed by `GetNextSample()`, `curSample`.

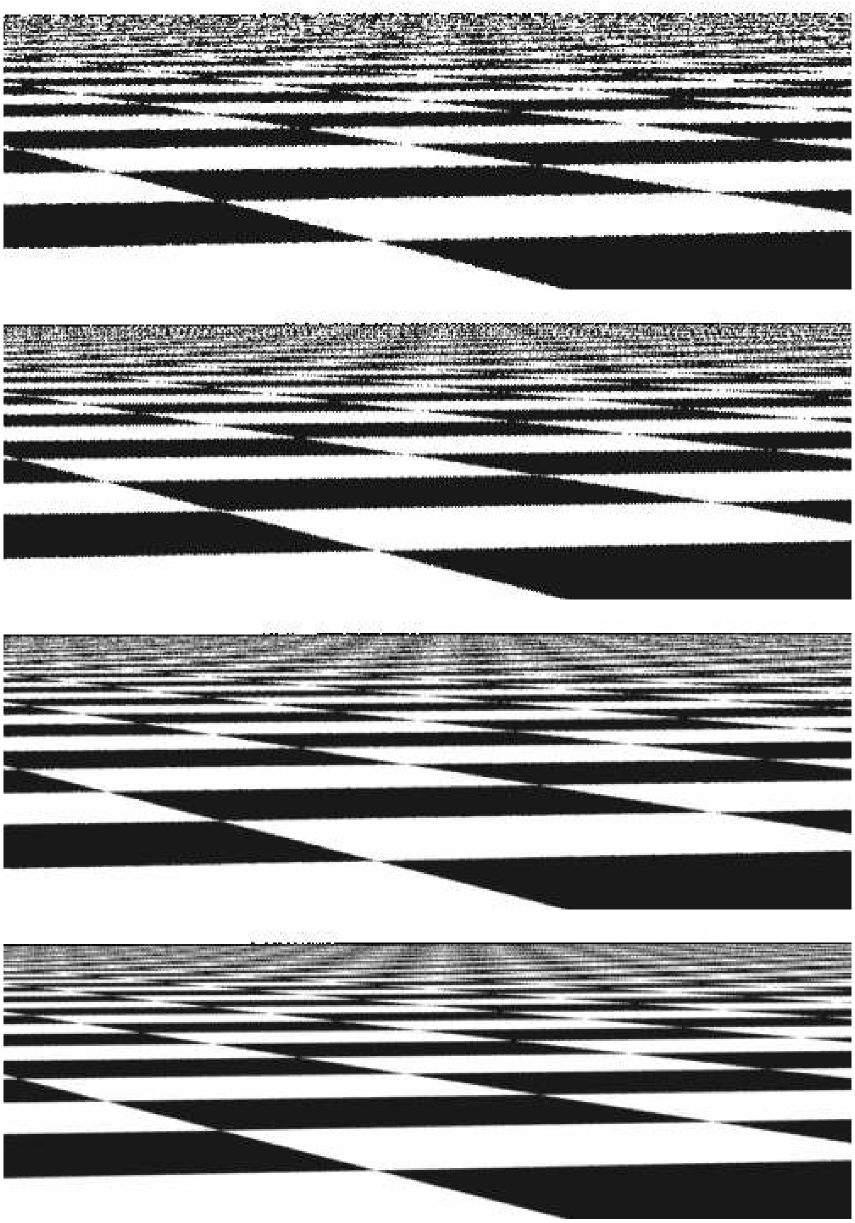


Figure 7.22: Images comparing the stratified sampler to the low discrepancy sampler. On top is the jittered stratified sampler with a single sample per pixel, and beneath it is the low discrepancy sampler with a single sample per pixel. Next is the stratified sampler with four samples per pixel and finally the low discrepancy sampler with four samples per pixel. Note that although the low discrepancy pattern is able to reproduce the checker pattern farther than the stratified pattern, there is a regular structure to the error in the low discrepancy pattern that is visually distracting.

⟨*LDSampler Method Definitions*⟩≡

```
LDSampler::LDSampler(int xstart, int xend, int ystart, int yend,
        int xs, int ys, bool uf)
    : Sampler(xstart, xend, ystart, yend, xs, ys) {
    scale = max(xPixelEnd - xPixelStart, yPixelEnd - yPixelStart);
    nSamples = xPixelSamples * yPixelSamples * scale * scale;
    invNSamples = 1.f / nSamples;
    curSample = 0;
    useFolded = uf;
    scrambles = imageSamplesDone = NULL;
}
```

⟨*LDSampler Private Data*⟩≡

```
int scale, nSamples, curSample;
Float invNSamples;
bool useFolded;
```

The `LDSampler` keeps generating points until a total of `nSamples` have been returned, after which it returns `false`, indicating that it has no more to provide. In the implementation below, we use either the `FoldedRadicalInverse()` function (to give a Hammersley-Zaremba point set) or the `RadicalInverse()` function (to give a Hammersley point set), based on the `useFolded` parameter.

FoldedRadicalInverse()	256
LDSampler	258
ampler::imageSamplesDone	264
LDSampler::scrambles	264
RadicalInverse()	254
Sample::imageX	239
Sample::imageY	239
Sampler	237
Sampler::xPixelEnd	238
Sampler::xPixelSamples	238
Sampler::xPixelStart	238
Sampler::yPixelEnd	238
Sampler::yPixelSamples	238
Sampler::yPixelStart	238

I removed the goto here – please check that I didn't mess this up.

⟨*LDSampler Method Definitions*⟩+≡

```
bool LDSampler::GetNextSample(Sample *sample) {
    do {
        ++curSample;
        if (curSample > nSamples) return false;
        ⟨Compute Hammersley (x,y) image sample location⟩
    } while (Float2Int(sample->imageX) >= xPixelEnd ||
        Float2Int(sample->imageY) >= yPixelEnd);
    ⟨Compute remaining dimensions of Hammersley sample⟩
    return true;
}
```

We start by computing the raster-space (x,y) image sample position. We immediately check to make sure that it is inside the region of pixels that need samples generated for it, so that we can skip computing values for the remainder of the dimensions in case it is out of bounds.

⟨*Compute Hammersley* (x,y) *image sample location*⟩≡

```
Float x = (curSample - .5f) * invNSamples;
Float y = useFolded ? FoldedRadicalInverse(curSample, 2) :
    RadicalInverse(curSample, 2);
sample->imageX = xPixelStart + x * scale;
sample->imageY = yPixelStart + y * scale;
```

Now that we know that we've got a valid image sample, we compute the sample points for the rest of the dimensions. Time and lens samples use Hammersley points in dimensions 3, 5, and 7.

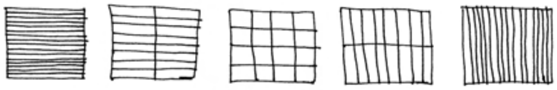


Figure 7.23: some elementary intervals, base 2...

```
⟨Compute remaining dimensions of Hammersley sample⟩≡
  if (useFolded) {
      sample->time  = FoldedRadicalInverse(curSample, 3);
      sample->lensX = FoldedRadicalInverse(curSample, 5);
      sample->lensY = FoldedRadicalInverse(curSample, 7);
  }
  else {
      sample->time  = RadicalInverse(curSample, 3);
      sample->lensX = RadicalInverse(curSample, 5);
      sample->lensY = RadicalInverse(curSample, 7);
  }
  ⟨Compute low-discrepancy integrator samples⟩
```

To generate samples for the integrators, we can take advantage of a remarkable property of certain low-discrepancy patterns that allows us to generate a set of sample positions for a pixel's worth of image samples such that each sample is well stratified with respect not only to the other samples in the set but also to the sample positions at neighboring image samples–both halves of the goal that we only got one part of with the `StratifiedSampler`.

256	`FoldedRadicalInverse()`
260	`LDSampler::curSample`
260	`LDSampler::useFolded`
254	`RadicalInverse()`
239	`Sample::lensX`
239	`Sample::lensY`
239	`Sample::time`
244	`StratifiedSampler`

A useful low-discrepancy sequence in 2D can be constructed using the Van De Corput sequence in one dimension and a sequence based on a radical inverse function due to Sobol' in the other direction. The resulting sequence is a special type of low-discrepancy sequence known as an $(0,2)$-sequence. $(0,2)$-sequences are stratified in a very general way. For example, the first 16 samples in an $(0,2)$-sequence satisfy the stratification constraint, such that there is just one sample in each of the boxes of extent $(1/4,1/4)$. However, the also satisfy the LHS constraint, such that only one of them is in each of the boxes of extent $(1/16,1)$ and $(1,1/16)$. Furthermore, there is only one sample in each of the boxes of extent $(1/2,1/8)$ and $(1/8,1/2)$. Figure 7.23 shows all of the regions that the first 16 samples of an $(0,2)$-sequence simultaneously satisfy stratification properties with respect to. Furthermore, each succeeding sequence of 16 additional samples from the sequence satisfies this same distribution property.

In general, any sequence of length $2^{l_1 l_2}$ ($l_i \geq 0$ an integer) from an $(0,2)$-sequence satisfies a general stratification constraint. Define the set of *elementary intervals* in 2 dimensions, base two as

$$E = \left\{ \left[\frac{a_1}{2^{l_1}}, \frac{a_1+1}{2^{l_1}}\right) \times \left[\frac{a_2}{2^{l_2}}, \frac{a_2+1}{2^{l_2}}\right) \right\},$$

where $a_i = 0,\ldots,2^{l_i}-1$ and is an integer. Then, one sample from each of the first $2^{l_1 l_2}$ values in the sequence will be in each of the elementary intervals. Furthermore, the same property is true for each subsequent set of $2^{l_1 l_2}$ values.

Consider now how $(0,2)$-sequences can be applied to generating 2D samples for the integrators: for example, consider a pixel with 2 by 2 image samples, each with

4 by 4 integrator samples. The first $2 \times 2 \times 4 \times 4 = 2^6$ values of an $(0,2)$-sequence are well-distributed with respect to each other according to the corresponding set of elementary intervals. Furthermore, the first $4 \times 4 = 2^4$ samples are themselves well-distributed according to their corresponding elementary intervals, as are the next 2^4 of them, the subsequent ones, etc. Therefore, we can use the first 16 $(0,2)$-sequence samples for the first image sample in the region around the pixel, and so forth. The result is an extremely well-distributed set of sample points.

However, there a handful of details that must be addressed before $(0,2)$-sequences can be used in practice. The first is that we need to generate multiple sets of 2D sample values for each image sample, and we would like to generate different sample values in the areas around different pixels. One approach to this problem would be to use carefully-chosen non-overlapping sub-sequences of the $(0,2)$-sequence for each of these. Another approach, which we will use here is to *randomly scramble* the $(0,2)$-sequence, giving a $(0,2)$-sequence built by randomly permuting the base b digits of the values of the original net.

In two dimensions, the scrambling approach we will use repeatedly partitions and shuffles the unit square $[0,1]^2$. In each of the dimensions, it first divides the square in half at 0.5. It then swaps the two halves with 50% probability. Then, for the intervals $[0,0.5)$ and $[0.5,1)$, it splits each of them in half down the middle and randomly exchanges each of those two halves. This process continues recursively until floating-point precision intervenes and continuing the process would make no difference to the values computed.

`Sobol2()`	263
`VanDerCorput()`	263

Two things make this process efficient: first, because we are scrambling two sequences that are computed base 2, the digits a_i of the sequences are all 0 or 1, and scrambling a particular digit is equivalent to exclusive-or'ing it with 0 or 1. Second, we make the simplification that at each level l of the recursive scrambling, we make the same decision as to whether to swap each pair of the 2^{l-1} pairs of sub-intervals or not. The result of these two decisions is that the scrambling can be encoded as a set of bits stored in an `unsigned long` and can be applied to the original digits via exclusive-or operations.

⟨*Sampling Inline Functions*⟩≡

```
inline void Sample02Net(u_int n, u_int scramble[2], Float sample[2]) {
    sample[0] = VanDerCorput(n, scramble[0]);
    sample[1] = Sobol2(n, scramble[1]);
}
```

check capitalization of Van Der Corput

Here are implementations of the Van Der Corput and Sobol' low-discrepancy sequences, specialized for the base 2 case. Each of them takes a `u_int` value `scramble` that encodes a random permutation to apply. It computes the `n`th value from each of the sequences as it simultaneously applies the permutation.

This just reverses the bits, does the xor, turns it to a float...

⟨*Sampling Inline Functions*⟩+≡

```
inline Float VanDerCorput(u_int n, u_int scramble) {
    n = (n << 16) | (n >> 16);
    n = ((n & 0x00ff00ff) << 8) | ((n & 0xff00ff00) >> 8);
    n = ((n & 0x0f0f0f0f) << 4) | ((n & 0xf0f0f0f0) >> 4);
    n = ((n & 0x33333333) << 2) | ((n & 0xcccccccc) >> 2);
    n = ((n & 0x55555555) << 1) | ((n & 0xaaaaaaaa) >> 1);
    n ˆ= scramble;
    return (Float)n / (Float)0x100000000LL;
}
```

⟨*Sampling Inline Functions*⟩+≡

```
inline Float Sobol2(u_int n, u_int scramble) {
    for (u_int v = 1 << 31; n != 0; n >>= 1, v ˆ= v >> 1)
        if (n & 0x1) scramble ˆ= v;
    return (Float)scramble / (Float)0x100000000LL;
}
```

We can now define the fragment that computes integrator sample values for the `LDSmapler`. For each local area around a pixel in the image, we will compute the random `u_int` values that will be used to scramble each of the 1D and 2D integrator sample values that we will compute at that pixel. These are stored in the `scrambles` array. We also keep a count of the number of samples that have been generated for each pixel's region so that we can choose the appropriate offsets into the $(0,2)$-sequences for the current sample in the current pixel.

264 `LDSampler::scrambles`
237 `Sampler`

⟨*Compute low-discrepancy integrator samples*⟩≡

```
if (!scrambles) {
    ⟨Allocate and initialize arrays for per-pixel records⟩
}
⟨Get pointer to scramble values for current pixel⟩
⟨Compute low-discrepancy 1D samples⟩
⟨Compute low-discrepancy 2D samples⟩
```

The low-discrepancy `Sampler` doesn't have an explicit notion of generating samples for a particular image pixel, yet we need to use the same scramble values for all of the samples within the area around each pixel so that they are well-distributed with respect to each other. Because all of the samples nearby a particular pixel aren't generated at once, we will allocate memory to keep track of the `u_int` values used for scrambling each set of 1D and 2D sample values at each pixel, as well as the number of samples generated for the pixel so far, `imageSamplesDone` so that we know which subsequence of the $(0,2)$-sequence to use.

⟨*Allocate and initialize arrays for per-pixel records*⟩≡

```
int nPix = (1 + xPixelEnd - xPixelStart) * (1 + yPixelEnd - yPixelStart);
imageSamplesDone = new u_int[nPix];
for (int i = 0; i < nPix; ++i)
    imageSamplesDone[i] = 0;
int nScrambles = nPix * (sample->n1D.size() + 2 * sample->n2D.size());
scrambles = new u_int[nScrambles];
for (int i = 0; i < nScrambles; ++i)
    scrambles[i] = RandomUInt();
```

⟨*LDSampler Private Data*⟩+≡

```
u_int *scrambles, *imageSamplesDone;
```

We can map the floating-point (x,y) image sample position to map to an integer pixel value, and from that, an offset into the `scramble` and `imageSamplesDone` arrays.

⟨*Get pointer to scramble values for current pixel*⟩≡

```
int xPixel = Floor2Int(sample->imageX);
int yPixel = Floor2Int(sample->imageY);
int pixelNum = (xPixel - xPixelStart) +
    (yPixel - yPixelStart) * (xPixelEnd - xPixelStart);
Assert(pixelNum < (xPixelEnd-xPixelStart)*(yPixelEnd-yPixelStart));
Assert(xPixel >= xPixelStart && xPixel < xPixelEnd);
Assert(yPixel >= yPixelStart && yPixel < yPixelEnd);
u_int *scramble = &scrambles[pixelNum *
    (sample->n1D.size() + 2 * sample->n2D.size())];
int pixelSamples = imageSamplesDone[pixelNum]++;
```

`Assert()`	659
`RoundUpPow2()`	678
`Sample02Net()`	262
`Sample::imageX`	239
`Sample::imageY`	239
`Sample::n1D`	241
`Sample::n2D`	241
`Sample::twoD`	241
`Sampler::xPixelEnd`	238
`Sampler::xPixelStart`	238
`Sampler::yPixelEnd`	238
`Sampler::yPixelStart`	238

We will omit the fragment that computes low-discrepancy 1D sample values; it just uses a randomly scrambled Van Der Corput sequence and is analogous to the 2D fragment, which is shown below. It loops over all of the sets of samples that need to be generated around the given pixel. For each one, it computes `nextPow2`, the next power-of-two that is greater than or equal to the number of samples needed–this tells us how far ahead in the sample sequence to jump for the start of the next set of samples. Using this as a base, we compute the points of the scrambled $(0,2)$-sequence in turn, using the appropriate shuffle. When done, the `scramble` pointer is incremented, so that it points to the scramble values for the next set of points.

⟨*Compute low-discrepancy 2D samples*⟩≡

```
for (u_int i = 0; i < sample->n2D.size(); ++i) {
    u_int nextPow2 = RoundUpPow2(sample->n2D[i]);
    int sampStart = pixelSamples * nextPow2;
    for (u_int j = 0; j < sample->n2D[i]; ++j) {
        int sampNum = sampStart + j + 1;
        Sample02Net(sampNum, scramble, &sample->twoD[i][2*j]);
    }
    scramble += 2;
}
```

With the folded radical inverse functions in dimensions greater than two, such that the implementations are not easily optimized, it is substantially more expensive computationally to generate image samples than it is for the `StratifiedSampler` of the previous section or the `BestCandidateSampler` that will be introduced in the next section (roughly ten times slower.) For very simple scenes, where the cost of tracing a camera ray and computing its contribution is low, it may be more efficient to trace more rays generated by a lower-quality sample generation method to render an image of a particular quality level than it is to trace fewer rays that are "better", since the cost of generating their samples may dominate overall running time. For more complex scenes, however, where computing the contribution of a camera ray is more expensive, we can afford to spend more time to compute very good samples, since a reduction in the total number of samples that need to be taken can make up for the expense of computing the samples.

7.6 ***ADV***: Best-Candidate Sampling Patterns

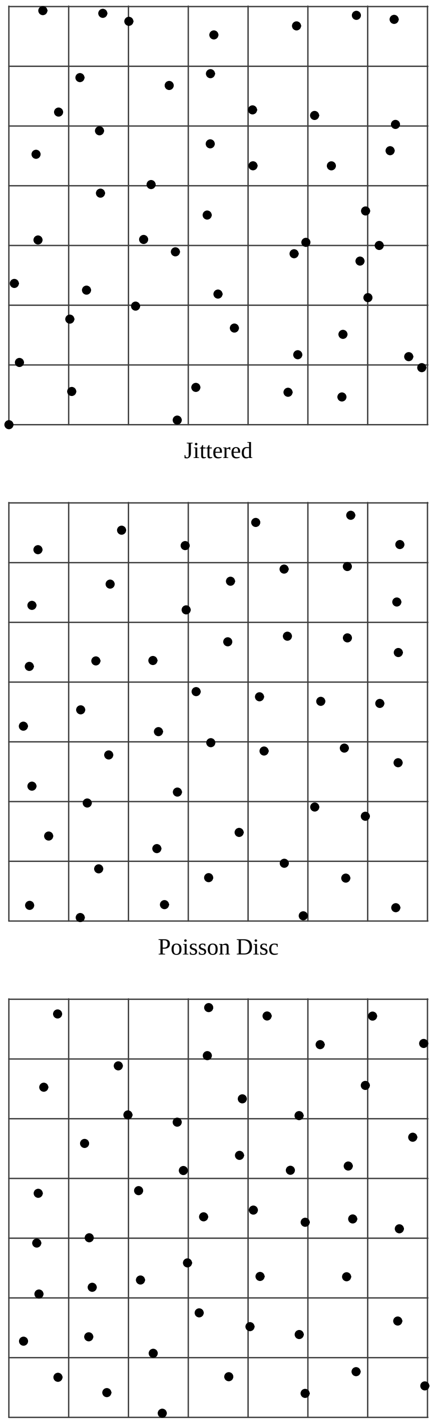
So far, we have two imperfect solutions to the sampling problem: jittered stratified sampling, which has randomness that reduces the visual impact of undersampling high-frequency image functions but suffers from bunching up of samples and undersampling of some areas, and low-discrepancy sampling, which addresses the sample distribution problem with carefully-constructed sample locations but can be prone to aliasing. Ideally, both of these problems could be solved simultaneously; the best candidate sampler in this section tries to do just this.

275 `BestCandidateSampler`
244 `StratifiedSampler`

The *Poisson disk pattern* addresses both the issue of randomization as well as the issue of well-separated sample placement; it has been shown to be an excellent image sampling pattern. The Poisson disk pattern is a group of points such that no two of them are closer than some specified distance. Studies have shown that the rods and cones in the eye are distributed in a Poisson disk-like pattern, which suggests that this pattern might be effective for imaging. Poisson disk patterns are usually generated by *dart throwing*: we keep generating random samples, throwing away all that are closer to a previous sample than a fixed threshold distance. This can be a very expensive process, since many darts may need to be thrown before one is accepted.

A related approach due to Don Mitchell is the *best candidate* algorithm. When a new sample is to be computed, a large number of random candidates are generated; all of these candidates are compared to the previous samples and the one that is farthest away from all of the previous samples is added to the pattern. Although this algorithm doesn't guarantee the Poisson disk property, it usually does quite well at finding well-separated points if enough candidates are generated. Another advantage it has is that any prefix of the final pattern is itself a well-distributed sampling pattern. Furthermore, it's easier to generate a good pattern with a pre-chosen number of samples with the best candidate algorithm than it is with a dart throwing algorithm.

In this section we will present an implementation of the best-candidate algorithm and its extension to computing sampling patterns that include good distributions of samples in additional dimensions. Because generating the sample positions is a computationally-intensive algorithm, we will compute a good sampling pattern



Jittered

Poisson Disc

Best Candidate

Figure 7.24: Comparison of sampling patterns. On the top is a jittered pattern: note clumping of samples and undersampling in some areas. In the middle is a Poisson disk pattern generated by dart-throwing. No two samples are closer than a fixed threshold, and although there is no guarantee that there will be one sample in each of the strata, this is usually the case. On the bottom is a pattern generated with the best-candidate algorithm; it is nearly as good as the Poisson disk pattern. (Due to its toroidal topology, the two strata at the top left with no samples have samples very close to them from the bottom left part, etc.)

once in a pre-process. The pattern can then be stored in a table and efficiently used at rendering-time.

Rather than computing a sampling pattern large enough to sample the most enormous image we'd ever render, we'll compute a pattern that can be reused by tiling it over the image plane by translating and scaling it appropriately. This means that we must consider the pattern to have *toroidal topology*; when computing the distance between two samples, we must compute the distance between them as if the square sampling region was rolled into a torus. Thus, for these purposes points at the top of the region may have a very small distance to points at the bottom, etc.

7.6.1 Generating the best-candidate pattern

We will now show the program that generates the samples in an off-line computation.

⟨*samplepat.cpp*⟩≡

```
  #include "lrt.h"
  #include "sampling.h"
  ⟨BestCandidate Sampling Constants⟩
  ⟨Sample Pattern Precomputation⟩
```

First we need to define the size of the table that we will be generating.

⟨*BestCandidate Sampling Constants*⟩≡

```
  #define SQRT_SAMPLE_TABLE_SIZE 64
  #define SAMPLE_TABLE_SIZE (SQRT_SAMPLE_TABLE_SIZE * \
                             SQRT_SAMPLE_TABLE_SIZE)
```

We will generate sample points in a dive-dimensional space: two dimensions for the image sample location, one for the time, and two more determine a point on a lens. Because we don't know ahead of time what types of sample patterns the integrators will need, we will just use scrambled $(0,2)$-sequences for them, as in the low-discrepancy sampler. When generating the samples we store each of these sets of samples in a separate array.

⟨*Pattern Precomputation Local Data*⟩≡

```
  static Float imageSamples[SAMPLE_TABLE_SIZE][2];
  static Float timeSamples[SAMPLE_TABLE_SIZE];
  static Float lensSamples[SAMPLE_TABLE_SIZE][2];
```

Here is the `main()` function for the off-line sample computation program. We compute sample values in a multi-stage process. First, we generate a well-distributed set of image sample positions. Then, given the image samples, we generate a good set of time samples, accounting for the positions of the time samples for nearby image samples. Finally, we generate good samples for the lens, again taking into account the positions of nearby lens samples.

```
⟨Sample Pattern Precomputation⟩+≡
  int main() {
      ⟨Compute image sample positions⟩
      ⟨Compute time samples⟩
      ⟨Compute lens samples⟩
      ⟨Write sample table to disk⟩
      return 0;
  }
```

In order to speed up the candidate evaluation, we will store the accepted samples in a grid. This allows us to only check nearby samples when computing distances between samples. The grid splits up the 2D sample domain $[0,1]^2$ into `BC_GRID_SIZE` strata in each direction and stores a list of the integer sample numbers of the samples that overlap each cell. The `GRID()` macro maps a position in $[0,1]$ to the corresponding grid cell. **explain this better – the text makes it sound like GRID goes from R^2 to R^1, which is not true.**

```
⟨Global Forward Declarations⟩+≡
  #define BC_GRID_SIZE 40
  typedef vector<int> SampleGrid[BC_GRID_SIZE][BC_GRID_SIZE];
  #define GRID(v) (int((v) * BC_GRID_SIZE))
```

ProgressReporter	660
ProgressReporter::Done()	660
ogressReporter::Update()	660
SAMPLE_TABLE_SIZE	267
vector	658

To compute the image samples, we start by creating a sample grid and calling a function to run the 2D best candidate algorithm.

```
⟨Compute image sample positions⟩≡
  SampleGrid pixelGrid;
  BestCandidate2D(imageSamples, SAMPLE_TABLE_SIZE, &pixelGrid);
```

For the best candidate algorithm, the first image sample position is chosen completely arbitrarily and recorded in the grid. For all subsequent samples, we generate a set of candidates that are compared to the already-computed samples.

```
⟨Sample Pattern Precomputation⟩+≡
  void BestCandidate2D(Float table[][2], int totalSamples,
          SampleGrid *grid) {
      SampleGrid localGrid;
      if (!grid) grid = &localGrid;
      ProgressReporter progress(totalSamples-1, "Throwing Darts");
      ⟨Generate first 2D sample arbitrarily⟩
      for (int currentSample = 1; currentSample < totalSamples;
          ++currentSample) {
          ⟨Generate next best 2D image sample⟩
          progress.Update();
      }
      progress.Done();
  }
```

To start off the process, we can choose any random point for the first sample; only the second sample and beyond need to be checked against previous samples.

⟨*Generate first 2D sample arbitrarily*⟩≡

```
table[0][0] = RandomFloat();
table[0][1] = RandomFloat();
addSampleToGrid(table, 0, grid);
```

A short utility function adds a particular item (given by `offset`) in the table of samples to a `SampleGrid`.

why not pass a reference to the SampleGrid instead of a pointer?

⟨*Pattern Precomputation Utility Functions*⟩≡

```
static void addSampleToGrid(Float sample[][2], int offset,
        SampleGrid *grid) {
    int u = GRID(sample[offset][0]), v = GRID(sample[offset][1]);
    (*grid)[u][v].push_back(offset);
}
```

To generate the rest of the samples, we will use a dart throwing algorithm that throws a number of candidate darts for each needed sample. The number of darts thrown is proportional to the number of samples we have already; this ensures that the quality of the samples as we go is in some sense consistent. After throwing a dart, we see how close it is to all of the samples we've generated so far. If it's farther away from all of the accepted samples than the previous best candidate was, we keep it. At the end of the loop, the remaining candidate is kept.

268	`currentSample`
268	`GRID`
679	`RandomFloat()`
268	`SampleGrid`
237	`Sampler`

⟨*Generate next best 2D image sample*⟩≡

```
Float maxDist2 = 0.;
int numCandidates = 500 * currentSample;
for (int currentCandidate = 0; currentCandidate < numCandidates;
        ++currentCandidate) {
    ⟨Generate a random candidate sample⟩
    ⟨Loop over neighboring grid cells and check distances⟩
    ⟨Keep this sample if it is the best one so far⟩
}
addSampleToGrid(table, currentSample, grid);
```

Candidate positions are chosen completely at random. Note that we're computing image sample locations in the range $[0,1)$; it'll be up to the `Sampler` that uses the sampling pattern to scale and translate image samples into raster-space appropriately.

⟨*Generate a random candidate sample*⟩≡

```
Float candidate[2];
candidate[0] = RandomFloat();
candidate[1] = RandomFloat();
```

Now that we have a candidate, we see if it's the best candidate we've come up with so far. We compute the distances to all of the already-generated samples, keeping track of the minimum of the distances to all of the previously accepted samples. Whichever candidate that has the largest minimum distance is the best. For efficiency, we will actually compute the squared distance, which gives the same result for this test and saves us a lot of expensive square root computations.

We actually only compute distances to the eight neighboring grid cells and the cell that the candidate is in; although this means that the first few samples are not optimally distributed relative to each other, this doesn't matter by the time we are done computing samples, so long as BC_GRID_SIZE $<$ SQRT_SAMPLE_TABLE_SIZE.

⟨*Loop over neighboring grid cells and check distances*⟩≡

```
Float sampleDist2 = INFINITY;
int gu = GRID(candidate[0]), gv = GRID(candidate[1]);
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Compute (u,v) grid cell to check⟩
        ⟨Update minimum squared distance from cell's samples⟩
    }
}
```

We do need to handle the toroidal topology here, though; if the grid cell we'd like to consider is out of bounds, we wrap around to the other end of the grid.

⟨*Compute (u,v) grid cell to check*⟩≡

```
int u = gu + du, v = gv + dv;
if (u < 0)             u += BC_GRID_SIZE;
if (u >= BC_GRID_SIZE) u -= BC_GRID_SIZE;
if (v < 0)             v += BC_GRID_SIZE;
if (v >= BC_GRID_SIZE) v -= BC_GRID_SIZE;
```

BC_GRID_SIZE	268
GRID	268
INFINITY	678
SQRT_SAMPLE_TABLE_SIZE	267

We now loop over the list of sample numbers for the samples in the grid cell we're considering. For each one, we compute the squared distance to the current candidate, recording the lowest squared distance of all the ones we check.

⟨*Update minimum squared distance from cell's samples*⟩≡

```
for (u_int g = 0; g < (*grid)[u][v].size(); ++g) {
    int s = (*grid)[u][v][g];
    Float xdist = Wrapped1DDist(candidate[0], table[s][0]);
    Float ydist = Wrapped1DDist(candidate[1], table[s][1]);
    Float d2 = xdist*xdist + ydist*ydist;
    sampleDist2 = min(sampleDist2, d2);
}
```

When we compute the 1D distance between two values in $[0,1]$, we need to handle the wrap-around issue. Consider two samples with x coordinates of .01 and .99, respectively. Direct computation will find their distance to be .98, though with wrap-around, the actual distance should be .02. Because we're only checking distances to samples in adjacent grid cells, we can easily detect this situation when one of the distances is greater than 0.5. In that case, the true distance is just the sum of the distance from the higher sample to one plus the distance from zero to the lower sample.

⟨*Pattern Precomputation Utility Functions*⟩+≡

```
inline Float Wrapped1DDist(Float a, Float b) {
    Float d = fabsf(a - b);
    if (d < 0.5f) return d;
    else return 1.f - max(a, b) + min(a, b);
}
```

Finally, we see if this candidate has the highest squared distance to its neighbors. If so, we record its distance and tentatively put it in the output `table`.

⟨*Keep this sample if it is the best one so far*⟩≡

```
if (sampleDist2 > maxDist2) {
    maxDist2 = sampleDist2;
    table[currentSample][0] = candidate[0];
    table[currentSample][1] = candidate[1];
}
```

Now that we've got all of the image samples that we want, we turn to computing the sample positions for the rest of the dimensions. One might think that a good sample pattern could be computed by generalizing the Poisson disk concept to a higher-dimensional Poisson sphere. Interestingly enough, we can do better than this. (In the five-dimensional case in particular, a large number of candidate samples would be needed to find good ones, anyway.)

Consider the problem of choosing time values for two nearby image samples: not only do we want the time values to not be too close together, but in fact, it's even better if the time values are as far apart as possible—in any local 2D region of the image, we'd like the best possible coverage of the complete three-dimensional sample space.

An intuition for why this is the case comes from how the sampling pattern will be used. Although we're generating a five-dimensional pattern overall, what we're interested in is optimizing its distribution across local areas of the two-dimensional image plane; optimizing its distribution over the five-dimensional space is at best a secondary concern.

268	`currentSample`
660	`ProgressReporter`
660	`ProgressReporter::Update()`
679	`RandomFloat()`
267	`SAMPLE_TABLE_SIZE`
270	`sampleDist2`

Therefore, we'll use a two stage process for generating the sample positions. First, we will generate a well-distributed sampling pattern for the time and lens positions. Then, we will associate these samples with image samples in a way that ensures that nearby image samples have sample values for the other dimensions that are well spread-out.[7]

For time, we generate a set of one-dimensional stratified sample values over $[0,1]$. When we're done, we will rearrange the `timeValues` array so that the ith time sample is a good one for the ith image sample.

⟨*Compute time samples*⟩≡

```
ProgressReporter timeProgress(SAMPLE_TABLE_SIZE, "Time samples");
for (int i = 0; i < SAMPLE_TABLE_SIZE; ++i)
    timeSamples[i] = (i + RandomFloat()) / SAMPLE_TABLE_SIZE;
for (int currentSample = 1; currentSample < SAMPLE_TABLE_SIZE;
    ++currentSample) {
    ⟨Select best time sample for current image sample⟩
    timeProgress.Update();
}
```

[7] As if that wasn't enough to worry about, we should also be considering *correlation*. Not only should nearby image samples have distant sample values for the other dimensions, but we should also make sure that, for example, the time and lens values aren't correlated: if we somehow kept choosing samples such that the time value was always similar to the lens u sample value, the sample pattern is not as good as it would be if the two were uncorrelated. We won't address this issue in our approach here, though at least the technique we use isn't prone to introducing correlation.

⟨*Select best time sample for current image sample*⟩≡

```
int best = -1;
⟨Find best time relative to neighbors⟩
Assert(best != -1);
swap(timeSamples[best], timeSamples[currentSample]);
```

Given that we're working on finding a good time for the sample number `currentSample`, the elements of `timeSamples` from zero to `currentSample-1` have already been assigned to previous image samples and are unavailable to us. The rest of the times, from `currentSample` to `SAMPLE_TABLE_SIZE-1`, are the ones we will choose from.

⟨*Find best time relative to neighbors*⟩≡

```
Float maxMinDelta = 0.;
for (int t = currentSample; t < SAMPLE_TABLE_SIZE; ++t) {
    ⟨Compute min delta for this time⟩
    ⟨Update best if this is best time so far⟩
}
```

As when we were doing dart-throwing for image samples, we only look at the samples in the adjoining few grid cells. Of these, we will select the one that is most different than the time samples that have already been assigned to the nearby image samples.

`Assert()`	659
`currentSample`	268
`GRID`	268
`INFINITY`	678
`SAMPLE_TABLE_SIZE`	267
`Wrapped1DDist()`	270

⟨*Compute min delta for this time*⟩≡

```
int gu = GRID(imageSamples[currentSample][0]);
int gv = GRID(imageSamples[currentSample][1]);
Float minDelta = INFINITY;
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Check distance from times of nearby samples⟩
    }
}
```

We loop through the samples in each of the grid cells, though we need to be careful to only consider the ones that already have time samples associated with them. Therefore, we skip over the ones that have sample numbers greater than the sample we're currently working to find a time value for. For the remaining ones, we compute the distance for their time sample to the current candidate time sample, keeping track of the minimum difference.

⟨*Check distance from times of nearby samples*⟩≡

```
⟨Compute (u,v) grid cell to check⟩
for (u_int g = 0; g < pixelGrid[u][v].size(); ++g) {
    int otherSample = pixelGrid[u][v][g];
    if (otherSample < currentSample) {
        Float dt = Wrapped1DDist(timeSamples[otherSample],
            timeSamples[t]);
        minDelta = min(minDelta, dt);
    }
}
```

If the minimum distance from the current time sample is greater than the minimum distance of the previous best time sample, we update our records.

⟨*Update* `best` *if this is best time so far*⟩≡

```
if (minDelta > maxMinDelta) {
    maxMinDelta = minDelta;
    best = t;
}
```

We now go ahead and take care of the lens positions. We generate good sampling patterns using dart throwing and then associate these sample values with image samples in the same manner that we assigned times to image samples, selecting lens positions that are far away from the lens positions of nearby image samples.

⟨*Compute lens samples*⟩≡

```
BestCandidate2D(lensSamples, SAMPLE_TABLE_SIZE);
Redistribute2D(lensSamples, pixelGrid);
```

After the `BestCandidate2D()` function generates a good set of 2D samples, the `Redistribute2D()` utility function takes the set of samples to assign to the image samples and reshuffles them like we reshuffled the time samples to give them a good distribution with respect to their neighbors.

⟨*Sample Pattern Precomputation*⟩+≡

```
static void Redistribute2D(Float samples[][2],
        SampleGrid &pixelGrid) {
    ProgressReporter progress(SAMPLE_TABLE_SIZE, "Redistribution");
    for (int currentSample = 1;
        currentSample < SAMPLE_TABLE_SIZE; ++currentSample) {
        ⟨Select best sample for current image sample⟩
        progress.Update();
    }
    fprintf(stderr, "\n");
}
```

659	`Assert()`
268	`BestCandidate2D()`
268	`currentSample`
660	`ProgressReporter`
660	`ProgressReporter::Update()`
267	`SAMPLE_TABLE_SIZE`
268	`SampleGrid`

⟨*Select best sample for current image sample*⟩≡

```
int best = -1;
⟨Find best 2D sample relative to neighbors⟩
Assert(best != -1);
swap(samples[best][0], samples[currentSample][0]);
swap(samples[best][1], samples[currentSample][1]);
```

As with time, we want to choose the sample from the available ones that maximizes the minimum distance to the sample values that have already been assigned to the neighboring image samples.

There are four chunks in a row here, what's up with that. More text.

⟨*Find best 2D sample relative to neighbors*⟩≡

```
Float maxMinDist2 = 0.f;
for (int samp = currentSample; samp < SAMPLE_TABLE_SIZE;
        ++samp) {
    ⟨Check distance to nearby samples⟩
    ⟨Update best for 2D sample if it is best so far⟩
}
```

⟨*Check distance to nearby samples*⟩≡

```
int gu = GRID(imageSamples[currentSample][0]);
int gv = GRID(imageSamples[currentSample][1]);
Float minDist2 = INFINITY;
for (int du = -1; du <= 1; ++du) {
    for (int dv = -1; dv <= 1; ++dv) {
        ⟨Check 2D samples in current grid cell⟩
    }
}
```

⟨*Check 2D samples in current grid cell*⟩≡

```
⟨Compute (u,v) grid cell to check⟩
for (u_int g = 0; g < pixelGrid[u][v].size(); ++g) {
    int s2 = pixelGrid[u][v][g];
    if (s2 < currentSample) {
        Float dx = Wrapped1DDist(samples[s2][0],
            samples[samp][0]);
        Float dy = Wrapped1DDist(samples[s2][1],
            samples[samp][1]);
        Float d2 = dx*dx + dy*dy;
        minDist2 = min(d2, minDist2);
    }
}
```

currentSample	268
GRID	268
INFINITY	678
Wrapped1DDist()	270

⟨*Update* `best` *for 2D sample if it is best so far*⟩≡

```
if (minDist2 > maxMinDist2) {
    maxMinDist2 = minDist2;
    best = samp;
}
```

When we're all done, we open up a file and write out C++ code that initializes the table. When `lrt` is compiled, it will `#include` this file to initialize its sample table. **since we're including it, why not generate `sampledata.h` intead?**

⟨*Write sample table to disk*⟩≡

```
FILE *f = fopen("sampledata.cpp", "w");
Assert(f);
fprintf(f, "\n/* Automatically generated %dx%d sample "
    "table (%s @ %s) */\n\n",
    SQRT_SAMPLE_TABLE_SIZE, SQRT_SAMPLE_TABLE_SIZE,
    __DATE__, __TIME__);
fprintf(f, "const Float BestCandidateSampler::sampleTable[%d][5] "
    "= {\n", SAMPLE_TABLE_SIZE);
for (int i = 0; i < SAMPLE_TABLE_SIZE; ++i) {
    fprintf(f, "  { ");
    fprintf(f, "%10.10ff, %10.10ff, ", imageSamples[i][0],
        imageSamples[i][1]);
    fprintf(f, "%10.10ff, ", timeSamples[i]);
    fprintf(f, "%10.10ff, %10.10ff, ", lensSamples[i][0],
        lensSamples[i][1]);
    fprintf(f, "},\n");
}
fprintf(f, "};\n");
```

659	Assert()
267	SAMPLE_TABLE_SIZE
237	Sampler
267	SQRT_SAMPLE_TABLE_SIZE
244	StratifiedSampler

7.6.2 Using the best-candidate pattern

⟨*bestcandidate.cpp**⟩≡

```
#include "sampling.h"
#include "paramset.h"
#include "film.h"
⟨BestCandidate Sampling Constants⟩
⟨BestCandidateSampler Declarations⟩
⟨BestCandidateSampler Method Definitions⟩
```

`BestCandidateSampler`, the `Sampler` that uses our sample table, is pretty straightforward. A single copy of the sample table covers

```
SQRT_SAMPLE_TABLE_SIZE / xPixelSamples
```

pixel separation extents in the x direction and analogously in y. As with the `StratifiedSampler`, we scan across the image from the upper left of the crop window, going left-to-right and then top-to-bottom. Here, we generate all samples inside the sample table's extent before advancing to the next region of the image that it covers.

⟨*BestCandidateSampler Declarations*⟩≡

```
class BestCandidateSampler : public Sampler {
public:
    ⟨BestCandidateSampler Public Methods⟩
private:
    ⟨BestCandidateSampler Private Data⟩
};
```

We store our current raster-space pixel position in `xTablePos` and `yTablePos` where `xTableWidth` and `yTableWidth` are the raster-space widths in pixel separations that the precomputed sample table spans. `tableOffset` holds the current offset into the sample table; when it is advanced to the point where we are at the end of the table, we advance to the next region of the image that the table covers.

Figure 7.25

⟨*BestCandidateSampler Method Definitions*⟩≡

```
BestCandidateSampler::BestCandidateSampler(int xstart, int xend,
        int ystart, int yend, int xs, int ys)
    : Sampler(xstart, xend, ystart, yend, xs, ys) {
    xTablePos = xPixelStart;
    yTablePos = yPixelStart;
    xTableWidth = (Float)SQRT_SAMPLE_TABLE_SIZE / xPixelSamples;
    yTableWidth = (Float)SQRT_SAMPLE_TABLE_SIZE / yPixelSamples;
    tableOffset = 0;
    ⟨Update sample shifts⟩
    scrambles = imageSamplesDone = NULL;
}
```

BestCandidateSampler	275
LDSampler	258
SAMPLE_TABLE_SIZE	267
Sampler	237
Sampler::xPixelSamples	238
Sampler::xPixelStart	238
Sampler::yPixelSamples	238
Sampler::yPixelStart	238
SQRT_SAMPLE_TABLE_SIZE	267
StratifiedSampler	244

⟨*BestCandidateSampler Private Data*⟩≡

```
int tableOffset;
Float xTablePos, yTablePos;
Float xTableWidth, yTableWidth;
u_int *scrambles, *imageSamplesDone;
```

Here we incorporate the precomputed sample data.

⟨*BestCandidateSampler Private Data*⟩+≡

```
static const Float sampleTable[SAMPLE_TABLE_SIZE][5];
```

⟨*BestCandidateSampler Method Definitions*⟩+≡

```
#include "sampledata.cpp"
```

One problem that sometimes comes up when using replicated precomputed sample patterns is that there may be subtle image artifacts aligned with the extent of the pattern on the image plane due to the same values being used repeatedly for time and lens position in each replicated sample region. Not only are the same `SAMPLE_TABLE_SIZE` samples used and re-used (whereas the `StratifiedSampler` and `LDSampler` will at least generate different time values and so forth for each different image sample), but the upper left sample in each block of samples will always have the same time value, etc.

On approach to this problem is to transform the set of sample values each time before starting to re-use the pattern. Here, we use *Cranley-Patterson rotations*, where we compute in each dimension

$$X_i' = (X_i + \xi_i) \quad \bmod 1,$$

where X_i is the sample value and ξ_i is a random number between zero and one. Because the various sampling patterns were computed with toroidal topology, the resulting pattern is still well-distributed and seamless. The table of random offsets

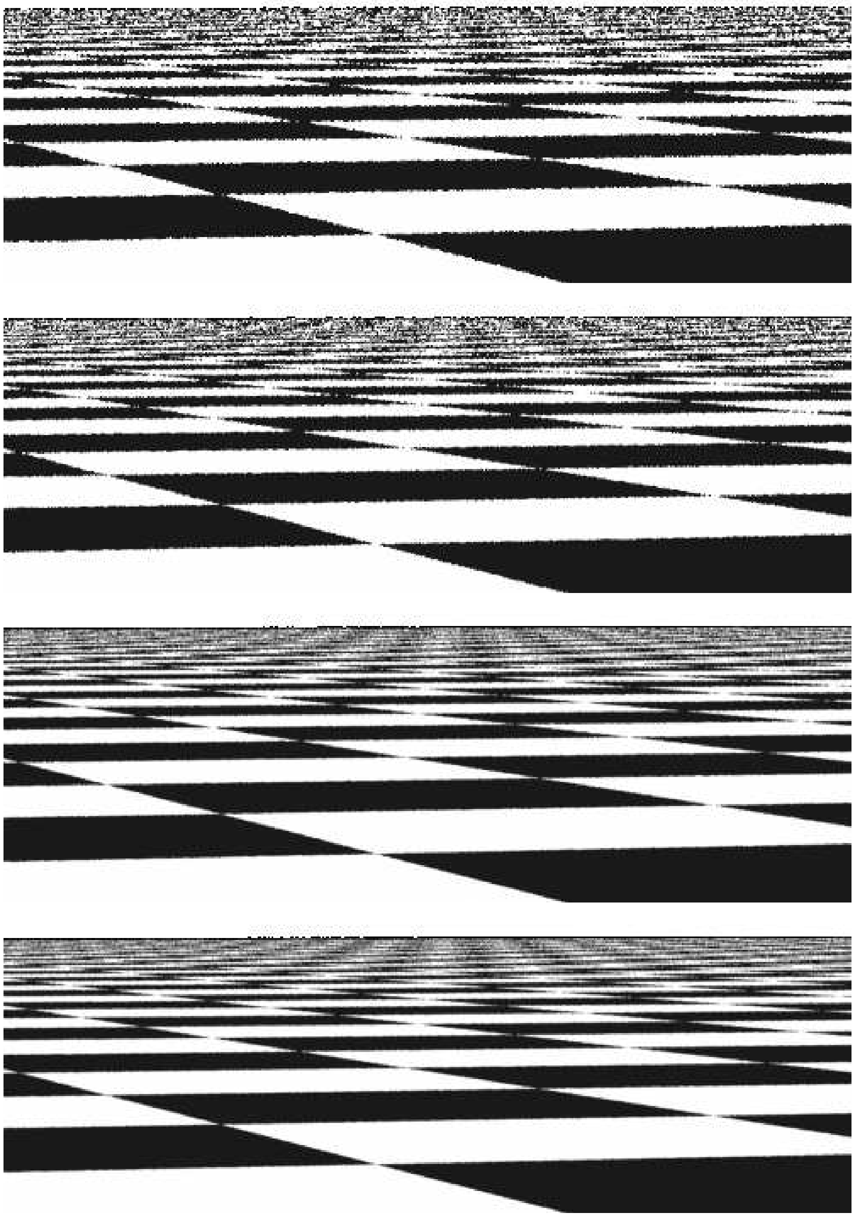


Figure 7.25: Comparisons of the stratified sampling pattern with the best candidate sampling pattern. The top image is the stratified pattern with a single sample per pixel, and the best candidate pattern with a single sample per pixel is beneath it. The third image shows the stratified pattern with four samples per pixel, with the four sample best candidate pattern at the bottom. Though the differences are subtle, note that the edges of the checks in the foreground aren't less noisy when the best candidate pattern is used, and it also does better at resolving the checks toward the horizon, particularly on the sides. Furthermore, the noise from the best candidate pattern tends to be higher-frequency, which is more visually acceptable.

for time and lens position ξ_i is updated each time we are about to reuse the table once again.

⟨*Update sample shifts*⟩≡

```
for (int i = 0; i < 3; ++i)
    sampleOffsets[i] = RandomFloat();
```

⟨*BestCandidateSampler Private Data*⟩+≡

```
Float sampleOffsets[3];
```

The GetNextSample() has a similar structure to the one for StratifiedSampler

⟨*BestCandidateSampler Method Definitions*⟩+≡

```
bool BestCandidateSampler::GetNextSample(Sample *sample) {
again:
    ⟨Return false if BestCandidateSampler is done⟩
    ⟨Compute raster sample from table⟩
    ⟨Advance to next sample table position⟩
    ⟨Check sample against crop window, goto again if outside⟩
    ⟨Compute low-discrepancy integrator samples⟩
    return true;
}
```

BestCandidateSampler	275
dateSampler::sampleTable	276
didateSampler::xTablePos	276
dateSampler::xTableWidth	276
didateSampler::yTablePos	276
dateSampler::yTableWidth	276
RandomFloat()	679
Sample::imageX	239
Sample::imageY	239
Sample::lensX	239
Sample::lensY	239
Sample::time	239
Sampler::yPixelEnd	238
StratifiedSampler	244

As with the StratifiedSampler, we are done generating samples when the upper *y* coordinate of the region goes below the bottom of the crop window.

⟨*Return* false *if* BestCandidateSampler *is done*⟩≡

```
if (yTablePos >= yPixelEnd)
    return false;
```

It just takes some simple indexing and scaling to compute the raster-space sample position from the positions in the table. We don't use the Cranley-Patterson shifting technique on image samples: this would cause the sampling points at the borders between repeated instances of the table to have a poor distribution; preserving good image-distribution is more important than reducing correlation. The rest of the camera dimensions are sampled using the shifting method described above, using the WRAP macro that ensures that the result stays between 0 and 1.

Would be nice to know if this made a difference; seems very subtle. Example renderings with blowups? Can we even do that?

⟨*Compute raster sample from table*⟩≡

```
#define WRAP(x) ((x) > 1 ? ((x)-1) : (x))
sample->imageX = xTablePos + xTableWidth *
    sampleTable[tableOffset][0];
sample->imageY = yTablePos + yTableWidth *
    sampleTable[tableOffset][1];
sample->time  = WRAP(sampleOffsets[0] +
    sampleTable[tableOffset][2]);
sample->lensX = WRAP(sampleOffsets[1] +
    sampleTable[tableOffset][3]);
sample->lensY = WRAP(sampleOffsets[2] +
    sampleTable[tableOffset][4]);
```

The sampler now steps to the next precomputed sample value; if it's hit the end of the sample table, it tries to move `xTablePos` forward. If this leaves the raster extent of the image, it moves `yTablePos` ahead.

⟨*Advance to next sample table position*⟩≡

```
if (++tableOffset == SAMPLE_TABLE_SIZE) {
    ⟨Update sample shifts⟩
    tableOffset = 0;
    xTablePos += xTableWidth;
    if (xTablePos >= xPixelEnd) {
        xTablePos = xPixelStart;
        yTablePos += yTableWidth;
    }
}
```

The sample table may partially spill off the end of the image plane, so some of the samples that we generate may be outside the necessary sample region. We detect this case by checking the sample against the region of pixels to be sampled and generating a new sample if it's out of bounds.

⟨*Check sample against crop window, goto* `again` *if outside*⟩≡

```
if (sample->imageX <  xPixelStart ||
    sample->imageX >= xPixelEnd   ||
    sample->imageY <  yPixelStart ||
    sample->imageY >= yPixelEnd)
    goto again;
```

276	BestCandidateSampler::tableOffse
276	BestCandidateSampler::xTablePos
276	BestCandidateSampler::xTableWidt
276	BestCandidateSampler::yTablePos
276	BestCandidateSampler::yTableWidt
239	Sample::imageX
239	Sample::imageY
267	SAMPLE_TABLE_SIZE
238	Sampler::xPixelEnd
238	Sampler::xPixelStart
238	Sampler::yPixelEnd
238	Sampler::yPixelStart

7.7 Image Reconstruction

We now turn to the task of turning our carefully-chosen image samples and their computed radiance values into pixel values for display or storage. Given the non-uniformly distributed set of image samples, we need to compute a final value for each of the pixels in the output image. According to the signal processing theory, we need to do three things:

1. Reconstruct a continuous image function $\tilde{L}$ from the set of image samples.

2. Prefilter the function $\tilde{L}$ to remove any frequencies past the Nyquist limit for the pixel spacing.

3. Sample $\tilde{L}$ at the pixel locations to compute the final pixel values.

Because we know that we will only be resampling the $\tilde{L}$ at the pixel locations, we don't need to construct an explicit representation of the function and can also aggregate the first two steps into a single filter function.

Recall that if the original function had been uniformly sampled at a frequency greater than the Nyquist frequency and reconstructed with the sinc filter, then the reconstructed function in the first step would match the original image function perfectly–quite a feat since we were only able to point-sample it. Because the original image function has higher frequencies than we were able to sample (due to edges, etc.), we chose to sample it non-uniformly, trading off noise for aliasing.

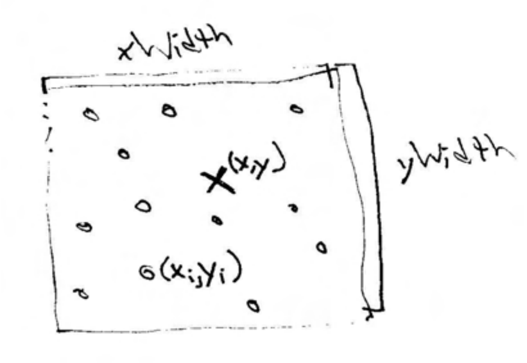


Figure 7.26: 2D image filtering: to compute a filtered pixel value for a pixel located at (x,y), all of the image samples inside the box around (x,y) with extent `xWidth` and `yWidth` need to be considered. Each one, (x_i,y_i) is weighted by a 2D filter function, $f(x-x_i,y-y_i)$; the weighted average of all samples is the final pixel value.

Film 294
Filter 281

The theory behind reconstruction (Equation **??**) depends on the samples being uniformly spaced. While a number of approaches have been used to try to extend the theory to non-uniform sampling, there is not yet as solid a footing for this. The most widely used method in graphics to compute pixel values is based on interpolation of the samples around a pixel. To compute a final value for a pixel $p(x,y)$, this interpolation results in computing a weighted average:

$$p(x,y) = \frac{\sum_i f(x-x_i,y-y_i)L(x_i,y_i)}{\sum_i f(x-x_i,y-y_i)} \qquad (7.7.3)$$

where $L(x_i,y_i)$ is the radiance value of the i'th sample, located at (x_i,y_i), and f is a filter function. Figure 7.26 shows a pixel at location (x,y), marked with an "x", that has a pixel filter with extent `xWidth` in the x direction and `yWidth` in the y direction. Image samples are denoted by dots, and all of the samples inside the box given by the filter extent may contribute to the pixel's value, depending on the filter function's value for $f(x-x_i,y-y-i)$.

Recall that the ideal sinc filter is prone to ringing when the underlying function has frequencies beyond the Nyquist limit (Gibbs phenomenon)–where edges in the image have faint replicated copies of the edge in nearby pixels. Furthermore, the sinc filter is generally avoided for efficiency reasons because it has *infinite support*: it doesn't fall off to zero at a finite distance from its center, so all of the image samples would need to be filtered for each output pixel. In practice, there is no single best filter function. Choosing the best one for a particular problem takes a mixture of quantitative evaluation and qualitative judgment.

7.7.1 Filter Functions

First we will define the abstract `Filter` class from which all our filter implementations will derive. `Filter` implements various functions $f(x,y)$ for use in pixel filtering (Equation 7.7.3). The `Film` object (described in the next chapter) stores a pointer to a `Filter` object that it uses for filtering.

⟨*Sampling Declarations*⟩+≡

```
class Filter {
public:
    ⟨Filter Interface⟩
    ⟨Filter Public Data⟩
};
```

All filters have widths beyond which they have a value of zero; these may be different in the *x* and *y* directions. The constructor takes values for these distances and stores them (and related values) for use by the filter implementations. The filter's overall extent in each direction (its *support*) is twice the value of its corresponding width.

this could really use a diagram; the fact that xw and yw are 1/2 the filter width could be made clearer that way.

⟨*Filter Interface*⟩+≡

```
Filter(Float xw, Float yw)
    : xWidth(xw), yWidth(yw), invXWidth(1.f/xw),
        invYWidth(1.f/yw) {
}
```

⟨*Filter Public Data*⟩≡

```
const Float xWidth, yWidth;
const Float invXWidth, invYWidth;
```

The sole function that `Filter` implementations need to provide is the `Evaluate()` method. It takes an *x* and *y* argument, which are the position of the sample point *relative to the center of the filter*. The return value specifies the weight of the sample. We will never call the filter function with points outside of the filter's extent; therefore, individual filters don't need to check for this case.

⟨*Filter Interface*⟩+≡

```
virtual Float Evaluate(Float x, Float y) const = 0;
```

Box Filter

⟨box.cpp*⟩≡

```
#include "sampling.h"
#include "paramset.h"
⟨Box Filter Declarations⟩
⟨Box Filter Method Definitions⟩
```

One of the most commonly used filters in graphics is the *box filter* (and in fact, when filtering and reconstruction isn't addressed explicitly, the box filter is the *de facto* result). The box filter equally weights all samples within a square region of the image. Though computationally efficient, it's just about the worst filter possible. In practice, it allows high frequency sample data to leak into the output pixels, causing postaliasing, where even if the original sample values were at a high enough frequency so that there was no aliasing, errors are introduced by poor filtering. The left side of Figure 7.27 shows a graph of the box filter.

Figure 7.28 shows the performance of the box filter for reconstructing two 1D functions. The top graph shows the step function we used previously to illustrate

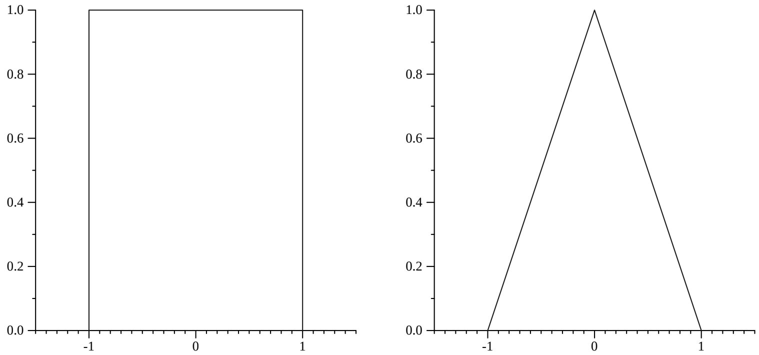


Figure 7.27: Graphs of the box filter (left) and triangle filter (right). Though neither of these is a particularly good filter, they are both computationally efficient, easy to implement, and good baselines when evaluating other filters.

Gibbs phenomenon. For this function, the box does reasonably well. The bottom graph shows the box used to reconstruct a sinusoidal function that has increasing frequency along the *x* axis. Not only does it do a poor job of reconstructing the function when the frequency is low, giving a highly discontinuous result even though the original function was smooth, but as the function's frequency approaches and passes the Nyquist limit, it also does an extremely poor job of reconstruction.

Filter 281

⟨*Box Filter Declarations*⟩≡

```
class BoxFilter : public Filter {
public:
    BoxFilter(Float xw, Float yw) : Filter(xw, yw) { }
    Float Evaluate(Float x, Float y) const;
};
```

Because the evaluation function isn't called with (x,y) values outside of the filter's extent, we can always return 1 for the filter function's value.

⟨*Box Filter Method Definitions*⟩≡

```
Float BoxFilter::Evaluate(Float x, Float y) const {
    return 1.;
}
```

Triangle Filter

⟨*triangle.cpp**⟩≡

```
#include "sampling.h"
#include "paramset.h"
⟨Triangle Filter Declarations⟩
⟨Triangle Filter Method Definitions⟩
```

The triangle filter gives slightly better results than the box: samples at pixel centers have a weight of one, and the weight linearly falls off to the square extent of the filter. See the right side of Figure 7.27 for a graph of the triangle filter.

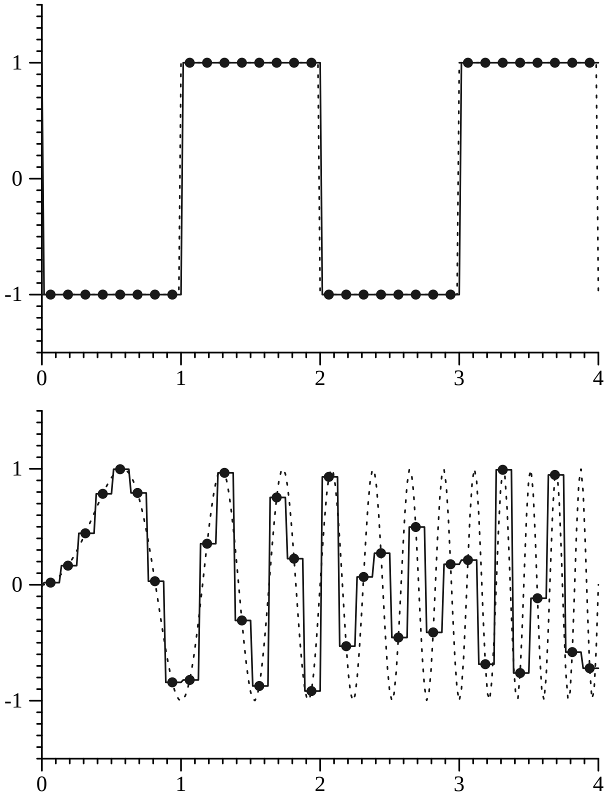


Figure 7.28: The box filter in action, reconstructing samples of the step funciton (top) and a sinusoidal function with increasing frequency as *x* increases. It does reasonably well with the step function (as would be expected), but an extremely poor job with the sinusoidal function.

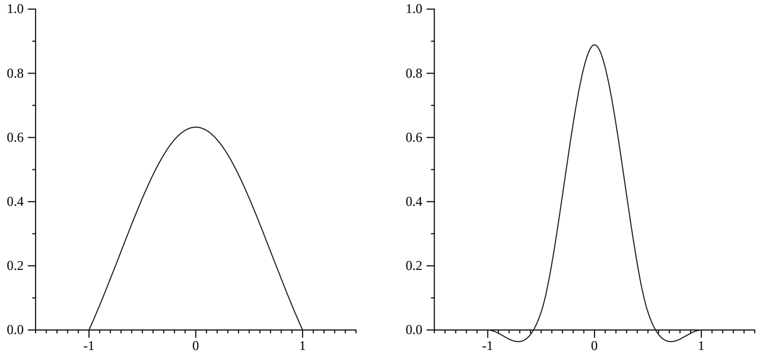


Figure 7.29: Graphs of the Gaussian filter (left) and the Mitchell filter (right). The Gaussian gives images that tend to be a bit blurry, while the negative lobes of the Mitchell filter help to accentuate and sharpen edges in final images.

⟨*Triangle Filter Declarations*⟩≡

```
class TriangleFilter : public Filter {
public:
    TriangleFilter(Float xw, Float yw) : Filter(xw, yw) { }
    Float Evaluate(Float x, Float y) const;
};
```

Filter	281
Filter::xWidth	281
Filter::yWidth	281

⟨*Triangle Filter Method Definitions*⟩≡

```
Float TriangleFilter::Evaluate(Float x, Float y) const {
    return max(0.f, xWidth - fabsf(x)) *
        max(0.f, yWidth - fabsf(y));
}
```

Gaussian Filter

⟨*gaussian.cpp**⟩≡

```
#include "sampling.h"
#include "paramset.h"
⟨Gaussian Filter Declarations⟩
⟨Gaussian Filter Method Definitions⟩
```

The Gaussian is the first filter in `lrt` that gives good performance in practice. It applies a Gaussian bump which is centered at the output pixel and radially symmetric around it. We subtract the Gaussian's value at the end of its extent from the filter value; this makes the filter go to zero at its limit–see the left side of Figure 7.29. The Gaussian does tend to give blurrier images than the next two filters, though its blurring tendencies can help mask any remaining aliasing in the image.

⟨*Gaussian Filter Declarations*⟩≡

```
class GaussianFilter : public Filter {
public:
    ⟨GaussianFilter Public Methods⟩
private:
    ⟨GaussianFilter Private Data⟩
    ⟨GaussianFilter Utility Functions⟩
};
```

The 1D Gaussian filter function of width *w* is

$$f(x) = e^{-\alpha x^2} - e^{-\alpha w^2},$$

where α controls the rate of falloff of the filter. Smaller values cause a slower falloff, giving a blurrier image. For efficiency, the constructor precomputes the constant term for $e^{-\alpha w^2}$ in each direction.

⟨*GaussianFilter Public Methods*⟩≡

```
GaussianFilter(Float xw, Float yw, Float a)
    : Filter(xw, yw) {
    alpha = a;
    expX = expf(-alpha * xWidth * xWidth);
    expY = expf(-alpha * yWidth * yWidth);
}
```

⟨*GaussianFilter Private Data*⟩≡

```
Float alpha;
Float expX, expY;
```

⟨*Gaussian Filter Method Definitions*⟩≡

```
Float GaussianFilter::Evaluate(Float x, Float y) const {
    return Gaussian(x, expX) * Gaussian(y, expY);
}
```

281 Filter

⟨*GaussianFilter Utility Functions*⟩≡

```
Float Gaussian(Float d, Float expv) const {
    return max(0.f, float(expf(-alpha * d * d) - expv));
}
```

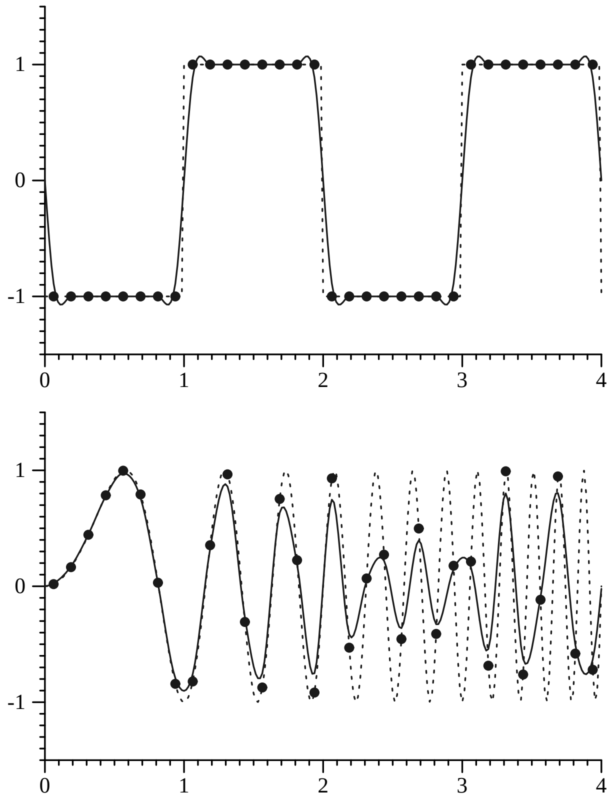
Mitchell Filter

⟨*mitchell.cpp**⟩≡

```
#include "sampling.h"
#include "paramset.h"
⟨Mitchell Filter Declarations⟩
⟨Mitchell Filter Method Definitions⟩
```

This whole section is lame – should show more of the math and say things about the filter like they're piecewise cubic. Also, Don told me that he wished he had emphasized more that he intended his filters to really be a 1D family of filters along the line B+2C=1 (or whatever); we should make sure to beat that horse here. I'll rewrite this text if I get back around to it, since I bothered to do the derivation of the mitchell filter by hand a year ago.

Filter design is a notoriously difficult craft, mixing mathematical analysis and perceptual experiments. Mitchell and Netravali have developed a family of parameterized filter functions in order to be able to explore this space in a systematic manner. After analyzing test subjects' subjective responses to images filtered with a variety of parameter values, they developed a filter that tends to do a good job of trading off between *ringing*–phantom edges next to actual edges in the image–and *blurring*–overly blurred results–two common artifacts from poor reconstruction filters.



Filter 281

Figure 7.30: The Mitchell–Netravali filter used to reconstruct the example functions. It does a good job with both of these functions, introducing minimal ringing with the step function, and accurately representing the sinusoid.

Figure/whatever showing experimental evaluation of B and C parameters...

Note that in the graph of this filter on the right side of Figure 7.29 that this filter function takes on negative values out by its edges; it has *negative lobes*. In practice these negative regions improve the sharpness of edges, giving crisper images (reduced blurring). If they become too large, however, ringing tends to start to enter the image. Figure 7.30 shows this filter reconstructing our two test functions. It does extremely well with both of them–there is minimal ringing with the step function, and it does a very good job with the sinusoidal function, up until the sampling rate isn't sufficient to capture the function's detail.

⟨*Mitchell Filter Declarations*⟩≡

```
class MitchellFilter : public Filter {
public:
    ⟨MitchellFilter Public Methods⟩
private:
    Float B, C;
};
```

⟨*MitchellFilter Public Methods*⟩≡

```
MitchellFilter(Float b, Float c, Float xw, Float yw)
    : Filter(xw, yw) { B = b; C = c; }
```

Like many 2D image filtering functions, the Mitchell-Netravali filter is the product of two one-dimensional filter functions in the *x* and *y* directions. Such filters are called *separable*. (In fact, all of the filters in `lrt` are separable, though this wasn't always made explicit in the previous ones.)

⟨*Mitchell Filter Method Definitions*⟩≡

```
Float MitchellFilter::Evaluate(Float x, Float y) const {
    return Mitchell1D(x * invXWidth) *
        Mitchell1D(y * invYWidth);
}
```

⟨*MitchellFilter Public Methods*⟩+≡

```
Float Mitchell1D(Float r) const {
    r = fabsf(2.f * r);
    if (r > 1.f)
        return ((-B - 6*C) * r*r*r + (6*B + 30*C) * r*r +
            (-12*B - 48*C) * r + (8*B + 24*C)) * (1.f/6.f);
    else
        return ((12 - 9*B - 6*C) * r*r*r +
            (-18 + 12*B + 6*C) * r*r +
            (6 - 2*B)) * (1.f/6.f);
}
```

```
281 Filter::invXWidth
281 Filter::invYWidth
286 MitchellFilter
289 SincFilter
```

Windowed Sinc Filter

⟨*sinc.cpp**⟩≡

```
#include "sampling.h"
#include "paramset.h"
⟨Sinc Filter Declarations⟩
⟨Sinc Filter Method Definitions⟩
```

Finally, we provide the `SincFilter` class, which implements a filter based on the sinc function. In practice, the sinc filter is often multiplied by another function that goes to zero after some distance; this gives a filter function with finite extent, which is necessary for an implementation with reasonable performance. An additional parameter τ controls how many cycles the sinc function passes through before it is clamped to a value of zero; the left side of Figure 7.31 shows a graph of three cycles of the sinc function (solid line) and a graph of the windowing function we use, which was developed by Lanczos (dashed line). The Lanczos window is just the central lobe of the sinc function, scaled to cover the τ cycles:

$$w(x) = \frac{\sin \pi x/\tau}{\pi x/\tau}.$$

The right side of Figure 7.31 shows the product of the sinc function and the windowing function, giving the filter that we will implement here.

Figure 7.32 shows the windowed sinc's reconstruction performance for uniform 1D samples. Thanks to the windowing, it exhibits far less ringing than the infinite-extent sinc function when reconstructing the step function (compare to Figure 7.11), and it does extremely well with reconstructing the sinusoidal function until prealiasing begins.

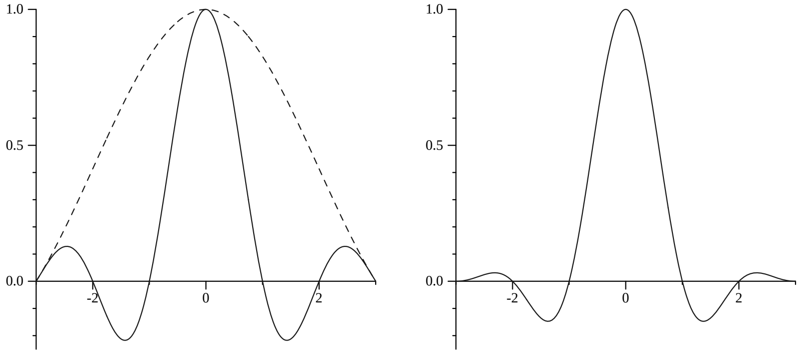


Figure 7.31: Graphs of the sinc filter. On the left is the sinc function, truncated after three cycles (solid line) and the Lanczos windowing function (dashed line). On the right is the product of these two functions, which we implement in the `SincFilter`.

`SincFilter 289`

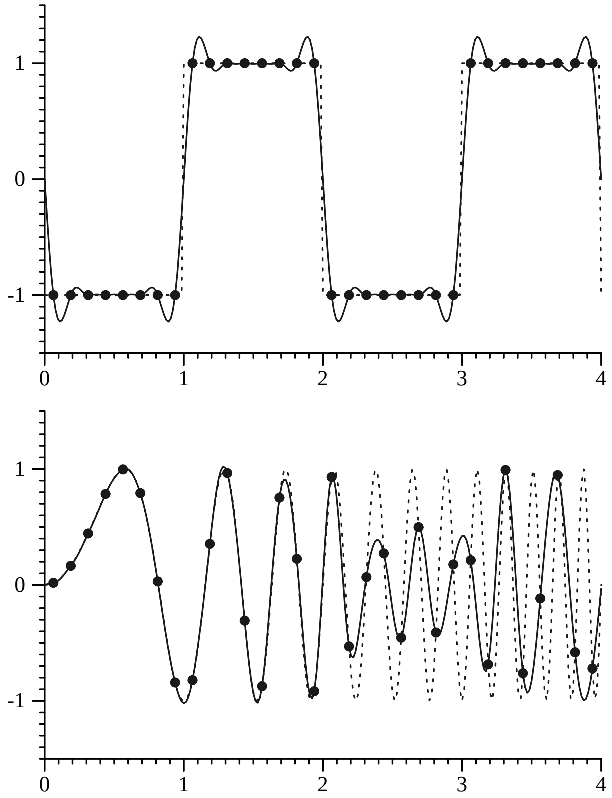


Figure 7.32: Results of the windowed sinc being used to reconstruct the example functions; here $\tau = 3$. Like the infinite sinc, it suffers from ringing with the step function, though there is less ringing in the windowed version. The filter does quite well with the sinusoid, however.

⟨*Sinc Filter Declarations*⟩≡

```
class SincFilter : public Filter {
public:
    SincFilter(Float xw, Float yw, Float t) : Filter(xw, yw) {
        tau = t;
    }
    Float Evaluate(Float x, Float y) const;
    Float Sinc1D(Float x) const;
private:
    Float tau;
};
```

Like the Mitchell-Netravali filter, the sinc filter is also separable.

⟨*Sinc Filter Method Definitions*⟩≡

```
Float SincFilter::Evaluate(Float x, Float y) const{
    return Sinc1D(x * invXWidth) * Sinc1D(y * invYWidth);
}
```

The implementation straightforward; we compute the value of the sinc function and then multiply it by the value of the Lanczos windowing function.

⟨*Sinc Filter Method Definitions*⟩+≡

```
Float SincFilter::Sinc1D(Float x) const {
    x = fabsf(x);
    if (x < 1e-5) return 1.f;
    if (x > 1.)   return 0.f;
    x *= M_PI;
    Float sinc = sinf(x * tau) / (x * tau);
    Float lanczos = sinf(x) / x;
    return sinc * lanczos;
}
```

```
281 Filter
281 Filter::invXWidth
281 Filter::invYWidth
678 M_PI
```

Further Reading

One of the best books on signal processing, sampling, reconstruction, and the Fourier transform is Bracewell(Bracewell 2000). Glassner's *Principles of Digital Image Synthesis* (Glassner 1995) has a series of chapters on the theory and application uniform and non-uniform sampling and reconstruction to computer graphics. For an extensive survey of the history of and techniques for interpolation of sampled data, including the sampling theorem, see Meijering's survey article (Meijering 2002).

Crow first identified aliasing as a major source of artifacts in computer generated images (Crow 1977). Using non-uniform sampling to turn aliasing into noise was introduced by Cook et al(Cook 1986) and Dippé and Wold(Dippé and Wold 1985); this work was based on experiments by Yellot, who investigated the distribution of photoreceptors in the eyes of monkeys (Yellot 1983). Dippé and Wold also first introduced the pixel filtering equation to graphics and developed a Poisson sample pattern with a minimum distance between samples. Lee et al developed a technique for adaptive sampling based on statistical tests to compute images to a given error tolerance (Lee, Redner, and Uselton 1985).

Mitchell has extensively investigated sampling patterns for ray tracing; his 1987 and 1991 SIGGRAPH papers have many key insights, and the best candidate approach described in this chapter is based on the latter paper (Mitchell 1987; Mitchell 1991). Another efficient technique to generate Poisson disk patterns was developed by McCool and Fiume (McCool and Fiume 1992) and Hiller et al applied a technique based on relaxation that takes a random point set and improves its distribution (Hiller, Deussen, and Keller 2001).

Shirley first introduced the use of discrepancy to evaluate the quality of sample patterns in computer graphics (Shirley 1991). This work was built upon by Mitchell (Mitchell 1992) and Dobkin and Mitchell (Dobkin and Mitchell 1993), Dobkin et al (Dobkin, Eppstein, and Mitchell 1996).

Mitchell's first paper on discrepancy introduced the idea of using deterministic low-discrepancy sequences for sampler, removing all randomness in the interest of lower-discrepancy (Mitchell 1992). Such *quasi-random* sequences are the basis of Quasi Monte Carlo methods, which will be described in Chapter 14. The seminal book on quasi-random sampling and algorithms for generating low-discrepancy patterns was written by Niederreiter (Niederreiter 1992).

More recently, Keller and collaborators have investigated quasi-random sampling patterns for a variety of applications in graphics (Keller 1996; Keller 1997; Keller 2001). The $(0,2)$-sequence sampling techniques we have used in the `LDSampler` and `BestCandidateSampler` are based on a paper by Köllig and Keller (Kollig and Keller 2002). Some of their techniques are based on algorithms developed by Friedel and Keller (Friedel and Keller 2000). Wong et al compared numeric error with various low-discrepancy sampling schemes (Wong, Luk, and Heng 1997), though one of Mitchell's interesting findings was that low-discrepancy sampling sequences sometimes lead to visually-objectionable artifacts in images that aren't present with other sampling patterns.

`BestCandidateSampler` 275
`LDSampler` 258

Chiu et al suggested a *multi-jittered* 2D sampling technique that combined the properties of stratified and Latin hypercube approaches, though their technique doesn't ensure good distributions across all elementary intervals as $(0,2)$-sequences do.

Mitchell has recently investigated how much better stratified sampling patterns are than random patterns in practice (Mitchell 1996); in general, the smoother the function being sampled is, the more effective they are. For very quickly-changing functions (e.g. pixel regions with complex geometry overlapping them), more sophisticated stratified patterns perform no better than unstratified random patterns. As such, for complex scenes with complex variation in the high-dimensional image function, the advantages of fancy sampling schemes compared to a simple stratified pattern are likely to be minimal.

Cook first introduced the Gaussian filter to graphics (Cook 1986). Mitchell and Netravali investigated a family of filters by doing experiments with human observers to find the most effective ones; the Mitchell filter in this chapter is the one they chose as best (Mitchell and Netravali 1988). Kajiya and Ullner have investigated image filtering methods that account for the effect of the reconstruction characteristics of Gaussian falloff from pixels in CRTs (Kajiya and Ullner 1981), and more recently, Betrisey et al describe Microsoft's *ClearType* technology for display of text on LCDs (Betrisey, Blinn, Dresevic, Hill, Hitchcock, Keely, Mitchell, Platt, and Whitted 2000). However, we are not aware of any work that has investigated this issue for image display.

There has been quite a bit of research into reconstruction filters for image resampling applications; while this application is slightly different than reconstructing non-uniform samples for image synthesis, much of this experience is applicable. Turkowski reports that the Lanczos-windowed sinc filter gives the best results for image resampling (Turkowski 1990b). Meijering et al tested a variety of filters for image resampling by applying a series of transformations to images such that if perfect resampling had been done, the final image would be the same as the original; they also found that the Lanczos window performed well (as did a few others), and that truncating the sinc without a window gave some of the worst results (Meijering, Niessen, Pluim, and Viergever 1999). Our figures comparing the results of 1D reconstruction of samples of step functions and sinusoids were inspired by Hoffman's (Hoffman 2002).

Exercises

7.1 incremental faster computation of folded radical inverse function

7.2 Rushmeier and Ward have suggested a non-linear filter function that works well for reducing the visual impact of noise in images generated with Monte Carlo light transport algorithms (Rushmeier and Ward 1994). Their observation was that if a single sample is substantially brighter than all of the samples around it, then it is likely that the other nearby samples should also have detected the bright feature that caused the spike of brightness. If the standard filters described in this chapter are used to reconstruct the final image, the spike contributes to a small number of pixels, resulting in a visually unappealing bright area in the final image.

The filters in `lrt` are all *linear*: the value of the filter function is determined solely by the position of the sample with respect to the pixel position; the value of the sample has no impact on the value of the filter funciton. In an effort to reduce noise without changing the overall brightness of the image, the Rushmeier–Ward filter widens the extent of the filter function for a given sample depending on the brightness of that sample compared to the brightness of the nearby samples. For example, if the base filter being used was a box filter with extent such that nine samples contributed to each pixel, if any of the nine samples contributed more than 25% to the final pixel value, it would be detected as a spike and a wider filter extent would be used for it, dispersing its energy to a wider set of pixels in the final image.

Because `lrt` assumes that filters are linear, and because it doesn't store sample values after adding their contribution to the image, implementing the Rushmeier–Ward filter in `lrt` is not straightforward. Investigate approaches for modifying `lrt` so that this filter can be used. Is it possible to add it without storing all of the image samples (which may take a large amount of memory for high-resolution images with many samples)?

7.3 Mitchell and Netravali note that there are reconstruction filters that use both the value of a function and its derivative at the point to be able to do substantially better reconstruction than if just the value of the function is known (Mitchell and Netravali 1988). Furthermore, they report that they have derived closed form expressions for the screen-space derivatives of Lambertian and Phong reflection models, though they do not include these expressions in their paper. Investigate derivative-based reconstruction and extend `lrt` to support this technique. Because it will likely be difficult to derive expressions for the screen-space derivatives for general shapes and BSDF models, investigate approximations based on finite-differencing. Techniques built on the ideas behind the ray differentials of Section 11.2 may be fruitful for this effort.

8. Film and the Imaging Pipeline

294 Film

As radiance values are found for image samples, they are passed to the `Film` class, which is repsonsible for computing their weighted contributions to the pixels around them. Just as the particular type of film in a (non-digital) camera determines how light arriving at the film plane is transformed into color on the film, the `Film` class models the response of a virtual sensing device in the simulated camera.

After the main rendering loop exits and radiance values have been computed for all of the image samples, the film applies the imaging pipeline, which is the third and last major phase of `lrt`'s execution. It is responsible for transforming the spectral pixel values at each pixel into final output values for display or storage in a file. All of the transformations it applies seek to reduce the impact of the limitations of the display devices that are currently available for displaying digital images. For example, computer displays generally expect an RGB color triplet to describe the color of each pixel, not an arbitrary spectral power distribution. Spectra described by general basis function coefficients must be converted to RGB before they can be displayed. A related problem, which will be discussed in Section 8.3, is that displays have substantially less dynamic range from the brightest to the darkest radiance value that they can display than the range of radiance values that are present in many realistic scenes. Therefore, the radiance values computed by `lrt` must be compressed to the displayable range in a way that causes the final displayed image to appear as close as possible to the way it would appear on an ideal display device without this restriction. These and other issues are addressed by various phases of the imaging pipeline.

8.1 Film Interface

The `Film` base-class, defined in `core/film.h`, defines the abstract interface for `Film` implementations.

⟨*Film Declarations*⟩≡

```
class Film {
public:
    ⟨Film Interface⟩
    ⟨Film Public Data⟩
};
```

The `Film` constructor must be given the overall resolution of the image in the *x* and *y* directions; these are stored in the public member variables `xResolution` and `yResolution` since the `Camera`s in Chapter 6 need these values to compute some of the camera-related transformations (e.g. the raster to camera space transformations).

Is there any reason not to have multiple films per camera? I think it would be cleaner to have ColorFilm, BWFilm, DepthFilm, etc, and you might want to generate multiple images for a single rendering. You could also set up multiple films with different imaging parameters. Of course this would all go away if we provided a way to run the film pipeline as a post-process, which we really don't. We can write out the SPD coefficients, but we don't show how to use those things.

`Ray`	36
`Sampler`	237
`Spectrum`	181

⟨*Film Interface*⟩≡

```
Film(int xres, int yres)
    : xResolution(xres), yResolution(yres) {
}
```

⟨*Film Public Data*⟩≡

```
const int xResolution, yResolution;
```

The first key `Film` method is `AddSample()`, which takes a sample and corresponding camera ray, radiance value, and alpha value, and updates the image.

⟨*Film Interface*⟩+≡

```
virtual void AddSample(const Sample &sample, const Ray &ray,
    const Spectrum &L, Float alpha) = 0;
```

After the main rendering loop exits, `Scene::Render()` calls the film's `WriteImage()` method, which is the film's cue to do any processing necessary to compute the final image and display it or store it as appropriate.

⟨*Film Interface*⟩+≡

```
virtual void WriteImage() = 0;
```

The `Film`'s final responsibility is to be able to determine the range of integer pixel values that the `Sampler` is responsible for generating samples for. While this range would be from $(0,0)$ to $(\mathrm{xResolution}-1, \mathrm{yResolution}-1)$ for a basic film implementation, in practice it is usually necessary to generate samples that extend a bit beyond the edges of the final image. The `ImageFilm`'s implementation of this method in the next section will make the reason for its existence more clear.

⟨*Film Interface*⟩+≡

```
virtual void GetSampleExtent(int *xstart,
    int *xend, int *ystart, int *yend) const = 0;
```

8.2 Image Film

we should explain why there is a Film interface at all. What other kinds of film are possible? Why are we only providing one? I like the idea of having a color, black-and-white, and a depth film be separate, since this corresponds more naturally to the concept of a film in a camera. However, since we're digital, we should show ways that you could do something really sweet with the Film interface. Perhaps I'll write an OpenGL previewer film – I think this is the right place to do it.

We will provide only one specific `Film` implementation for `lrt` here, `ImageFilm`, which is a general-purpose implementation with a highly configurable image pipeline. It is in `film/image.cpp`. The exercises suggest a few other types of `Film` that are useful for specialized applications.

⟨*ImageFilm Declarations*⟩≡

```
class ImageFilm : public Film {
public:
    ⟨ImageFilm Public Methods⟩
private:
    ⟨ImageFilm Private Data⟩
};
```

294 `Film`
301 `ImageInfo`

The `ImageFilm` constructor takes three extra parameters beyond the overall image resolution. First is a filter function that it uses to compute the weighted contribution that each sample makes to the pixels around it. Next is `crop`, which specifies a crop window that can be used to select a rectangular subset of pixels to be rendered– the crop window can be useful for debugging as well as for breaking a large image into chunks that can then be reassembled later. **why isn't crop in the base Film class?** The crop window is specified in NDC space, with each coordinate ranging from zero to one–see Figure 8.1. The third extra parameter is an `ImageInfo` structure that gathers all of the parameters used in the various stages of the imaging pipeline. Because there are many such parameters, we've gathered them up into this single structure rather than passing them individually here. (This also allows us to defer describing these parameters until later in this chapter, together with the image pipeline implementation.)

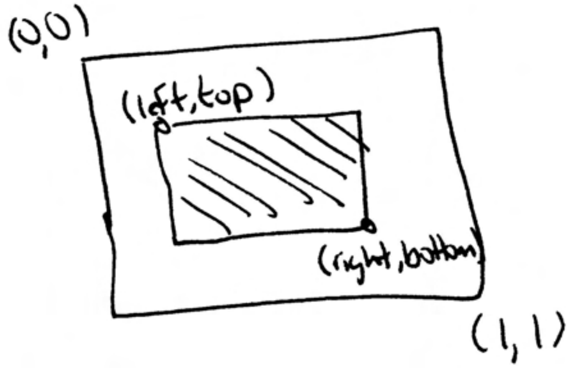


Figure 8.1: The image crop window specifies a subset of the image to be rendered. It is specified in NDC space, with coordinates ranging from $(0,0)$ to $(1,1)$. The `Film` class only allocates space for and stores pixel values in the region inside the crop window.

⟨*ImageFilm Method Definitions*⟩≡

```
ImageFilm::ImageFilm(int xres, int yres, Filter *filt, const Float crop[4],
        int wf, const ImageInfo &ii)
    : Film(xres, yres) {
    filter = filt;
    memcpy(cropWindow, crop, 4 * sizeof(Float));
    imageInfo = ii;
    writeFrequency = sampleCount = wf;
    ⟨Compute film image extent⟩
    ⟨Allocate film image storage⟩
    ⟨Precompute filter weight table⟩
}
```

Film	294
Filter	281
ImageInfo	301

⟨*ImageFilm Private Data*⟩≡

```
Filter *filter;
ImageInfo imageInfo;
Float cropWindow[4];
int writeFrequency, sampleCount;
```

In conjunction with the overall image resolution, the crop window gives the extent of pixels that need to be actually stored and written to disk. `xPixelStart` and `yPixelStart` store the pixel position of the upper left corner of the crop window, and `xPixelCount` and `yPixelCount` give the total number of pixels in each direction. Their values are easily computed from the overall resolution and the crop window, though the calculations are done carefully such that if an image is rendered in pieces with crop windows that cover the entire image, each final pixel will be present in only one of the sub-images.

⟨*Compute film image extent*⟩≡

```
xPixelStart = Ceil2Int(xResolution * cropWindow[0]);
xPixelCount = Ceil2Int(xResolution * cropWindow[1]) - xPixelStart;
yPixelStart = Ceil2Int(yResolution * cropWindow[2]);
yPixelCount = Ceil2Int(yResolution * cropWindow[3]) - yPixelStart;
```

⟨*ImageFilm Private Data*⟩+≡

```
int xPixelStart, yPixelStart, xPixelCount, yPixelCount;
```

Given the pixel resolution of the possibly-cropped image, the constructor next allocates an array of `Pixel` structures, one for each pixel. Pixel radiance values are stored in the `L` member variable, their alpha values are stored in `alpha`, their *z* depths are stored in `depth`, and `weightSum` holds the sum of filter weight values for the sample contributions to the pixel; it is used to perform pixel filtering (Equation 7.7.3).

Because small rectangular blocks of pixels need to be updated for each image sample, the `ImageFilm` uses a `BlockedArray` to store the pixels in order to reduce the number of cache misses as samples arrive and pixel values are updated.

⟨*Allocate film image storage*⟩≡

```
pixels = new BlockedArray<Pixel>(xPixelCount, yPixelCount);
```

⟨*ImageFilm Private Data*⟩+≡

```
struct Pixel {
    Pixel() : L(0.f) {
        alpha = 0.f;
        depth = INFINITY;
        weightSum = 0.f;
    }
    Spectrum L;
    Float alpha, depth, weightSum;
};
BlockedArray<Pixel> *pixels;
```

672	`BlockedArray`
294	`Film::xResolution`
294	`Film::yResolution`
678	`INFINITY`
181	`Spectrum`

With `lrt`'s default settings, each image sample ends up contributing to an average of sixteen pixels in the final image. Particularly for simple scenes, where relatively little time is spent on ray intersection testing and shading computations, the time spent updating the image for each sample can be significant. Therefore, the `ImageFilm` precomputes a table of filter values for use in the `AddSample()` method. Here we are making the assumption that the filter only varies according to the absolute value of the offset from the filter kernel's origin in the *x* and *y* directions and only compute values for the positive quadrant of offsets.

This approach saves both the expense of the virtual function calls to the `Filter::Evaluate()` method as well as the expense of evaluating the filter. The error introduced by not evaluating the filter at each sample's precise location generally isn't noticeable in practice.

```
⟨Precompute filter weight table⟩≡
  #define FILTER_TABLE_SIZE 16
  filterTable = new Float[FILTER_TABLE_SIZE * FILTER_TABLE_SIZE];
  Float *ftp = filterTable;
  for (int y = 0; y < FILTER_TABLE_SIZE; ++y) {
      Float fy = ((Float)y + .5f) * filter->yWidth /
          FILTER_TABLE_SIZE;
      for (int x = 0; x < FILTER_TABLE_SIZE; ++x) {
          Float fx = ((Float)x + .5f) * filter->xWidth /
              FILTER_TABLE_SIZE;
          *ftp++ = filter->Evaluate(fx, fy);
      }
  }
```

```
⟨ImageFilm Private Data⟩+≡
  Float *filterTable;
```

ImageFilm::AddSample() applies the ideas of image sampling and reconstruction theory to take image samples and filter them to compute pixel values. Recall the pixel filtering equation:

$$p(x,y) = \frac{\sum_i f(x-x_i, y-y_i) L(x_i,y_i)}{\sum_i f(x-x_i, y-y_i)}$$

Filter	281
Filter::Evaluate()	281
Filter::xWidth	281
Filter::yWidth	281
ImageFilm::filter	296
Ray	36
Spectrum	181

which describes each final pixel radiance value $p(x,y)$ as the weighted sum of nearby samples' radiance values, according to a filter function f. Because all of the Filters in lrt have finite extent, this method starts by computing which pixels will be affected by the current sample. Then, turning the pixel filtering equation inside out, it updates two running sums for each pixel (x,y) that is affected by the sample. One sum accumulates for the numerator of the sample interpolation equation and the other the denominator. When all of the samples have been processed, final pixel values will be computed by performing the division.

```
⟨ImageFilm Method Definitions⟩+≡
  void ImageFilm::AddSample(const Sample &sample, const Ray &ray,
      const Spectrum &L, Float alpha) {
      ⟨Compute sample's raster extent⟩
      ⟨Loop over filter support and add sample to pixel arrays⟩
      ⟨Possibly write out in-progress image⟩
  }
```

To find the raster-space bounds of the pixels that the sample potentially contributes to, AddSample() converts the continuous sample coordinate to a discrete coordinate by subtracting 0.5. It then offsets this value by the filter width in each direction–this process is shown in Figure 8.2. The ceiling of the coordinates of the extent is taken on its lower end and the floor on the upper end, since pixels outside the bound of the extent are guaranteed to not be affected by the sample.

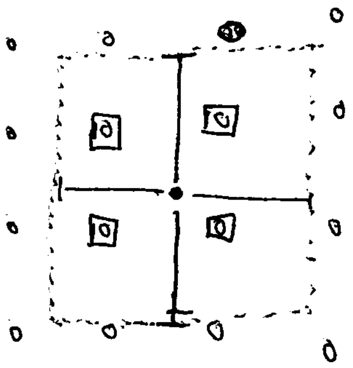


Figure 8.2: Sample Raster Extent Figure.

⟨*Compute sample's raster extent*⟩≡

```
Float dImageX = sample.imageX - 0.5f, dImageY = sample.imageY - 0.5f;
int x0 = Ceil2Int (dImageX - filter->xWidth);
int x1 = Floor2Int(dImageX + filter->xWidth);
int y0 = Ceil2Int (dImageY - filter->yWidth);
int y1 = Floor2Int(dImageY + filter->yWidth);
x0 = max(x0, xPixelStart);
x1 = min(x1, xPixelStart + xPixelCount - 1);
y0 = max(y0, yPixelStart);
y1 = min(y1, yPixelStart + yPixelCount - 1);
```

281 Filter::xWidth
281 Filter::yWidth
296 ImageFilm::filter
297 ImageFilm::xPixelCount
297 ImageFilm::xPixelStart
297 ImageFilm::yPixelCount
297 ImageFilm::yPixelStart
239 Sample::imageX
239 Sample::imageY

Given the extent of pixels that are affected by this sample–(x0,y0) to (x1,y1), inclusive–we can now loop over all of those pixels and then filter the sample value appropriately.

⟨*Loop over filter support and add sample to pixel arrays*⟩≡

```
Float depth = ray(ray.maxt).z;
⟨Precompute x and y filter table offsets⟩
for (int y = y0; y <= y1; ++y)
    for (int x = x0; x <= x1; ++x) {
        ⟨Evaluate filter value at (x,y) pixel⟩
        ⟨Update pixel values with filtered sample contribution⟩
    }
```

All of the pixels along each row in the *x* direction have the same *y* offset to the sample position, and similarly all of the pixels in each column in *y* have the same offset in *x*. Therefore, here it's possible to precompute the indices into the filter weight table needed for sample filtering before looping over the pixels, thus saving repeated work in that loop.

note still need to do that min call due to floating point error...

⟨*Precompute x and y filter table offsets*⟩≡

```
int *ifx = (int *)alloca((x1-x0+1) * sizeof(int));
for (int x = x0; x <= x1; ++x) {
    Float fx = fabsf((x - dImageX) * filter->invXWidth * FILTER_TABLE_SIZE);
    ifx[x-x0] = min(Floor2Int(fx), FILTER_TABLE_SIZE-1);
}
int *ify = (int *)alloca((y1-y0+1) * sizeof(int));
for (int y = y0; y <= y1; ++y) {
    Float fy = fabsf((y - dImageY) * filter->invYWidth * FILTER_TABLE_SIZE);
    ify[y-y0] = min(Floor2Int(fy), FILTER_TABLE_SIZE-1);
}
```

Each discrete integer pixel (x, y) has an instance of the filter function centered around it. To compute the filter weight for a particular sample, it's necessary to find the offset from the pixel to the sample's position in discrete coordinates and evaluate the filter function. If we were evaluating the filter explicitly, the appropriate computation would be:

```
filterWt = filter->Evaluate(x - dImageX, y - dImageY);
```

Instead, we retrieve the appropriate filter weight from the table.

⟨*Evaluate filter value at* (x, y) *pixel*⟩≡

```
int offset = ify[y-y0]*FILTER_TABLE_SIZE + ifx[x-x0];
Float filterWt = filterTable[offset];
```

explain why we don't filter the depth samples.

⟨*Update pixel values with filtered sample contribution*⟩≡

```
Pixel &pixel = (*pixels)(x - xPixelStart, y - yPixelStart);
pixel.L.AddWeighted(filterWt, L);
pixel.alpha += alpha * filterWt;
pixel.depth = min(pixel.depth, depth);
pixel.weightSum += filterWt;
```

Because the pixel reconstruction filter spans a number of pixels, the `Sampler` must generate image samples a bit outside of the range of pixels that will actually be output. In this manner, even pixels at the boundary of the image will have an equal density of samples around them in all directions, not just toward the interior of the image.

⟨*ImageFilm Method Definitions*⟩+≡

```
void ImageFilm::GetSampleExtent(int *xstart,
        int *xend, int *ystart, int *yend) const {
    *xstart = Floor2Int(xPixelStart - filter->xWidth);
    *xend   = Ceil2Int (xPixelStart + xPixelCount  +
        filter->xWidth);
    *ystart = Floor2Int(yPixelStart - filter->yWidth);
    *yend   = Ceil2Int (yPixelStart + yPixelCount +
        filter->yWidth);
}
```

`Filter::invXWidth`	281
`Filter::invYWidth`	281
`Filter::xWidth`	281
`Filter::yWidth`	281
`FILTER_TABLE_SIZE`	298
`ImageFilm::filter`	296
`ImageFilm::filterTable`	298
`ImageFilm::pixels`	297
`ImageFilm::xPixelCount`	297
`ImageFilm::xPixelStart`	297
`ImageFilm::yPixelCount`	297
`ImageFilm::yPixelStart`	297
`Pixel`	297
`Pixel::alpha`	297
`Pixel::depth`	297
`Pixel::L`	297
`Pixel::weightSum`	297
`Sampler`	237
`Spectrum::AddWeighted()`	182

For images that take a long time to render, it can be helpful to the user if the renderer periodically writes out the image that has been computed so far; this is easily handled by calling the `WriteImage()` method periodically.

⟨*Possibly write out in-progress image*⟩≡

```
if (--sampleCount == 0) {
    WriteImage();
    sampleCount = writeFrequency;
}
```

8.2.1 Image Output

After the main rendering loop finishes, `Scene::Render()` calls the `Film`'s `WriteImage()` method. The simplest thing for it to do is to save the array of floating-point SPD coefficients at each pixel for later processing or display by programs with knowledge of the basis functions used. This case is handled by the ⟨*Write raw coefficient image*⟩ fragment. More commonly, the user will want to store an image that can be directly displayed on a CRT or LCD. In this case, the pixels pass through an imaging pipeline that uses information about the particular display device to compute a new image for that device. A number of tricky issues, ranging from limitations of display devices to the behavior of the human visual system, need to be carefully addressed during this process.

296 `ImageFilm::sampleCount`
296 `ImageFilm::writeFrequency`

As mentioned above, the `ImageInfo` structure holds parameters used by the imaging pieline that describe the characteristics of a particular display device.

⟨*ImageInfo Declarations*⟩≡

```
struct ImageInfo {
    ImageInfo() {
        ⟨ImageInfo Constructor Implementation⟩
    }
    ⟨ImageInfo Public Data⟩
};
```

⟨*ImageFilm Method Definitions*⟩+≡

```
void ImageFilm::WriteImage() {
    ⟨Allocate working imaging memory and compute normalized pixel values⟩
    ⟨Compute premultiplied alpha color values⟩
    if (imageInfo.writeCoefficientImage) {
        ⟨Write raw coefficient image⟩
    }
    else {
        ⟨Apply display imaging pipeline⟩
    }
    ⟨Release temporary image memory⟩
}
```

⟨*ImageInfo Public Data*⟩≡

```
bool writeCoefficientImage;
```

⟨*ImageInfo Constructor Implementation*⟩≡

```
writeCoefficientImage = false;
```

In either case, this method starts by making a copy of the pixel values so that changes to them during the image processing don't change the film's pixel values. Thus, this method could be called multiple times during rendering to write partial images.

⟨*Allocate working imaging memory and compute normalized pixel values*⟩≡

```
int nPix = xPixelCount * yPixelCount;
Spectrum *Lout = new Spectrum[nPix];
Float *AlphaOut = new Float[nPix];
int offset = 0;
for (int y = 0; y < yPixelCount; ++y) {
    for (int x = 0; x < xPixelCount; ++x) {
        Lout[offset] = (*pixels)(x, y).L;
        AlphaOut[offset] = (*pixels)(x, y).alpha;
        ⟨Normalize pixel with weight sum⟩
        ++offset;
    }
}
```

ImageFilm::xPixelCount	297
ImageFilm::yPixelCount	297
INFINITY	678
Pixel::alpha	297
Pixel::L	297
Pixel::weightSum	297
Spectrum	181
Spectrum::Clamp()	183

As the pixels in this copy are being initialized, their final values based on the pixel filtering equation arecomputed for each of them by dividing each pixel sample value by the value of `Pixel::weightSum` that has been accumulated for that pixel.

⟨*Normalize pixel with weight sum*⟩≡

```
Float weightSum = (*pixels)(x, y).weightSum;
if (weightSum != 0.f) {
    Float invWt = 1.f / weightSum;
    Lout[offset] *= invWt;
    Lout[offset].Clamp(0, INFINITY);
    AlphaOut[offset] *= invWt;
}
```

Each spectral pixel value is also optionally multiplied by its alpha value; pixel colors scaled by alpha are known as having *premultiplied alpha* (also known as *associated alpha*). This representation has a number of advantages if image compositing operations are being performed by a separate program using images from `lrt` (see the further reading section for pointers.)

⟨*Compute premultiplied alpha color values*⟩≡

```
if (imageInfo.premultiplyAlpha)
    for (int i = 0; i < xPixelCount * yPixelCount; ++i)
        Lout[i] *= AlphaOut[i];
```

⟨*ImageInfo Public Data*⟩+≡

```
bool premultiplyAlpha;
```

⟨*ImageInfo Constructor Implementation*⟩+≡

```
premultiplyAlpha = true;
```

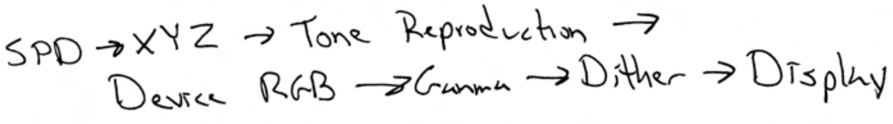

SPD → XYZ → Tone Reproduction →
Device RGB → Gamma → Dither → Display

Figure 8.3: There are four main stages in the `ImageFilm`'s display pipeilne. First tone reproduction algorithms may be used to remap the wide range of pixel radiance values to the more limited range that displays are capable of. Next, the pixel colors are converted to the color representation used by the display–typically red, green, and blue primary colors. Gamma correction then accounts for the non-linear relationship between color values sent to the display and their brightness on the display, and finally dithering adds a small amount of random noise to the pixel values to help break up transitions between different colors in different regions of the image.

If the raw per-pixel SPD coefficients are being stored directly, then no additional work is necessary and the image can be written immediately.

⟨*Write raw coefficient image*⟩≡

```
WriteImageFloat(imageInfo.filename, (Float *)Lout, AlphaOut,
    xPixelCount, yPixelCount, COLOR_SAMPLES,
    xResolution, yResolution);
```

180	`COLOR_SAMPLES`
294	`Film::xResolution`
294	`Film::yResolution`
297	`ImageFilm::xPixelCount`
297	`ImageFilm::yPixelCount`

After saving the image, the working memory is freed.

⟨*Release temporary image memory*⟩≡

```
delete[] Lout;
delete[] AlphaOut;
```

If raw coefficients aren't being written, the image is passed through a number of stages of the image pipeline, summarized in Figure 8.3. to convert the spectral image into a format suitable for display or printing while still creating the best possible image. The fragment ⟨*Apply display imaging pipeline*⟩ applies each of these stages in turn.

⟨*Apply display imaging pipeline*⟩≡
  ⟨*Possibly apply bloom effect to image*⟩
  ⟨*Apply tone reproduction to image*⟩
  ⟨*Convert image to display RGB*⟩
  ⟨*Scale image and handle out-of-gamut RGB values*⟩
  ⟨*Apply gamma correction to image*⟩
  ⟨*Map image to display range*⟩
  ⟨*Dither image*⟩
  ⟨*Save display image to disk*⟩

8.3 ***ADV***: Perceptual Issues and Tone Mapping

In the early days of computer graphics, typical shading models returned color values between zero and one, with no pretense of being associated with actual physical quantities. Thus, pixels had values in this range as well, and images could be directly displayed on a CRT with a framebuffer with RGB components from 0 to 255, just by scaling the pixel values. In the real-world, it is not unusual for scenes to have radiance values with magnitudes ranging from 0.01 to $1,000$, representing five orders of magnitude of variation from the brightest parts to the darkest parts. Remarkably, the human visual system generally handles this extreme range of brightness well, since the human eye is sensitive mostly to local constrast, not absolute brightness. Not only do computer displays not take radiance values as input, but they are unable to display very bright colors or very dim colors; they can generally display only about two orders of magnitude of brightness variation under ideal viewing conditions.

Because realisitic scenes rendered with physically-based rendering algorithms suffer from this mismatch between scene brightness and the display device's capabilities, it's important to address the issue of displaying the image such that it has as close an appearence to the actual scene as possible. It has recently been an active area of research to find good methods to compress those extra orders of magnitude for image display. This work has been broadly classified as *tone mapping*[1]; it draws on research into the human visual system (*HVS*) to guide the development of techniques for image display. By exploiting various properties of the HVS, tone mapping algorithms have been developed that do remarkably well at compensating for display device limitations. In this section, we will describe and implement a few such algorithms. Our coverage of this area touches on representative a subset of the possibilities, though the further reading section gives pointers to many recent papers in this field.

8.3.1 ***ADV***: Luminance and photometry

Because tone mapping algorithms are generally based on human perception of brightness, most tone mapping operators are based on the unit of *luminance*, which gives a sense of how bright a spectral power distribution appears to a human observer. For example, luminance accounts for the fact that a SPD with a particular amount of energy in the green wavelengths will appear much brighter to a human than a SPD with the same amount of energy in blue.

Luminance is closely related to radiance; given a spectral radiance value, a luminance value can be computed with a simple conversion formula. In fact, all of the radiometric quantities defined in Chapter 5 have analogs in the field of *photometry*, which is the study of visible electromagnetic radiation and its perception by the HVS. Each spectral radiometric quantity can be converted to its corresponding photometric quantity by integrating with the spectral response curve $V(\lambda)$, which describes the relative sensitivity of the human eye to various wavelengths.[2]

Luminance, which we will denote here by Y, is related to spectral radiance $L(\lambda)$ by

$$Y = \int_\lambda L(\lambda) V(\lambda) \mathrm{d}\lambda.$$

[1] **Jack Tumblin's rant re: tone reproduction vs. tone mapping XXX**

[2] The spectral response curve model is based on experiments done in a normally-illuminated indoor environment. Because sensitivity to color decreases in dark environments, it doesn't model HVS response well under all lighting situations. Nonetheless, it forms the basis for the definition of luminance and other related photometric properties.

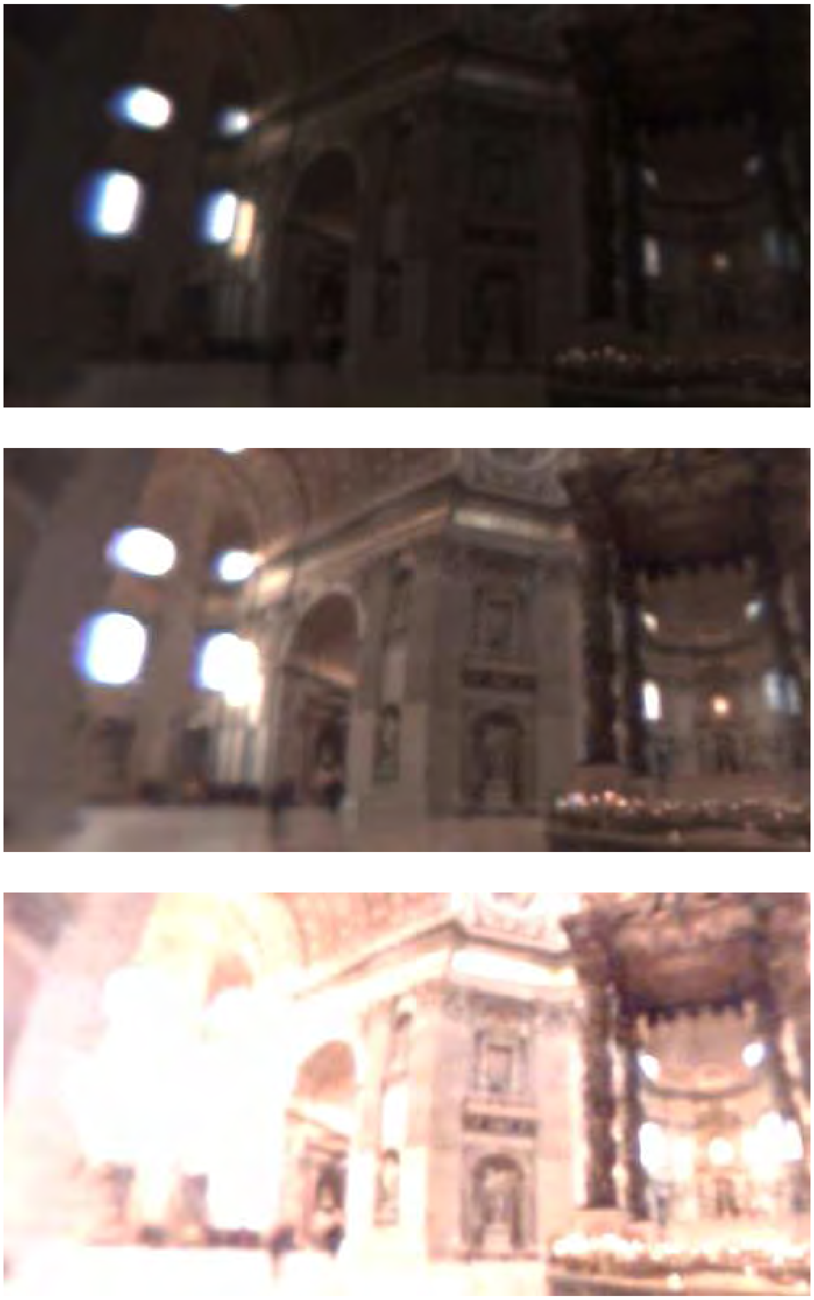


Figure 8.4: St. Peter's Basilica in Rome: a high dynamic range of St. Peter's accurately encodes the lighting inside the Basilica, including the multiple orders of magnitude of radiance values that are present. The standard approach of choosing a fixed radiance value to map to the brightest displayable color does poorly with this environment. Here, we have chosen three different maximum radiance values, each greater than the last by a factor of ten. Observe that even in the darkest image, the light from the windows is blown out. In the top and middle image, some areas are too dark to make out details, while in the brightest image, large regions of the image map to the maximum value, so that no detail remains. A human observer inside St. Peter's on the day these photographs were taken would have been able to see detail throughout the environment. The tone reproduction operators in this section will apply more sophisticated algorithms than simple scaling to map the wide range of radiance values to the display device's displayable range.

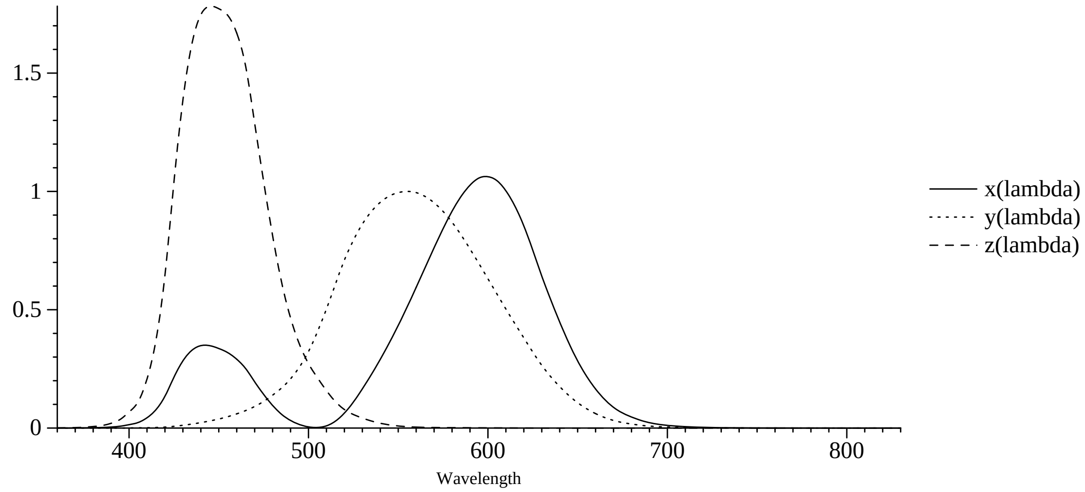


Figure 8.5: To compute the XYZ values for an arbitrary SPD, the SPD is convolved with each of the three matching curves shown here.

Luminance (cd/m^2, or nits)	
600,000	Sun at horizon
120,000	60 Watt light bulb
8,000	Clear sky
100–1000	Typical office
1–100	Typical computer display
1–10	Street lighting
0.25	Cloudy moonlight

Figure 8.6: Representative luminance values for a number of lighting contitions.

Luminance and the spectral response curve $V(\lambda)$ are closely related to the XYZ representation of color (Section 5.1.2): the CIE $Y(\lambda)$ tristimulus curve was chosen to be proportional to $V(\lambda)$ so that

$$Y = 683 \int_{\lambda} L(\lambda) Y(\lambda) \mathrm{d}\lambda.$$

Thus, we already have the luminance of each pixel in the image within a scale factor. The units of luminance are candelas per meter squared (cd/m^2), where the candela is the photometric equivalent of radiant intensity. The quantity cd/m^2 is often referred to as a *nit*. Some representative luminance values are given in Figure 8.6.

The human eye has two types of photoreceptor responsible for detecting light: rods and cones. Rods help with perception in dark environments (*scoptic* light levels), ranging from approximately 10^{-6} to 10 cd/m^2. Rods give little information about color and are not very good at resolving fine details. Cones handle light ranging from approximately .01 to 10^8 cd/m^2 (*photopic* light levels.) There are three types of cones, with sensitivity to different wavelengths of light. Computer displays generally display luminances from about 1 to 100 cd/m^2.

Figure 8.7: Bloom makes brights brighter

8.3.2 ***ADV***: Bloom

Before describing tone mapping algorithms for remapping images to the displayable range, we'll describe a technique that helps fool the HVS into percieving that an image on a display is brighter than it actually is. When part of an environment being viewed by the human eye is substantially brigher than the rest of it, an effect called "bloom" often causes a blurred glow in the area around the bright object. The origins of this effect aren't completely understood, but are largely believed to be due to scattering of light inside the human eye. Researchers in computer graphics have found that simulating this effect in rendered images can make images appear substantially more realistic; when this glow is present in part of an image, the human visual system naturally perceives that that part of the image is much brighter than the rest of it. Figure 8.7 shows an example of this effect applied to an image with a number of bright specular highlights.

`lrt` optionally applies a bloom effect to images as they go through the imaging pipeline. The filter used here is empirical and not directly based on a model of the human visual system though it works well in practice (see the further reading section for pointers to more physically-based glare effects). The basic idea is to apply a very wide filter that falls off quickly to all of the pixels in the image. Because the filter has a wide support, very bright pixels can contribute energy to other pixels nearby them; because it quickly falls off, it doesn't blur regions of the image that have similar brightness values but extremely bright pixels are able to overwhelm the low filter weight and spread out their contribution to other pixels. This bloom image is then mixed into the original image with a user-supplied weight.

This filter takes two parameters; `bloomRadius`, which gives the fraction of the image that the filter covers, and `bloomFraction`, which gives the weight that the blurred bloom image is given when mixed with the original image.

⟨*ImageInfo Public Data*⟩+≡

```
  Float bloomRadius, bloomFraction;
```

By default, `bloomRadius` is zero, indicating that the filter is disabled. Values around 0.1 or 0.2 are good starting points when using this filter in practice.

⟨*ImageInfo Constructor Implementation*⟩+≡

```
bloomRadius = 0.f;
bloomFraction = .2f;
```

If the user has set the bloom radius to be greater than zero, the filter is applied.

⟨*Possibly apply bloom effect to image*⟩≡

```
if (imageInfo.bloomRadius > 0.f) {
    ⟨Compute image-space extent of bloom effect⟩
    ⟨Initialize bloom filter table⟩
    ⟨Apply bloom filter to image pixels⟩
    ⟨Mix bloom effect into each pixel⟩
    ⟨Free memory allocated for bloom effect⟩
}
```

First the width of the filter in pixels must be determined; this is just based on the `bloomRadius` value times the larger of the *x* and *y* resolutions of the image.

⟨*Compute image-space extent of bloom effect*⟩≡

```
int bloomSupport = Float2Int(imageInfo.bloomRadius *
    max(xResolution, yResolution));
int bloomWidth = bloomSupport / 2;
```

Film::xResolution	294
Film::yResolution	294
ImageInfo::bloomRadius	307

Because the bloom filter function will be evaluated many times over the image, it's worth precomputing a table of its values. THe implementation here uses the radially-symmetric filter function

$$f(x,y) = \left(1 - \frac{\sqrt{x^2+y^2}}{d}\right)^8,$$

where *d* is the width of the filter. This filter was introduced by Chiu et al as an ad-hoc model of bloom (Chiu, Herf, Shirley, Swamy, Wang, and Zimmerman 1993). This filter is not separable, and thus the number of pixels that must be filtered for each output pixel is quadratic in the filter's width; this gives extra motivation for tabularizing the filter values.

⟨*Initialize bloom filter table*⟩≡

```
Float *bloomFilter = new Float[bloomWidth * bloomWidth];
for (int i = 0; i < bloomWidth * bloomWidth; ++i) {
    Float dist = sqrtf(i) / bloomWidth;
    bloomFilter[i] = powf(max(0.f, 1.f - dist), 8.f);
}
```

The filtering method computes the entries in a temporary image that holds the bloom contribution. It is important to not update the original image as the bloom values are computed, since this would result in errors due to *feedback* as pixel values with bloom would incorrectly be used to determine bloom at nearby pixels.

⟨*Apply bloom filter to image pixels*⟩≡

```
Spectrum *bloomImage = new Spectrum[nPix];
for (int y = 0; y < yPixelCount; ++y) {
    for (int x = 0; x < xPixelCount; ++x) {
        ⟨Compute bloom for pixel (x,y)⟩
    }
}
```

To find a pixel in the bloom image, first it is necessary to find the range of pixels that potentially contribute bloom to it. The filter is then applied to all of the relevant pixels.

⟨*Compute bloom for pixel* (x,y)⟩≡

```
⟨Compute extent of pixels contributing bloom⟩
int offset = y * xPixelCount + x;
Float sumWt = 0.;
for (int by = y0; by <= y1; ++by)
    for (int bx = x0; bx <= x1; ++bx) {
        ⟨Accumulate bloom from pixel (bx,by)⟩
    }
bloomImage[offset] /= sumWt;
```

The extent of contributing pixels is found by offsetting by the filter width in each direction and clamping to the overall image resolution, similar to how pixels that an image sample contributes to are found.

⟨*Compute extent of pixels contributing bloom*⟩≡

```
int x0 = max(0, x - bloomWidth);
int x1 = min(x + bloomWidth, xPixelCount - 1);
int y0 = max(0, y - bloomWidth);
int y1 = min(y + bloomWidth, yPixelCount - 1);
```

297 `ImageFilm::xPixelCount`
297 `ImageFilm::yPixelCount`
307 `ImageInfo::bloomFraction`
181 `Spectrum`

The current pixel isn't included in the bloom computation, since the intent here is to add contributions from other pixels to the current one. For all the others, the bloom filter weights the pixel's contribution to the bloom image.

⟨*Accumulate bloom from pixel* (*bx*,*by*)⟩≡

```
int dx = x - bx, dy = y - by;
if (dx == 0 && dy == 0) continue;
int dist2 = dx*dx + dy*dy;
if (dist2 < bloomWidth * bloomWidth) {
    int bloomOffset = bx + by * xPixelCount;
    Float wt = bloomFilter[dist2];
    sumWt += wt;
    bloomImage[offset] += wt * Lout[bloomOffset];
}
```

Once the bloom image is computed, it's mixed into the original image according to the `bloomFraction` value.

⟨*Mix bloom effect into each pixel*⟩≡

```
for (int i = 0; i < xPixelCount * yPixelCount; ++i)
    Lout[i] = (1.f - imageInfo.bloomFraction) * Lout[i] +
        imageInfo.bloomFraction * bloomImage[i];
```

⟨*Free memory allocated for bloom effect*⟩≡

```
delete[] bloomFilter;
delete[] bloomImage;
```

8.3.3 ***ADV***: Tone mapping interface

The basic approach to tone reproduction is to derive a scaling function that maps each pixel's value to the display's dynamic range. For simple tone reproduction operators, a single function is often used for all pixels in the image. Such operators are called *spatially-uniform* operators. They give a monotonic mapping of image luminance to display luminance. More sophisticated approaches use a function that varies based on each pixel's brightness and the brightness of nearby pixels; these are *spatially-varying* operators and they do necessarily guarantee a monotonic mapping.

That it is possible (and often more effective) to have a spatially-varying operator is interesting. This approach works well because the human eye is more sensitive to local contrast than overall luminance. Because of this characteristic, it is often possible to assign totally different pixel values to separate parts of the image that started with the same absolute luminance, without the human observer noticing that anything is amiss.

A key goal of many tone mapping operators is to preserve local contrast in the displayed image rather than preserving absolute brightness. It's more important to make sure that enough distinct colors are used in all regions of the image–bright and dim–so that different colors are seen, rather than mapping a wide range of image intensities to the same pixel values. Thus, an object that is twice as bright as another one in the scene doesn't necessarily need to be twice as bright on the display. Again, local changes in contrast are the most important thing for the human visual system.

The HVS's overall sensitivity to luminance changes varies depending on the *adaptation luminance*, which we will denote by Y^a. The adaptation luminance may vary over different parts of the image. In the methods below, we will use both the *display adaptation luminance* Y_d^a, which is the adaptation luminance of the human observer looking at the computer display, and the *world adaptation luminance*, Y_w^a, the adaptation luminance that the human would have if viewing the actual scene.

Because the rods in the human eye take over from the cones in very dim environments, the HVS has substantially different characteristics in the dark. For example, color perception is reduced and everything appears to be varying shades of dark gray. Furthermore, *spatial acuity* is reduced: at an adaptation luminance of 1000 nits, the HVS can resolve about 50 cycles of spatial detail per degree of vision, while at .001 nits, only about 2.2 cycles per degree can be made out. Tone reproduction operators that account for scoptic light levels often introduce some blurring to the image to account for this effect.

All of the tone mapping operators inherit from the `ToneMap` base class, which is defined in `core/tonemap.h` and specifies the interface method `ToneMap::Map()`.

⟨*ToneMap Declarations*⟩≡

```
class ToneMap {
public:
    ⟨ToneMap Interface⟩
};
```

The `ToneMap::Map()` method takes a pointer to the array of the image's pixel luminance values, the resolution of the image, and the maximum luminance that the display device being used is capable of generating. It is responsible for computing a scale-factor for each pixel with its tone mapping technique and storing the scale in the `scale` array The scale should be such that the luminances of the scaled pixels will be in the range `[0, maxDisplayY]`.

⟨*ToneMap Interface*⟩+≡

```
virtual void Map(const Float *y, int xRes, int yRes,
    Float maxDisplayY, Float *scale) const = 0;
```

For tone mapping, the `ImageInfo` structure holds a `ToneMap` pointer, initialized to `NULL` by default. For the tone reproduction operators that make use of information, it also holds a field that records the maximum luminance that the device is capable of displaying, `maxDisplayY`.

⟨*ImageInfo Public Data*⟩+≡

```
ToneMap *toneMap;
Float maxDisplayY;
```

By default, no tone mapping is performed, and the maximum display luminance is set to a reasonable value for common displays.

⟨*ImageInfo Constructor Implementation*⟩+≡

```
toneMap = NULL;
maxDisplayY = 100.f;
```

If a tone mapping operator was specified in the scene description, it is applied here. First, luminance values are computed or each pixel, then the operator computes a scale-factor for each pixel, and then the image is scaled.

297	ImageFilm::xPixelCount
297	ImageFilm::yPixelCount
301	ImageInfo
185	Spectrum::y()
310	ToneMap

⟨*Apply tone reproduction to image*⟩≡

```
if (imageInfo.toneMap) {
    Float *scale = new Float[nPix], *lum = new Float[nPix];
    ⟨Compute pixel luminance values⟩
    imageInfo.toneMap->Map(lum, xPixelCount, yPixelCount,
        imageInfo.maxDisplayY, scale);
    ⟨Apple scale to pixels for tone mapping and map to [0,1]⟩
    delete[] scale;
    delete[] lum;
}
```

⟨*Compute pixel luminance values*⟩≡

```
for (int i = 0; i < xPixelCount * yPixelCount; ++i)
    lum[i] = 683.f * Lout[i].y();
```

Because the scale values returned by the tone mapping operator should leave the pixel luminance values `lum[i]` in the range `[0, maxDisplayY]` but current display devices don't take luminance values as input, the results from the operator now need to be scaled to the range $[0,1]$ for the rest of the pipeline.

⟨*Apple scale to pixels for tone mapping and map to* $[0,1]$⟩≡

```
Float displayTo01 = 683.f / imageInfo.maxDisplayY;
for (int i = 0; i < xPixelCount * yPixelCount; ++i)
    Lout[i] *= scale[i] * displayTo01;
```

8.3.4 ***ADV***: Maximum to white

The most straightforward tone reproduction operator to apply (besides just hoping that the image's pixel values are already in a suitable range for the display) is the *maximum to white* operator. It loops over all of the pixels to find the one with the greatest luminance and computes the scale of the pixels so that the largest luminance maps to the maximum luminance value of the display. Its trivial implementation is in `tonemaps/maxwhite.cpp`.

⟨MaxWhiteOp Declarations⟩≡

```
class MaxWhiteOp : public ToneMap {
    ⟨MaxWhiteOp Public Methods⟩
};
```

⟨MaxWhiteOp Public Methods⟩≡

```
void Map(const Float *y, int xRes, int yRes,
         Float maxDisplayY, Float *scale) const {
    ⟨Compute maximum luminance of all pixels⟩
    Float s = maxDisplayY / maxY;
    for (int i = 0; i < xRes * yRes; ++i)
        scale[i] = s;
}
```

ToneMap 310

⟨Compute maximum luminance of all pixels⟩≡

```
Float maxY = 0.;
for (int i = 0; i < xRes * yRes; ++i)
    maxY = max(maxY, y[i]);
```

There are two main disadvantages to this operator in practice (as its application to the St. Peter's image in Figure 8.4 showed). First, it doesn't account for the human visual system at all: if the lights in the scene are turned up to be 100 times brighter and the scene is re-rendered, the maximum to white operator will give the same displayed image as before. Second, a small number of very bright pixels can cause the rest of the image to be too dark to be visible. Nonetheless, it can work well for scenes without too much dynamic range in the image and serves as a baseline that can show off the improvement that smarter operators offer.

8.3.5 ***ADV***: Contrast-based scale factor

The next tone reproduction operator focuses on preserving contrast in the displayed image. It was developed by Greg Ward (Ward 1994a). Built upon work by researchers who have studied the HVS and developed models that simulate it, this operator is based on a model that describes the smallest change in luminance that is noticeable to a human observer given a particular adaptation luminance (the *just noticeable difference*, otherwise known as *JND*). The larger the adaptation luminance, the larger a change in luminance is needed to be noticeable. The operator tries to set image luminances such that one JND in the displayed image corresponds to one JND in the actual environment.

This uniform scale factor attempts to preserve *contrast visibility*–given a region of the original image that would be is just noticeably different from its neighbor to a human observer, it tries to scale display pixel values such that the person

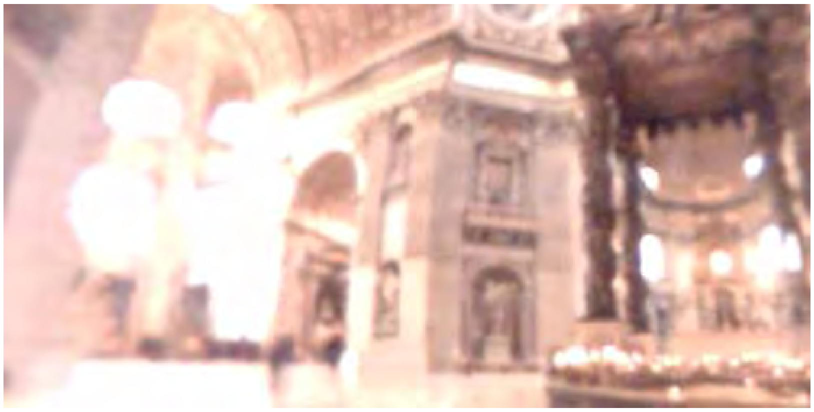


Figure 8.8: Application of the contrast-preserving scale factor to the image works well in some parts of the image, though it doesn't do well at preserving detail in the very bright areas. Any operator that uses a single global scale factor is susceptible to this problem.

looking at the display perceives that those two pixel values are just noticeably different. In particular, a scale factor that increased JNDs would be a waste of precious display dynamic range, while one that reduced JNDs would cause visually detectible features to disappear.

310 ToneMap

Blackwell found that given an adaptation luminance in the photopic range Y^a, a reasonable model of the minimum change in luminance necessary to be visible is given by

$$\Delta Y(Y^a) = 0.0594 \cdot (1.219 + (Y^a)^{0.4})^{2.5}.$$

Thus, this operator would like to determine a scale s such that

$$\Delta Y(Y_d^a) = s \cdot \Delta Y(Y_w^a),$$

where Y_d^a is the display adaptation luminance and Y_w^a is the world adaptation luminance for someone observing the actual scene.

Substituting Blackwell's model and solving for s, gives

$$s = \left(\frac{1.219 + (Y_d^a)^{0.4}}{1.219 + (Y_w^a)^{0.4}} \right)^{2.5}. \qquad (8.3.1)$$

⟨*ContrastOp Declarations*⟩≡

```
class ContrastOp : public ToneMap {
public:
    ContrastOp(Float day) { displayAdaptationY = day; }
    void Map(const Float *y, int xRes, int yRes, Float maxDisplayY,
        Float *scale) const;
    Float displayAdaptationY;
};
```

⟨*ContrastOp Method Definitions*⟩≡

```
void ContrastOp::Map(const Float *y, int xRes, int yRes,
        Float maxDisplayY, Float *scale) const {
    ⟨Compute world adaptation luminance, Ywa⟩
    ⟨Compute contrast-preserving scalefactor, s⟩
    for (int i = 0; i < xRes*yRes; ++i)
        scale[i] = s;
}
```

One unresolved issue is how to compute the world adaptation luminance Y_w^a. Ideally, this would be computed based on which part of the scene the viewer was looking at and how long they had been looking at it (it takes some time for the eye to adapt to luminace changes). Lacking this information, this operator computes a log average of all of the luminances in the original image. Taking the log average, rather than an average of the original luminances, helps prevent small bright regions from overwhelming luminance values in the rest of the image.

⟨*Compute world adaptation luminance,* `Ywa`⟩≡

```
Float Ywa = 0.;
for (int i = 0; i < xRes * yRes; ++i)
    if (y[i] > 0) Ywa += logf(y[i]);
Ywa = expf(Ywa / (xRes * yRes));
```

ContrastOp 313

The scale is directly computed from Equation 8.3.1.

⟨*Compute contrast-preserving scalefactor,* `s`⟩≡

```
Float s = powf((1.219f + powf(displayAdaptationY, 0.4f)) /
    (1.219f + powf(Ywa, 0.4f)), 2.5f);
```

Figure 8.8 shows this operator in action; it does a reasonable job on the St. Peter's image, but has trouble maintaining detail in the bright areas. This isn't too surprising; any operator that uses the same scale factor at all pixels will have trouble with images with many orders of magnitude of brightness variation. This operator does work well on typical indoor scenes, however, and is computationally efficient.

8.3.6 ***ADV***: Varying adaptation luminance

As mentioned earlier, it is often possible to make better use of the display's dynamic range by using a scale factor that varies over the image. Here we will implement a tone reproduction operator tailored for high-contrast scenes with many orders of magnitude of brightness variation. It computes a local adaptation luminance that smoothly varies over the image. The local adaptation luminance is then used to compute a scale-factor using a contrast-preserving tone reproduction operator, in a similar manner to the `ContrastOp` operator defined above.

The main difficulty with methods that compute a spatially-varying local adpatation luminance is that they are prone to artifacts at boundaries between very bright and very dim parts of the image. If the tone reproduction operator scales the dim pixels using an adaptation luminance that includes the effects of the bright pixels, the dim pixels will be mapped to black, causing a halo artifact at the boundary of the final image. (An example of this effect is shown in Figure 8.9.)

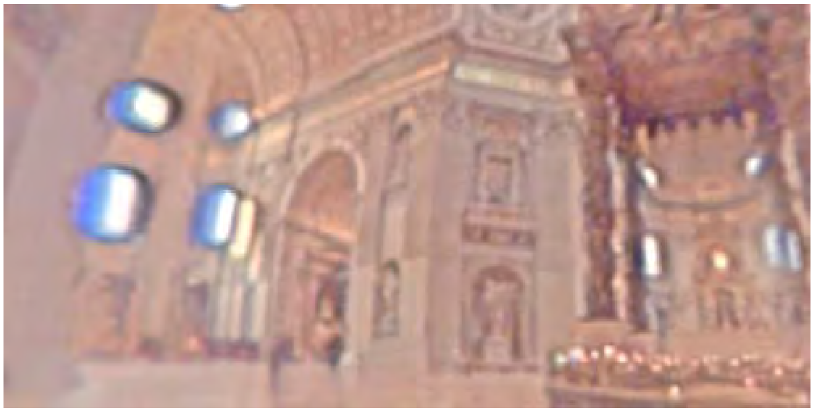


Figure 8.9: When the adaptation luminance is computed using a fixed search radius at each pixel (here roughly ten pixels), there are often unsightly halo artifacts at transitions between very bright and dimmer regions of the image. For example, there are black borders around the bright light from the windows, where the adaptation luminance has been computed using the bright window light, such that the tone mapping operator maps the adjacent, dimmer pixels to very low values.

313 ContrastOp
310 ToneMap

Instead, it is better if we can make sure that the dim pixels have an adaptation luminance based on just nearby dim pixels. This operator uses an image processing technique to detect these boundaries in an effort to address this problem. Over local regions of the image where the adaptation luminance is slowly changing, this tone reproduction operator gives a local scale factor, which is tuned to preserve contrast. However, since adaptation is allowed to vary over the image, details are also preserved–bright regions aren't blown out to be white, and dark regions aren't mapped down to black pixels. The approach implemented is based on a tone reproduction operator developed by Ashikhmin (Ashikhmin 2002). Reinhard et al simultaneously developed a different operator that uses the same technique to comupte local adaptation (Reinhard, Stark, Shirley, and Ferwerda 2002). The implementation is in `tonemaps/highcontrast.cpp`.

The results of applying this operator to the St. Peter's image are shown in Figure 8.10. It does substantially better than the contrast operator, thanks to its spatially-varying scale factor.

⟨*HighContrastOp Declarations*⟩≡

```
class HighContrastOp : public ToneMap {
public:
    void Map(const Float *y, int xRes, int yRes, Float maxDisplayY,
        Float *scale) const;
private:
    ⟨HighContrastOp Utility Methods⟩
};
```

The tone mapping function that `HighContrastOp` uses is based on the *threshold versus intensity* (TVI) function, which gives the just noticable luminance difference for given adaptation level $TVI(Y^a)$. This is similar in spirit to the JND function used in `ContrastOp`, but is based on a more complex model of the human visual system, including a model of response to scoptic light levels.

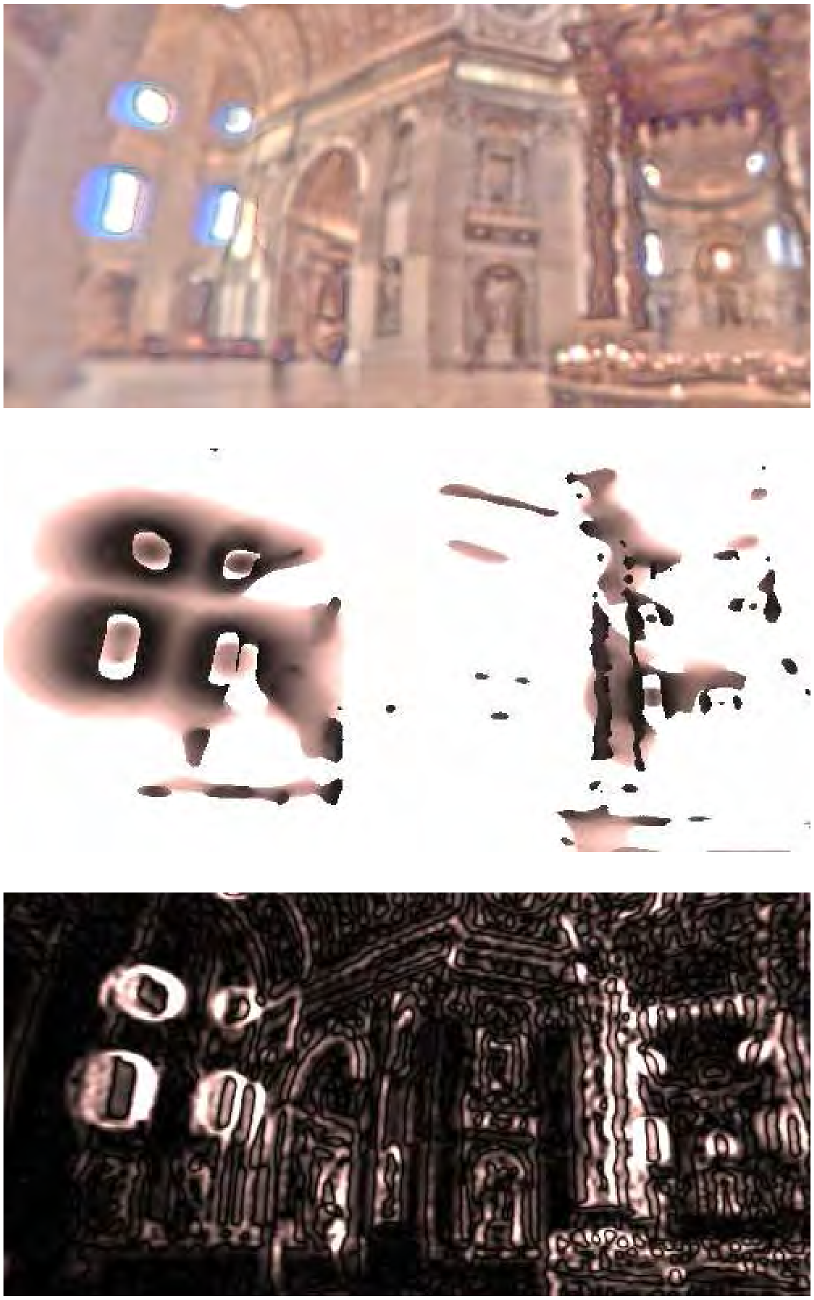


Figure 8.10: The high-contrast tone reproduction operator computes a spatially-varying adaptation luminance while paying attention to boundaries between areas with substantially different luminances. The result of its use on the St. Peter's image is shown at the top. It does an excellent job of remapping this image to a small dynamic range. The middle image is a grey-scale visualization of the blur radius that was used to compute adaptation luminance at each pixel; boundaries with large luminance changes are successfully detected, indicated by small radii, while areas with slower change in luminance compute adaptation luminance over a wider area. The bottom image shows the local contrast computed at each pixel with a blur radius of 1.5 and 3 pixels.

From the TVI function, we define the *perceptual capacity*, which tells us, given a particular adaptation level, how many JNDs a given luminance range covers:

$$\frac{Y_a - Y_b}{TVI(Y^a)}$$

Later, we will use this to remap local regions of the image in a way that preserves their perceptual capacity when displayed.

So that we can quickly compute the perceptual capacity of a given pair of luminance values, the auxilary capacity function $C(Y)$ is defined as the integral

$$C(Y) = \int_0^Y \frac{\mathrm{d}Y}{TVI(Y)},$$

where the approximation is made that the adaptation level to compute the differential perceptual capacity at a given luminance is assumed to be equal to the luminance. Then $C(Y_a) - C(Y_b)$ is the perceptual capacity from Y_a to Y_b.

Ashikhmin has made some simplifications to a widely-used TVI function in order to be able to integrate it analytically to compute $C(Y)$, giving the function

$$C(Y) = \begin{cases} Y/0.0014 & Y < 0.0034 \\ 2.4483 + \log(L/0.0034)/0.4027 & 0.0034 \leq Y < 1 \\ 16.563 + (Y-1)/0.4027 & 1 \leq Y < 7.2444 \\ 32.0693 + \log(Y/7.2444)/0.0556 & \text{otherwise} \end{cases}$$

⟨*HighContrastOp Utility Methods*⟩≡

```
static Float C(Float y) {
    if (y < 0.0034f) return y / 0.0014f;
    else if (y < 1) return 2.4483f + log10f(y/0.0034f)/0.4027f;
    else if (y < 7.2444f) return 16.563f + (y - 1)/0.4027f;
    else return 32.0693f + log10f(y / 7.2444f)/0.0556f;
}
```

Given $C(Y)$, we can now take a given luminance value and determine how many JND steps it is from the minimum luminance in the image,

$$C(Y) - C(Y_{\text{min}})$$

and we can also compute, of all of the JND steps that the image goes through, what fraction of the way through all of them it is:

$$\frac{C(Y) - C(Y_{\text{min}})}{C(Y_{\text{max}}) - C(Y_{\text{min}})}$$

This gives us a sense of how far through the range of display luminances this world luminance should be mapped. Thus, the overall tone mapping operator, giving a result in terms of display luminance, is

$$T(Y) = Y_d^{\text{max}} \frac{C(Y) - C(Y_{\text{min}})}{C(Y_{\text{max}}) - C(Y_{\text{min}})}.$$

⟨*HighContrastOp Utility Methods*⟩+≡
```
static Float T(Float y, Float CYmin, Float CYmax,
        Float maxDisplayY) {
    return maxDisplayY * (C(y) - CYmin) / (CYmax - CYmin);
}
```

We can now define the main tone reproduction function. It compues the minimum and maximum luminances of all pixels in the image so that $Y_{\max}$ and $Y_{\min}$ can be computed. In order to be able to quickly do the searches to compute adaptation luminances, it also builds an image pyramid data structure, where the original image is progressively filtered down into lower-resolution copies of itself. This is then used when the operator loops over all of the pixels and computes each pixel's scale factor.

⟨*HighContrastOp Method Definitions*⟩≡
```
void HighContrastOp::Map(const Float *y, int xRes, int yRes,
        Float maxDisplayY, Float *scale) const {
    ⟨Find minimum and maximum image luminances⟩
    ⟨Build luminance image pyramid⟩
    ⟨Apply high contrast tone mapping operator⟩
}
```

HighContrastOp 315
HighContrastOp::C() 317

⟨*Find minimum and maximum image luminances*⟩≡
```
Float minY = y[0], maxY = y[0];
for (int i = 0; i < xRes * yRes; ++i) {
    minY = min(minY, y[i]);
    maxY = max(maxY, y[i]);
}
Float CYmin = C(minY), CYmax = C(maxY);
```

Most previous approaches to computing local adaptation luminance used a blurred version of the original image, though this led to the halo artifact described previously. The insight behind the approach implemented here is that adaptation luminance shouldn't be based on a constant-sized region of luminances around the pixel (x,y), but should be based on a varying area: as long as the luminance is locally roughly constant, the area can be expanded until a significant change in luminance is reached. This gives us the best of both worlds: when luminance is changing slowly, we compute adaptation luminance over a larger area, giving smooth variation of adaptation luminance when we are far from high contrast features. When contrast is quickly changing, however, we detect this and avoid artifacts by computing a more local adaptation luminance.

A standard technique from image processing is to define the *local contrast* $lc(x,y)$ of a pixel as the magnitude of the difference between that pixel's value and its value in two blurred versions of the image, one blurred with twice a wide a filter than the other

$$lc(s,x,y) = \frac{B_s(x,y) - B_{2s}(x,y)}{B_s(x,y)}.$$

Here s is the filter width used for blurring the image, expressed in pixels and $B_s(x,y)$ is pixel (x,y)'s value in the blurred image. We would like to find the smallest local extent around each pixel (x,y) of radius s such that $|lc(s,x,y)|$ is less than some constant value–when it becomes greater than that value, we have passed the amount of acceptable local contrast. Having found such an s, adaptation luminance is computed by

$$Y^a(x,y) = B_s(x,y),$$

thus fulfilling the criteria above. The top image in Figure 8.10 shows this operator applied to the St. Peter's Basilica image, while the middle image shows the widths used for computing local adaptation luminance at each pixel, where the brighter the pixel, the wider a region was sampled, and the bottom image shows the local contrast computed at each pixel for $s = 1.5$. Notice how edges where there are large jumps in brightness in the original image are found by the local contrast function.

In order to be able to quickly find the value of pixels in the blurred image $B_s(x,y)$, this operator creates an image pyramid with the `MIPMap` class, which will be described in Section 11.5.2. For the purposes of this section, we will just make use of the fact that it can accurately and efficiently compute values of $B_s(x,y)$ for arbitrary values of s.

```
⟨Build luminance image pyramid⟩≡
  MIPMap<Float> pyramid(xRes, yRes, y);
```

Next, the adaptation luminance is computed for each pixel and the scale factor is computed. Note that it is necessary to convert from discrete to continuous pixel coordinates in `xc` and `yc` to give the correct continuous position for the `MIPMap` lookup.

417 MIPMap

```
⟨Apply high contrast tone mapping operator⟩≡
  for (int y = 0; y < yRes; ++y) {
      Float yc = (Float(y) + .5f) / Float(yRes);
      for (int x = 0; x < xRes; ++x) {
          Float xc = (Float(x) + .5f) / Float(xRes);
          ⟨Compute local adaptation luminance at (x,y)⟩
          ⟨Apply tone mapping based on local adaptation luminance⟩
      }
  }
```

To compute the adaptation luminance, this method looks up the value of the pixel in images with a given blur amount (specifically blurred by `width` and 2×`width`) to compute the value of the local contrast function, `lc`. If it's above the value stored in `maxLocalContrast`, 0.5 (an arbitrary constant, chosen after some experimentation), the adaptation luminance is set as the average of a slightly smaller region around the pixel and the loop terminates for this pixel. Otherwise, the blur radius is increased by one pixel's span–`dwidth`–and a new value of `lc` is computed. This process eventually stops once it reaches a large blur without finding a sufficient amount of contrast–at that point, just using the wide area to set adaptation luminance works well anyway.

⟨*Compute local adaptation luminance at* (x,y)⟩≡

```
Float dwidth = 1.f / Float(max(xRes, yRes));
Float maxWidth = 32.f / Float(max(xRes, yRes));
Float width = dwidth, prevWidth = 0.f;
Float Yadapt;
Float prevlc = 0.f;
const Float maxLocalContrast = .5f;
while (1) {
    ⟨Compute local contrast at (x,y)⟩
    ⟨If maximum contrast is exceeded, compute adaptation luminance⟩
    ⟨Increase search region and prepare to compute contrast again⟩
}
```

The local contrast computation is made trivial with the `MIPMap` image pyramid; the `MIPMap::Lookup()` method applies a filter of given width to the image at the given pixel position.

⟨*Compute local contrast at* (x,y)⟩≡

```
Float b0 = pyramid.Lookup(xc, yc, width, 0.f, 0.f, width);
Float b1 = pyramid.Lookup(xc, yc, 2.f*width, 0.f, 0.f, 2.f*width);
Float lc = fabsf((b0 - b1) / b0);
```

Lerp()	677
MIPMap	417
MIPMap::Lookup()	427

If the local contrast exceeds the maximum allowed contrast, an ad-hoc approximation determines the width s for which $lc(s,x,y) = \text{maxLocalContrast}$. Given the local contrast that was computed the last time through the loop in `prevlc` and the fact that

$$\text{prevlc} < \text{maxLocalContrast} < \text{lc},$$

a value t is found such that linear interpolation t of the way between `prevlc` and `lc` gives exactly `maxLocalContrast`. Under the assumption that contrast varies linearly between the last width at which contrast was computed and the current width, it's easy to compute the width between them where we assume that the contrast constraint was violated.

⟨*If maximum contrast is exceeded, compute adaptation luminance*⟩≡

```
if (lc > maxLocalContrast) {
    Float t = (maxLocalContrast - prevlc) / (lc - prevlc);
    Float w = Lerp(t, prevWidth, width);
    Yadapt = pyramid.Lookup(xc, yc, w, 0.f, 0.f, w);
    break;
}
```

⟨*Increase search region and prepare to compute contrast again*⟩≡

```
prevlc = lc;
prevWidth = width;
width += dwidth;
if (width >= maxWidth) {
    Yadapt = pyramid.Lookup(xc, yc, maxWidth, 0.f, 0.f, maxWidth);
    break;
}
```

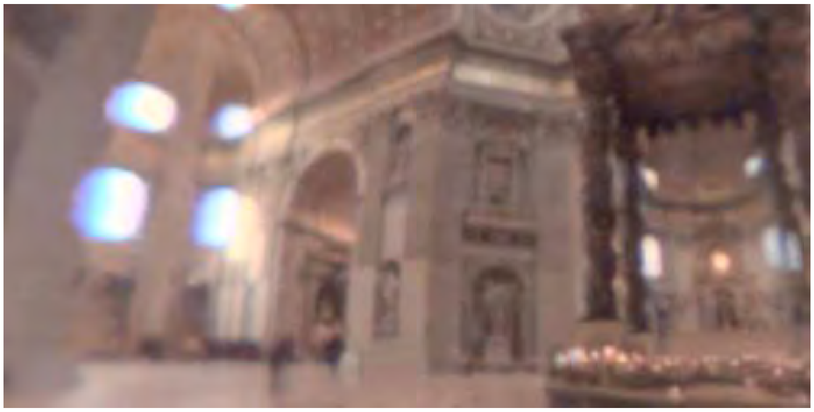


Figure 8.11: The ad-hoc non-linear scale factor works remarkably well on the image of St. Peter's Basilica, preserving detail over a wide range of image luminance values.

Given the tone mapping function $T(Y^a)$, the scale-factor at a given pixel (x,y) is defined by

$$s(x,y) = \frac{T(Y^a(x,y))}{Y^a(x,y)}.$$

As long as $Y^a(x,y)$ is slowly varying over the image, this is a essentially a locally-linear mapping.

318 HighContrastOp::T()
312 MaxWhiteOp

⟨*Apply tone mapping based on local adaptation luminance*⟩≡

```
scale[x + y*xRes] = T(Yadapt, CYmin, CYmax, maxDisplayY) /
    Yadapt;
```

8.3.7 ***ADV***: Spatially-varying non-linear scale

The last tone mapping approach implemented here is not at all grounded in the perception literature, but works remarkably well in practice. It is based on an ad-hoc formula that was chosen with perceptual issues in mind and was introduced by Reinhard et al (Reinhard, Stark, Shirley, and Ferwerda 2002). Its implementation is in the file `tonemaps/nonlinear.cpp`. A spatially varying factor is used to scale each pixel:

$$s(x,y) = \frac{\left(1 + \frac{y(x,y)}{y^2_{\max}}\right)}{1 + y(x,y)}.$$

Note that this operator is not based on luminance Y, but the y component of XYZ color (i.e. the scale of 683 is not included).

This scale factor maps black pixels to zero and the brightest pixels to one. In between, darker pixels require relatively less change in brightness to cause a given change in output pixel value than bright pixels do. This is in tune with properties of the HVS, which has a generally logarithmic response curve, rather than a linear one.

Like the `MaxWhiteOp` operator, it first computes the maximum luminance of all pixels in the image.

```
⟨NonLinearOp Declarations⟩≡
  class NonLinearOp : public ToneMap {
      ⟨NonLinearOp Public Methods⟩
  };
```

The implementation of the operator is straightforward, made messy only by the need to remove the scale of 683 from the luminance values.

```
⟨NonLinearOp Public Methods⟩≡
  void Map(const Float *y, int xRes, int yRes,
          Float maxDisplayY, Float *scale) const {
      ⟨Compute world adaptation luminance, Ywa⟩
      Ywa /= 683.f;
      Float invY2 = 1.f / (Ywa * Ywa);
      for (int i = 0; i < xRes * yRes; ++i) {
          Float ys = y[i] / 683.f;
          scale[i] = maxDisplayY / 683.f *
              (1.f + ys * invY2) / (1.f + ys);
      }
  }
```

ImageInfo	301
ToneMap	310

Figure 8.11 shows this operator in action. It does an excellent job of mapping the St. Peter's image to a reasonable range, preserving contrast over a wide range of brightnesses.

8.4 Device RGB Conversion and Output

After the tone reproduction step, we should have pixels with SPDs that range between zero and one. (Some tone reproduction operators don't guarantee this, or we may have not run the tone mapping step at all, so the values will be clamped to this range later in the pipeline just to be sure.) Given additional information about the particular display device being used, the device-independent XYZ pixel values are converted to device-dependent RGB values. This is another change of spectral basis, where the new basis is determined by the spectral response curves of the red, green, and blue elements of the display device. As before, weights to convert from XYZ to the device RGB can be precomputed. The `ImageInfo` structure holds the weights for the particular display being used.

```
⟨ImageInfo Public Data⟩+≡
  Float rWeight[3], gWeight[3], bWeight[3];
```

By default, these are initialized to the appropriate weights for the RGB primaries as specified by the HDTV standard. This is a good match for most modern display devices.

⟨*ImageInfo Constructor Implementation*⟩+≡

```
rWeight[0] =  3.240479f;
rWeight[1] = -1.537150f;
rWeight[2] = -0.498535f;
gWeight[0] = -0.969256f;
gWeight[1] =  1.875991f;
gWeight[2] =  0.041556f;
bWeight[0] =  0.055648f;
bWeight[1] = -0.204043f;
bWeight[2] =  1.057311f;
```

⟨*Convert image to display RGB*⟩≡

```
Float *rgb = new Float[3*nPix];
⟨Define RGB access macros⟩
for (int i = 0; i < nPix; ++i) {
    Float xyz[3];
    Lout[i].XYZ(xyz);
    R(i) = imageInfo.rWeight[0]*xyz[0] + imageInfo.rWeight[1]*xyz[1] +
        imageInfo.rWeight[2]*xyz[2];
    G(i) = imageInfo.gWeight[0]*xyz[0] + imageInfo.gWeight[1]*xyz[1] +
        imageInfo.gWeight[2]*xyz[2];
    B(i) = imageInfo.bWeight[0]*xyz[0] + imageInfo.bWeight[1]*xyz[1] +
        imageInfo.bWeight[2]*xyz[2];
}
```

301 `ImageInfo`

⟨*Define RGB access macros*⟩≡

```
#define R(i) (rgb[3*(i)])
#define G(i) (rgb[3*(i)+1])
#define B(i) (rgb[3*(i)+2])
```

Unfortunately, there are many colors that modern displays cannot reproduce (for example, saturated oranges and purples); such colors are called *out of gamut*. Out of gamut colors will have values outside the range $[0,1]$ after they have been converted to the display's RGB space. There aren't any completely satisfactory solutions to this problem, given that the display device can't reproduce those colors in the first place; it's mostly a matter of trading off different kinds of error. In `lrt`, we rescale out of gamut colors so that the maximum of the three components is one and the others are scaled proportionally.

⟨*ImageInfo Public Data*⟩+≡

```
bool clampToGamut;
Float gain;
```

⟨*ImageInfo Constructor Implementation*⟩+≡

```
clampToGamut = true;
gain = 1.f;
```

Before performing the this, a user-supplied scale is applied to the pixel values, allowing last minute brightness adjustment; the amount is controlled by the `ImageInfo`'s `gain` value.

⟨*Scale image and handle out-of-gamut RGB values*⟩≡
```
for (int i = 0; i < nPix; ++i) {
    R(i) *= imageInfo.gain;
    G(i) *= imageInfo.gain;
    B(i) *= imageInfo.gain;
}
if (imageInfo.clampToGamut) {
    for (int i = 0; i < nPix; ++i) {
        Float m = max(R(i), max(G(i), B(i)));
        if (m > 1.f) {
            R(i) /= m;
            G(i) /= m;
            B(i) /= m;
        }
    }
}
```

Next it is necessary to adjust the color values for the non-linear change in displayed brightness that displays based on cathode ray tubes (CRTs) exhibit. With these kinds of displays, the displayed brightness doesn't vary linearly with the pixel values: a pixel with value 100 isn't usually twice as bright as a pixel with value 50. (Although newer display technologies like LCD screens don't naturally have non-linear response like this, they are generally built with logic that mimics this characteristic of CRTs. **CHECK THIS**) This non-linear response is generally modeled with a power function

$$d = v^{\gamma},$$

where d is the display brightness, v is the voltage applied to the display's electron gun (which is directly proportional to the pixel value, unless the hardware or operating system also performs gamma correction.), and the *gamma value* γ is generally 2.2. Given a pixel value v, then, computing

$$v' = v^{1/\gamma}$$

and sending v' to the display will cause the displayed brightness to be linear.

⟨*ImageInfo Public Data*⟩+≡
```
Float invGamma;
```

⟨*ImageInfo Constructor Implementation*⟩+≡
```
invGamma = 1.f / 2.2f;
```

⟨*Apply gamma correction to image*⟩≡
```
if (imageInfo.invGamma != 1.f) {
    for (int i = 0; i < nPix; ++i) {
        R(i) = powf(R(i), imageInfo.invGamma);
        G(i) = powf(G(i), imageInfo.invGamma);
        B(i) = powf(B(i), imageInfo.invGamma);
    }
}
```

Given gamma corrected pixel values, we may need to map their values to the range that the display expects (e.g. 0 to 255.) Some image file formats can store floating-point pixel values, so it's not always necessary to perform this step.

⟨*ImageInfo Public Data*⟩+≡

```
bool integerFormat;
int maxDisplayValue;
```

⟨*ImageInfo Constructor Implementation*⟩+≡

```
integerFormat = true;
maxDisplayValue = 255;
```

⟨*Map image to display range*⟩≡

```
if (imageInfo.integerFormat) {
    for (int i = 0; i < nPix; ++i) {
        R(i) *= imageInfo.maxDisplayValue;
        G(i) *= imageInfo.maxDisplayValue;
        B(i) *= imageInfo.maxDisplayValue;
        AlphaOut[i] *= imageInfo.maxDisplayValue;
    }
}
```

679 RandomFloat()

Before converting these pixel values to integer values for the display, it is also helpful to *dither* their values, adding a small random value to each pixel's color component. Introducing this very small amount of noise improves the visual quality of displayed images by making the transition between areas with one pixel to another less well-delineated.

⟨*ImageInfo Public Data*⟩+≡

```
Float ditherAmount;
```

⟨*ImageInfo Constructor Implementation*⟩+≡

```
ditherAmount = 0.5f;
```

⟨*Dither image*⟩≡

```
if (imageInfo.ditherAmount > 0) {
    for (int i = 0; i < nPix; ++i) {
        R(i) += 2.f * imageInfo.ditherAmount *
            (RandomFloat() - .5f);
        G(i) += 2.f * imageInfo.ditherAmount *
            (RandomFloat() - .5f);
        B(i) += 2.f * imageInfo.ditherAmount *
            (RandomFloat() - .5f);
    }
}
```

Finally, the image is saved. The image output routines take care of converting the floating-point pixel values to integers, if appropriate for the file format being used.

⟨*ImageInfo Public Data*⟩+≡

```
string filename;
```

⟨*ImageInfo Constructor Implementation*⟩+≡

```
filename = "lrt.tiff";
```

⟨*Save display image to disk*⟩≡

```
if (imageInfo.integerFormat)
    WriteImage8Bit(imageInfo.filename, rgb, AlphaOut, xPixelCount,
        yPixelCount, 3, xResolution, yResolution, xPixelStart,
        yPixelStart);
else
    WriteImageFloat(imageInfo.filename, rgb, AlphaOut, xPixelCount,
        yPixelCount, 3, xResolution, yResolution, xPixelStart,
        yPixelStart);
delete[] rgb;
```

Further Reading

Porter and Duff's paper on compositing digital images is the classic paper on the uses of images with alpha channels and explains why pre-multiplied alpha is a preferable preresentation for color (Porter and Duff 1984). (The first use of an extra alpha channel in images in graphics dates to Smith and Catmull, however (Smith 1979). See also Wallace's paper for a refinement of Smith and Catmull's approach (Wallace 1981).)

Film::xResolution	294
Film::yResolution	294
ImageFilm::xPixelCount	297
ImageFilm::xPixelStart	297
ImageFilm::yPixelCount	297
ImageFilm::yPixelStart	297

Gamma correction has a a long history in computer graphics; Poynton has written comprehensive FAQs on issues related to color and gamma-correction in computer graphics (Poynton 2002b; Poynton 2002a).

Display issues, mapping to reasonable RGB values, out of gamut colors, ... See Rougeron and Péroche's survey article for discussion and references (Rougeron and Péroche 1998).

Malacara's monograph gives a concise overview of color theory and basic properties of how the the human visual system processes color (Malacara 2002).

Wandell's book?

wazczeki(?sp) and stiles

Glassner has written an article on the under-constrained problem of converting RGB values (e.g. as selected by the user from a display) to a SPD (Glassner 1989).

Tone reproduction for computer graphics became an active area of research around 1993 with the work of Tumblin and Rushmeier (Tumblin and Rushmeier 1993), Chiu et al (Chiu, Herf, Shirley, Swamy, Wang, and Zimmerman 1993), and Ward (Ward 1994a). The bloom technique is based the one described in Chiu et al's paper. The non-linear mapping we presented was developed by Reinhard et al (Reinhard, Stark, Shirley, and Ferwerda 2002).

Poynton gamma FAQ. Catmull color compensation tables paper?

Ward suggested an early technique for compactly storing floating-point image data from realistic image synthesis (Ward 1991) and later developed an extention to the TIFF image file format for accurate high dynamic range color representaiton (Larson 1998).

Tumblin and Rushmeier first introduced a the first tone mapping algorithms to computer graphics and sparked the recent focus on tone reproduction (Tumblin and Rushmeier 1993). Other early work included Chiu et al's spatially-varying scale (Chiu, Herf, Shirley, Swamy, Wang, and Zimmerman 1993), and Ward's contrast-preservation scale (Ward 1994a), which we have implemented in Section 8.3.5.

Since the initial work in tone reproduction, there has been an explosion of research in this area. The survey article of Delvin et al summarizes most of the work in this area through 2002, giving pointers to the original papers (Devlin, Chalmers, Wilkie, and Purgathofer 2002). For background information on properties of the human visual system, Wandell's book on vision is an excellent starting point (Wandell 1995).

The local contrast detection approach we implemented in the `HighContrastOp` is based on Ashikhmin's technique (Ashikhmin 2002). The same local contrast operator was also introduced by Reinhard et al, who developed an operator based on principles from photography, rather than the human visual system (Reinhard, Stark, Shirley, and Ferwerda 2002; Reinhard 2002).

Chiu et al!!!

Simulating the scattering processes inside the human eye can also improve the perceived quality of the image. Nakame et al (Nakamae, Kaneda, Okamoto, and Nishita 1990) and Spencer et al (Spencer, Shirley, Zimmerman, and Greenberg 1995) both modeled glare, where very bright areas in an environment mask dimmer areas next to them. Simulating this in displayed images (e.g. by blurring a bit around very bright areas) can substantially increase the perceived brightness of objects.

294 `Film`
315 `HighContrastOp`

Another interesting application for knowledge of the characteristics of the HVS is perceptually-driven rendering, where less work is done in areas of the image that are known to be visually unimportant and more work is done in areas that are known to be important. Bolin and Meyer paper (Bolin and Meyer 1998) and Ramasubramanian et al (Ramasubramanian, Pattanaik, and Greenberg 1999) both developed rendering algorithms using this approach.

misc: (Tumblin, Hodgins, and Guenter 1999), (Larson, Rushmeier, and Piatko 1997)

boundary preservation (Tumblin and Turk 1999)

Durand and Dorsey (Durand and Dorsey 2002)

vision overview (Ferwerda 2001)

Interactive Durand and Dorsey (Durand and Dorsey 2000)

complex/sophisticated (Pattanaik, Ferwerda, Fairchild, and Greenberg 1998)

adaptation and masking (Ferwerda, Pattanaik, Shirley, and Greenberg 1996) extended Ward's contrast-based method to handle scoptic lighting levels, including reduced color sensitivity and spatial acuity. (Ferwerda, Pattanaik, Shirley, and Greenberg 1997)

Time dependence (Pattanaik, Tumblin, Yee, and Greenberg 2000)

Frankle and McCann 83 retinex paper

Exercises

8.1 Write a new `Film` implementation that opens a window and interactively displays pixel values as the scene is rendered. What sort of modifications (and simplifications) to the `ImageFilm`'s imaging pipeline are necessary to make this film implementation be efficient?

8.2 Image-based rendering is the general name for a set of techniques that use one or more images of a scene to synthesize new images from viewpoints different than the original ones. One such approach is lightfield rendering, where a set of images from a densely-spaced set of positions is used (Levoy and Hanrahan 1996; Gortler, Grzeszczuk, Szeliski, and Cohen 1996). Extend `lrt` to directly generate lightfields of scenes, without requiring that the renderer be run multiple times, once for each camera position. It will probably be necessary to write a specialized `Camera`, `Sampler`, and `Film` to do this. Also, write an interactive lightfield viewer that uses these lightfields to generate new views of the scene.

8.3 Implement a tone reproduction operator for low-light environments (e.g. (Ferwerda, Pattanaik, Shirley, and Greenberg 1996; Ferwerda, Pattanaik, Shirley, and Greenberg 1997)) and compare its results to the tone reproduction operators described in this chapter. (Rather than re-rendering a complex scene multiple times in order to experiment with different tone mapping operators, a faster approach is to start by rendering the scene once and storing the image in a floating-point image format. Then, set up a new scene up with an orthographic camera pointing at a texture-mapped quadrilateral facing the camera with the image of the scene mapped onto it. If this quadrilateral is illuminated with a distant light source, then a rendered image of it will look almost the same as the original scene. Because the texture has the full floating-point representation of the scene's radiance values, appropriate input is available for the tone reproduction algorithm.)

8.4 Ward style histogram-based tone repro stuff: don't waste dynamic range in parts of the histogram where not many image samples lie

8.5 The bloom effect implemented in this chapter works well, but is completely ad-hoc. Read Spencer et al's paper, which describes glare and bloom algorithms based on a more detailed model of how light scatters in the human eye and implement their algorithms as a tone reproduction operator in `lrt` (Spencer, Shirley, Zimmerman, and Greenberg 1995).

8.6 darkroom: interactive dodge and burn

8.7 Deep framebuffers: rather than just storing spectral values and an alpha channel in an image, it's often useful to store additional information about the objects in the scene that were visible at each pixel. See for example the SIGGRAPH papers by Perlin (Perlin 1985) and Saito and Takahashi (Saito and Takahashi 1990). For example, if the 3D position, surface normal, and BRDF information about the object at a pixel is stored, then the scene can be efficiently re-rendered after moving the light sources (Gershbein and Hanrahan 2000). If each sample stores information about all of the objects visible along its camera ray, rather than just the first one, new images from shifted viewpoints can be re-rendered (Shade, Gortler, wei He, and Szeliski 1998). Investigate representations for deep framebuffers and algorithms that use them; extend `lrt` to support rendering and saving more complex images and develop tools that operate on them.

9. Reflection Models

This chapter defines a set of classes for describing light scattering at surfaces. Recall that the bidirectional reflectance distribution function (BRDF) abstraction was introduced in Section 5.4.1 to describe light scattering at surfaces, the BTDF describes transmission at a surface, and the BSDF encompasses both of these effects. In this chapter, we will start by defining a generic interface to these surface reflection and transmission functions. Scattering from realistic surfaces is often best described as a mixture of multiple BRDFs and BTDFs; in Chapter 10, we will introduce a BSDF object that combines multiple BRDFs and BTDFs to represent overall scattering from the surface. This chapter also sidesteps the issue of reflection and transmission properties that vary over the surface; the texture classes of Chapter 11 will address that problem.

Specific reflection models come from a number of sources:

1. Measured data: reflection distribution properties of a number of real-world surfaces have been measured. These data may be presented in tabular form, or as coefficients for a set of basis functions.

2. Phenomenological: equations that attempt to describe the qualitative properties of real-world surfaces can be remarkably effective at mimicking them. These BSDFs can be particularly easy to use, since they tend to have intuitive parameters (e.g. “roughness”) that modify their behavior. The majority of reflection functions used in computer graphics fall into this category.

3. Simulation: if low-level information is known about the composition of a surface (e.g. that a paint is comprised of colored particles of some average size suspended in a medium, or that a particular fabric is comprised from two types of thread, each with known reflectance properties), light scattering

from micro-geometry with these reflectance properties can be simulated to generate reflection data. This simulation can either be done during rendering or as a pre-process, after which it may be fit to a set of basis functions for use at rendering-time.

4. Geometric optics: as with simulation approaches, if the surface's lower-level scattering and geometric properties are known, then models can often be derived directly from these descriptions. This approach is much more tractable if geometric optics is used to model light's interaction with the surface–this is a much simpler model, ignoring wave effects like polarization, etc.

5. Physical (wave) optics: some reflection models have been derived using a detailed model of light, treating it as a wave and computing the solution to Maxwell's equations to find how it scatters from a surface with known properties. These models tend to be computationally expensive however, and usually aren't appreciably more accurate than models based on geometric optics.

In this chapter, we will define implementations of reflection models based on measured data, phenomenological models, and geometric optics. The further reading section at the end of this chapter gives pointers to a variety of other models.

Before we define the reflection and transmission interfaces and classes, a brief review of how they fit into the overall system and are used in the process of computing outgoing radiance at a point being shaded is in order[1]. The integrator classes, defined in Chapter 16, are responsible for determining which surface is first visible along a ray and computing the scattered radiance at that point. One the hit point is found, the integrator runs the surface shader that was bound to the surface. The surface shader is a short procedure that is responsible for deciding what the BSDF is at a particular point on the surface (see Chapter 10); it returns a BSDF object that holds BRDFs and BTDFs for that point. The integrators then use the BSDF to compute the scattered light at the point, based on the incoming illumination from the light sources in the scene.

9.0.1 Basic terminology

In order to be able to compare the visual appearance of different reflection models, we will introduce some basic terminology for describing reflection from surfaces. Although reflection from real surfaces often won't cleanly fit into the categories below, they offers a general framework to start out with.

Reflection from surfaces can be split into four broad categories: *diffuse*, *glossy specular*, *perfect specular*, and *retro-reflective* (Figure 9.1). Most real surfaces exhibit reflection that is a mixture of these four types. Diffuse surfaces scatter light equally in all directions. Although a perfectly diffuse surface isn't physically plausible, examples of near-diffuse surfaces include dull chalkboards and matte paint. Glossy specular surfaces (for example, gloss paint or plastic) scatter light preferentially in a set of reflected directions–they show blurry reflections of other objects. Specular surfaces reflect incident light in a single outgoing direction. Mirrors and glass are examples of specular surfaces. Finally, retro-reflective surfaces

[1] The reader may wish to revisit Chapter 1 at this point.

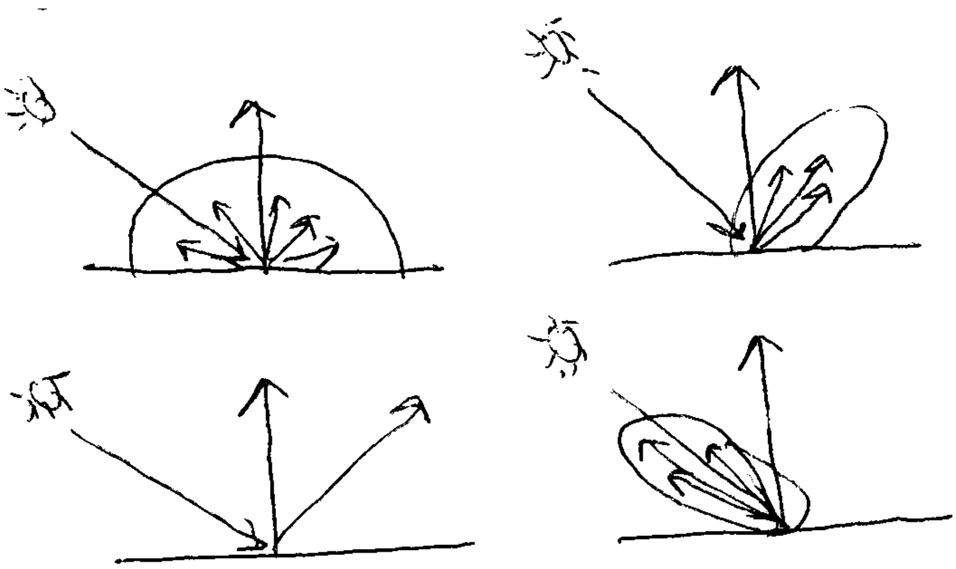


Figure 9.1: Four main types of reflection: diffuse, glossy specular, perfect specular, retro-reflection.

mainly scatter light back along the incident direction; fabrics such as velvet and the Earth's moon are two examples of retro-reflective surfaces.

Given a particular general type of reflection, the reflectance distribution function may be *isotropic* or *anisotropic*. Most objects are isotropic: if you choose a point on the surface and rotate it around its normal axis around that point, the amount of light reflected doesn't change. In contrast, anisotropic materials reflect different amounts of light as you rotate them in this way. Examples include brushed metal, phonographic records, and compact disks.

XXX image showing these differences.

9.0.2 Geometric Setting

Reflection computations in `lrt` are evaluated in a reflection coordinate system where two tangent vectors and the normal vector at the point being shaded are aligned with the *x*, *y*, and *z* axes, respectively (Figure 9.2). All direction vectors passed to and returned from the BRDF and BTDF routines will be defined with respect to this coordinate system. This is a natural coordinate system for implementing reflection computations; doing all BSDF calculations in this coordinate system helps make some of the ideas more clear. It is important to understand this coordinate system in order to understand the BSDF implementations in this chapter.

This coordinate system also gives us a frame for expressing directions in spherical coordinates (θ, ϕ); the angle θ is measured from the given direction to the *z* axis, and ϕ is the angle formed with the *x* axis after projection of the direction onto the *xy* plane. Given a direction vector ω in this coordinate system, it is easy to compute quantities like the cosine of the angle that it forms with the normal direction:

$$\cos\theta = (\mathbf{n} \cdot \omega) = ((0,0,1) \cdot \omega) = \omega_z.$$

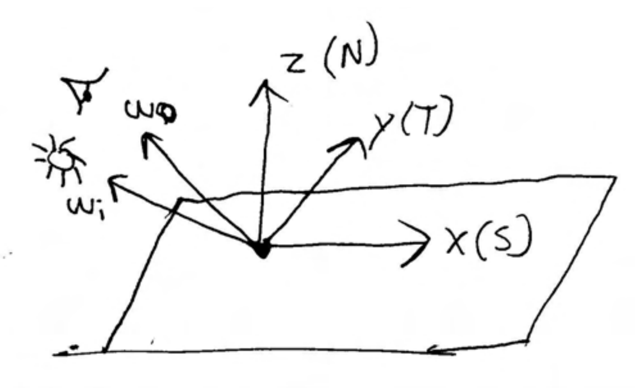


Figure 9.2: basic BSDF interface setting: the shading coordinate system is defined by the orthonormal basis vectors $(\mathbf{s}, \mathbf{t}, \mathbf{n})$. We will orient these vectors such that they lie along the x, y, and z axes in this coordinate system. Direction vectors ω in world space are transformed into the shading coordinate system before any of the BRDF or BTDF methods are called.

We will provide a utility function to compute this value; this mostly serves to make the intent of BSDF code more clear.

Vector 27

⟨*BSDF Inline Functions*⟩≡

```
inline Float CosTheta(const Vector &w) { return w.z; }
```

Some additional algebra shows that the absolute value of $\sin\theta$ is

$$|\sin\theta| = \sqrt{1-\omega_z^2}$$

⟨*BSDF Inline Functions*⟩+≡

```
inline Float AbsSinTheta(const Vector &w) {
    return sqrtf(max(0.f, 1.f - w.z*w.z));
}
```

The signed value of $\sin\theta$ can be determined by examining the sign of ω_z.

⟨*BSDF Inline Functions*⟩+≡

```
inline Float SinTheta(const Vector &w) {
    Float s = sqrtf(max(0.f, 1.f - w.z*w.z));
    if (w.z < 0.f) s *= -1.f;
    return s;
}
```

The value of $\sin^2\theta$ can be computed more efficiently using the identity $\omega_x^2 + \mathit{YCOMP}\omega^2 + \mathit{ZCOMP}\omega^2 = 1$:

⟨*BSDF Inline Functions*⟩+≡

```
inline Float SinTheta2(const Vector &w) {
    return w.x*w.x + w.y*w.y;
}
```

We can similarly use the shading coordinate system to simplify the calculations for the sine and cosine of the ϕ angle (Figure 9.3). In the plane of the point being

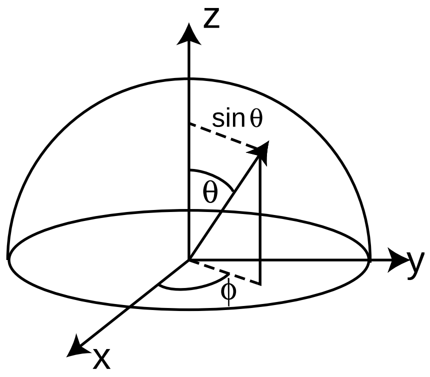


Figure 9.3: As was done for the `SpecularTransmission` BTDF, the $\sin\theta$ term is found by computing the length of the dashed line, which is the magnitude of the xy components of the vector. The $\sin\phi$ and $\cos\phi$ terms can be computed using the circular coordinate equations $x = r\cos\phi$ and $y = r\sin\phi$, where r, the length of the dashed line, was already computed for $\sin\theta$.

shaded, the vector ω has coordinates (x, y), which are given by $r\cos\phi$ and $r\sin\phi$, respectively. The radius r is just $\sin\theta$, so

$$\begin{aligned}\cos\phi &= \frac{x}{r} = \frac{x}{\sin\theta}\\ \sin\phi &= \frac{y}{r} = \frac{y}{\sin\theta}.\end{aligned}$$

```
334 BxDF
332 SinTheta()
348 SpecularTransmission
 27 Vector
```

⟨*BSDF Inline Functions*⟩+≡

```
inline Float CosPhi(const Vector &w) {
    return w.x / SinTheta(w);
}
inline Float SinPhi(const Vector &w) {
    return w.y / SinTheta(w);
}
```

Another convention we will follow is that the incident light direction, ω_i, and the outgoing viewing direction, ω_o, will be normalized and outward facing after being transformed into the local coordinate system at the surface. By convention, the surface normal **n** always points to the "outside" of the object, which helps us determine if light is entering or exiting transmissive objects: if the incident light direction ω_i is in the same hemisphere as **n**, then light is entering; otherwise it is exiting.

Note that the local coordinate system used for shading may not be exactly the same as the coordinate system returned by the `Shape::Intersect()` routines from Chapter 3; they can be modified between intersection and shading to achieve effects like bump-mapping. See Chapter 10 for examples of this kind of modification.

The `BxDF` implementations do not need to concern themselves with whether ω_i and ω_o lie in the same hemisphere. For example, although a reflective `BxDF` should, in principle, return zero reflection if the incident direction is above the surface and the outgoing direction is below, we will expect the BxDF to compute and return the amount of light reflected as if they were in the same hemisphere. Higher-level code in lrt will ensure that only reflective or transmissive scattering routines are evaluated as appropriate. (The need for this convention will be explained in Section 10.1.)

9.1 Basic Interface

We will first define the interface for the individual BRDF and BTDF functions. BRDFs and BTDFs share a common base-class, BxDF, which defines the basic interface that they adhere to. Because both have the exact same interface, this reduces repeated code and allows some parts of the system to work with BxDFs generically without distinguishing between BRDFs and BTDFs.

We assume that light in different wavelengths is *decoupled*; energy at one wavelength will not be reflected at a different wavelength. Thus, fluorescent materials are not supported.

⟨*BxDF Declarations*⟩≡

```
class BxDF {
public:
    ⟨BxDF Interface⟩
    ⟨BxDF Public Data⟩
};
```

BSDF 370

The BSDF class, which will be introduced in Section 10.1, holds a collection of BxDF objects that together describe the scattering at a point on a surface. Although we are hiding the implementation details nature of the BxDF behind a common interface for reflective and transmissive materials, some of the light transport algorithms in Chapter 16, will need to distinguish between these two types. Therefore, all BxDFs have a BxDF::type member that holds the bitwise and of flags from BxDFType. For each BxDF, the flags should have exactly one of BSDF_REFLECTIVE or BSDF_TRANSMISSIVE set, and exactly one of the diffuse, glossy, and specular flags.

⟨*BSDF Declarations*⟩≡

```
enum BxDFType {
    BSDF_REFLECTION   = 1<<0,
    BSDF_TRANSMISSION = 1<<1,
    BSDF_DIFFUSE      = 1<<2,
    BSDF_GLOSSY       = 1<<3,
    BSDF_SPECULAR     = 1<<4,
    BSDF_ALL_TYPES = BSDF_DIFFUSE | BSDF_GLOSSY | BSDF_SPECULAR,
    BSDF_ALL_REFLECTION = BSDF_REFLECTION | BSDF_ALL_TYPES,
    BSDF_ALL_TRANSMISSION = BSDF_TRANSMISSION | BSDF_ALL_TYPES,
    BSDF_ALL          = BSDF_ALL_REFLECTION | BSDF_ALL_TRANSMISSION,
};
```

⟨*BxDF Public Data*⟩≡

```
const BxDFType type;
```

⟨*BxDF Interface*⟩+≡

```
BxDF(BxDFType t) : type(t) { }
```

The key method that BxDFs provide is the BxDF::f() method that returns the value of the distribution function for the given pair of directions.

⟨*BxDF Interface*⟩+≡

```
virtual Spectrum f(const Vector &wo, const Vector &wi) const = 0;
```

Not all BxDFs can be easily evaluated in this manner. For example, perfectly specular objects like mirror, glass, or water only scatter light from a single incident direction in a single outgoing direction. Such BxDFs are best described with *delta distributions* that are zero except for the single direction where light is scattered.

These BxDFs need special handling in lrt, so we will also provide method: BxDF::Sample_f(). This method is used both for handling scattering that is described by delta functions as well as for randomly sampling directions from BxDFs that scatter light along multiple directions–this second application will be explained in Chapter 14.

BxDF::Sample_f() computes the direction of incident light ω_i given an outgoing direction ω_o and returns the value of the BxDF for the given pair of directions. For delta distributions, it is necessary for the BxDF to choose the direction, since the caller has no chance of generating the appropriate ω_i direction.[2] When this method is used for sampling random directions of non-specular BxDFs for Monte Carlo integration, two random numbers from zero to one, u1 and u2, are used to help choose the direction. The pdf parameter is used to return the value of the probability density function for the sampled direction. These parameters aren't needed for delta distribution BxDFs, so they will be explained in the Monte Carlo chapter, when we provide implementations of this method for non-specular reflection functions.

334	BxDF
334	BxDFType
181	Spectrum
27	Vector

⟨*BxDF Interface*⟩+≡

```
virtual Spectrum Sample_f(const Vector &wo, Vector *wi,
    Float u1, Float u2, Float *pdf) const;
```

9.1.1 Reflectance

It can be useful to take the aggregate behavior of the 4D BxDF, defined as a function over pairs of directions, and reduce it to a 2D function over a single direction, or even to a constant value that describes its overall scattering behavior.

The *hemispherical-directional reflectance* is a 2D function that gives the total reflection in a given direction due to constant illumination over the hemisphere, or, equivalently, total reflection over the hemisphere due to light from a given direction[3]. It is defined as:

$$\rho_{dh}(\omega_o) = \frac{1}{\pi}\int_{\mathcal{H}^2(\mathbf{n})} f_r(p, \omega_o, \omega)|cos\theta| d\omega.$$

[2] Delta functions in BxDFs have some additional subtle implications for light transport algorithms. Section 16.2.4 describes the issues in detail.

[3] The fact that these two quantities are equal is due to the reciprocity of real-world reflection functions. If we are using a non-physically based BRDF that does not obey reciprocity, this assumption, along with many others in lrt, breaks down.

The BxDF::rho() method computes the reflectance function $\rho_d h$:
explain the samples stuff... Yes, what is this? —Greg

⟨*BxDF Interface*⟩+≡

```
virtual Spectrum rho(const Vector &wo, int nSamples = 16,
    Float *samples = NULL) const;
```

For some BxDFs, this integral can be computed analytically. If not, we will provide a method to estimate the value of ρ_{dh} in Section 15.3.4 the chapter on Monte Carlo integration.

The *hemispherical-hemispherical reflectance* of a surface, denoted by ρ_{hh}, is a constant spectral value that gives the fraction of incident light reflected by a surface when the incident light is the same from all directions. It is:

$$\rho_{hh} = \frac{1}{\pi} \int_{\mathcal{H}^2(\mathbf{n})} \int_{\mathcal{H}^2(\mathbf{n})} f_r(\mathrm{p}, \omega_o, \omega_i) \, |\cos\theta_o \cos\theta_i| \mathrm{d}\omega_o \mathrm{d}\omega_i$$

We overload the BxDF::rho method to compute ρ_{hh} if no direction ω_o is provided.

⟨*BxDF Interface*⟩+≡

```
virtual Spectrum rho(int nSamples = 16, Float *samples = NULL) const;
```

BSDF_REFLECTION	334
BSDF_TRANSMISSION	334
BxDF	334
BxDFType	334
Spectrum	181
Vector	27

9.1.2 BRDF→BTDF Adapter

It's handy to be define an adapter class that lets us re-use an already-defined BRDF class as a BTDF, especially for phenomenological models that may be equally plausible models of transmission. The BRDFToBTDF class takes a BRDF pointer in the constructor and uses it to implement the BTDF interface. In particular, this means forwarding method calls on to the BRDF, possible switching the ω_i direction to lie in the same hemisphere as ω_o, as the BRDF expects.

⟨*BxDF Declarations*⟩+≡

```
class BRDFToBTDF : public BxDF {
public:
    ⟨BRDFToBTDF Public Methods⟩
private:
    BxDF *brdf;
};
```

The constructor for our adapter class is simple. It simply switches the reflection and transmission flags of the BxDF::type member.

⟨*BRDFToBTDF Public Methods*⟩≡

```
BRDFToBTDF(BxDF *b)
    : BxDF(BxDFType(b->type ^ (BSDF_REFLECTION | BSDF_TRANSMISSION))) {
    brdf = b;
}
```

The adapter will need to convert an incoming vector to the corresponding vector in the opposite hemisphere. Fortunately, this is a simple calculation in the shading coordinate system; we simply negate the vector's *z* coordinate.

⟨*BRDFToBTDF Public Methods*⟩+≡

```
static Vector otherHemisphere(const Vector &w) {
    return Vector(w.x, w.y, -w.z);
}
```

Finally, evaluating the adapted BRDF or the corresponding reflectances ρ_{dh} and ρ_{hh} is just a matter of reflecting the given ray into the other hemisphere before calling the BTDF's corresponding method.

⟨*BRDFToBTDF Public Methods*⟩+≡

```
Spectrum rho(const Vector &w, int nSamples,
        Float *samples) const {
    return brdf->rho(otherHemisphere(w), nSamples, samples);
}
Spectrum rho(int nSamples, Float *samples) const {
    return brdf->rho(nSamples, samples);
}
```

⟨*BxDF Method Definitions*⟩≡

```
Spectrum BRDFToBTDF::f(const Vector &wo, const Vector &wi) const {
    return brdf->f(wo, otherHemisphere(wi));
}
```

```
336 BRDFToBTDF
336 BxDF::rho()
181 Spectrum
 27 Vector
```

9.2 Specular Reflection and Transmission

The behavior of light at perfectly smooth surfaces is relatively easy to characterize analytically in both the physical and geometric optics models. These surfaces exhibit *perfect specular reflection and transmission* of incident light; for a given ω_i direction, all light is scattered in a single outgoing direction. For specular reflection, this direction is the outgoing direction that makes the same angle with the normal that the incoming direction does (See Figure 9.4).

For transmission, this direction is given by *Snell's law*, which relates the angle θ_t between the transmitted direction and the flipped surface normal $-\mathbf{n}$ to the angle θ_i between the incident ray and the surface normal $\mathbf{n}$. (One of the exercises at the end of this chapter is to derive Snell's law using Fermat's Principle from optics.) Snell's law is dependent on the *index of refraction* for the medium the incident ray is in and the index of refraction of the medium it is entering. The index of refraction describes how much more slowly light travels in a particular medium compared to the speed of light in a vacuum. We will use the "eta" symbol, η, to denote the index of refraction. Snell's law is:

$$\eta_i \sin\theta_i = \eta_t \sin\theta_t.$$

In general, the index of refraction η varies by the wavelength of light. Thus, incident light generally scatters in multiple directions at the boundary between two different media, an effect known as *dispersion*: this effect can be seen when incident white light is split into spectral components, by a prism. Common practice in graphics is to ignore this wavelength dependence, since this effect is generally not crucial for visual accuracy, and doing so simplifies the light transport calculations substantially. Alternatively, the paths of multiple beams of light (e.g. at a series of

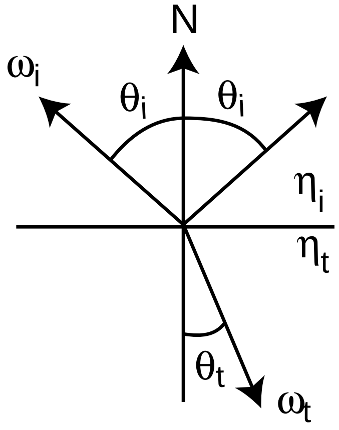


Figure 9.4: Basic setting for specular reflection and transmission. **Why isn't w_r labeled in this diagram?** The reflected direction ω_r is the direction ω_i opposite the outgoing direction ω_o that makes the same angle θ_i with the surface normal as the incident ray. The transmitted direction makes an angle θ_t with the negated surface normal, where θ_t is given by Snell's law, which depends on the indices of refraction of the incident and transmitted media, η_o and η_t, respectively.

discrete wavelengths) can be tracked through the environment which a dispersive object is found. The further reading section at the end of this chapter has pointers to previous research in this area.

9.2.1 Fresnel reflectance

In addition to knowing the reflected and transmitted directions, we also need to compute the fraction of incoming light that is reflected or transmitted. In simple raytracers, these fractions are typically just given as "reflectivity" or "transmissiveness" values, which are uniform over the entire surface. For physical reflection or refraction, however, these effects are view dependent and cannot be captured simply by constant per-surface scaling amounts. The *Fresnel equations* describe the amount of light reflected from a surface; they are the solution to Maxwell's equations at smooth surfaces. There are two sets of Fresnel equations; one for *dielectric media*–objects that don't conduct electricity, like glass–and one for *conductors*, like metals.

For each of these cases, the Fresnel equations have two forms, depending on the polarization of the incident light. Properly accounting for polarization in rendering is a complex task; in `lrt` we will make the common assumption that light is *circularly polarized*; i.e., that it is randomly oriented with respect to the light wave. With this simplifying assumption, the Fresnel reflectance is simply the sum of the squares of the parallel and perpendicular polarization terms.

To compute the Fresnel reflectance of a dielectric, we need to know the indices of refraction for the two media (Figure 9.4). Table 9.1 has the indices of refraction for a number of dielectric materials.

The Fresnel formulae for dielectrics are **Nolan says that these are approximations too. It would be nice to say something about the theory here, and why we need approximations. He said something about the solutions being highly**

Medium	Index of refraction η
Vacuum	1.0
Air at sea level	1.00029
Ice	1.31
Water (20° C)	1.333
Fused Quartz	1.46
Glass	1.5 - 1.6
Sapphire	1.77
Diamond	2.42

Table 9.1: Indices of refraction for a variety of objects, giving the ratio of the speed of light in a vacuum to the speed of light in the medium. Though this is a generally a wavelength-dependent quantity, these values are just averages over the visible wavelengths.

non-linear, and these approximations drop the higher order terms. Can we get some learning on this?:

$$\begin{aligned} r_{\|} &= \frac{\eta_t(N\cdot\omega_o) - \eta_o(N\cdot\omega_t)}{\eta_t(N\cdot\omega_o) + \eta_o(N\cdot\omega_t)} \\ r_{\perp} &= \frac{\eta_o(N\cdot\omega_o) - \eta_t(N\cdot\omega_t)}{\eta_o(N\cdot\omega_o) + \eta_t(N\cdot\omega_t)} \end{aligned}$$

181 Spectrum

where $r_{\|}$ is the Fresnel reflectance for parallel polarized light and $r_{\perp}$ is the reflectance for perpendicular polarized light. η_o and η_t are the indices of refraction for the incident and transmitted media, and ω_o and ω_t are the incident and transmitted directions, where ω_t was computed with Snell's law. For light with circular polarization (the usual assumption in graphics),

$$r = \frac{1}{2}(r_{\|}^2 + r_{\perp}^2).$$

The function `FrDiel()` computes the Fresnel reflection formula for circularly polarized light. Note that the quantities $\mathbf{n}\cdot\omega_o$ and $\mathbf{n}\cdot\omega_t$ are passed in the variables `coso` and `cost`.

⟨*BxDF Utility Functions*⟩≡

```
Spectrum FrDiel(Float coso, Float cost, const Spectrum &etao,
        const Spectrum &etat) {
        Spectrum Rparl = ((etat * coso) - (etao * cost)) /
                         ((etat * coso) + (etao * cost));
        Spectrum Rperp = ((etao * coso) - (etat * cost)) /
                         ((etao * coso) + (etat * cost));
    return (Rparl*Rparl + Rperp*Rperp) / 2.f;
}
```

Due to conservation of energy, the energy transmitted by a dielectric is $1 - F_r$, if F_r is the Fresnel reflectance.

Conductors don't transmit light. Some of the incident light is absorbed by the material and turned into heat; the Fresnel formula for conductors tells how much is reflected. In addition to depending on $\cos\theta_o$, it depends on η, the index of

Object	η	k
Gold	0.37	2.82
Silver	0.177	3.638
Copper	0.617	2.63
Steel	2.485	3.433

Table 9.2: Representative measured values of η and k for a few conductors (data from Hall.)

refraction of the conductor, and k, its *absorption coefficient*. Values for η and k for a few conductors are given in Figure 9.2. As with the index of refraction for dielectrics, these quantities are wavelength-dependent, though are represented as averages here.

again, explain dropping higher-order terms; why are we approximating here? A widely used approximation to the Fresnel reflectance for conductors is:

$$r_{\parallel}^2 = \frac{(\eta^2+k^2)(N\cdot\omega_i)^2 - 2\eta(N\cdot\omega_i)+1}{(\eta^2+k^2)(N\cdot\omega_i)^2 + 2\eta(N\cdot\omega_i)+1} \tag{9.2.1}$$

$$r_{\perp}^2 = \frac{(\eta^2+k^2) - 2\eta(N\cdot\omega_i)+(N\cdot\omega_i)^2}{(\eta^2+k^2) + 2\eta(N\cdot\omega_i)+(N\cdot\omega_i)^2} \tag{9.2.2}$$

Spectrum 181

⟨*BxDF Utility Functions*⟩+≡

```
Spectrum FrCond(Float coso, const Spectrum &eta, const Spectrum &k) {
    Spectrum tmp = (eta*eta + k*k) * coso*coso;
    Spectrum Rparl2 = (tmp - (2.f * eta * coso) + 1) /
        (tmp + (2.f * eta * coso) + 1);
    Spectrum tmp_f = eta*eta + k*k;
    Spectrum Rperp2 = (tmp_f - (2.f * eta * coso) + coso*coso) /
       (tmp_f + (2.f * eta * coso) + coso*coso);
    return (Rparl2 + Rperp2) / 2.f;
}
```

For many conductors, values for η and/or k aren't known–much less work has gone into measuring these values for conductors than for dielectrics. Two approximation methods have been applied in graphics to find plausible values for these quantities. Both assume that the reflectance of the object has been measured at normal incidence: the viewer and the light are both looking directly down on the surface. By fixing the value of either η or k and substituting into the Fresnel conductor formula, we can approximate the other value so that the proper reflectance is computed for normal incidence.

The first method computes an approximate value of η, assuming that the absorption coefficient is equal to zero. If $k = 0$ (assumed), and $\mathbf{n}\cdot\omega_i = 1$ (normal incidence), then equations 9.2.1 and 9.2.2 both simplify to:

$$r_{\parallel}^2 = r_{\perp}^2 = \frac{\eta^2-2\eta-1}{\eta^2+2\eta+1} = \left(\frac{\eta-1}{\eta+1}\right)^2$$

Since the Fresnel reflectance I is given at normal incidence, we can solve for η, giving:

$$\eta = \frac{1+\sqrt{I}}{1-\sqrt{I}}$$

⟨*BxDF Utility Functions*⟩+≡

```
Spectrum FresnelApproxEta(const Spectrum &i) {
    Spectrum intensity = i.Clamp(0.f, .999f);
    return (Spectrum(1.) + intensity.Sqrt()) /
        (Spectrum(1.) - intensity.Sqrt());
}
```

We can perform the same process to approximate the absorption coefficient k, assuming that $\eta = 1$. In this case, the fresnel equations simplify to:

$$r_{\parallel}^2 = r_{\perp}^2 = \frac{k^2}{k^2+4}$$

And we can easily solve for k:

$$k = 2\sqrt{\frac{q}{1-q}}$$

⟨*BxDF Utility Functions*⟩+≡

```
Spectrum FresnelApproxK(const Spectrum &i) {
    Spectrum intensity = i.Clamp(0.f, .999f);
    return 2.f * (intensity / (Spectrum(1.) - intensity)).Sqrt();
}
```

342 FresnelDielectric
181 Spectrum
183 Spectrum::Clamp()
182 Spectrum::Sqrt()

For convenience, we will define an abstract `Fresnel` class that provides an interface for computing Fresnel reflection coefficients. The `FresnelConductor` and `FresnelDielectric` realizations of this interface help simplify the implementation of subsequent BRDFs that may need to support both forms.

⟨*BxDF Declarations*⟩+≡

```
class Fresnel {
public:
    ⟨Fresnel Interface⟩
};
```

The only function provided by the `Fresnel` interface is the `Fresnel::Evaluate()` method. Given the cosine of the angle made by the incoming direction and the surface normal, it returns the amount of light reflected by the surface.

⟨*Fresnel Interface*⟩+≡

```
virtual Spectrum Evaluate(Float cosi) const = 0;
```

Fresnel Conductors

Our first implementation of this interface, `FresnelConductor` is (as the name implies) for conductors.

⟨*BxDF Declarations*⟩+≡

```
class FresnelConductor : public Fresnel {
public:
    ⟨FresnelConductor Public Methods⟩
private:
    ⟨FresnelConductor Private Data⟩
};
```

The `FresnelConductor` constructor simply stores the index of refraction η and the absorption coefficient k.

⟨*FresnelConductor Public Methods*⟩+≡

```
FresnelConductor(const Spectrum &e, const Spectrum &kk)
    : eta(e), k(kk) {
}
```

⟨*FresnelConductor Private Data*⟩≡

```
Spectrum eta, k;
```

The evaluation routine for `FresnelConductor` is simple; it just calls the `FrCond` function defined above.

`FrCond()`	340
`Fresnel`	341
`Spectrum`	181

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum FresnelConductor::Evaluate(Float cosi) const {
    return FrCond(fabsf(cosi), eta, k);
}
```

Fresnel Dielectrics

⟨*BxDF Declarations*⟩+≡

```
class FresnelDielectric : public Fresnel {
public:
    ⟨FresnelDielectric Public Methods⟩
private:
    ⟨FresnelDielectric Private Data⟩
};
```

The constructor for `FresnelDielectric` simply stores the indices of refraction on the two sides η_o and η_t.

⟨*FresnelDielectric Public Methods*⟩+≡

```
FresnelDielectric(Float eo, Float et) {
    eta_o = eo;
    eta_t = et;
}
```

⟨*FresnelDielectric Private Data*⟩≡

```
Float eta_o, eta_t;
```

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum FresnelDielectric::Evaluate(Float coso) const {
    ⟨Compute Fresnel reflectance for dielectric⟩
}
```

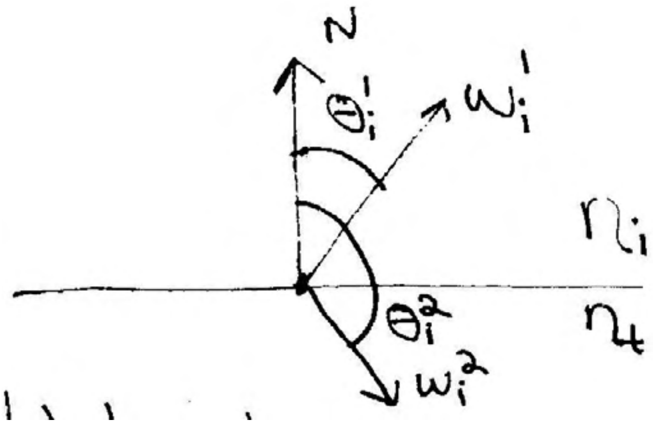


Figure 9.5: The cosine of the angle θ that a direction ω_i makes with the surface normal tells us if the direction is pointing outside the surface (in the same hemisphere as the normal), or inside the surface. In the standard BxDF coordinate system, this test just requires checking the z component of the direction vector. Here, $\omega_i{}^1$ is in the upper hemisphere, with a positive-valued cosine, while $\omega_i{}^2$ is in the lower hemisphere.

677 Clamp()
339 FrDiel()

Evaluating the Fresnel formula for dielectric media is a bit more complicated than for conductors. First, we need to determine if the incident direction is on the outside of the medium or in the inside of it to know how to interpret the two indices of refraction. Next, we apply Snell's law to compute the sine of the angle the transmitted direction makes with the surface normal. We can then compute the cosine of this angle using the identity $\sin^2 x + \cos^2 x = 1$.

⟨*Compute Fresnel reflectance for dielectric*⟩≡

```
coso = Clamp(coso, -1.f, 1.f);
⟨Compute indices of refraction for dielectric⟩
⟨Compute sint using Snell's law⟩
if (sint > 1.) {
    ⟨Handle total internal reflection⟩
}
else {
    Float cost = sqrtf(max(0.f, 1.f - sint*sint));
    return FrDiel(fabsf(coso), cost, eo, et);
}
```

The sign of the cosine of the incident direction indicates on which side of the medium the direction lies; see Figure 9.5. If the cosine is between 0 and 1, the direction is on the outside, and if it is between -1 and 0, it's on the inside. We set the variables `eo` and `et` such that `eo` has the index of refraction of the medium the incident ray is in.

⟨*Compute indices of refraction for dielectric*⟩≡

```
bool entering = coso > 0.;
Float eo = eta_o, et = eta_t;
if (!entering)
    swap(eo, et);
```

Once we know which index of refraction is which, we can easily compute $\sin\theta_t$ using Snell's law:

⟨*Compute* `sint` *using Snell's law*⟩≡

```
  Float sint = eo/et * sqrtf(max(0.f, 1.f - coso*coso));
```

When light is traveling from one medium to another with a lower index of refraction, incident angles near grazing have no transmission into the other medium. The angle at which this happens is called the *critical angle*; when θ_o is greater than the critical angle, *total internal reflection* occurs–all of the light is just reflected. That case is detected here by a value of $\sin\theta_t$ greater than one; we just set `F` to 1, rather than using the Fresnel equations.

⟨*Handle total internal reflection*⟩≡

```
  return 1.;
```

A Special Fresnel Interface

The `FresnelNoOp` implementation of the `Fresnel` interface simply returns 100% reflection for all incoming directions. Although this is physically implausible, it allows `lrt` to implement certain kinds of materials that are popular in computer graphics, such as perfect mirror reflection.

`Fresnel`	341
`Spectrum`	181
`SpecularReflection`	345

⟨*BxDF Declarations*⟩+≡

```
  class FresnelNoOp : public Fresnel {
  public:
      Spectrum Evaluate(Float) const { return Spectrum(1.); }
  };
```

9.2.2 Specular reflection

We can now implement the `SpecularReflection` class, which describes physically plausible specular reflection using the Fresnel interface from the previous section. First, we will derive the BRDF for a specular reflector. Since the Fresnel equations give the fraction of light reflected $F_r(\omega_o)$, then we need a BRDF such that

$$L_\mathrm{o}(\omega_o) = F_r(\omega_o)\, L_\mathrm{i}(\omega_i)$$

where ω_i is the reflection vector for ω_o about the surface normal.

Such a BRDF can be constructed using the *Dirac delta distribution*, a special distribution $\delta(x)$ defined such that

$$\delta(x) = 0 \forall x \neq 0$$

but where

$$\int_{-\infty}^{\infty} \delta(x)\, dx = 1.$$

These functions have the important property that:

$$\int f(x)\, \delta(x - x_0)\, dx = f(x_0) \qquad (9.2.3)$$

The delta distribution requires special handling compared to standard functions. In particular, integrals with delta distributions must be evaluated by sampling the delta distribution; their values cannot be properly computed without doing so. Consider the delta distribution equation 9.2.3: if we tried to evaluate it using the trapezoid rule or some other numerical integration technique, there would be zero probability that any of the evaluation points x_i would have a non-zero value of $\delta(x_i)$. Rather, we must allow the delta distribution to determine the evaluation point itself. We will see this issue in practice for both specular `BxDF`s as well as some of the light sources in Chapter 13.

Intuitively, we want the BRDF to be zero everywhere except at the perfect reflection direction. This suggests the use of the delta distribution. A first guess might be to simply use delta functions to restrict the outgoing angle to the reflection angle. This would yield a BRDF of:

$$f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i) = \delta(\omega_i - \omega_r) = \delta(cos\theta_i - cos\theta_r)\delta(\phi_i - \phi_r \pm \pi)$$

Although this seems appealing, plugging into equation 5.4.9 reveals a problem:

$$\begin{aligned} L_r(\theta_r,\phi_r) &= \int_{HemiN} \delta(cos\theta_i - cos\theta_r)\delta(\phi_i - \phi_r \pm \pi) L_i(\theta_i,\phi_i)\cos\theta_i \, d\omega_i \\ &= L_i(\theta_r,\phi_r \pm \pi)\cos\theta_i \end{aligned}$$

This is not correct, because it contains an extra factor of $\cos\theta_i$. We can simply divide out this factor to find the correct BRDF for perfect specular reflection:

334	`BSDF_REFLECTION`
334	`BSDF_SPECULAR`
334	`BxDF`
334	`BxDFType`
341	`Fresnel`
181	`Spectrum`

$$f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i) = F_r(\omega_o)\frac{\delta(\omega_i - \mathrm{R}(\omega_o,\mathbf{n}))}{|\cos\theta_i|}$$

if $\mathrm{R}(\omega_o,\mathbf{n})$ is the specular reflection vector for ω_o reflected about the surface normal $\mathbf{n}$. (Note that $F_r(\omega_o) = F_r(\omega_i)$ when the BRDF is non-zero, sine then the two directions make the same angle with the surface normal.)

⟨*BxDF Declarations*⟩+≡

```
class SpecularReflection : public BxDF {
public:
    ⟨SpecularReflection Public Methods⟩
private:
    ⟨SpecularReflection Private Data⟩
};
```

The `SpecularReflection` BxDF takes a `Fresnel` object to describe dielectric or conductor Fresnel properties and an additional spectrum, which is used to scale the reflected color.

⟨*SpecularReflection Public Methods*⟩≡

```
SpecularReflection(const Spectrum &r, Fresnel *f)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_SPECULAR)),
      R(r), fresnel(f) {
}
```

⟨*SpecularReflection Private Data*⟩≡

```
Spectrum R;
Fresnel *fresnel;
```

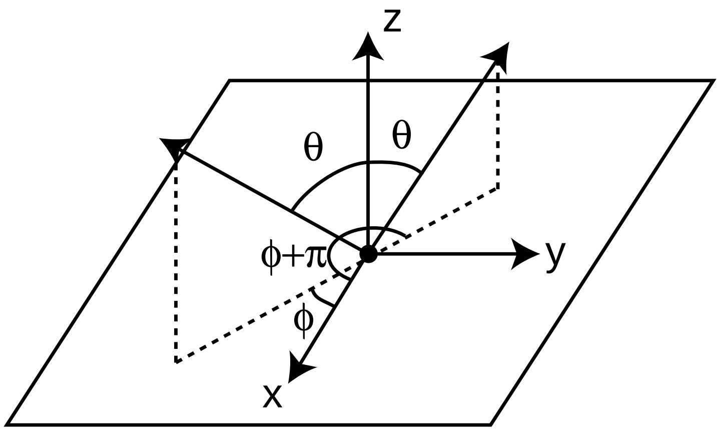


Figure 9.6: Given an incident direction that makes an angle θ with the surface normal and an angle φ with the x axis, the reflected ray about the normal makes an angle θ with the normal and $\phi+\pi$ with the x axis. The (x,y,z) coordinates of this direction can be found by scaling the incident direction by $(-1,-1,1)$.

`CosTheta()`	332
`Fresnel::Evaluate()`	341
`Spectrum`	181
`SpecularReflection`	345
`SpecularReflection::R`	345
`Vector`	27

The rest of the implementation is completely straightforward; we return no scattering from `SpecularReflection::f()`, since for an arbitrary pair of directions, the delta function returns no scattering[4].

⟨*SpecularReflection Public Methods*⟩+≡

```
Spectrum f(const Vector &, const Vector &) const {
    return Spectrum(0.);
}
```

However, we do implement the `Sample_f()` method, which selects an appropriate direction according to the delta function.

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum SpecularReflection::Sample_f(const Vector &wo,
        Vector *wi, Float u1, Float u2, Float *pdf) const {
    ⟨Compute perfect specular reflection direction⟩
    *pdf = 1.f;
    return fresnel->Evaluate(CosTheta(wo)) * R /
        fabsf(CosTheta(*wi));
}
```

To compute the reflection direction, we need to compute the reflection of ω_o around the surface normal. Because all these computations take place in a shading coordinate system where the surface normal is defined to be $(0,0,1)$, the computation is quite simple–all we need to do is to rotate ω_o by π radians about **n**–see Figure 9.6.

Recall the transformation matrix from Chapter 2 for a rotation around the z axis;

[4] If the user happened to pass a vector and its perfect mirror direction, this function still returns zero. Although this might be a slightly confusing interface to these reflection functions, we still get the correct answer because reflection functions involving singularities like δ-functions require special handling by the light transport equations (see Chapter 16).

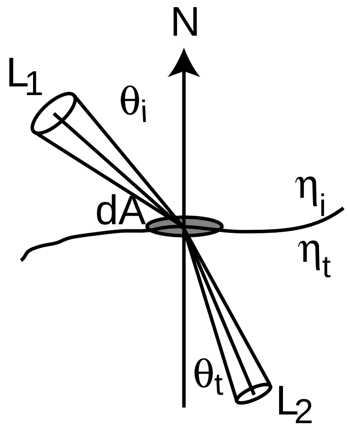


Figure 9.7: The amount of transmitted radiance at the boundary between media with different indices of refraction is scaled by the squared ratio of the two indices of refraction. Intuitively, this can be understood to be the result of the radiance's differential solid angle being squeezed down or expanded as a result of transmission.

if the angle of rotation is π radians, it is:

27 Vector

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

When a vector is multiplied by this matrix, the effect is just to negate the x and y components and thus it's easy to compute the reflection direction.

⟨*Compute perfect specular reflection direction*⟩≡

```
  *wi = Vector(-wo.x, -wo.y, wo.z);
```

9.2.3 Specular Transmission

there was some horrible confusion here between the *o* subscript and the *i* subscript on thetas, omegas, and etas. I'm not sure I got it all right. This merits some careful looking-at. Please.

We will now derive the BTDF for specular transmission. Snell's law does more than give us the direction for the transmitted ray– it can also be used to show that radiance along a ray changes as the ray goes between media of different indices of refraction.

Consider radiance arriving at the boundary between two media, with indices of refraction η_i and η_t for the incoming and transmitted media, respectively (Figure 9.7). We will denote the fraction of incident energy that is transmitted to the outgoing direction by τ (this will generally be given by the Fresnel equations). The amount of transmitted differential flux, then, is:

$$\mathrm{d}^2\Phi_i = \tau \mathrm{d}^2\Phi_t.$$

If we use the definition of radiance (Equation 5.2.4), we have

$$\tau(L_{\mathrm{t}} \cos\theta_t \, \mathrm{d}A \mathrm{d}\omega_t) = (L_{\mathrm{i}} \cos\theta_i \, \mathrm{d}A \mathrm{d}\omega_i).$$

Expanding the solid angles to spherical angles, we have

$$\tau(L_{\mathrm{t}} \cos\theta_t \, \mathrm{d}A \, \sin\theta_t \, \mathrm{d}\theta_t \, \mathrm{d}\phi_t) = (L_{\mathrm{i}} \cos\theta_i \, \mathrm{d}A \, \sin\theta_i \, \mathrm{d}\theta_i \, \mathrm{d}\phi_i). \qquad (9.2.4)$$

We can now differentiate Snell’s law with respect to θ, which gives the relation:

$$\eta_i \cos\theta_i \, \mathrm{d}\theta_i = \eta_t \cos\theta_t \, \mathrm{d}\theta_t.$$

Rearranging terms, we get:

$$\frac{\cos\theta_i \, \mathrm{d}\theta_i}{\cos\theta_t \, \mathrm{d}\theta_t} = \frac{\eta_t}{\eta_i}.$$

Substituting this and Snell’s law into Equation 9.2.4 and simplifying, we have:

$$\tau L_{\mathrm{t}} \eta_i^2 \mathrm{d}\phi_t = L_{\mathrm{i}} \eta_t^2 \mathrm{d}\phi_i.$$

Because $\phi_t = \phi_i + \pi$, $\mathrm{d}\phi_t = \mathrm{d}\phi_i$, so this gives the final relationship:

$$L_{\mathrm{i}} = \tau L_{\mathrm{t}} \frac{\eta_i^2}{\eta_t^2}. \qquad (9.2.5)$$

BSDF_SPECULAR	334
BSDF_TRANSMISSION	334
BxDF	334
BxDFType	334
FresnelDielectric	342
Spectrum	181
SpecularReflection	345
ecularTransmission::etat	349
SpecularTransmission::T	349

The BTDF for specular transmission is thus

$$f_{\mathrm{r}}(\mathrm{p}, \omega_i, \omega_t) = \frac{\eta_i^2}{\eta_t^2} (1 - F_r(\omega_i)) \, \frac{\delta(\omega_i - \mathrm{T}(\omega_t))}{|\cos\theta_t|}$$

The `SpecularTransmission` class is almost exactly the same as `SpecularReflection` except that the sampled direction is the direction for perfect specular transmission.

⟨*BxDF Declarations*⟩+≡

```
class SpecularTransmission : public BxDF {
public:
    ⟨SpecularTransmission Public Methods⟩
private:
    ⟨SpecularTransmission Private Data⟩
};
```

The `SpecularTransmission` constructor stores the two indices of refraction on either side of the surface, as well as a “transmissiveness” scaling factor `T`.

⟨*SpecularTransmission Public Methods*⟩≡

```
SpecularTransmission(const Spectrum &t, Float ei, Float et)
    : BxDF(BxDFType(BSDF_TRANSMISSION | BSDF_SPECULAR)),
      fresnel(ei, et) {
    T = t;
    etai = ei;
    etat = et;
}
```

Because conductors do not transmit light, we always use a `FresnelDielectric` object to do the Fresnel computations.

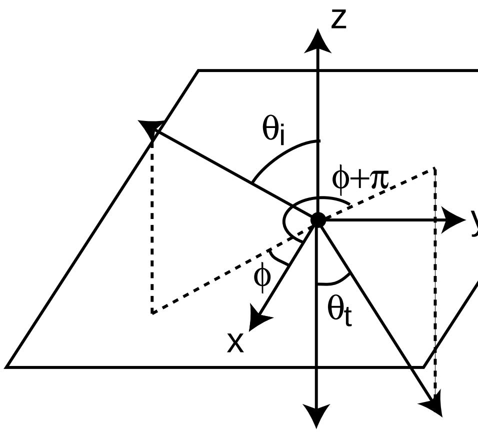


Figure 9.8: The specularly transmitted direction make an angle θ_t with the negated surface normal, $-z$. Like specular reflection, the angle it makes with the x axis is π greater than the incident ray's angle.

⟨*SpecularTransmission Private Data*⟩≡

```
Spectrum T;
Float etai, etat;
FresnelDielectric fresnel;
```

Like the `SpecularReflection` class, we return zero from `SpecularTransmission::f()`, since this reflection distribution is a multiple of a δ-function.

332	`CosTheta()`
341	`Fresnel::Evaluate()`
342	`FresnelDielectric`
181	`Spectrum`
345	`SpecularReflection`
348	`SpecularTransmission`
27	`Vector`

⟨*SpecularTransmission Public Methods*⟩+≡

```
Spectrum f(const Vector &, const Vector &) const {
    return Spectrum(0.);
}
```

Figure 9.8 shows the basic setting for specular transmission. The incident ray is refracted about the surface normal, with the angle θ_t given by Snell's law.

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum SpecularTransmission::Sample_f(const Vector &wo,
        Vector *wt, Float u1, Float u2, Float *pdf) const {
    ⟨Figure out which η is incident and which is transmitted⟩
    ⟨Compute transmitted ray direction⟩
    *pdf = 1.f;
    Spectrum F = fresnel.Evaluate(CosTheta(wo));
    return (ei*ei)/(et*et) * (Spectrum(1.)-F) * T /
        fabsf(CosTheta(*wt));
}
```

We start by seeing if the incident ray is entering or exiting the refractive medium; we use the convention that the surface normal (and thus the $(0,0,1)$ direction in our local reflection space) is oriented such that it points outside of the object. Therefore, if the z component of the ω_o direction is greater than zero, the incident ray is coming from outside of the object.

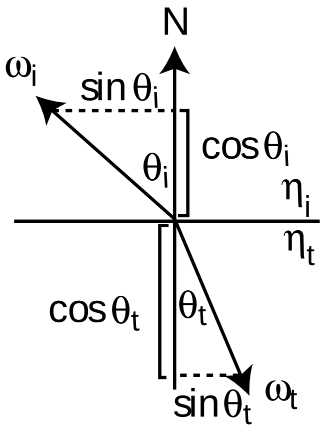


Figure 9.9: Basic geometry for computing the transmitted direction ω_t from the incident direction ω_i. The $\cos\theta$ terms are equal to the z components of the direction vectors and the $\sin\theta$ terms are equal to the xy lengths of the corresponding direction vectors.

⟨*Figure out which* η *is incident and which is transmitted*⟩≡

```
bool entering = CosTheta(wo) > 0.;
Float ei = etai, et = etat;
if (!entering)
    swap(ei, et);
```

CosTheta()	332
SinTheta2()	332
pecularTransmission::etat	349
Vector	27

Figure 9.9 shows the basic setting for computing the transmitted ray direction.

We next compute `sini2` and `sint2`, which are the squares of $\sin\theta_i$ and $\sin\theta_t$, respectively. In the reflection coordinate system, $\sin\theta_i$ is equal to the sum of the squares of the x and y components of ω_o. $(\sin\theta_t)^2$ can be computed directly from $(\sin\theta_i)^2$ using Snell's law.

We then apply the trigonometric identity $\sin^2\theta + \cos^2\theta = 1$ to compute $\cos\theta_t$ from $\sin\theta_t$; this directly gives us the z component of the transmitted direction. To compute the x and y components, we first mirror ω_o about the normal, as we did for specular reflection, but then scale it by the ratio $\sin\theta_t / \sin\theta_i$ to give it the proper magnitude. From Snell's law, this ratio is just η_i/η_t, though, which we happen to have computed previously.

⟨*Compute transmitted ray direction*⟩≡

```
Float sini2 = SinTheta2(wo);
Float eta = ei / et;
Float sint2 = eta * eta * sini2;
⟨Handle total internal reflection for transmission⟩
Float cost = sqrtf(max(0.f, 1.f - sint2));
if (entering) cost = -cost;
Float sintOverSini = eta;
*wt = Vector(sintOverSini * -wo.x, sintOverSini * -wo.y, cost);
```

We need to handle the case of total internal reflection here as well; if the squared value of $\sin\theta_t$ is greater than one, no transmission is possible, so we just return black.

⟨*Handle total internal reflection for transmission*⟩≡

```
if (sint2 > 1.) return 0.;
```

9.3 Lambertian Reflection

One of the simplest BRDFs is the Lambertian model; it models a perfect diffuse surface that scatters incident illumination equally in all directions. Although this reflection model is not physically plausible, it is a good approximation to many real-world surfaces such as matte paint.

⟨*BxDF Declarations*⟩+≡

```
class Lambertian : public BxDF {
public:
    ⟨Lambertian Public Methods⟩
private:
    ⟨Lambertian Private Data⟩
};
```

The `Lambertian` constructor takes a reflectance SPD R which gives the fraction of incident light that is scattered. We also precompute the quantity $\frac{R}{\pi}$ for convenience in later calculations.

⟨*Lambertian Public Methods*⟩≡

```
Lambertian(const Spectrum &reflectance)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)),
      R(reflectance), RoverPI(reflectance * INV_PI) {
}
```

334	BSDF_DIFFUSE
334	BSDF_REFLECTION
334	BxDF
334	BxDFType
678	INV_PI
181	Spectrum
27	Vector

⟨*Lambertian Private Data*⟩≡

```
Spectrum R, RoverPI;
```

The reflection distribution function for `Lambertian` is quite straightforward, since its value is constant. However, we don't just return the original reflectance value R, since BRDF's must integrate to unity over the hemisphere to preserve energy (see Section 5.4.1). Since the true BRDF must be a multiple of R, we know that:

$$\int_{\mathcal{H}^2(N)} cR \, \mathrm{d}\omega = 1$$

which gives $=\frac{1}{\pi}$. This $\frac{R}{\pi}$ value was precomputed in the `Lambertian` constructor.

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum Lambertian::f(const Vector &wo,
        const Vector &wi) const {
    return RoverPI;
}
```

The directional-hemispherical and hemispherical-hemispherical reflectance values for a Lambertian BRDF are trivial to compute analytically; the derivations are omitted.

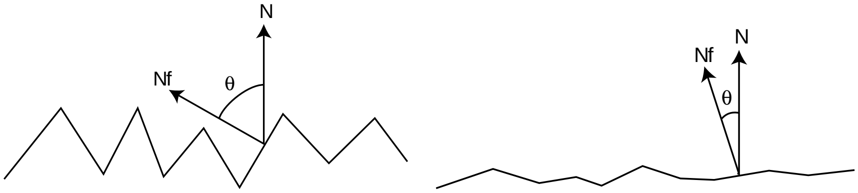


Figure 9.10: Microfacet surface models are often described by a function that describes the distribution of microfacet normals $\mathbf{n_f}$ with respect to the surface normal $\mathbf{n}$. The greater the variation of microfacet normals, the rougher the surface is (left). Smooth surfaces have relatively little variation of microfacet normals (right).

⟨*Lambertian Public Methods*⟩+≡

```
Spectrum rho(const Vector &w, int nSamples,
        Float *samples) const {
        return R;
}
```

⟨*Lambertian Public Methods*⟩+≡

```
Spectrum rho(int, Float *) const { return R; }
```

Lambertian::R	351
Spectrum	181
Vector	27

9.4 Microfacet Models

Most geometric optics-based approaches to modeling surface reflection are based on the idea that rough surfaces can be modeled as a collection of small *microfacets*. A surface comprised of microfacets is essentially a heightfield, where the distribution of faces is described statistically. For example, the left half of Figure 9.10 shows the cross-section of a relatively rough surface. On the right is a much smoother microfacet surface.

Microfacet-based BRDF models work by statistically modeling the scattering of light from a large collection of such microfacets. If we assume that the differential area being illuminated, dA, is relatively large compared to the size of individual microfacets, then a large number of microfacets are illuminated, so their aggregate behavior can be modeled.

The two main components of microfacet models are an expression for the distribution of facets and a BRDF that describes how light scatters from individual microfacets. Given these, we would like to derive a closed form expression that gives the BRDF that describes scattering from such a surface. Perfect mirror reflection is typically used for the microfacet BRDF, though the Oren–Nayar model (described below) treats them as Lambertian reflectors.

Finally, local lighting effects at the microfacet level need to be considered (Figure 9.11). Consider an individual microfacet of interest, indicated by a heavy line (`WHAT HEAVY LINE`) in the figure. On the left, we can see that the viewer may not be able to see it, due to occlusion from another microfacet. As shown in the middle figure, it may be in the shadow of a neighboring microfacet. Finally, as shown in the right figure, inter-reflection among the microfacets may cause the microfacet to receive more light than predicted by direct illumination. A common simplification

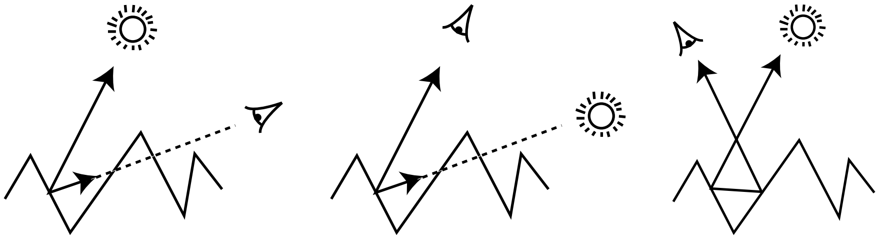


Figure 9.11: There are three important geometric effects to consider with microfacet reflection models. On the left is *masking*, where the microfacet of interest isn't visible to the viewer due to occlusion by another microfacet. In the middle is *shadowing*, where analogously light doesn't reach the microfacet. On the right is *inter-reflection*, where light bounces among the microfacets before reaching the viewer.

is to assume that all of the microfacets are symmetric V-shaped grooves. If this assumption is made, then interreflection with most of the other microfacets can be ignored; only the neighboring microfacet in the groove needs to be considered.

Individual microfacet-based BRDFs consider each of these effects with varying degrees of accuracy. The general approach is to make the best approximations to these effects possible, while still obtaining an easily evaluated expression.

9.4.1 Oren–Nayar diffuse reflection

Oren and Nayar observed that real-world objects tend not to exhibit perfect Lambertian. Specifically, rough surfaces generally appear brighter as the illumination direction approaches the viewing direction. They developed a reflection model that describes rough surfaces as a collection of symmetric V-shaped grooves. They further assumed that each individual microfacet (groove face) exhibited perfect Lambertian reflection, and derived a BRDF that models the aggregate reflection of the collection of grooves. The distribution of microfacets was assumed to be Gaussian with a single parameter σ, the standard deviation of the orientation angle.

The resulting model, which accounts for shadowing, masking, and inter-reflection among the microfacets didn't have a closed-form solution, so they found the following approximation that fit it well: result is:

$$f_r(\omega_i, \omega_o) = \frac{\rho}{\pi}\left(A + B \max(0, \cos(\phi_i - \phi_o)) \sin\omega_+ \tan\omega_-\right)$$

where if σ is in radians,

$$\begin{aligned} A &= 1 - \frac{\sigma^2}{2(\sigma^2 + 0.33)} \\ B &= \frac{0.45\sigma^2}{\sigma^2 + 0.09} \\ \omega_+ &= \max(\theta_i, \theta_o) \\ \omega_- &= \min(\theta_i, \theta_o) \end{aligned}$$

We can precompute the values of the A and B parameters to the model and store them away in the constructor; this will save us work in evaluating the BRDF later.

⟨*OrenNayar Public Methods*⟩+≡

```
OrenNayar(const Spectrum &reflectance, Float sig)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)),
      R(reflectance) {
    Float sigma = Radians(sigma);
    Float sigma2 = sigma*sigma;
    A = 1.f - (sigma2 / (2.f * (sigma2 + 0.33f)));
    B = 0.45f * sigma2 / (sigma2 + 0.09f);
}
```

⟨*OrenNayar Private Data*⟩≡

```
Spectrum R;
Float A, B;
```

Evaluating the model is relatively straightforward; just a matter of applying some trigonometry to compute the values for the terms in the model. We start by computing and storing $\sin\theta_i$ and $\sin\theta_o$; recall from to the section on specular transmission and Figure 9.3 that the *xy* magnitude of the direction vectors gives these values.

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum OrenNayar::f(const Vector &wo,
        const Vector &wi) const {
    Float sinthetai = SinTheta(wi), sinthetao = SinTheta(wo);
    ⟨Compute cosine term of Oren–Nayar model⟩
    ⟨Compute sine and tangent terms of Oren–Nayar model⟩
    return R * INV_PI * (A + B * maxcos * sinalpha * tanbeta);
}
```

BSDF_DIFFUSE	334
BSDF_REFLECTION	334
BxDF	334
BxDFType	334
CosPhi()	333
INV_PI	678
SinPhi()	333
SinTheta()	332
Spectrum	181
Vector	27

We now need to compute the $\max(0, \cos(\phi_i - \phi_o))$ term. We can apply the trigonometric identity

$$\cos(a-b) = \cos a \cos b + \sin a \sin b,$$

such that we just need to compute the sines and cosines of ϕ_i and ϕ_o.

⟨*Compute cosine term of Oren–Nayar model*⟩≡

```
Float sinphii = SinPhi(wi), cosphii = CosPhi(wi);
Float sinphio = SinPhi(wo), cosphio = CosPhi(wo);
Float dcos = cosphii * cosphio + sinphii * sinphio;
Float maxcos = max(0.f, dcos);
```

Finally, we compute the $\sin\alpha$ and $\tan\beta$ terms. Note that whichever of ω_i or ω_o has a larger value for $\cos\theta$ (i.e. a larger value of its *z* component), has a *smaller* value for θ. Given the knowledge of which angle is smaller, we can set $\sin\alpha$ directly from the appropriate $\sin\theta$ value already computed. The tangent can then just be computed using the identity $\tan a = \sin a / \cos a$.

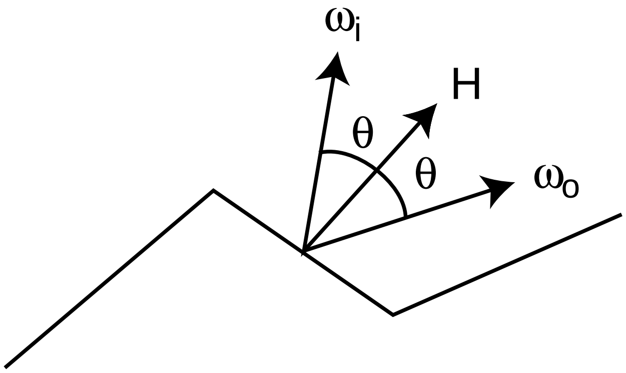


Figure 9.12: For perfectly specular microfacets and a given pair of directions ω_i and ω_o, only those microfacets with normal $\omega_h = \widehat{\omega_i + \omega_o}$ will reflect any light from ω_i to ω_o.

⟨*Compute sine and tangent terms of Oren–Nayar model*⟩≡

```
Float sinalpha, tanbeta;
if (fabsf(CosTheta(wi)) > fabsf(CosTheta(wo))) {
    sinalpha = sinthetao;
    tanbeta = sinthetai / fabsf(CosTheta(wi));
}
else {
    sinalpha = sinthetai;
    tanbeta = sinthetao / fabsf(CosTheta(wo));
}
```

332 CosTheta()

9.4.2 Torrance–Sparrow model

One of the first microfacet models for computer graphics was developed by Torrance and Sparrow to model metallic surfaces. They modeled surfaces as collections of perfectly smooth mirror microfacets. The surface is statistically described by a distribution function $D(\theta)$ that gives the probability that a microfacet has orientation θ (recall Figure 9.10 which shows how roughness and the microfacet normal distribution function are related).

Because the microfacets are perfectly specular, only those that are oriented exactly so that they reflect the incident direction ω_i to the outgoing direction ω_o give any reflection for that pair of directions. In other words, only those microfacets with a normal equal to the *half-angle vector*,

$$\omega_h = \widehat{\omega_i + \omega_o}$$

cause perfect specular reflection from ω_i to ω_o (and vice-versa). (Figure 9.12.)

The derivation of the Torrance–Sparrow has a number of interesting steps; we'll go through it in detail here. Consider the differential flux incident on the microfacets oriented with half-angle ω_h for directions ω_i and ω_o, $\mathrm{d}^2\Phi_h$. From the definition of radiance, Equation 5.2.4, it is: **I don't like the "2" in the superscript of** $\mathrm{d}^2\Phi_h$

$$\mathrm{d}^2\Phi_h = L_{\mathrm{i}}(\omega_i)\,\mathrm{d}\omega\,\mathrm{d}A^{\perp}(\omega_h) = L_{\mathrm{i}}(\omega_i)\,\mathrm{d}\omega\,\cos\theta_h\,\mathrm{d}A(\omega_h),$$

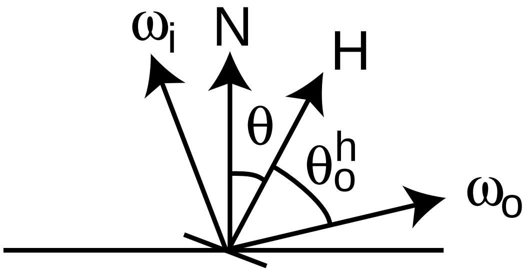


Figure 9.13: Setting for the derivation of the Torrance–Sparrow model. For directions ω_i and ω_i, only microfacets with normal ω_h reflect light. The angle between ω_h and **n** is denoted by θ and the angle between ω_h and ω_o is denoted by θ_h. (The angle between ω_h and ω_i is also necessarily θ_h.)

where we have written dA (ω_h) for the area measure of the microfacets with orientation ω_h and $\cos\theta_h$ for the cosine of the angle between ω_i and ω_h (Figure 9.13).

The differential area of microfacets with orientation ω_h is just

$$\mathrm{d}A(\omega_h) = D(\omega_h)\mathrm{d}\omega_h \mathrm{d}A.$$

The first two terms describe the differential area of facets per unit area that have the proper orientation, and the dA term converts this to differential area.

Therefore,

$$\mathrm{d}^2\Phi_h = L_{\mathrm{i}}(\omega_i)\,\mathrm{d}\omega\,\cos\theta_h\,D(\omega_h)\,\mathrm{d}\omega_h\,\mathrm{d}A(\omega_h). \tag{9.4.6}$$

If we assume that the microfacets individually reflect light according to Fresnel's law, the outgoing flux is

$$\mathrm{d}^2\Phi_o = F(\omega_o,\omega_i)\mathrm{d}^2\Phi_h. \tag{9.4.7}$$

Again using the definition of radiance, the reflected outgoing radiance is

$$L(\omega_o) = \frac{\mathrm{d}^2\Phi_o}{\mathrm{d}\omega_o\,\cos\theta_o \mathrm{d}A}.$$

If we substitute Equation 9.4.7 into this and then Equation 9.4.6 into the result, we have

$$L(\omega_o) = \frac{F(\omega_o,\omega_i)\,L_{\mathrm{i}}(\omega_o)\,\mathrm{d}\omega_i\,D(\omega_h)\,\mathrm{d}\omega_h\,\mathrm{d}A\,\cos\theta_h}{\mathrm{d}\omega_o\,\mathrm{d}A\,\cos\theta_o}$$

In Section 15.3.1, we will derive an important relation between $\mathrm{d}\omega_h$ and $\mathrm{d}\omega_o$:

$$\mathrm{d}\omega_h = \frac{\mathrm{d}\omega_o}{4\cos\theta_h}.$$

We can substitute this into the previous equation and simplify, giving

$$L(\omega_o) = \frac{F(\omega_o,\omega_i)\,L_{\mathrm{i}}(\omega_o)\,D(\omega_h)\,\mathrm{d}\omega_i}{4\,\cos\theta_o}.$$

We can now apply the definition of the BRDF, Equation 5.4.8, giving us the Torrance–Sparrow BRDF:

$$f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i) = \frac{D(\omega_h)\,F(\omega_i,\omega_o)}{4\,\cos\theta_i\,\cos\theta_o}$$

The Torrance–Sparrow model also includes a *geometric attenuation* term, which describes the fraction of microfacets that are masked or shadowed, given directions ω_i and ω_o. This G term can just be included in the derivation as the Fresnel term was above. The full model, then, is

$$f_{\mathrm{r}}(\mathrm{p}, \omega_o, \omega_i) = \frac{D(\omega_h)\, G(\omega_o, \omega_i)\, F(\omega_o, \omega_i)}{4 \cos\theta_o \cos\theta_i}. \tag{9.4.8}$$

One of the nice things about the Torrance–Sparrow model is that the derivation doesn't depend on the particular microfacet distribution being used. Furthermore, because it doesn't depend on a particular Fresnel function, it can be used for both conductors and dielectrics. However, reflection functions besides perfect specular reflection can *not* be easily substituted: the relationship between $d\omega_h$ and $d\omega_o$ used in its derivation depends on the specular reflection assumption.

We now use the Torrance–Sparrow model to implement a general microfacet-based BRDF. It takes a pointer to an abstract `MicrofacetDistribution` class, which provides a single method to compute the D term of the Torrance–Sparrow model. This function, `MicrofacetDistribution::D()`, gives the probability density for microfacets to be oriented with normal ω_h.

⟨*BxDF Declarations*⟩+≡

```
class MicrofacetDistribution {
public:
    ⟨MicrofacetDistribution Interface⟩
};
```

334 BSDF_GLOSSY
334 BSDF_REFLECTION
334 BxDF
334 BxDFType
341 Fresnel
358 Microfacet::distribution
358 Microfacet::fresnel
358 Microfacet::R
181 Spectrum
27 Vector

⟨*MicrofacetDistribution Interface*⟩+≡

```
virtual Float D(const Vector &wh) const = 0;
```

The `Microfacet` BRDF, then, just takes a pointer to a distribution, the reflectance of the object, and a Fresnel function.

⟨*BxDF Declarations*⟩+≡

```
class Microfacet : public BxDF {
public:
    ⟨Microfacet Public Methods⟩
private:
    ⟨Microfacet Private Data⟩
};
```

⟨*BxDF Method Definitions*⟩+≡

```
Microfacet::Microfacet(const Spectrum &reflectance, Fresnel *f,
        MicrofacetDistribution *d)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_GLOSSY)),
     R(reflectance), distribution(d), fresnel(f) {
}
```

⟨*Microfacet Private Data*⟩≡

```
Spectrum R;
MicrofacetDistribution *distribution;
Fresnel *fresnel;
```

Evaluating the terms of the Torrance–Sparrow BRDF is straightforward. For the Fresnel term, recall that the angle θ_h is the same between ω_h and both ω_i and ω_o, so it doesn't matter which vector we use to compute the cosine of θ_h. We arbitrarily choose to use ω_i.

⟨BxDF Method Definitions⟩+≡

```
Spectrum Microfacet::f(const Vector &wo, const Vector &wi) const {
    Float cosThetaO = fabsf(CosTheta(wo));
    Float cosThetaI = fabsf(CosTheta(wi));
    Vector wh = (wi + wo).Hat();
    Spectrum F = fresnel->Evaluate(Dot(wi, wh));
    return R * distribution->D(wh) * G(wo, wi, wh) * F /
         (4.f * cosThetaI * cosThetaO);
}
```

The final component of the Torrance–Sparrow model is a geometric attenuation term that accounts for masking and shadows. This term is derived by assuming that the microfacets were made of infinitely long V-shaped grooves. This assumption is more restrictive than the general term $D(\omega_h)$ used to model the microfacet distribution, and does not account for the roughness of the surface, but yields a closed form result, which is crucial. It is also easy to evaluate, and matches real-world surfaces well. The attenuation term is (we omit the derivation):

AbsDot()	31
CosTheta()	332
Dot()	30
Fresnel	341
Fresnel::Evaluate()	341
Microfacet	357
MicrofacetDistribution	357
rofacetDistribution::D()	357
Spectrum	181
Vector	27
Vector::Hat()	30

$$G(\omega_o, wi) = \min\left(1, \min\left(\frac{2(N\cdot H)(N\cdot\omega_o}{\omega_o\cdot H}, \frac{2*(N\cdot H)(N\cdot\omega_i)}{\omega_o\cdot H}\right)\right)$$

⟨Microfacet Public Methods⟩+≡

```
Float G(const Vector &wo, const Vector &wi,
        const Vector &wh) const {
    Float NdotH = fabsf(CosTheta(wh));
    Float NdotWO = fabsf(CosTheta(wo)), NdotWI = fabsf(CosTheta(wi));
    Float WOdotH = AbsDot(wo, wh);
    return min(1.f, min((2.f * NdotH * NdotWO / WOdotH),
                    (2.f * NdotH * NdotWI / WOdotH)));
}
```

9.4.3 Blinn Microfacet Distribution

The Blinn microfacet model gives an exponential falloff of distribution of microfacet normal orientations with respect to the underlying surface normal. The most likely microfacet orientation is in the surface normal direction, falling off to no microfacets oriented perpendicular to the normal. For smooth surfaces, this falloff happens very quickly, and for rough surfaces, it is more gradual.

```
⟨BxDF Declarations⟩+≡
  class Blinn : public MicrofacetDistribution {
  public:
      Blinn(Float e) { exponent = e; }
      ⟨Blinn Public Methods⟩
  private:
      Float exponent;
  };
```

The Blinn model uses a popular theme in computer graphics: it raises the cosine of the angle between the half-vector and the normal to a power:

$$D(\omega_h) \propto (\omega_h \cdot N)^e$$

where e is a user-supplied exponent that controls the rate. Like BRDF's, microfacet distribution functions must be *normalized* to ensure that they are physically plausible. In the case of microfacets, what we need to ensure is that there exists some heightfield with the given distribution of face normals $D(\omega_h)$. Another way to think about this normalization requirement is that if we sum the projected area of all the microfacet faces over some area $\mathrm{d}A$, the sum should equal $\mathrm{d}A$. Mathematically, this means we must enforce:

357 MicrofacetDistribution

$$\int_{\mathcal{H}^2(N)} D(\omega_h) \cos\theta_h \, \mathrm{d}\omega_h = 1$$

A common error is to perform this integral over solid angle instead of projected solid angle (i.e., to leave out the $\cos\theta_h$ term), which is not too bad, but it does not guarantee the existence of a heightfield with the right distribution. Of course, we don't care about the actual heightfield, just that it exists.

For the Blinn model, we have that $D(\omega_h) \propto (\omega_h \cdot N)^e$. The above normalization requirement gives:

$$\begin{aligned}
\int_{\mathcal{H}^2(N)} c(\omega_h \cdot N)^e \cos\theta_h \, \mathrm{d}\omega_h &= \int_0^{2\pi} \int_0^{\frac{\pi}{2}} c(\cos\theta_h)^{e+1} \sin\theta_h \, d\theta \, d\phi \\
&= 2c\pi \int_0^1 u^{e+1} du \\
&= 2c\pi \left. \frac{u^{e+2}}{e+2} \right|_0^1 \\
&= \frac{2c\pi}{e+2} = 1
\end{aligned}$$

Therefore,

$$c = \frac{e+2}{2\pi}$$

So the properly normalized Blinn microfacet distribution is:

$$D(\omega_h) = \frac{e+2}{2\pi}(\omega_h \cdot N)^e \tag{9.4.9}$$

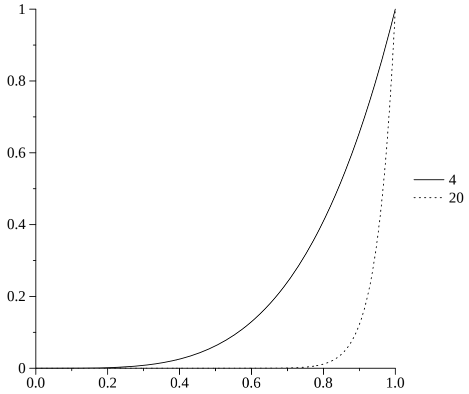


Figure 9.14: Graph showing the effect of varying the exponent for the Blinn microfacet distribution model. The solid line shows the graph of the non-normalized distribution function x^4, and the dotted line shows the graph of x^{20}. The larger the exponent, the more likely it is that a microfacet will be oriented close to the surface normal, as would be the case for a smooth surface.

CosTheta()	332
INV_TWOPI	678
Vector	27

Figure 9.14 shows how the exponent affects the distribution **renderings of this**. The solid line shows the distribution of cosines of the angle between the surface normal and the microfacet normal with an exponent of 4. This corresponds to a fairly rough surface, so there is a high probability of microfacets being oriented in a direction far away from the normal. The dashed line shows the effect of a higher exponent of 20, corresponding to a smoother surface. For this case, there is a much lower probability that any microfacets will be oriented very far from the surface normal direction.

⟨*Blinn Public Methods*⟩≡

```
Float D(const Vector &wh) const {
    Float costhetah = fabsf(CosTheta(wh));
    return (exponent+2) * INV_TWOPI * powf(max(0.f, costhetah), exponent);
}
```

9.4.4 Anisotropic microfacet model

Because the Blinn distribution from the last section only depends on the angle made between the half-angle and the surface normal, it is *radially symmetric*, and yields an isotropic BRDF. Ashikhmin and Shirley have developed a microfacet distribution function for modeling the appearance of anisotropic surfaces. Recall that an anisotropic BRDF is one where the reflection characteristics at a point vary as the surface is rotated about that point in the plane perpendicular to the surface normal. Brushed metals and some types of fabric exhibit anisotropy.

Their model is physically-based, has intuitive parameters, is efficient, and fits well into the Monte Carlo integration techniques that will be introduced in later chapters. We won't present the derivation here; the reader is referred to the further reading section for pointers to their original papers. Their model takes two parameters: e_x, which gives an exponent for the distribution function for half-angle vectors with an azimuthal angle that orients them exactly along the ±x axis, and

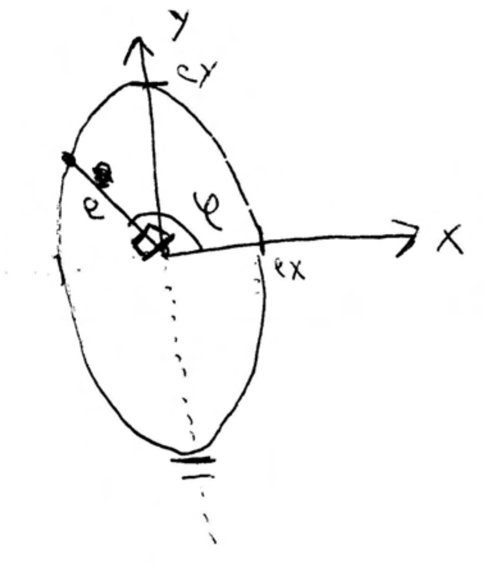


Figure 9.15: The two exponents e_x and e_y for the anisotropic microfacet distribution function give specular exponents for microfacets facing exactly along the x and y axes, respectively. For microfacets with other orientations, the exponent e is computed by finding the radius e of the super-ellipse for the actual orientation angle ϕ.

357 MicrofacetDistribution

e_y, an exponent for microfacets oriented along the $\pm y$ axis. Exponents for intermediate orientations are found by considering these two values as the lengths of the axes of a super-ellipse and finding the appropriate value for the actual microfacet orientation–see Figure 9.15.

The resulting microfacet distribution function is

$$D(\omega_h) = \sqrt{(e_x+1)\cdot(e_y+1)}(\omega_h \cdot N)^{e_x \cos^2\phi + e_y \sin^2\phi}.$$

⟨*BxDF Declarations*⟩+≡
```
class Anisotropic : public MicrofacetDistribution {
public:
    ⟨Anisotropic Public Methods⟩
private:
    Float ex, ey;
};
```

⟨*Anisotropic Public Methods*⟩≡
```
Anisotropic(Float x, Float y) { ex = x; ey = y; }
```

The terms of the distribution function can be computed quite efficiently. Recall from the Oren–Nayar BRDF that $\cos\phi = x/\sin\theta$ and $\sin\phi = y/\sin\theta$. Since we want to compute $\cos^2\phi$ and $\sin^2\phi$, however, we can use the substitution $\sin^2\theta +$

$\cos^2\theta = 1$, so that

$$\cos^2\phi = \frac{x^2}{1-z^2}$$
$$\sin^2\phi = \frac{y^2}{1-z^2}.$$

Thus, the implementation is:

⟨*Anisotropic Public Methods*⟩+≡

```
Float D(const Vector &wh) const {
    Float costhetah = fabsf(CosTheta(wh));
    Float e = (ex * wh.x * wh.x + ey * wh.y * wh.y) /
        (1.f - costhetah * costhetah);
    return sqrtf((ex+1)*(ey+1)) * powf(costhetah, e);
}
```

9.5 Lafortune Model

CosTheta() 332
Vector 27

A recent trend in computer graphics is to enable the use of measured data for rendering realistic images. Although this approach can lead to very realistic renderings, one drawback is the enormous amount of data required to accurately represent a measured BRDF.

Lafortune, Foo, Torrance, and Greenberg have developed a BRDF model designed to fit measured BRDF data to a parameterized model with a relatively small number of parameters. Their model is both easy to implement and efficient. The genesis of their model is the *Phong model*–one of the first BRDF models developed for graphics. The original Phong model has a number of shortcomings (especially that it is neither reciprocal nor energy-conserving) that the Lafortune model avoids.

The *modified Phong BRDF*, which is reciprocal, is

$$f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i) = (\omega_i \cdot R(\omega_o,\mathbf{n}))^e = (\omega_o \cdot R(\omega_i,\mathbf{n}))^e,$$

where $R(\omega,\mathbf{n})$ is the operator that reflects the vector ω about the surface normal $\mathbf{n}$. Like the Blinn microfacet distribution model, the cosine of the angle between the two vectors is raised to a given power. In the canonical BRDF coordinate system, the Phong model can be equivalently written as

$$f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i) = (\omega_i \cdot (\omega_o \times (-1,-1,1)))^e = (\omega_o \cdot (\omega_i \times (-1,-1,1)))^e.$$

Lafortune points out that the vector $(-1,-1,1)$ in the modified Phong model can itself be a parameter to the BRDF. We will call this vector the *orientation vector*, since it orients the direction of maximum reflection. For example, if the orientation vector were $(-1,-1,0.5)$, the main reflection vector would be lowered from the perfect specular direction to be closer to the surface. Many glossy surfaces exhibit this *off-specular* reflective behavior.

If the orientation vector were $(1,1,1)$, the surface would be *retro-reflective*–light would be primarily reflected back along the direction it arrived along. The moon is an example of a retro-reflective surface.

Using this generalized Phong model as a building block, the Lafortune model expresses the BRDF as the sum of multiple Phong *lobes*, each with a different orientation vector and specular exponent, plus a Lambertian diffuse term. The contribution of each lobe is determined by the magnitude of the orientation vector–the re-oriented incident vector is no longer necessarily of unit length and its length affects the magnitude of the dot product[5]. Thus, we have:

$$f_{\mathrm{r}}(\mathrm{p},\omega_o,\omega_i) = \frac{\rho_d}{\pi} + \sum_{i}^{\text{nlobes}} (\omega_i \cdot (\omega_o \times o_i))^{e_i},$$

that's supposed to be e_i on the end there, right? where ρ_d is the diffuse reflectance, o_i are the orientation vectors, and e_i are the specular exponents.

As a further generalization, each orientation vector and specular exponent is allowed to vary as a function of wavelength; we represent each of them with `Spectrum` objects in the implementation below. This gives a natural way to express wavelength-dependent reflection variation in the model.

In the original paper, this model defined the orientation vectors so that the vector $(1,1,1)$ would give the classic Phong model. To be consistent with our specular reflection BRDF, we will instead use the convention that $(-1,-1,1)$ gives the Phong model.

334 BxDF
334 BxDFType
181 Spectrum

⟨*BxDF Declarations*⟩+≡

```
class Lafortune : public BxDF {
public:
    ⟨Lafortune Public Methods⟩
private:
    ⟨Lafortune Private Data⟩
};
```

The `Lafortune` `BxDF` here follows a slightly different convention for managing the values passed into it than the rest of the `BxDF`s: rather than making a local copy of the values in the arrays of coefficients passed to its constructor, it just stores pointers to the arrays and assumes that the calling code will be responsible for freeing this memory if it was dynamically allocated. The motivation for this is to save unnecessary copying of data for the common case where the coefficients are statically allocated, as they are in all of the materials that use `Lafortune` in Chapter 10. If a material uses textures to compute the coeffecients in some manner, it can use the `BSDF_ALLOC()` macro from Section to allocate the coefficient data to ensure that it is freed at an appropriate time.

Because the particular values of the parameters to this BRDF may affect whether it is diffuse or glossy, the `BxDFType` is left as an additional parameter to the constructor.

[5] This makes for an unintuitive control for manual adjustment of the BRDF's characteristics, though it is less troublesome if the BRDF is being automatically fit to measured data.

⟨*BxDF Method Definitions*⟩+≡

```
Lafortune::Lafortune(const Spectrum &r, u_int nl,
    const Spectrum *xx, const Spectrum *yy,
    const Spectrum *zz,
    const Spectrum *e, BxDFType t)
    : BxDF(t), R(r) {
    nLobes = nl;
    x = xx;
    y = yy;
    z = zz;
    exponent = e;
}
```

⟨*Lafortune Private Data*⟩≡

```
Spectrum R;
u_int nLobes;
const Spectrum *x, *y, *z, *exponent;
```

To evaluate this reflection model, we simply sum the contribution of each lobe.

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum Lafortune::f(const Vector &wo, const Vector &wi) const {
    Spectrum ret = R * INV_PI;
    for (u_int i = 0; i < nLobes; ++i) {
        ⟨Add contribution for ith Phong lobe⟩
    }
    return ret;
}
```

BxDF	334
BxDFType	334
INV_PI	678
Lafortune	363
Spectrum	181
Spectrum::Pow()	183
Vector	27

Evaluating each lobe is straightforward. We simultaneously compute the re-oriented ω_o vector by multiplying its x, y, and z coefficients with the appropriate spectral orientation coefficients and compute the dot product of the result with ω_i, giving a spectral result which is itself then raised to the spectral exponent provided.

⟨*Add contribution for ith Phong lobe*⟩≡

```
Spectrum v = x[i] * wo.x * wi.x + y[i] * wo.y * wi.y +
    z[i] * wo.z * wi.z;
ret += v.Pow(exponent[i]);
```

9.6 Fresnel Incidence Effects

Shirley and collaborators have often made the observation that most BRDF Models in graphics do not account for the effect of Fresnel reflection reducing the amount of light reaching the bottom level of layered objects. Consider a polished wood table or a wall with glossy paint: if you look at their surfaces head-on, you primarily see the wood or the paint pigment color. As you move your viewpoint toward a glancing angle, you see less of the underlying color as it is overwhelmed by increasing glossy reflection due to Fresnel effects. The images in Figure XXX show this effect. **do these renderings please**

In this section, we will implement a BRDF model due to Ashikhmin and Shirley that models a diffuse underlying surface with a glossy specular surface above it.

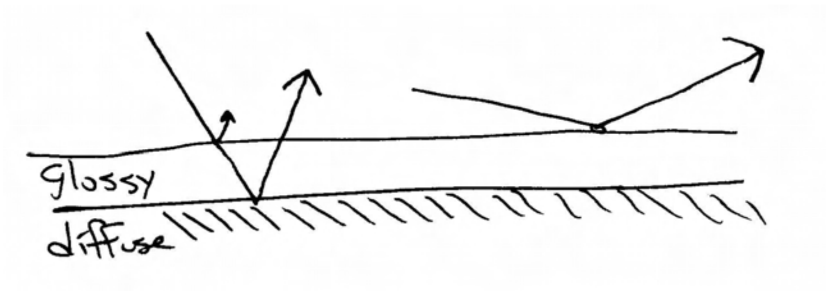


Figure 9.16: The `FresnelBlend` BRDF models the effect of a surface with a glossy layer on top of a diffuse substrate. As on the angle of incidence of the direction vectors ω_i and ω_o heads toward glancing (right), the amount of light that reaches the diffuse substrate is reduced by Fresnel effects and the diffuse layer becomes less visibly apparent.

The effect of reflection from the diffuse surface is modulated by the amount of energy left after Fresnel effects have been considered. Figure 9.16 shows this; on the left, the incident direction is close to the normal, so most light is transmitted to the diffuse layer and the diffuse term dominates. On the right, the incident direction is close to glancing, so glossy reflection is the primary mode of reflection.

334	BSDF_GLOSSY
334	BSDF_REFLECTION
334	BxDF
334	BxDFType
357	MicrofacetDistribution
181	Spectrum

⟨*BxDF Declarations*⟩+≡

```
class FresnelBlend : public BxDF {
public:
    ⟨FresnelBlend Public Methods⟩
private:
    ⟨FresnelBlend Private Data⟩
};
```

The model takes two spectra, representing diffuse and specular reflectance, and a microfacet distribution function for the glossy layer.

⟨*BxDF Method Definitions*⟩+≡

```
FresnelBlend::FresnelBlend(const Spectrum &d, const Spectrum &s,
        MicrofacetDistribution *dist)
    : BxDF(BxDFType(BSDF_REFLECTION | BSDF_GLOSSY)),
      Rd(d), Rs(s) {
    distribution = dist;
}
```

⟨*FresnelBlend Private Data*⟩≡

```
Spectrum Rd, Rs;
MicrofacetDistribution *distribution;
```

This model is based on the weighted sum of a glossy specular term and a diffuse term. Accounting for reciprocity and energy conservation, the glossy specular term is derived as

$$f_{\mathrm{r}}(\mathrm{p}, \omega_o, \omega_i) = \frac{D(\omega_h)\, F(\omega_o, \omega_i)}{8\pi\, (\omega_h \cdot \omega_i)\, (\max((\mathbf{n} \cdot \omega_o), (\mathbf{n} \cdot \omega_i)))}$$

where $D(\omega_h)$ is a microfacet distribution term and $F(\omega_o, \omega_i)$ represents Fresnel reflectance. Note that this is quite similar to the Torrance–Sparrow model.

The key to Ashikhmin and Shirley's model was deriving a diffuse term such that the model still obeyed reciprocity and conserved energy. One key to making the derivation practical was using an approximation to the Fresnel reflection equations due to Schlick, who computed Fresnel reflection as

$$F(\cos\theta) = R + (1-R)(1-\cos\theta)^5,$$

where R is the reflectance of the surface at normal incidence.

Given this Fresnel term, they showed that the diffuse term below successfully modeled Fresnel-based reduced diffuse reflection in a physically plausible manner: **sweet jesus, how is this related to the above equation? Explain this.**

$$f_r(\omega_i, \omega_o) = \frac{28R_d}{23\pi}(1-R_s)\left(1-\left(1-\frac{(N\cdot\omega_i)}{2}\right)^5\right)\left(1-\left(1-\frac{(N\cdot\omega_o)}{2}\right)^5\right)$$

⟨FresnelBlend Public Methods⟩$+\equiv$

```
Spectrum SchlickFresnel(Float costheta) const {
    return Rs + powf(1 - costheta, 5.f) * (Spectrum(1.) - Rs);
}
```

⟨BxDF Method Definitions⟩$+\equiv$

```
Spectrum FresnelBlend::f(const Vector &wo, const Vector &wi) const {
    Spectrum diffuse = (28.f/(23.f*M_PI)) * Rd *
        (Spectrum(1.) - Rs) *
        (1 - powf(1 - .5f * fabsf(CosTheta(wi)), 5)) *
        (1 - powf(1 - .5f * fabsf(CosTheta(wo)), 5));
    Vector H = (wi + wo).Hat();
    Spectrum specular = distribution->D(H) /
        (8.f * M_PI * AbsDot(wi, H) *
        max(fabsf(CosTheta(wi)), fabsf(CosTheta(wo)))) *
        SchlickFresnel(Dot(wi, H));
    return diffuse + specular;
}
```

CosTheta()	332
FresnelBlend	365
FresnelBlend::Rd	365
FresnelBlend::Rs	365
M_PI	678
rofacetDistribution::D()	357
Spectrum	181
Vector	27
Vector::Hat()	30

Further Reading

Phong developed and early empirical reflection model for glossy surfaces in computer graphics (Phong 1975). Though not reciprocal or energy-conserving, it was a cornerstone of the first synthetic images of non-Lambertian objects. The Torrance–Sparrow microfacet model is described in (Torrance and Sparrow 1967); a variant of it was applied to computer graphics by Cook and Torrance (Cook and Torrance 1981; Cook and Torrance 1982).

Hall's book collected and described the state of the art in physically-based surface reflection models for graphics in 1989; it remains a seminal reference (Hall 1989). It discussed the physics of surface reflection in detail, with many pointers to the original literature and with many tables of useful measured data about reflection from real surfaces.

Cite wavelength-dependent IOR stuff. Incl Glassner PDIS (Glassner 1995, Section 11.8), Delvin et al survey. Smits, musgrave stuff. McCool stratified wavelength clusters.

Beckmann developed an early physical optics model of surface reflection XXX, which Kajiya used to derive an anisotropic model for computer graphics (Kajiya 1985). Beckmann's work was built upon more recently by He et al (He, Torrance, Sillion, and Greenberg 1991). However, Nayar et al have shown that some reflection models based on physical (wave) optics have substantially the same characteristics as those based on geometric optics–the geometric optics approximations don't seem to cause too much error (except for very smooth surfaces) (Nayar, Ikeuchi, and Kanade 1991). This is a helpful result, giving experimental basis to the general belief that wave optics models aren't usually worth their computational expense for computer graphics applications.

The Oren–Nayar Lambertian model is described in their 1994 SIGGRAPH paper (Oren and Nayar 1994). Other notable BRDF models recently developed in computer graphics include Ward's anisotropic model (Ward 1992) and Hanrahan and Krueger's model of subsurface reflection (Hanrahan and Krueger 1993). Schlick developed an anisotropic model that is both computationally efficient and easy to importance sample for Monte Carlo integration (Schlick 1993). Ashikhmin et al recently developed techniques for computing self-shadowing terms for arbitrary microfacet distributions, without needing to make the assumptions that Torrance and Sparrow did (Ashikhmin, Premoze, and Shirley 2000), though unfortunately their solutions cannot be evaluated in closed form, but must be approximated numerically. Other good references for anisotropic models are Poulin and Fournier's and Lu et al's (Poulin and Fournier 1990; Lu, Koenderink, and Kappers 1999).

Lafortune et al (Lafortune, Foo, Torrance, and Greenberg 1997).

Ashikhmin and Shirley anisotropic model (Ashikhmin and Shirley 2002; Ashikhmin and Shirley 2000)

Kajiya and Kay developed a reflection model for hair based on a model of individual hairs as cylinders with diffuse and glossy reflection properties that determined the overall reflection from these cylinders accounting for the effect of variation in surface normal over the hemisphere along the cylinder (Kajiya and Kay 1989). See also the paper by Banks, which XXX (Banks 1994). More recently, Goldman has developed a probabilistic shading model that models reflection from collections of short hairs (Goldman 1997) and Marschner et al have developed a more accurate of light scattering from long hair (Marschner, Jensen, Cammarano, Worley, and Hanrahan 2003).

A number of researchers have investigated how to find BRDFs based on modeling the small-scale geometric features of a reflective surface. This work includes Cabral et al's computing BRDFs from bump maps (Cabral, Max, and Springmeyer 1987), Fournier's normal distribution functions (Fournier 1992), and Westin et al (Westin, Arvo, and Torrance 1992).

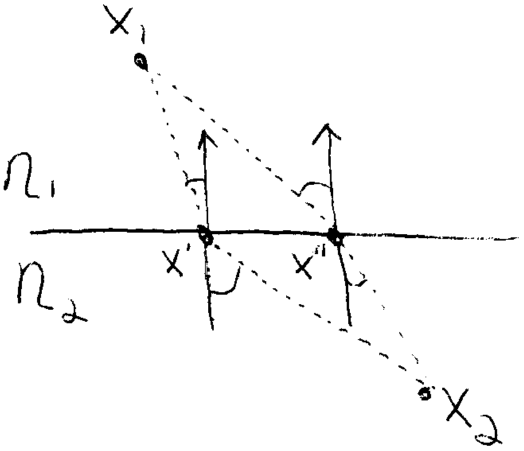


Figure 9.17: Fermat's principle for Snell's law...

Exercises

9.1 simulation: geom and brdf, fire rays at it, tabularize BRDF. isotropic a big win–3d table θ_o, θ_i, $d\phi$...

9.2 Hanrahan–Krueger subsurface stuff.

9.3 A consequence of Fermat's principle from optics is that light traveling from a point x_1 in a medium with index of refraction η_1 to a point x_2 medium with another index of refraction η_2 will follow a path that minimizes the time to get from the first point to the second point. Snell's law can be shown to follow from this fact directly.

Consider light traveling between two points x_1 and x_2 separated by a planar boundary. The light could potentially pass through the boundary while traveling from x_1 to x_2 at any point on the boundary x' (see Figure 9.17, which shows two such possible points x' and x''.) Recall that the time it takes light to travel between two points in a medium with a constant index of refraction is proportional to the distance between them times the index of refraction in the medium. Using this fact, show that the point x' on the boundary that minimizes the total time to travel from x_1 to x_2 is the point where $\eta_1 \sin\theta_1 = \eta_2 \sin\theta_2$.

10. Materials

370 BSDF

The low-level BRDFs and BTDFs introduced in Chapter 9 solve only part of the problem of describing how a surface scatters light. Although they describe how light is scattered at a particular point on the surface, but we still need to know *which* BRDFs and BTDFs describe the scattering at a point, and what the parameters that describe the behavior of these scattering functions.

In this chapter, we provide a general procedural shading mechanism to generate BRDFs and BTDFs for points on surfaces. The basic idea is that a *surface shader* is bound to each primitive in the scene. The surface shader is a small procedure that is executed at a point to be shaded; it returns the BSDF, which holds a collection of BRDFs and BTDFs that describes the scattering at the point. This is a somewhat different shading paradigm than many rendering systems use—most combine the function of the surface shader and the lighting integrator (see Chapter 16) into a single shader. By separating these two pieces, a more flexible system results that is better able to handle new light transport algorithms.

10.1 BSDFs

We now present the implementation of our the general BSDF class. It represents a weighted mixture of BRDFs and BTDFs, allowing the rest of the system to work with composite BSDFs directly, rather than having to consider all of the components they are built from.

Equally important, the BSDF class hides the mechanics of shading normals from the rest of the system. Shading normals, either from per-vertex normals on polygonal meshes, or from bump mapping, can substantially improve the visual richness of scenes. However, because they are an *ad hoc* construct, they are tricky to incorporate into a physically-based renderer. Those issues will all be handled in the BSDF, simplifying other parts of the system.

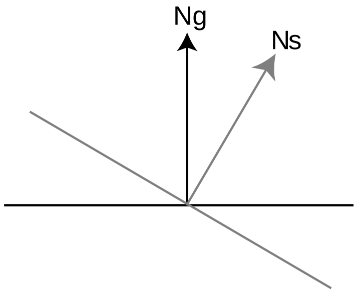


Figure 10.1: The geometric normal, $\mathbf{n}_\mathrm{g}$, defined by the surface geometry, and the shading normal, $\mathbf{n}_\mathrm{s}$, given by per-vertex normals and/or bump mapping will generally specify different hemispheres for integrating incident illumination to compute surface reflection. This inconsistency is important to handle carefully since it can lead to unsightly artifacts in images.

⟨*BSDF Declarations*⟩+≡

```
class BSDF {
public:
    ⟨BSDF Public Methods⟩
    ⟨BSDF Public Data⟩
private:
    ⟨BSDF Private Methods⟩
    ⟨BSDF Private Data⟩
};
```

BSDF::nBxDFs	371
Cross()	31
DifferentialGeometry	58
Normal	34

XXX text needs update XXX

The `BSDF` constructor takes two pieces of `DifferentialGeometry`: `dgS`, is the *shading* differential geometry, where the normal, S, and T vectors may have been modified from the true geometric normal and tangent vectors of the original surface and `dgG`, which represents the true geometric characteristics at the point being shaded–see Figure 10.1. Throughout this section, we will use the convention that $\mathbf{n}_\mathrm{s}$ is the shading normal and $\mathbf{n}_\mathrm{g}$ is the geometric normal.

⟨*BSDF Method Definitions*⟩≡

```
BSDF::BSDF(const DifferentialGeometry &dg, const Normal &ngeom,
        Float e)
    : dgShading(dg), eta(e) {
    ng = ngeom;
    nn = dgShading.nn;
    sn = dgShading.dpdu.Hat();
    tn = Cross(nn, sn);
    nBxDFs = 0;
}
```

⟨*BSDF Public Data*⟩≡

```
const DifferentialGeometry dgShading;
const Float eta;
```

⟨*BSDF Private Data*⟩≡

```
Normal nn, ng;
Vector sn, tn;
int nBxDFs;
#define MAX_BxFS 8
BxDF * bxdfs[MAX_BxFS];
```

⟨*BSDF Inline Method Definitions*⟩≡

```
inline void BSDF::Add(BxDF *b) {
    Assert(nBxDFs < MAX_BxFS);
    bxdfs[nBxDFs++] = b;
}
```

A short utility routine checks to see if the flags for a particular BxDF are compatible with the flags passed in by a user.

⟨*BSDF Inline Method Definitions*⟩+≡

```
inline bool BSDF::MatchesFlags(const BxDF *bxdf,
        BxDFType flags)  const {
    return (bxdf->type & flags) == bxdf->type;
}
```

⟨*BSDF Public Methods*⟩+≡

```
int NumComponents() const { return nBxDFs; }
int NumComponents(BxDFType flags) const {
    int num = 0;
    for (int i = 0; i < nBxDFs; ++i)
        if (MatchesFlags(bxdfs[i], flags)) ++num;
    return num;
}
```

659	Assert()
370	BSDF
334	BxDF
334	BxDF::type
334	BxDFType
58	DifferentialGeometry
30	Dot()
34	Normal
27	Vector

We also provide a transformation to and from the local coordinate system expected by BxDFs (as described in Section 9.1). In this coordinate system, the surface normal is along $(0,0,1)$, the primary tangent is $(1,0,0)$ and the secondary tangent is $(0,1,0)$. This transformation into "shading space" simplified many of the BxDF equations in Chapter 9. These transformations are computed in the same way as the DifferentialGeometry methords for transforming to and from the differential geometry's frame; see Section 2.8 for more information.

The transformation to shading space normalizes the resulting vector, since many BxDF implementations depend on this. However, we don't normalize directions in world space, since there's not a corresponding assumption for world-space rays.

⟨*BSDF Public Methods*⟩+≡

```
Vector WorldToLocal(const Vector &v) const {
    return Vector(Dot(v, sn), Dot(v, tn), Dot(v, nn));
}
```

⟨*BSDF Public Methods*⟩+≡

```
Vector LocalToWorld(const Vector &v) const {
    return Vector(sn.x * v.x + tn.x * v.y + nn.x * v.z,
                  sn.y * v.x + tn.y * v.y + nn.y * v.z,
                  sn.z * v.x + tn.z * v.y + nn.z * v.z);
}
```

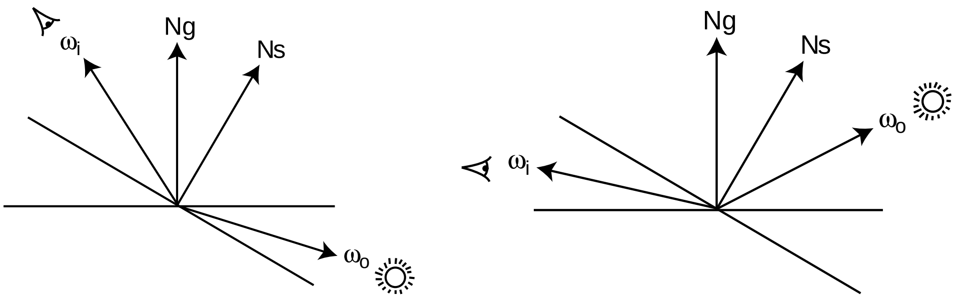


Figure 10.2: The two types of error that result from using shading normals: on the left, a light leak, where the geometric normal indicates that the light is on the backside of the surface, but the shading normal indicates the light is visible (assuming a reflective and not transmissive surface.) On the right is a dark spot, where the geometric normal indicates that the surface is illuminated but the shading normal indicates that the viewer is behind the lit side of the surface.

`BSDF::nn`	371
`BSDF::sn`	371
`BSDF::tn`	371
`BxDF`	334
`Vector`	27

Shading normals can cause a variety of undesirable artifacts in practice–see Figure 10.2. On the left is a *light leak*: the geometric normal indicates that ω_i and ω_o lie on opposide sides of the surface, so if the surface is not transmissive, the light should have no contribution. However, if we directly evaluate the scattering equation 5.4.9 about the hemisphere centered around the shading normal, we will incorrectly incorporate the light from ω_i. Thus, we can see that $\mathbf{n_s}$ can't just be used as a direct replacement for $\mathbf{n_g}$ in rendering computations.

The right side of Figure 10.2 shows a similar situation: the shading normal indicates that no light should be reflected to the viewer, since it is not in the same hemisphere as the illumination, while the geometric normal incidates that they are in the same hemisphere. Direct use of $\mathbf{n_s}$ would cause ugly black spots on the surface where this situation happens.

Fortinately, there is an elegant solution to these problems. When evaluating the BSDF, we use the geometric normal to decide if we should be evaluating reflection or transmission: if ω_i and ω_o lie in the same hemisphere with respect to $\mathbf{n}_g$, we evaluate the BRDFs, and otherwise we evaluate the BTDFs.

Given that convention, recall from Section 9.1 that `BxDF`s in `lrt` should evaluate their values without regard to whether ω_i and ω_o are in the same or are in different hemispheres. Thus, light leaks are avoided, since we only evaluate the BTDFs for the situation in the left side of Figure 10.2, giving us no reflection for a purely reflective surface. Similarly, black spots are avoided since we would evaluate the BRDFs for the situation on the right side of the figure, even though the shading normal thinks that the directions are in different hemispheres. Because the BRDFs evaluate their values in this case, we get a reasonable result.

Given all that, evaluating the BSDF is easy. We just transform the world-space direction vectors to local BSDF space, determine whether we should be using the BRDFs or the BTDFs, and loop over the appropriate set, evaluating a weighted sum of their contributions.

⟨*BSDF Method Definitions*⟩+≡

```
Spectrum BSDF::f(const Vector &woW,
        const Vector &wiW, BxDFType flags) const {
    Vector wi = WorldToLocal(wiW), wo = WorldToLocal(woW);
    Spectrum f = 0.;
    if (Dot(wiW, ng) * Dot(woW, ng) > 0)
        flags = BxDFType(flags & ~BSDF_TRANSMISSION);
    else
        flags = BxDFType(flags & ~BSDF_REFLECTION);
    for (int i = 0; i < nBxDFs; ++i)
        if (MatchesFlags(bxdfs[i], flags))
            f += bxdfs[i]->f(wo, wi);
    return f;
}
```

We'll also provide BSDF methods that sum up the reflectance values of their individual BxDFs; the implementation of these methods are straightforward loops over the BxDFs and won't be shown here.

⟨*BSDF Public Methods*⟩+≡

```
Spectrum rho(BxDFType flags = BSDF_ALL) const;
Spectrum rho(const Vector &wo, BxDFType flags = BSDF_ALL) const;
```

370	BSDF
371	BSDF::bxdfs
371	BSDF::MatchesFlags()
371	BSDF::nBxDFs
371	BSDF::WorldToLocal()
334	BSDF_ALL
334	BSDF_REFLECTION
334	BSDF_TRANSMISSION
334	BxDF
335	BxDF::f()
334	BxDFType
30	Dot()
375	Material
670	MemoryArena
181	Spectrum
563	SurfaceIntegrator
27	Vector

10.1.1 BSDF Memory Management

For each camera ray that intersects geometry in the scene, one or more BSDF objects will be created by the SurfaceIntegrator in the process of computing the reflected radiance from the intersection point. (Integrators that account for multiple inter-reflections of light will generally create a number of BSDFs in this process.) Each of these BSDFs in turn has a number of BRDFs and BTDFs stored inside it, as returned by the Material at the intersection point. A straightforward implementation would use new and delete to dynamically allocate storage for both the BSDF as well as each of the BxDFs that it holds.

Unfortunately, such an approach is relatively inefficient: too much time is spent in the dynamic memory management routines for a series of small memory allocations. Instead, we will use a specialized allocation scheme based on the MemoryArena described in Appendix **??**. The MemoryArena allocates a large block of memory and responds to allocation requests via the MemoryArena::Alloc() call by returning sections of that block from beginning to end. It does not support freeing individual allocations. Instead, the MemoryArena::FreeAll() method is called when *all* of the allocated items are no longer in use. The result is that individual allocations are extremely efficient, and freeing memory is also efficient and done relatively infrequently.

The BSDF class holds a static MemoryArena that will be used for BSDF and BxDF allocations. After each camera ray has been handled, the Scene::Render() method frees up all of the memory used for allocating BSDF memory for that ray. (There should be no BSDF pointers held anywhere in the system at this point.)

The BSDF provides allocation and freeing routines that mirror those in the MemoryArena; the requests are just passed on to the static MemoryArena in the BSDF.

⟨*BSDF Public Methods*⟩+≡

```
static void *Alloc(u_int sz) { return zone.Alloc(sz); }
static void FreeAll() { zone.FreeAll(); }
```

⟨*BSDF Private Data*⟩+≡

```
static MemoryArena zone;
```

For the convenience of code that allocates BSDFs and BxDFs (e.g. the Materials throughout the rest of this chapter), we will provide a macro that hides some of the messiness of using the memory zone approach. Insead of using the new operator to allocate those objects, like this:

```
BSDF *b = new BSDF;
BxDF *lam = new Lambertian(Spectrum(1.0);
```

code should instead be written with the BSDF_ALLOC() macro, like this:

```
BSDF *b = BSDF_ALLOC(BSDF);
BxDF *lam = BSDF_ALLOC(Lambertian)(Spectrum(1.0));
```

The macro calls the BSDF::Alloc() routine to allocate the appropriate amount of memory for the object, and then uses the placement operator new to run the constructor for the object at the given memory location.

BSDF	370
BxDF	334
Material	375
MemoryArena	670
MemoryArena::Alloc()	670
MemoryArena::FreeAll()	671

⟨*BSDF Declarations*⟩+≡

```
#define BSDF_ALLOC(T)  new (BSDF::Alloc(sizeof(T))) T
```

We will make the BSDF destructor a private method, in order to ensure that it isn't inadvertently called. This could lead to subtle errors, since a pointer in the middle memory managed by the MemoryArena would be passed to the system's dynamic allocation freeing routine. We also declare a friend class of a non-existent class; this is just to silence compiler warnings related to having a private destructor.

⟨*BSDF Private Methods*⟩≡

```
~BSDF() { }
friend class NoSuchClass;
```

10.2 Material Interface and Bump Mapping

⟨*material.h**⟩≡

```
#include "lrt.h"
#include "primitive.h"
#include "texture.h"
#include "color.h"
#include "reflection.h"
⟨Material Class Declarations⟩
⟨Material Creation Macros⟩
```

⟨*material.cpp**⟩≡

```
#include "material.h"
⟨Material Method Definitions⟩
```

⟨*Material Class Declarations*⟩≡

```
class Material : public ReferenceCounted {
public:
    ⟨Material Interface⟩
    ⟨Material Public Methods⟩
private:
    ⟨Material Private Data⟩
};
```

There are two main functions that `Material`s are responsible for implementing. The first is a pure virtual function that returns the `BSDF` for a point on a surface represented by its `DifferentialGeometry`. The material is responsible for synthesizing relevant information about the texture and geometric surface properties at the point to generate the scattering function at the point.

⟨*Material Interface*⟩≡

```
virtual BSDF *GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) const = 0;
```

Since the usual interface to the hit point that various `SurfaceIntegrators` is through a `Intersection`, we will also add a convenience method to `Intersection` that returns the `BSDF` at the hit point. It does some setup work, computes the shading differential geometry, and forwards the request on to the `Material`.

⟨*Intersection Method Definitions*⟩+≡

```
BSDF *Intersection::GetBSDF(const RayDifferential &ray) const {
    dg.ComputeDifferentials(ray);
    return primitive->GetBSDF(dg, WorldToObject);
}
```

⟨*GeometricPrimitive Method Definitions*⟩+≡

```
const AreaLight *GeometricPrimitive::GetAreaLight() const {
    return areaLight;
}
```

⟨*GeometricPrimitive Method Definitions*⟩+≡

```
BSDF *GeometricPrimitive::GetBSDF(const DifferentialGeometry &dg,
        const Transform &WorldToObject) const {
    DifferentialGeometry dgs;
    shape->GetShadingGeometry(WorldToObject.GetInverse(),
        dg, &dgs);
    material->Bump(&dgs, dg);
    return material->GetBSDF(dgs, dg.nn);
}
```

491	`AreaLight`
370	`BSDF`
58	`DifferentialGeometry`
400	`DifferentialGeometry::ComputeDif`
132	`GeometricPrimitive`
132	`GeometricPrimitive::material`
131	`Intersection`
131	`Intersection::dg`
131	`Intersection::primitive`
131	`Intersection::WorldToObject`
377	`Material::Bump()`
34	`Normal`
132	`Primitive::GetBSDF()`
37	`RayDifferential`
663	`ReferenceCounted`
67	`Shape::GetShadingGeometry()`
563	`SurfaceIntegrator`
43	`Transform`

10.2.1 Bump mapping

All materials take an optional `Float` texture map that defines a displacement at each point on the surface: each point p has a displaced point p' associated with it, defined by $\mathrm{p}' = \mathrm{p} + d\mathbf{n}(\mathrm{p})$, where d is the offset returned by the displacement texture at p and $\mathbf{n}(\mathrm{p})$ is the surface normal at p–see Figure 10.3. We will use this texture to compute bump-mapped shading normals below, though it could also be used in an implementation of displacement mapping.

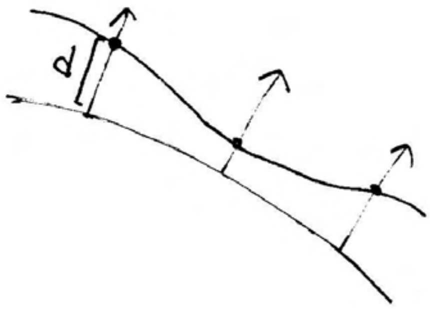


Figure 10.3: The displacement texture associated with each material defines a new surface based on the old one, offset by the displacement amount along the normal at each point. `lrt` doesn't compute a geometric representation of this displaced surface, though it does use it to compute shading normals for bump-mapping.

⟨*Material Public Methods*⟩≡

```
Material(Texture<Float> *disp) {
    displace = disp;
}
```

```
Material  375
 Texture  394
```

⟨*Material Private Data*⟩≡

```
Texture<Float> *displace;
```

The second important `Material` method, `Bump()`, is responsible for computing the effect of bump mapping at the point being shaded. It allows the `Material` to perturb its normal or tangent vectors in order to simulate the effect of rough surfaces or modify the mapping for anisotropy, respectively.

To compute a shading normal at a point, we will evaluate the displacement texture at two auxiliary points next to the current point x: see Figure 10.4. We move to the nearby positions on the tangent plane $\mathrm{p}_\mathrm{x} = \mathrm{p} + dx \times \Delta\mathrm{p}/\Delta x$ and $\mathrm{p}_\mathrm{y} = \mathrm{p} + dy \times \Delta\mathrm{p}/\Delta y$, the intersection points of the auxiliary differential rays, and thus the estimated points one pixel to the side in the x and y directions. By evaluating the displacement at these three points, we compute three points on the displaced surface, p', p'_x and p'_y. The cross product of new tangent vectors $\mathbf{v_x} = \mathrm{p}'_\mathrm{x} - \mathrm{p}$ and $\mathbf{v_y} = \mathrm{p}'_\mathrm{y} - \mathrm{p}$ gives the shading normal $\mathbf{n_s}$.

This approach is based on the assumption that the surface is locally flat around p: if it has a large curvature, then $\mathrm{p} + dx \times \Delta\mathrm{p}/\Delta x$ may be far from the actual surface. This error is rarely a problem in practice.

⟨*Material Method Definitions*⟩+≡

```
void Material::Bump(DifferentialGeometry *dgs,
        const DifferentialGeometry &dgg) const {
    if (displace) {
        ⟨Compute offset positions and evaluate displacement texture⟩
        ⟨Compute bump-mapped differential geometry⟩
    }
    ⟨Orient shading normal to match geometric normal⟩
}
```

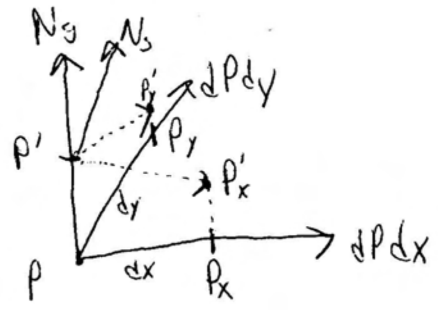


Figure 10.4: To compute the shading normal at a point, we evaluate the displacement texture at that point and at two auxiliary points. By taking the cross product of the vectors from the main point to the auxiliary point, we find the shading normal. The auxiliary points are found by offsetting by the parametric distances *du* and *dv* along the ∂p/∂*u* and ∂p/∂*v* vectors.

Given the offset distances, we use the `DifferentialGeometry::Shift()` method to compute the differential geometry at the auxiliary points. We can then evaluate the displacement texture at the three points and compute the three displaced positions.

58	DifferentialGeometry
58	DifferentialGeometry::nn
58	DifferentialGeometry::p
378	DifferentialGeometry::Shift()
375	Material
376	Material::displace
33	Point
395	Texture::Evaluate()
27	Vector

⟨*Compute offset positions and evaluate displacement texture*⟩≡

```
if (dgs->dudx == 0 && dgs->dvdx == 0 &&
    dgs->dudy == 0 && dgs->dvdy == 0)
    dgs->dudx = dgs->dvdy = .01;
DifferentialGeometry dgdx, dgdy;
dgs->Shift(1, 0, &dgdx);
dgs->Shift(0, 1, &dgdy);
Point p = dgs->p + Vector(dgs->nn) * displace->Evaluate(*dgs);
Point px = dgdx.p + Vector(dgdx.nn) * displace->Evaluate(dgdx);
Point py = dgdy.p + Vector(dgdy.nn) * displace->Evaluate(dgdy);
```

The `DifferentialGeometry::Shift()` method uses the local-flatness assumption mentioned above. New (u, v) coordinates are easily computed based on the (x, y) offset the caller provided. We assume that the partial derivatives, `dpdu` and `dpdv` are the same at the shifted point due to the flatness assumption, though we approximate the new surface normal using `dndu` and `dndv`. We also assume that the Jacobians ∂*u*/∂*x*, ∂*v*/∂*x*, etc., are loally constant so that we can compute `du` by `dx * dudx + dy * dudy`, etc. The new point p, then, is just computed by moving the appropriate distances along ∂p/∂*u* and ∂p/∂*v*.

⟨*DifferentialGeometry Public Methods*⟩+≡

```
void Shift(Float dx, Float dy, DifferentialGeometry *g) const {
    g->p = p + dx * (dudx * dpdu + dvdx * dpdv) +
        dy * (dudy * dpdu + dvdy * dpdv);
    g->u = u + dx * dudx + dy * dudy;
    g->v = v + dx * dvdx + dy * dvdy;
    g->nn = Normal(Cross(dpdu, dpdv) + dx * (dudx * dndu + dvdx * dndv) +
        dy * (dudy * dndu + dvdy * dndv)).Hat();

    g->dpdu = dpdu;
    g->dpdv = dpdv;
    g->dpdx = dpdx;
    g->dpdy = dpdy;
    g->dudx = dudx;
    g->dvdx = dvdx;
    g->dudy = dudy;
    g->dvdy = dvdy;
    g->dndu = dndu;
    g->dndv = dndv;
    g->shape = shape;
}
```

Cross()	31
DifferentialGeometry	58
fferentialGeometry::dndu	58
fferentialGeometry::dndv	58
fferentialGeometry::dpdu	58
fferentialGeometry::dpdv	58
fferentialGeometry::dpdx	400
fferentialGeometry::dpdy	400
fferentialGeometry::dudx	400
fferentialGeometry::dudy	400
fferentialGeometry::dvdx	400
fferentialGeometry::dvdy	400
DifferentialGeometry::nn	58
DifferentialGeometry::p	58
ferentialGeometry::shape	58
DifferentialGeometry::u	58
DifferentialGeometry::v	58
Dot()	30
Normal	34
Vector	27

Given the new positions, we compute partial derivatives with forward differences. This is all that we need to do here; the `DifferentialGeometry` constructor takes care of computing the resulting normal, etc.

⟨*Compute bump-mapped differential geometry*⟩≡

```
Float dx = sqrtf(dgs->dudx*dgs->dudx + dgs->dvdx*dgs->dvdx);
Float dy = sqrtf(dgs->dudy*dgs->dudy + dgs->dvdy*dgs->dvdy);
dgs->UpdateBasis((p-px) / dx, (p-py) / dy);
```

XXX dndu and dndv? XXX

⟨*DifferentialGeometry Method Definitions*⟩+≡

```
void DifferentialGeometry::UpdateBasis(const Vector &DPDU,
        const Vector &DPDV) {
    dpdu = DPDU;
    dpdv = DPDV;
    nn = Normal(Cross(dpdu, dpdv)).Hat();
    ⟨Adjust normal based on orientation and handedness⟩
}
```

Finally, this method flips the shading coordinate frame if needed, so that the shading normal lies in the hemisphere around the geometric normal–the assumption is that the shading normal represents a relatively small perturbation of the geometric normal, so should be in the same hemisphere.

⟨*Orient shading normal to match geometric normal*⟩≡

```
if (Dot(dgg.nn, dgs->nn) < 0)
    dgs->nn = -dgs->nn;
```

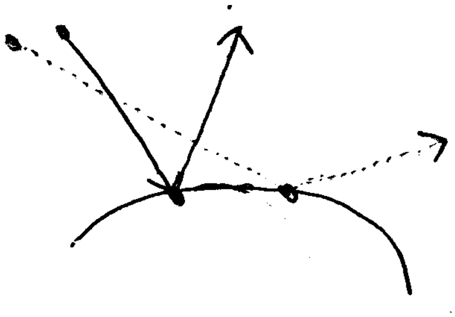


Figure 10.5: The specular reflection formula gives the direction of the reflected ray (solid line) at a point on a surface. An offset ray for a ray differential (dashed line) will generally intersect the surface at a different point and be reflected in a different direction. The new direction is affected by both the different surface normal at the point as well as its different incident direction.

10.2.2 ***ADV***: Ray differentials for specular reflection and transmission

Where should this go???

This is so poorly explained, it's unbelievable. Needs serious editing/rewriting.

36 Ray
37 RayDifferential

`RayDifferential`s were described in Section 1.3.3 as generalizations of `Ray`s that also stored the rays that would be traced at the image samples offset one pixel in the *x* and *y* directions. Section 11.2.1 used these auxiliary rays to estimate the texture-space sampling rate for the image being rendered, which was crucial for texture filtering to reduce aliasing. Given the success of this approach for camera rays, one might want to extend the method to make it possible to determine texture-space sampling rates for objects that are seen indirectly via specular reflection or refraction–objects seen in mirrors, for example, should also not have texture aliasing if possible. Igehy has developed an elegant solution to the problem of how to find the appropriate differential rays for specular reflection and refraction (Igehy 1999) which is the approach used in `lrt` [1].

Given an intersection at a point on a surface that a ray differential hit, we'd like to find an approximation to the reflected or refracted rays that would have been traced at the intersection points for the two offset rays (see Figure 10.5.) The new ray for the main ray is computed by the BSDF, so just the updated rays for the differentials need to be computed.

For both reflection and refraction, the origin of each differential is easily found. The differential geometry at the intersection point stores approximations for how much the surface position changes with respect to changes $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$ on the image plane. Thus, adding these offsets to the intersection point gives the origins for the new rays.

[1] Igehy's formulation is slightly different than the one here–he effectively stored the differences between the main ray and the offset rays, while we store the offset rays explicitly. The mathematics all work out to be the same in the end; we chose this alternative since it makes the algorithm's operation for camera rays easier to understand.

⟨*Compute ray differential* `rd` *for specular reflection*⟩≡

```
RayDifferential rd(p, wi);
rd.hasDifferentials = true;
rd.rx.o = p + isect.dg.dpdx;
rd.ry.o = p + isect.dg.dpdy;
⟨Compute differential reflected directions⟩
```

Finding the directions of these rays is slightly more tricky. Igehy observed that If we knew how much the direction ω_i changed with respect to a shift of a pixel in the x and y directions on the image plane, we could use this to approximate the direction of the offset rays:

$$\omega_x \approx \omega_i + \frac{\partial \omega_i}{\partial x}.$$

For a general world-space surface normal and outgoing direction, the direction for perfect specular reflection is

$$\omega_i = -\omega_o + 2(\omega_o \cdot \mathbf{n})\mathbf{n}.$$

Fortunately, the partial derivatives of this expression are easily computed.

fferentialGeometry::dndx	400
fferentialGeometry::dpdx	400
fferentialGeometry::dpdy	400
Intersection::dg	131
RayDifferential	37
Vector	27

$$\begin{aligned}\frac{\partial \omega_i}{\partial x} &= \frac{\partial}{\partial x}(\omega_i = -\omega_o + 2(\omega_o \cdot \mathbf{n})\mathbf{n}) \\ &= -\frac{\partial \omega_o}{\partial x} + 2((\omega_o \cdot \mathbf{n})\partial \mathbf{n}/\partial x + \frac{\partial(\omega_o \cdot N)}{\partial x}\mathbf{n}).\end{aligned}$$

Using the properties of the dot product, it can be shown that

$$\frac{\partial(\omega_o \cdot N)}{\partial x} = \frac{\partial \omega_o}{\partial x}\mathbf{n} + \omega_o \frac{\partial N}{\partial x}$$

All of the necessary quantities are readily available from the `DifferentialGeometry`, and the implementation is straightforward.

⟨*Compute differential reflected directions*⟩≡

```
const Vector &dndx = bsdf->dgShading.dndx, &dndy = bsdf->dgShading.dndy;
Vector dwodx = -ray.rx.d - wo, dwody = -ray.ry.d - wo;
Float dDNdx = Dot(dwodx, n) + Dot(wo, dndx);
Float dDNdy = Dot(dwody, n) + Dot(wo, dndy);
rd.rx.d = wi - dwodx + 2 * (Dot(wo, n) * dndx + Vector(dDNdx * n));
rd.ry.d = wi - dwody + 2 * (Dot(wo, n) * dndy + Vector(dDNdy * n));
```

A similar process of differentiating the equation for the direction of a specularly transmitted ray gives the equation to find the differential change in transmitted direction. We won't include the derivation or our implementation here, but refer the interested reader to the original paper and to the source code, respectively.

10.3 Matte

```
⟨matte.cpp*⟩≡
  #include "lrt.h"
  #include "material.h"
  ⟨Matte Class Declarations⟩
  ⟨Matte Method Definitions⟩
```

```
⟨Matte Class Declarations⟩≡
  class Matte : public Material {
  public:
      ⟨Matte Public Methods⟩
  private:
      ⟨Matte Private Data⟩
  };
```

The simplest surface is `Matte`. It describes a diffusely-reflecting surface. A `Matte::Kd` texture parameter gives the overall reflectivity of the surface at each point.

```
⟨Matte Public Methods⟩≡
  Matte(Texture<Spectrum> *kd, Texture<Float> *sig,
          Texture<Float> *disp)
      : Material(disp) {
      Kd = kd;
      sigma = sig;
  }
```

370	BSDF
334	BxDF
351	Lambertian
375	Material
181	Spectrum
394	Texture

```
⟨Matte Private Data⟩≡
  Texture<Spectrum> *Kd;
  Texture<Float> *sigma;
```

We need to destroy the `Texture` when the material is deleted. For brevity, we won't include the destructors for the rest of the materials in this chapter.

```
⟨Matte Method Definitions⟩≡
  Matte::~Matte() {
      delete Kd;
      delete sigma;
  }
```

The `BSDF` method just puts the pieces together. The `Matte::Kd` texture is evaluated to compute the `Kd` color at the point being shaded. This is then passed on to create a `Lambertian` `BxDF`, which is returned inside a `BSDF` object.

```
⟨Matte Method Definitions⟩+≡
  BSDF *Matte::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) const
      Spectrum r = Kd->Evaluate(dg);
      Float sig = sigma->Evaluate(dg);
      BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom);
      if (sig == 0.)
          ret->Add(BSDF_ALLOC(Lambertian)(r));
      else
          ret->Add(BSDF_ALLOC(OrenNayar)(r, sig));
      return ret;
  }
```

10.4 Plastic

```
⟨plastic.cpp*⟩≡
  #include "lrt.h"
  #include "material.h"
  ⟨Plastic Class Declarations⟩
  ⟨Plastic Method Definitions⟩
  ⟨Plastic Dynamic Creation Routine⟩
```

```
⟨Plastic Class Declarations⟩≡
  class Plastic : public Material {
  public:
      ⟨Plastic Public Methods⟩
  private:
      ⟨Plastic Private Data⟩
  };
```

A more interesting surface is plastic. Plastic can be modelled as a mixture of a diffuse and glossy scattering function, with appropriate parameters controlling the particular colors and glossiness. The parameters to `Plastic` are two reflectivities, `Kd` and `Ks`, which control how much diffuse reflection there is and how much glossy specular reflection there is. Next is a roughness parameter (which should range from zero to one) that determines the size of the specular highlight; the higher it is, the rougher the surface and the smaller the highlight.

```
⟨Plastic Public Methods⟩+≡
  Plastic(Texture<Spectrum> *kd, Texture<Spectrum> *ks,
          Texture<Float> *rough, Texture<Float> *disp)
          : Material(disp) {
      Kd = kd;
      Ks = ks;
      roughness = rough;
  }
```

```
⟨Plastic Private Data⟩≡
  Texture<Spectrum> *Kd, *Ks;
  Texture<Float> *roughness;
```

`BSDF`	370
`BSDF::Add()`	371
`BSDF_ALLOC`	374
`DifferentialGeometry`	58
`Lambertian`	351
`Material`	375
`Matte`	381
`Matte::Kd`	381
`Matte::sigma`	381
`Normal`	34
`Spectrum`	181
`Texture`	394
`Texture::Evaluate()`	395

⟨*Plastic Method Definitions*⟩+≡

```
BSDF *Plastic::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) const {
    Spectrum kd = Kd->Evaluate(dg);
    BxDF *diff = BSDF_ALLOC(Lambertian)(kd);
    Fresnel *fresnel = BSDF_ALLOC(FresnelDielectric)(1.5f, 1.f);
    Spectrum ks = Ks->Evaluate(dg);
    Float rough = roughness->Evaluate(dg);
    BxDF *spec = BSDF_ALLOC(Microfacet)(ks, fresnel, BSDF_ALLOC(Blinn)(1.f / rough));
    BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom);
    ret->Add(diff);
    ret->Add(spec);
    return ret;
}
```

10.5 Translucent

⟨*translucent.cpp**⟩≡

```
#include "lrt.h"
#include "material.h"
⟨Translucent Class Declarations⟩
⟨Translucent Method Definitions⟩
```

⟨*Translucent Class Declarations*⟩≡

```
class Translucent : public Material {
public:
    ⟨Translucent Public Methods⟩
private:
    ⟨Translucent Private Data⟩
};
```

359	Blinn
370	BSDF
371	BSDF::Add()
374	BSDF_ALLOC
334	BxDF
58	DifferentialGeometry
341	Fresnel
342	FresnelDielectric
351	Lambertian
375	Material
357	Microfacet
34	Normal
382	Plastic
382	Plastic::Kd
382	Plastic::Ks
382	Plastic::roughness
181	Spectrum
395	Texture::Evaluate()

⟨*Translucent Method Definitions*⟩+≡

```
BSDF *Translucent::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom)
    BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom);
    Spectrum r = reflect->Evaluate(dg);
    Spectrum t = transmit->Evaluate(dg);
    if (r.Black() && t.Black()) return ret;

    Spectrum kd = Kd->Evaluate(dg);
    if (!kd.Black()) {
        if (!r.Black()) ret->Add(BSDF_ALLOC(Lambertian)(r * kd));
        if (!t.Black()) ret->Add(BSDF_ALLOC(BRDFToBTDF)(BSDF_ALLOC(Lambertian)(
    }
    Spectrum ks = Ks->Evaluate(dg);
    if (!ks.Black()) {
        Float rough = roughness->Evaluate(dg);
        if (!r.Black()) {
            Fresnel *fresnel = BSDF_ALLOC(FresnelDielectric)(1.5f, 1.f);
            ret->Add(BSDF_ALLOC(Microfacet)(r * ks, fresnel,
                BSDF_ALLOC(Blinn)(1.f / rough)));
        }
        if (!t.Black()) {
            Fresnel *fresnel = BSDF_ALLOC(FresnelDielectric)(1.5f, 1.f);
            ret->Add(BSDF_ALLOC(BRDFToBTDF)(BSDF_ALLOC(Microfacet)(t * ks, fres
                BSDF_ALLOC(Blinn)(1.f / rough))));
        }
    }
    return ret;
}
```

10.6 Glass

⟨*glass.cpp**⟩≡

```
#include "lrt.h"
#include "material.h"
⟨Glass Class Declarations⟩
⟨Glass Method Definitions⟩
```

⟨*Glass Class Declarations*⟩≡

```
class Glass : public Material {
public:
    ⟨Glass Public Methods⟩
private:
    ⟨Glass Private Data⟩
};
```

Another surface shader simulates glass. Nevertheless, a combination of specular reflection and refraction brings us to the heart of recursive ray tracing and can lead to some nifty images. Our parameters include reflection and transmission coefficients as well as the index of refraction of the object.

⟨*Glass Public Methods*⟩+≡

```
Glass(Texture<Spectrum> *r, Texture<Spectrum> *t,
        Texture<Float> *i, Texture<Float> *disp)
        : Material(disp) {
    Kr = r;
    Kt = t;
    index = i;
}
```

⟨*Glass Private Data*⟩≡

```
Texture<Spectrum> *Kr, *Kt;
Texture<Float> *index;
```

As usual, we start by computing new parameters from the primitive's user-supplied values. We then generate a new BSDF that holds reflective and transmissive BRDFs as appropriate given the parameter values.

⟨*Glass Method Definitions*⟩+≡

```
BSDF *Glass::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) const {
    Spectrum R = Kr->Evaluate(dg);
    Spectrum T = Kt->Evaluate(dg);
    Float ior = index->Evaluate(dg);
    BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom, ior);
    if (!R.Black())
        ret->Add(BSDF_ALLOC(SpecularReflection)(R,
            BSDF_ALLOC(FresnelDielectric)(1., ior)));
    if (!T.Black())
        ret->Add(BSDF_ALLOC(SpecularTransmission)(T, 1., ior));
    return ret;
}
```

370	BSDF
371	BSDF::Add()
374	BSDF_ALLOC
58	DifferentialGeometry
342	FresnelDielectric
384	Glass
375	Material
34	Normal
181	Spectrum
345	SpecularReflection
348	SpecularTransmission
394	Texture
395	Texture::Evaluate()

10.7 Mirror

XXX like glass, but just reflection, no fresnel effects

⟨*mirror.cpp**⟩≡

```
#include "lrt.h"
#include "material.h"
⟨Mirror Class Declarations⟩
⟨Mirror Method Definitions⟩
```

⟨*Mirror Class Declarations*⟩≡

```
class Mirror : public Material {
public:
    ⟨Mirror Public Methods⟩
private:
    ⟨Mirror Private Data⟩
};
```

⟨*Mirror Private Data*⟩≡

```
Texture<Spectrum> *Kr;
```

As usual, we start by computing new parameters from the primitive's user-supplied values. We then generate a new `BSDF` that holds reflective and transmissive `BRDFs` as appropriate given the parameter values.

⟨*Mirror Method Definitions*⟩+≡

```
BSDF *Mirror::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) cons
    Spectrum R = Kr->Evaluate(dg);
    BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom);
    if (!R.Black())
        ret->Add(BSDF_ALLOC(SpecularReflection)(R,
            BSDF_ALLOC(FresnelNoOp)()));
    return ret;
}
```

10.8 Shiny Metal

⟨*shinymetal.cpp**⟩≡

```
#include "lrt.h"
#include "material.h"
⟨ShinyMetal Class Declarations⟩
⟨ShinyMetal Method Definitions⟩
```

BSDF	370
BSDF::Add()	371
BSDF_ALLOC	374
DifferentialGeometry	58
FresnelNoOp	344
Material	375
Mirror	385
Normal	34
Spectrum	181
SpecularReflection	345
Texture	394
Texture::Evaluate()	395

⟨*ShinyMetal Class Declarations*⟩≡

```
class ShinyMetal : public Material {
public:
    ⟨ShinyMetal Public Methods⟩
private:
    ⟨ShinyMetal Private Data⟩
};
```

Another basic combination of scattering functions gives us something that looks like a shiny metal surface. We have both a glossy specular reflection, with reflectance `Ks`, and perfect mirror specular reflection, with reflectance `Kr`.

⟨*ShinyMetal Public Methods*⟩+≡

```
ShinyMetal(Texture<Spectrum> *ks, Texture<Float> *rough,
        Texture<Spectrum> *kr, Texture<Float> *disp)
        : Material(disp) {
    Ks = ks;
    roughness = rough;
    Kr = kr;
}
```

⟨*ShinyMetal Private Data*⟩≡

```
Texture<Spectrum> *Ks, *Kr;
Texture<Float> *roughness;
```

⟨*ShinyMetal Method Definitions*⟩+≡

```
BSDF *ShinyMetal::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) const {
    Spectrum spec = Ks->Evaluate(dg);
    Float rough = roughness->Evaluate(dg);
    Spectrum R = Kr->Evaluate(dg);

    MicrofacetDistribution *md = BSDF_ALLOC(Blinn)(1.f / rough);
    Spectrum k = 0.;
    Fresnel *frMf = BSDF_ALLOC(FresnelConductor)(FresnelApproxEta(spec), k);
    Fresnel *frSr = BSDF_ALLOC(FresnelConductor)(FresnelApproxEta(R), k);
    BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom);
    ret->Add(BSDF_ALLOC(Microfacet)(1., frMf, md));
    ret->Add(BSDF_ALLOC(SpecularReflection)(1., frSr));
    return ret;
}
```

10.9 Diffuse Substrate

XXX need a better name

⟨*substrate.cpp**⟩≡

```
#include "lrt.h"
#include "material.h"
⟨Substrate Class Declarations⟩
⟨Substrate Method Definitions⟩
```

⟨*Substrate Class Declarations*⟩≡

```
class Substrate : public Material {
public:
    ⟨Substrate Public Methods⟩
private:
    ⟨Substrate Private Data⟩
};
```

359	Blinn
370	BSDF
371	BSDF::Add()
374	BSDF_ALLOC
58	DifferentialGeometry
341	Fresnel
341	FresnelApproxEta()
375	Material
357	Microfacet
357	MicrofacetDistribution
34	Normal
386	ShinyMetal
386	ShinyMetal::Kr
386	ShinyMetal::Ks
386	ShinyMetal::roughness
181	Spectrum
345	SpecularReflection
395	Texture::Evaluate()

A reasonably good model of glossy paint can be constructed using some of the pieces we have put together so far. There are two main types of light reflection with glossy paint: some of the incident light is specularly reflected at the surface, and the rest is transmitted into a substrate with suspended colored particles–see Figure 10.6. The transmitted light is interacts with the particles, and some wavelengths of light are absorbed, based on the particle color. The remaining light eventually exits.

If we make the assumption that the exiting light exits in random directions, reflection from the substrate can be modeled with a Lambertian BRDF. We will use the Fresnel formula for dielectrics to determine how much light is reflected and how much is transmitted, giving us weighting terms for the specular reflection and the body reflection BRDFs.

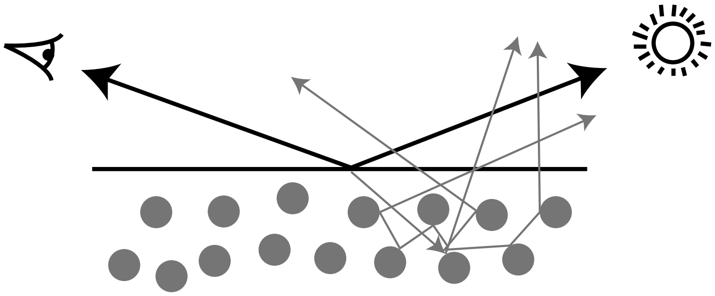


Figure 10.6:

⟨*Substrate Public Methods*⟩≡
```
Substrate(Texture<Spectrum> *kd, Texture<Spectrum> *ks,
        Texture<Float> *u, Texture<Float> *v, Texture<Float> *disp)
    : Material(disp) {
    Kd = kd;
    Ks = ks;
    nu = u;
    nv = v;
}
```

Anisotropic	361
BSDF	370
BSDF::Add()	371
BSDF_ALLOC	374
DifferentialGeometry	58
FresnelBlend	365
Material	375
Normal	34
Spectrum	181
Substrate	387
Texture	394
Texture::Evaluate()	395

⟨*Substrate Private Data*⟩≡
```
Texture<Spectrum> *Kd, *Ks;
Texture<Float> *nu, *nv;
```

⟨*Substrate Method Definitions*⟩+≡
```
BSDF *Substrate::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) c
    Spectrum d = Kd->Evaluate(dg);
    Spectrum s = Ks->Evaluate(dg);
    Float u = nu->Evaluate(dg);
    Float v = nv->Evaluate(dg);

    BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom);
    ret->Add(BSDF_ALLOC(FresnelBlend)(d, s, BSDF_ALLOC(Anisotropic)(1.f/u, 1.f/
    return ret;
}
```

10.10 Measured Data

⟨*clay.cpp**⟩≡
```
#include "lrt.h"
#include "material.h"
⟨Clay Class Declarations⟩
⟨Clay Method Definitions⟩
```

Cornell Program of Computer Graphics...

⟨*Clay Class Declarations*⟩≡

```
class Clay : public Material {
public:
    Clay(Texture<Float> *disp) : Material(disp) { }
};
```

⟨*Clay Method Definitions*⟩≡

```
BSDF *Clay::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) const {
    ⟨Declare clay coefficients⟩
    BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom);
    ret->Add(BSDF_ALLOC(Lafortune)(Spectrum(diffuse), 3, xy, xy, z, e,
        BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE)));
    return ret;
}
```

⟨*Declare clay coefficients*⟩≡

```
static Float diffuse[3] = {   0.383626f,   0.260749f,   0.274207f };
static Float xy0[3] =     {  -1.089701f,  -1.102701f,  -1.107603f };
static Float z0[3] =      {  -1.354682f,  -2.714801f,  -1.569866f };
static Float e0[3] =      {  17.968505f,  11.024489f,  21.270282f };
static Float xy1[3] =     {  -0.733381f,  -0.793320f,  -0.848206f };
static Float z1[3] =      {   0.676108f,   0.679314f,   0.726031f };
static Float e1[3] =      {   8.219745f,   9.055139f,  11.261951f };
static Float xy2[3] =     {  -1.010548f,  -1.012378f,  -1.011263f };
static Float z2[3] =      {   0.910783f,   0.885239f,   0.892451f };
static Float e2[3] =      { 152.912795f, 141.937171f, 201.046802f };
static Spectrum xy[3] = { Spectrum(xy0), Spectrum(xy1), Spectrum(xy2) };
static Spectrum z[3] = { Spectrum(z0), Spectrum(z1), Spectrum(z2) };
static Spectrum e[3] = { Spectrum(e0), Spectrum(e1), Spectrum(e2) };
```

370	BSDF
371	BSDF::Add()
374	BSDF_ALLOC
334	BSDF_DIFFUSE
334	BSDF_REFLECTION
334	BxDFType
58	DifferentialGeometry
363	Lafortune
375	Material
34	Normal
181	Spectrum
394	Texture

Will ifdraft felt, primer, skin... , Felt, , Primer, , Skin, , BluePaint, , BrushedMetal.

10.11 Uber Material

```
⟨uber.cpp*⟩≡
  #include "lrt.h"
  #include "material.h"
  ⟨UberMaterial Class Declarations⟩
  ⟨UberMaterial Method Definitions⟩
  ⟨UberMaterial Dynamic Creation Routine⟩
```

```
⟨UberMaterial Class Declarations⟩≡
  class UberMaterial : public Material {
  public:
      // UberMaterial Method Declarations
      ~UberMaterial();
      UberMaterial(Texture<Spectrum> *kd, Texture<Spectrum> *ks,
             Texture<Float> *rough, Texture<Spectrum> *op,
             Texture<Float> *disp)
              : Material(disp) {
          Kd = kd;
          Ks = ks;
          roughness = rough;
          opacity = op;
      }
      BSDF *GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) const;
  private:
      // UberMaterial Private Data
      Texture<Spectrum> *Kd, *Ks, *opacity;
      Texture<Float> *roughness;
  };
```

BSDF	370
DifferentialGeometry	58
Material	375
Normal	34
Spectrum	181
Texture	394

```
⟨UberMaterial Method Definitions⟩≡
  UberMaterial::~UberMaterial() {
      delete Kd;
      delete Ks;
      delete roughness;
      delete opacity;
  }
```

⟨*UberMaterial Method Definitions*⟩+≡

```
BSDF *UberMaterial::GetBSDF(const DifferentialGeometry &dg, const Normal &ngeom) const {
    BSDF *ret = BSDF_ALLOC(BSDF)(dg, ngeom);
    Spectrum kd = Kd->Evaluate(dg);
    if (!kd.Black()) {
        BxDF *diff = BSDF_ALLOC(Lambertian)(kd);
        ret->Add(diff);
    }

    Spectrum ks = Ks->Evaluate(dg);
    if (!ks.Black()) {
        Fresnel *fresnel = BSDF_ALLOC(FresnelDielectric)(1.5f, 1.f);
        Float rough = roughness->Evaluate(dg);
        BxDF *spec = BSDF_ALLOC(Microfacet)(ks, fresnel, BSDF_ALLOC(Blinn)(1.f / rough));
        ret->Add(spec);
    }

#if 0
    Spectrum op = opacity->Evaluate(dg);
    if (op != Spectrum(1.)) {
        BxDF *tr = BSDF_ALLOC(SpecularTransmission)(-op + Spectrum(1.), 1., 1.);
        ret->Add(tr);
    }
#endif
    return ret;
}
```

359	Blinn
370	BSDF
374	BSDF_ALLOC
334	BxDF
58	DifferentialGeometry
341	Fresnel
342	FresnelDielectric
351	Lambertian
357	Microfacet
34	Normal
181	Spectrum
348	SpecularTransmission

Further Reading

Phong and Crow first introduced the idea of interpolating per-vertex shading normals to give the appearence of smooth surfaces from polygonal meshes (Phong and Crow 1975). Blinn later developed the bump-mapping technique to give the appearence of geometric complexity on coarse meshes (Blinn 1978).

Snyder and Barr noted the light leak problem from per-vertex shading normals and proposed a number of work-arounds (Snyder and Barr 1987). The method we have used in this chapter is from Veach's thesis (Veach 1997, Section 5.3); it is a more robust solution than those of Snyder and Barr.

Kajiya generalized the idea of bump mapping the normal to *frame mapping* (Kajiya 1985).

Shading normals introduce a number of subtle problems to physically-based light transport algorithms that we have not addressed here. For example, they can easily lead to surfaces that reflect more energy than was incident upon them, which can wreak havoc with light transport algorithms. Veach has investigated this issue in depth and proposed a number of solutions (Veach 1996).

Amanatides's cone tracing method (Amanatides 1984) and Heckbert and Hanrahan (Heckbert and Hanrahan 1984) were the first to extend ray tracing to incorporate an area associated with each image sample, rather than just an infinitessimal ray.

Ray differentials (Igehy 1999). Extended by Suykens and Willems to handle glossy reflection as well (Suykens and Willems 2001). See also Turkowski's technical report (Turkowski 1993). Also Shinya et al (Shinya, Takahashi, and Naito 1987), and Mitchell and Hanrahan (Mitchell and Hanrahan 1992). Gritz and Hahn (Gritz and Hahn 1996), though theirs doesn't get good anisotropic filter regions and doesn't account for the variation in angle that the separation between adjacent pixels subtends as you go from the center to the edges of the image plane. Collins estimated ray footprint by keeping tree of all rays traced from a given eye ray, examining corresponding rays at the same level and position (Collins 1994).

Displacement mapping (Cook 1984; Cook, Carpenter, and Catmull 1987). Dice stuff up (Pharr and Hanrahan 1996). Inverse displacement mapping (Patterson, Hoggar, and Logie 1991; Logie and Patterson 1994). Wang adaptive stuff (Wang, Maillot, Fiume, Ng-Thow-Hing, Woo, and Bakshi 2000). Smits et al (Smits, Shirley, and Stark 2000). Heidrich and Seidel (Heidrich and Seidel 1998).

Dana et al BTF stuff (Dana, van Ginneken, Nayar, and Koenderink 1999).

Gondek et al investigated reflection from glossy painted surfaces (Gondek, Meyer, and Newman 1994); some of the observations from their paper influenced the *ad-hoc* paint material model introduced here.

Lafortune coefficients from measurements taken for Marschner at al paper, included here with permission of the Program of Computer Graphics, Cornell University (Marschner, Westin, Lafortune, Torrance, and Greenberg 1999).

BSDF 370
BxDF 334
DifferentialGeometry 58

Exercises

10.1 One form of aliasing that `lrt` doesn't make efforts to eliminate is specular highlight aliasing. Glossy specular surfaces with high specular exponents, particularly if they have high curvature, are succeptable to aliasing as small changes in incident direction or surface position (and thus surface normal) may cause the highlight's contribution to change substantially. Read Amanatides's paper on this topic (Amanatides 1992) and extend `lrt` to reduce specular aliasing, either using his technique or by developing your own. Most of the quantities needed are already available—$\partial\mathbf{n}/\partial x$ and $\partial\mathbf{n}/\partial y$ in the `DifferentialGeometry`, etc., though it will be necessary to extent the `BxDF` interface to give more information about specular exponents for glossy specular reflection components.

10.2 Another approach to specular highlight aliasing is to super-sample the BSDF, evaluating it multiple times around the point being shaded. Modify the `BSDF` class to use the `DifferentialGeometry::Shift()` method to move to a set of positions within the pixel sampling rate around the intersection point and evaluate the BSDF at each one of them when the BSDF evaluation routines are called. How well does this combat specular highlight aliasing?

11. Texture

We will now describe a set of interfaces and classes that allow us to incorporate *texture* into our material models. All of the materials in Chapter 10 have a few parameters that describe their characteristics (diffuse reflectance, glossiness, etc.). Because properties of realistic materials typically vary over the surface, we need an abstractions for describing this variation, and the code in this chapter addresses this problem. By separating the pattern generation methods here from the material descriptions in the previous chapter, it is easy to mix and match, thereby creating a wide variety of appearances.

In graphics, the techniques used to compute these varying surface parameters fall under the area of *texturing*. In `lrt`, a texture is extremely general; it is a function that maps positions in some domain (typically either (s,t) parametric space or (x,y,z) object space) to some other domain (typically spectra, vectors, or the real numbers). We support a variety of mappings; for surfaces with no variation in some parameter, we support zero-dimensional functions that always return a constant. For more typical variation on a surface, we support two-dimensional functions of (s,t) parameter values on a surface. Finally, for variation in space, we provide three-dimensional functions (e.g. of position in the scene). The arguments to these texture functions are generally referred to as *texture coordinates*. This chapter will include all three types of textures. Two-dimensional image maps are a well-known type of texturing–they are incorporated into our texturing framework in Section 11.5.

Texture functions may themselves be a source of high-frequency variation in the image function. See Figure 11.1, which shows an severly aliased image of a checkerboard on a plane. At the horizon, the number of checks between a pair of pixels is very large–the middle of that figure shows a close-up view of four pixels

Figure 11.1: Texture aliasing: on the left is an image of a checkerboard texture with one sample per pixel–it has severe aliasing artifacts at the horizon. The middle image shows a zoomed-in area from near the horizon, which gives a sense of how much high frequency detail is present between adjacent pixel sample positions. On the right, the texture function has taken into account the image sampling rate to prefilter the checkerboard function and remove high frequency detail, resulting in an anti-aliased image, even with a single sample per pixel.

DifferentialGeometry 58

at the horizon. Although the visual impact of this aliasing is reduced with the non-uniform sampling techniques from Chapter 7, a better solution to this problem is to implement texture functions that are aware of their frequency content so that they can reduce high frequencies based on the rate at which they are being sampled. For many texture functions, computing their frequency content isn't too difficult and is substantially more efficient than increasing the image sampling rate and tracing more rays into the scene.

The first section of this chapter will describe the basic texture interface and illustrate its use with a few basic texture functions. We will then discuss general approaches to texture anti-aliasing. Finally, we will present more complex texture implementations, showing a number of texture anti-aliasing techniques along the way.

11.1 Texture Interface and Basic Textures

`Texture` is a template class parameterized by the return type of its evaluation function. This allows us to reuse almost all of the texturing code between textures that return different types (`float`s, spectra, vectors, normals, etc.).

⟨*Texture Class Declarations*⟩≡

```
template <class T> class Texture {
public:
    ⟨Texture Interface⟩
};
```

The key to `Texture`'s interface is its evaluation function; it returns a value of the template type `T`. It has access to the `DifferentialGeometry` at the point being shaded; different textures in this chapter will use different parts of this structure to perform their evaluation.

⟨*Texture Interface*⟩≡

```
virtual T Evaluate(const DifferentialGeometry &) const = 0;
```

11.1.1 Constant Texture

ConstantTexture returns the same value no matter where it is evaluated. Because this represents a pure DC signal, it can be properly reconstructed with any sampling rate or pattern, and therefore needs no anti-aliasing. Although this texture appears trivial, it is actually incredibly useful. By providing this class, all parameters to all materials can be represented as a Texture, whether they are spatially varying or not. For example, a red diffuse object will have a ConstantTexture that always returns red as the diffuse color of the material. This way, the shading system will always evaluate a texture to get the surface color at a point, avoiding the need for separate textured and non-textured versions of materials.

⟨*Texture Class Declarations*⟩+≡

```
template <class T>
class ConstantTexture: public Texture<T> {
public:
    ConstantTexture(const T &v) { value = v; }
    T Evaluate(const DifferentialGeometry &) const;
private:
    T value;
};
```

58 DifferentialGeometry
394 Texture

⟨*Texture Template Method Definitions*⟩≡

```
template <class T>
T ConstantTexture<T>::Evaluate(
        const DifferentialGeometry &) const {
    return value;
}
```

11.1.2 Scale Texture

We have defined the texture interface in a way that makes it easy to use the output of one texture function when computing another. This is useful since it lets us define generic texture operations using any of the other texture types we have. The ScaleTexure takes two textures, a base map and a scale, and returns the product of their values when evaluated. This texture can also ignore anti-aliasing, leaving it to its members to handle.

⟨*Texture Class Declarations*⟩+≡

```
template <class T1, class T2>
class ScaleTexture : public Texture<T2> {
public:
    ⟨ScaleTexture Public Methods⟩
private:
    Texture<T1> *scale;
    Texture<T2> *value;
};
```

⟨*ScaleTexture Public Methods*⟩≡

```
ScaleTexture(Texture<T1> *s, Texture<T2> *v) {
    scale = s;
    value = v;
}
```

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T1, class T2>
T2 ScaleTexture<T1, T2>::Evaluate(
        const DifferentialGeometry &dg) const {
    return scale->Evaluate(dg) * value->Evaluate(dg);
}
```

We need to delete the child textures used by `ScaleTexture`s when they are deleted. We won't show the destructors for the rest of the textures in this chapter; if they hold pointers to other textures they will delete them in their destructors.

⟨*ScaleTexture Public Methods*⟩+≡

```
~ScaleTexture() {
    delete scale;
    delete value;
}
```

`ConstantTexture`	395
`DifferentialGeometry`	58
`Texture`	394
`Texture::Evaluate()`	395

11.1.3 Mix Textures

The `MixTexture` class is a more general variation of `ScaleTexture`. It takes three textures as input: two may be of any type, and the third must return a floating point value. The floating point texture is then used to linearly interpolate between the two other textures. Note that we can use a `ConstantTexture` for the floating point values to achieve a uniform blend, or a more complex `Texture` to blend in a spatially nonuniform way.

⟨*Texture Class Declarations*⟩+≡

```
template <class T>
class MixTexture : public Texture<T> {
public:
    ⟨MixTexture Public Methods⟩
private:
    Texture<T> *tex1, *tex2;
    Texture<Float> *amount;
};
```

⟨*MixTexture Public Methods*⟩≡

```
MixTexture(Texture<T> *t1, Texture<T> *t2, Texture<Float> *amt) {
    tex1 = t1;
    tex2 = t2;
    amount = amt;
}
```

To evaluate the mixture, we just evaluate the three textures and use the floating point value to linearly interpolate between the two. When the blend amount `amt` is zero, the first texture's value is returned and when it is one, the second one's value is returned. We will generally assume that `amt` will be between zero and one, but this behavior is not enforced, so texture extrapolation is possible.

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T>
T MixTexture<T>::Evaluate(
        const DifferentialGeometry &dg) const {
    T t1 = tex1->Evaluate(dg), t2 = tex2->Evaluate(dg);
    Float amt = amount->Evaluate(dg);
    return (1. - amt) * t1 + amt * t2;
}
```

11.2 Sampling and Anti-Aliasing

The sampling task in chapter 7 can be frustrating since the aliasing problem is known to be unsolvable from the start. The infinite frequency content of geometric edges and hard shadows *guarantees* aliasing in our images, regardless of the image sampling rate. Fortunately, for textures things are not so hopeless. We often either have a convenient analytic form of the texture function available, making it possible to remove excessively high frequencies before sampling it, or we can be careful when we evaluate the function so as to not introduce high frequencies in the first place. Doing the extra work to anti-alias the texture functions themselves is much more computationally efficient than increasing the pixel sampling rate to reduce texture aliasing and is a far more elegant solution to this problem. When complex surface detail is represented by a texture, we can often render an almost completely alias-free image with just a single sample per pixel.

```
58 DifferentialGeometry
396 MixTexture
395 Texture::Evaluate()
```

We need to address two problems in order to anti-alias textures:

1. Determine what the sampling rate is in texture space. The screen space sampling rate is known from the image resolution and pixel sampling rate, but here we need to determine what the resulting sampling rate is on a surface in the scene, and from that determine the rate at which the texture function is being sampled.

2. Given the texture sampling rate, we need to apply sampling theory (e.g. by removing excess frequencies beyond the Nyquist limit from the texture function) in order to return a texture value that doesn't have high-frequency variation.

These two issues will be addressed in the next two sub-sections.

11.2.1 Computing the texture-space sampling rate

Figure 11.2 shows the problem we face: texture coordinates (s,t) have been assigned to an object in the scene, such that at every point on the surface, the (s,t) coordinates at that point can be computed. For example, we might just compute (s,t) directly from the (u,v) coordinates of the point on a parametric surface. Given a particular point being shaded with some (s,t) texture coordinates, we need to

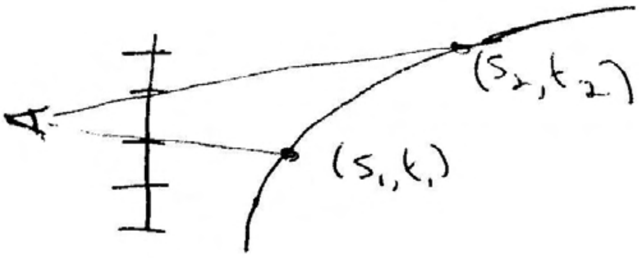


Figure 11.2: Given a set of image samples (here denoted by hash marks on the image plane) at some sampling rate, the 3D scene is sampled at some other rate by the camera rays. If our texture function depends on 2D surface coordinates (s,t), then the sampling rate in (s,t) can be approximated by the change in sample values $((s_1-s_2)/2,(t_1-t_2)/2)$ between adjacent image sample locations. This texture sampling rate is difficult to compute exactly: it is likely to be varying from sample to sample on the image plane, and it depends on the image sampling rate, the camera imaging model, the geometry of the 3D shape, and the association of (s,t) coordinates with 3D points on the object.

Figure 11.3: different sampling rates in x and y

estimate the local sampling rate at the point. If the two adjacent image smaples intersected the same surface and we know knew their texture coordinates (s_1,t_1) and (s_2,t_2), then we could use these points to approximate the local texture sampling rate as $((s_1-s_2)/2,(t_1-t_2)/2)$.

Since we are creating a 2D image, we both need to consider the texture sampling rate as we vary our position in both the x and y directions on the image pane. Figure 11.3 shows an example where adjacent samples in the x direction lead to a high sampling rate in texture space, while adjacent samples in y lead to a relatively low rate.

An alternate approach is to analytically compute the partial derivatives of the texture parameterization function in terms of image coordinates, $\partial s/\partial x$, $\partial t/\partial x$, $\partial s/\partial y$, and $\partial t/\partial y$. These derivatives also give a first-order approximation that we could use to compute the sampling rate. Though this approach is the basis of texture anti-aliasing in most triangle scan-conversion based renderers, it can be difficult to extend to general curved geometry and non-linear camera projections.

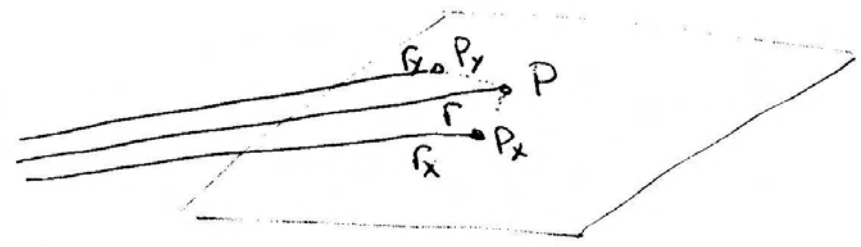


Figure 11.4: rays intersecting the tangent plane lets us approximate the relevant variations...

For a two-dimensional image, there are four such differential estimates, one pair for the adjacent image sample in the x direction and the other for y. We will denote these approximations to the differential change in texture coordinate (s,t) as a function of image position (x,y) by $\partial s/\partial x$, $\partial t/\partial x$, $\partial s/\partial y$, and $\partial t/\partial y$, and by `dsdx`, etc., in source code.

Some assumptions and approximations need to be made in order to make the problem tractable. It is far better to make these assumptions and do some form of anti-aliasing than to give up and just increase the image sampling rate to reduce aliasing, since additional camera rays are very expensive. The key to our calculations lies in the `RayDifferential` structure, which was defined long ago in Section 2.4.1. This structure is initialized in the `Scene::Render()` function, and contains not only the ray actually being traced through the scene, but also two *offset rays*, one offset horizontally one pixel from the camera ray and the other offset vertically by one pixel.

36 `Ray`
37 `RayDifferential`

All of the geometric ray intersection routines only use the main camera ray for their computations; the auxiliary rays are ignored (this is trivially accomplished because `RayDifferential` is a subclass of `Ray`). Once we find an intersection and are evaluating textures, however, we use the offset rays to estimate the local texture sampling rate. The key to this estimate is that we assume the surface is locally flat with respect to the sampling rate at the point being shaded. This is a reasonable approximation in practice, and it is hard to do much better; since ray tracing is a point-sampling technique, we have no additional information about the scene in between the rays we traced.

Given this approximation, we compute the plane through the point intersected by the main ray and tangent to the surface there. This plane is given by the implicit plane equation

$$ax + by + cz + d = 0,$$

where $a = \mathbf{n}_x$, $b = \mathbf{n}_y$, $c = \mathbf{n}_z$, and $d = -(\mathbf{n} \cdot p)$.

Next, we compute the intersection of the auxiliary rays r_x and r_y with this plane (Figure 11.4). Given their hit positions, we would like the find the amount of variation in position on the surface and variation in parametric (u,v) coordinates between adjacent camera ray samples. These give us the sampling rate in texture parameter space, which individual textures can use to determine their maximum allowed frequency content.

HUMPER STOPPED EDITING HERE

XXXX should this be moved back to shapes, rolled into constructor there?? XXXX

⟨*DifferentialGeometry Public Data*⟩+≡

```
mutable Vector dpdx, dpdy;
mutable Vector dndx, dndy;
mutable Float dudx, dvdx, dudy, dvdy;
```

⟨*Initialize* `DifferentialGeometry` *from parameters*⟩+≡

```
dudx = dvdx = dudy = dvdy = 0;
```

⟨*DifferentialGeometry Method Definitions*⟩+≡

```
DifferentialGeometry::DifferentialGeometry(const Point &P,
        const Vector &DPDU, const Vector &DPDV, const Vector &DNDU,
        const Vector &DNDV, Float uu, Float vv,
        const Shape *sh, Float DUDX, Float DVDX,
        Float DUDY, Float DVDY)
    : p(P), dpdu(DPDU), dpdv(DPDV), dndu(DNDU), dndv(DNDV) {
    ⟨Initialize DifferentialGeometry from parameters⟩
    ⟨Adjust normal based on orientation and handedness⟩
    dudx = DUDX;
    dvdx = DVDX;
    dudy = DUDY;
    dvdy = DVDY;
    dndx = dndu * dudx + dndv * dvdx;
    dndy = dndu * dudy + dndv * dvdy;
    dpdx = dpdu * dudx + dpdv * dvdx;
    dpdy = dpdu * dudy + dpdv * dvdy;
}
```

DifferentialGeometry	58
fferentialGeometry::dndu	58
fferentialGeometry::dndv	58
DifferentialGeometry::p	58
Point	33
RayDifferential	37
ential::hasDifferentials	38
Shape	63
Vector	27

⟨*DifferentialGeometry Method Definitions*⟩+≡

```
void DifferentialGeometry::ComputeDifferentials(
        const RayDifferential &ray) const {
    if (ray.hasDifferentials) {
        ⟨Estimate screen-space change in p, n, and (u,v)⟩
    }
    else {
        dudx = dvdx = 0.;
        dudy = dvdy = 0.;
        dpdx = dpdy = Vector(0,0,0);
    }
}
```

⟨*Estimate screen-space change in p, n, and* (u,v)⟩≡

```
⟨Compute auxiliary intersection points with plane⟩
dpdx = px - p;
dpdy = py - p;
⟨Compute (u,v) offsets at auxiliary points⟩
dndx = dndu * dudx + dndv * dvdx;
dndy = dndu * dudy + dndv * dvdy;
```

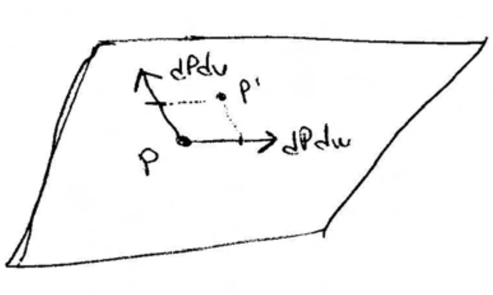


Figure 11.5: computing the change in (u,v) using the position of a point with respect to the tangent frame coordinate system.

Given their hit positions, we approximate the positions $\mathrm{p_x}$ and $\mathrm{p_y}$ on the surface with the intersection locations on the tangent plane. The ray–plane intersection algorithm says that if a ray is described by origin o and direction **d**, then the *t* value where it intersects a plane described by $ax+by+cz+d=0$ is

$$t = \frac{-((a,b,c)\cdot \mathrm{p})+d}{(a,b,c)\cdot \mathbf{d}}$$

To compute this value for the two auxiliary rays, we first compute the plane's *d* coefficient. We don't need to compute the *a*, *b*, and *c* coefficients, since they're just in `dg.nn`. We can then apply this formula directly.

⟨*Compute auxiliary intersection points with plane*⟩≡

```
  Float D = -Dot(nn, Vector(p.x, p.y, p.z));
  Vector rxv(ray.rx.o.x, ray.rx.o.y, ray.rx.o.z);
  Float tx = -(Dot(nn, rxv) + D) / Dot(nn, ray.rx.d);
  Point px = ray.rx.o + tx * ray.rx.d;
  Vector ryv(ray.ry.o.x, ray.ry.o.y, ray.ry.o.z);
  Float ty = -(Dot(nn, ryv) + D) / Dot(nn, ray.ry.d);
  Point py = ray.ry.o + ty * ray.ry.d;
```

58	`DifferentialGeometry::nn`
30	`Dot()`
33	`Point`
35	`Ray::o`
38	`RayDifferential::rx`
38	`RayDifferential::ry`
27	`Vector`

We compute their parametric (u,v) coordinates by taking advantage of the face that the surface's $\partial\mathrm{p}/\partial u$ and $\partial\mathrm{p}/\partial v$ form a (not-necessarily orthogonal) coordinate system on the plane and that the coordinates of the auxiliary intersection points in terms of this coordinate system are their coordinates with respect to the (u,v) parameterization (see Figure 11.5). Given a position p' on the plane, we can compute its position with respect to the coordinate system by

$$(\mathrm{p}'-\mathrm{p}) = (\partial\mathrm{p}/\partial u \; \partial\mathrm{p}/\partial v)\begin{pmatrix} du \\ dv \end{pmatrix}$$

or

$$\begin{pmatrix} \mathrm{p}'-\mathrm{p}_x \\ \mathrm{p}'-\mathrm{p}_y \\ \mathrm{p}'-\mathrm{p}_z \end{pmatrix} = \begin{pmatrix} \partial\mathrm{p}/\partial u_x & \partial\mathrm{p}/\partial v_x \\ \partial\mathrm{p}/\partial u_y & \partial\mathrm{p}/\partial v_y \\ \partial\mathrm{p}/\partial u_z & \partial\mathrm{p}/\partial v_z \end{pmatrix}\begin{pmatrix} du \\ dv \end{pmatrix}$$

This is a linear system in three equations of two unknowns–i.e. it's over-constrained. However, we need to be careful since one of the equations may be degenerate–e.g. if $\partial\mathrm{p}/\partial u$ and $\partial\mathrm{p}/\partial v$ are in the *xy* plane such that their *z* components are both zero, then the third equation will be degenerate. To deal with this, since we only need two equations to solve the system, we'd like to choose two that won't have degeneracies. Easy way to do this is to take the cross product of ∂p/∂*u* and ∂p/∂*v* and see which coordinate of the result has the largest magnitude; throw away that coordinate and use the other two. But that cross product is already available in `Nn`...

⟨*Compute* (u,v) *offsets at auxiliary points*⟩≡

```
⟨Initialize A, Bx, and By matrices for offset computation⟩
SolveLinearSystem2x2(A, Bx, x);
dudx = x[0];
dvdx = x[1];
SolveLinearSystem2x2(A, By, x);
dudy = x[0];
dvdy = x[1];
```

⟨*Initialize* `A`, `Bx`, *and* `By` *matrices for offset computation*⟩≡

```
Float A[2][2], Bx[2], By[2], x[2];
int axes[2];
if (fabsf(nn.x) > fabsf(nn.y) && fabsf(nn.x) > fabsf(nn.z)) {
    axes[0] = 1; axes[1] = 2;
}
else if (fabsf(nn.y) > fabsf(nn.z)) {
    axes[0] = 0; axes[1] = 2;
}
else {
    axes[0] = 0; axes[1] = 1;
}
⟨Initialize matrices for chosen projection plane⟩
```

⟨*Initialize matrices for chosen projection plane*⟩≡

```
A[0][0] = dpdu[axes[0]];
A[0][1] = dpdv[axes[0]];
A[1][0] = dpdu[axes[1]];
A[1][1] = dpdv[axes[1]];
Bx[0] = px[axes[0]] - p[axes[0]];
Bx[1] = px[axes[1]] - p[axes[1]];
By[0] = py[axes[0]] - p[axes[0]];
By[1] = py[axes[1]] - p[axes[1]];
```

XXX "filter region" often used to describe this area around a sample. but is a misnomer, since filter's extent not necessarily the same XXX

XXX discuss errors, filtering over areas on the surface that are invisible due to occlusion, silhouettes, etc. XXX

XXX note that anti-aliasing via tex coord differentials values is only really right if the end up being fed into a linear function to compute pixel contribution afterward... True for e.g. diffuse coefficient, but not for, say, a specular exponent... XXX

HUMPER STARTED EDITING AGAIN HERE

11.2.2 Anti-Aliasing Methods

Once we know the sampling rate in texture space, we need to remove frequencies in the texture function that are past the Nyquist limit for that sampling rate. What we would like to do, with as few approximations as possible, is to compute the result of the *ideal texture resampling* process. Given an arbitrary texture function $T(s,t)$, defined on a surface in the scene, consider the frequency content of the function $T'(x,y)$, which is $T(s,t)$ projected onto the image plane and expressed in terms of image coordinates:

$$T'(x,y) = T(s\partial x/\partial s + t\partial x/\partial t, s\partial y/\partial s + t\partial y/\partial t),$$

which can be approximated using the first-order differential values computed previously:

$$T'(x,y) \approx T(s/(\Delta s/\Delta x) + t/(\Delta t/\Delta x), s/(\Delta s/\Delta y) + t/(\Delta t/\Delta y)).$$

As we move away from the (x,y) position on the image plane for which $\partial s/\partial x$, etc. were computed, this approximation will become progressively more inaccurate.

The ideal resampling process says that in order to evaluate $T'(x,y)$ without aliasing, we must:

- *Band-limit* it, removing frequencies beyond the Nyquist limit by convolving it with the ideal sinc reconstruction filter:

$$T'_b(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} r(x')r(y')T'(x+x',y+y')dx'dy'$$

- Convolve this function with the pixel filter $f(x,y)$ centered at the (x,y) point on the screen at which we want to evaluate the texture function.

$$T'_f(x,y) = \int_{-\mathrm{xWidth}/2}^{\mathrm{xWidth}/2}\int_{-\mathrm{yWidth}/2}^{\mathrm{yWidth}/2} f(x',y')T'_b(x+x',y+y')dx'dy'$$

We will usually ignore the second step, effectively acting as if the pixel filter was a box filter, which allows us to do the anti-aliasing work completely in texture-space and simplifies the implementation significantly. In practice, both these steps can both be highly simplified; making some effort at applying them is much better than making no effort at all. Finding efficient solutions to this problem is made more difficult in practice because the texture sampling rate is generally different at every point that is shaded in the image, depending on the orientation and distance to the visible object at each image sample.

As a concrete example that can be solved in closed-form, consider a texture based on the sine function

$$T(s,t) = 1 + \sin(2\pi s)\sin(2\pi t).$$

In both the s and t directions, it has a single component with frequency $\omega = 1$ plus a constant component. According to sampling theory, point-sampling this function will lead to aliasing if the sampling frequency in texture space is less than $\omega_s = 2$ (corresponding to a sample spacing of $1/2$.)

The best approach is to *pre-filtering* the texture function before evaluating it at a particular (s,t) position by convolving with a sinc filter with width $1/\omega_s$, such that it removes all frequencies beyond half of the sampling frequency ω_s. The result of this can be worked out in closed form–if $\omega_s < 2$, the sinc filter will remove the sin components of $T(s,t)$ completely, giving us a filtered texture function

$$T_f(s,t) = 1.$$

If the texture sampling rate is high enough to capture the sin terms without aliasing, the sinc leaves this texture function unchanged, giving a filtered version

$$T_f(s,t) = 1 + \sin(2\pi s)\sin(2\pi t).$$

This example is unusual in that it is easy to work out in closed-form what the effect of pre-filtering with the ideal reconstruction filter would be. For more realistic texture functions, this is not possible. However, having seen the ideal process, some simplifications for more complex texture functions immediately suggest themselves.

First, The same disadvantages that the sinc has for image sample filtering are present in texture filtering. Recalll that the sinc has infinite extent and exhibits ringing artifacts when there are discontinuities in the function being filtered. A finite extent filter such as the Gaussian or the Mitchell filter might be used. Even the box filter, with its shortcomings, gives acceptable results for texture filtering in many cases. The box can be particularly easy to apply, since it merely requires averaging the the texture function over some (s,t) region. Intuitively, this is a reasonable approach to the texture filtering problem, and it can be computed directly for many texture functions.

Another approach is use the observation that the effect of the sinc filter is to let frequency components below the Nyquist limit pass through unchanged but to remove frequencies past them. Therefore, if we have some awareness of the frequency content of the texture function (e.g. if it is a sum of terms, each one with known frequency content), then if we replace the high-frequency terms with their average values, we are effectively doing the work of the sinc pre-filter. This approach is known as *clamping*. **XXX explain shortcomings w.r.t. sinc XXX**

Finally, for texture functions where none of these techniques is easily applied, we can use *super-sampling*, where the function is evaluated and filtered at multiple locations around (s,t) to approximate pre-filtering. If a box filter is used to filter the samples, then this is equivalent to averaging the value of the function around (s,t). This approach can be expensive if the texture function is complex to evaluate, and as with image sampling, a very large number of samples may be needed to remove aliasing. Recall that sampling theory shows that jittering the locations of the sample points can turn the aliasing to noise. Though this is a brute-force solution, it is at least more efficient than increasing the image sampling rate, since it saves us the cost of tracing more rays through the scene.

11.3 Texture Coordinate Generation

We should probably have example renderings for all these mappings...

The textures in this chapter are functions that take a two-dimensional (s,t) coordinate or a three-dimensional (x,y,z) coordinate and compute a texture value at the given position. Sometimes there are obvious ways to choose these texture coordinates; for parametric surfaces, such as the quadrics in Chapter 3, there is a natural two-dimensional parameterization of the surface, and for all surfaces the shading point p is a natural choice for a three-dimensional coordinate. In `lrt`, we will use the convention that 2D texture coordinates are denoted by (s,t); this helps make clear the distinction between the intrinsic (u,v) parameterization of the underlying surface and the possibly-different coordinate values used for texturing.

In general, however, there is often not a natural parameterization of complex surfaces. For instance, given an arbitrary subdivision surface, there is no simple and general-purpose way to assign (s,t) texture values to the whole shape so that the entire $[0,1]^2$ (s,t) space is covered continuously and without distortion. In fact, creating smooth parameterizations of complex meshes with low distortion is an active area of research in computer graphics. This section will introdouce two abstract base classes–`TextureMapping2D` and `TextureMapping3D`–that provide an interface for computing 2D and 3D texture coordinates. We will then implement a number of standard mappings using them.

The `TextureMapping2D` base class has a single method, `Map()`, which is given the `DifferentialGeometry` at the shading point and returns the (s,t) texture coordinates via `Float` pointers. Also, it returns estimates for the change in s and t with respect to pixel x and y coordinates in `dsdx`, `dtdx`, `dsty`, and `dtdy` so that textures that use the mapping can determine the (s,t) sampling rate and filter accordingly.

```
58  DifferentialGeometry
410 TextureMapping3D
```

⟨Texture Class Declarations⟩+≡

```
class TextureMapping2D {
public:
    ⟨TextureMapping2D Interface⟩
};
```

⟨TextureMapping2D Interface⟩+≡

```
virtual void Map(const DifferentialGeometry &dg,
    Float *s, Float *t, Float *dsdx, Float *dtdx,
    Float *dsdy, Float *dtdy) const = 0;
```

11.3.1 2D Identity Mapping

The simplest texture mapping uses the 2D parametric (u,v) coordinates in the `DifferentialGeometry` to compute the texture coordinates. These can be offset and scaled with user-supplied values in each dimension.

⟨Texture Class Declarations⟩+≡

```
class IdentityMapping2D : public TextureMapping2D {
public:
    ⟨IdentityMapping2D Public Methods⟩
private:
    Float su, sv, du, dv;
};
```

⟨*Texture Method Definitions*⟩≡

```
IdentityMapping2D::IdentityMapping2D(Float _su, Float _sv,
        Float _du, Float _dv) {
    su = _su; sv = _sv;
    du = _du; dv = _dv;
}
```

The scale-and-shift computation to compute (s,t) coordinates is quite straightforward.

⟨*Texture Method Definitions*⟩+≡

```
void IdentityMapping2D::Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const {
    *s = su * dg.u + du;
    *t = sv * dg.v + dv;
    ⟨Compute texture differentials for 2D identity mapping⟩
}
```

Computing the differential change in s and t in terms of the original change in u and v and the scale amounts is also easy. Using the chain rule,

$$\frac{\partial s}{\partial x} = \frac{\partial u}{\partial x}\frac{\partial s}{\partial u} + \frac{\partial v}{\partial x}\frac{\partial s}{\partial v}$$

DifferentialGeometry	58
DifferentialGeometry::u	58
DifferentialGeometry::v	58
SphericalMapping2D	407
TextureMapping2D	405

and similarly for the three other partial derivatives. From the mapping method,

$$s = s_u \cdot u,$$

so

$$\frac{\partial s}{\partial u} = s_u, \frac{\partial s}{\partial v} = 0,$$

and thus

$$\frac{\partial s}{\partial x} = s_u \frac{\partial u}{\partial x},$$

and so forth.

⟨*Compute texture differentials for 2D identity mapping*⟩≡

```
*dsdx = su * dg.dudx;
*dtdx = sv * dg.dvdx;
*dsdy = su * dg.dudy;
*dtdy = sv * dg.dvdy;
```

11.3.2 Spherical Mapping

Another useful mapping effectively wraps a sphere around the object. Each point is projected along the vector from the sphere's center through the point, up to the sphere's surface. There, the same (u,v) mapping as was used for the sphere shape is used.

The `SphericalMapping2D` object stores a transformation that is applied to points before this mapping is performed; this effectively allows the sphere to be arbitrarily positioned and oriented with respect to the object.

⟨*Texture Class Declarations*⟩+≡

```
class SphericalMapping2D : public TextureMapping2D {
public:
    ⟨SphericalMapping2D Public Methods⟩
private:
    void sphere(const Point &P, Float *s, Float *t) const;
    Transform WorldToTexture;
};
```

⟨*Texture Method Definitions*⟩+≡

```
void SphericalMapping2D::Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const {
    sphere(dg.p, s, t);
    ⟨Compute texture coordinate differentials for sphere (u,v) mapping⟩
}
```

A short utility function computes the mapping for a single point; it will be useful to have this logic separated out for computing texture coordinate differentials.

⟨*Texture Method Definitions*⟩+≡

```
void SphericalMapping2D::sphere(const Point &p, Float *s,
        Float *t) const {
    Vector vec = (WorldToTexture(p) - Point(0,0,0)).Hat();
    Float theta = SphericalTheta(vec);
    Float phi = SphericalPhi(vec);
    *s = theta * INV_PI;
    *t = phi * INV_TWOPI;
}
```

58	DifferentialGeometry
58	DifferentialGeometry::p
678	INV_PI
678	INV_TWOPI
33	Point
193	SphericalPhi()
193	SphericalTheta()
405	TextureMapping2D
43	Transform
27	Vector
30	Vector::Hat()

We could here again use the chain rule to compute the texture coordinate differentials, but will instead use a forward differencing approximation to give a flavor of another way to compute these values which is useful for more involved mapping functions. Recall that the DifferentialGeometry has fields that hold the change in position as a function of change in image sample position, so if the s coordinate is computed by some function $f_s(\mathrm{p})$, it's easy to compute approximations like

$$\frac{\partial s}{\partial x} \approx \frac{f_s(\mathrm{p} + \Delta \times \partial \mathrm{p}/\partial x) - f_s(\mathrm{p})}{\Delta}.$$

As the distance $\Delta \to 0$, this approximates the partial derivative.

One other detail is that the sphere mapping has a discontinuity in the mapping formula; there is a seam at $t = 1$ where the t texture coordinate then discontinuously jumps back to zero again. We can detect this by checking to see if the dtdx or dtdy estimate we computed was greater than 0.5 and then adjusting the estimate appropriately.

⟨*Compute texture coordinate differentials for sphere* (u,v) *mapping*⟩≡

```
Float sx, tx, sy, ty;
const Float delta = .01;
sphere(dg.p + delta * dg.dpdx, &sx, &tx);
*dsdx = (sx - *s) / delta;
*dtdx = (tx - *t) / delta;
if (*dtdx > .5) *dtdx = 1. - *dtdx;
sphere(dg.p + delta * dg.dpdy, &sy, &ty);
*dsdy = (sy - *s) / delta;
*dtdy = (ty - *t) / delta;
if (*dtdy > .5) *dtdy = 1. - *dtdy;
```

11.3.3 Cylindrical Mapping

Like the spherical mapping, the cylindrical mapping effectively wraps a cylinder around the object having texture coordinates computed for it. It also supports a transformation to orient the mapping cylinder.

⟨*Texture Class Declarations*⟩+≡

```
class CylindricalMapping2D : public TextureMapping2D {
public:
    ⟨CylindricalMapping2D Public Methods⟩
private:
    void cylinder(const Point &P, Float *s, Float *t) const;
    Transform WorldToTexture;
};
```

DifferentialGeometry	58
fferentialGeometry::dpdx	400
fferentialGeometry::dpdy	400
DifferentialGeometry::p	58
M_PI	678
Point	33
TextureMapping2D	405
Transform	43
Vector	27
Vector::Hat()	30

The cylindrical mapping has the same basic structure as the sphere mapping; the mapping function is just different. Therefore, we will omit the fragment that computes texture coordinate differentials, since it is essentially the same as the spherical version.

⟨*Texture Method Definitions*⟩+≡

```
void CylindricalMapping2D::Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const {
    cylinder(dg.p, s, t);
    ⟨Compute texture coordinate differentials for cylinder (u,v) mapping⟩
}
```

⟨*Texture Method Definitions*⟩+≡

```
void CylindricalMapping2D::cylinder(const Point &p, Float *s,
        Float *t) const {
    Vector vec = (WorldToTexture(p) - Point(0,0,0)).Hat();
    *s = (M_PI + atan2f(vec.y, vec.x)) / (2.f * M_PI);
    *t = vec.z;
}
```

11.3.4 Planar Mapping

Another classic mapping method is the planar mapping. The point to have texture coordinates computed is effectively projected onto a plane; a 2D parameterization of the plane then gives texture coordinates for the point. For example, a point p could be projected on the $z = 0$ plane to yield texture coordinates given by $u = \mathrm{p}_x$ and $v = \mathrm{p}_y$.

More generally, we can define such a parameterized plane with two non-parallel vectors $\mathbf{v_u}$ and $\mathbf{v_v}$ and offsets du and dv. The texture coordinates are given by finding the coordinates of the point with respect to the plane coordinate system, which is done by taking the dot product of the vector from the point to the origin with each vector $\mathbf{v_u}$ and $\mathbf{v_v}$ and then adding the offset. For the example in the previous paragraph, we'd have $\mathbf{v_u} = (1,0,0)$, $\mathbf{v_v} = (0,1,0)$, and $du = dv = 0$.

⟨*Texture Class Declarations*⟩+≡

```
class PlanarMapping2D : public TextureMapping2D {
public:
    ⟨PlanarMapping2D Public Methods⟩
private:
    Vector vs, vt;
    Float ds, dt;
};
```

58	DifferentialGeometry
400	DifferentialGeometry::dpdx
400	DifferentialGeometry::dpdy
58	DifferentialGeometry::p
30	Dot()
33	Point
405	TextureMapping2D
27	Vector

⟨*Texture Method Definitions*⟩+≡

```
PlanarMapping2D::PlanarMapping2D(const Vector &_v1,
        const Vector &_v2, Float _ds, Float _dt) {
    vs = _v1;
    vt = _v2;
    ds = _ds;
    dt = _dt;
}
```

The planar mapping differentials can be computed directly by finding the differentials of p in texture coordinate space; the result is below.

⟨*Texture Method Definitions*⟩+≡

```
void PlanarMapping2D::Map(const DifferentialGeometry &dg,
        Float *s, Float *t, Float *dsdx, Float *dtdx,
        Float *dsdy, Float *dtdy) const {
    Vector vec = dg.p - Point(0,0,0);
    *s = ds + Dot(vec, vs);
    *t = dt + Dot(vec, vt);
    *dsdx = Dot(dg.dpdx, vs);
    *dtdx = Dot(dg.dpdx, vt);
    *dsdy = Dot(dg.dpdy, vs);
    *dtdy = Dot(dg.dpdy, vt);
}
```

11.3.5 3D Mapping

We will define a `TextureMapping3D` class that defines the interface for generating three-dimensional texture coordinates.

⟨*Texture Class Declarations*⟩+≡

```
class TextureMapping3D {
public:
    ⟨TextureMapping3D Interface⟩
};
```

⟨*TextureMapping3D Interface*⟩+≡

```
virtual Point Map(const DifferentialGeometry &dg,
    Vector *dpdx, Vector *dpdy) const = 0;
```

The natural three dimensional mapping just takes the world-space coordinate of the point being shaded and applies a linear transformation to it. This will often be a transformation that takes the point back to the primitive's object space, so the texture does not appear to change as the object is transformed. **XXX shader space, actually XXX**

DifferentialGeometry	58
fferentialGeometry::dpdx	400
fferentialGeometry::dpdy	400
DifferentialGeometry::p	58
Point	33
Transform	43
Vector	27

⟨*Texture Class Declarations*⟩+≡

```
class IdentityMapping3D : public TextureMapping3D {
public:
    IdentityMapping3D(const Transform &x)
        : WorldToTexture(x) { }
    Point Map(const DifferentialGeometry &dg, Vector *dpdx,
        Vector *dpdy) const;
private:
    Transform WorldToTexture;
};
```

⟨*Texture Method Definitions*⟩+≡

```
Point IdentityMapping3D::Map(const DifferentialGeometry &dg,
        Vector *dpdx, Vector *dpdy) const {
    Point p = WorldToTexture(dg.p);
    Float delta = .1f;
    *dpdx = (WorldToTexture(dg.p + delta * dg.dpdx) - p) / delta;
    *dpdy = (WorldToTexture(dg.p + delta * dg.dpdy) - p) / delta;
    return p;
}
```

11.4 Interpolated Textures

The simplest non-constant 2D textures interpolate between multiple given values based on the relation of the (s,t) coordinates of the point being shaded to values at the four corners of $[0,1]^2$ or at the vertices of a triangle mesh. These textures also don't need to consider anti-aliasing, since they are typically not the source of high frequency variations.

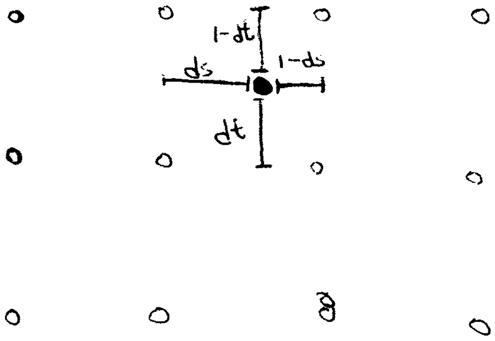


Figure 11.6: basic bilerp idea

11.4.1 Bilinear Interpolation

⟨*Texture Class Declarations*⟩+≡

```
template <class T>
class BilerpTexture : public Texture<T> {
public:
    ⟨BilerpTexture Public Methods⟩
private:
    ⟨BilerpTexture Private Data⟩
};
```

394 Texture
405 TextureMapping2D

The `BilerpTexture` class provides bilinear interpolation between four constant values. Figure 11.6 shows the basic idea: values are defined at $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$ in (s,t) parameter space. The value at a particular (s,t) position is found by interpolating between them.

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T>
BilerpTexture<T>::BilerpTexture(TextureMapping2D *m,
        const T &t00, const T &t01, const T &t10,
        const T &t11) {
    mapping = m;
    v00 = t00;
    v01 = t01;
    v10 = t10;
    v11 = t11;
}
```

⟨*BilerpTexture Private Data*⟩≡

```
TextureMapping2D *mapping;
T v00, v01, v10, v11;
```

The interpolated value of the four values at a (s,t) position can be computed by three linear interpolations. For example, we can first interpolate u of the way between the values at $(0,0)$ and $(1,0)$ and store that in a temporary `tmp1`. We can then interpolate u of the way between the $(0,1)$ and $(1,1)$ values and store the result in tmp2. Finally, by interpolating v of the way between tmp1 and tmp2 gives us our final result. Mathematically, this is:

$$\begin{aligned}
\mathrm{tmp}_1 &= (1-u)\mathrm{val}_{00} + u\mathrm{val}_{10} \\
\mathrm{tmp}_2 &= (1-u)\mathrm{val}_{01} + u\mathrm{val}_{11} \\
\mathrm{result} &= (1-v)\mathrm{tmp}_1 + v\mathrm{tmp}_2
\end{aligned}$$

Rather than storing the intermediate values explicitly, some algebraic rearrangement gives us the same result from an appropriately weighted average of the four corner values:

$$\begin{aligned}
\mathrm{result} &= (1-u)(1-v)\mathrm{val}_{00} + (1-u)v\,\mathrm{val}_{01} + \\
&\quad u(1-v)\mathrm{val}_{10} + uv\,\mathrm{val}_{11}
\end{aligned}$$

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T>
T BilerpTexture<T>::Evaluate(const DifferentialGeometry &dg) const {
    Float u, v, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &u, &v, &dsdx, &dtdx, &dsdy, &dtdy);
    return (1-u)*(1-v) * v00 + (1-u)*v * v01 + u*(1-v) * v10 +
        u*v * v11;
}
```

BilerpTexture	411
BilerpTexture::mapping	411
BilerpTexture::v00	411
BilerpTexture::v01	411
BilerpTexture::v10	411
BilerpTexture::v11	411
DifferentialGeometry	58
Texture	394
TextureMapping2D::Map()	405

11.5 Image Texture

The ImageTexture class stores a 2D array of point sampled values of a texture function. It uses these samples to reconstruct a continuous image function that can be evaluated at an arbitrary (s,t) position. These sample values are often called *texels*, since they are similar to pixels but are used in the context of a texture. Image textures are the most widely used type of texture in computer graphics: digital photographs, scanned artwork, images created with image editing programs, and images generated by renderers are all extremely useful sources of data for this particular texture representation. The term *texture map* is often used to refer to this type of texture, though this usage blurs the distinction between the mapping that computes texture coordinates and the texture function itself.

⟨*Texture Class Declarations*⟩+≡

```
template <class T>
class ImageTexture : public Texture<T> {
public:
    ⟨ImageTexture Public Methods⟩
private:
    ⟨ImageTexture Private Methods⟩
    ⟨ImageTexture Private Data⟩
};
```

The caller provides the ImageTexture with the filename of an image. The data in this file is used to create a MIPMap that stores the texels in memory and handles the details of reconstruction and filtering to reduce aliasing.

For an ImageTexture that returns Spectrum values from Texture::Evaluate(), the MIPMap stores the image data with type Spectrum. This can be a somewhat wasteful representation, since most image file formats use a single byte for red, green, and blue channels at each texel, while the Spectrum class stores spectra with 32-bit floating point values for each color coefficient. However, it would be unnecessarily restrictive to only support that representation. It's important that the system support image maps with arbitrary floating-point values for the accuracy and high contrast that come from this generality; it is particularly useful for bringing measured data into the renderer. As such, we will accept some wasted memory and store all color-oriented textures with Spectrum objects here.

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T>
ImageTexture<T>::ImageTexture(TextureMapping2D *m,
        const string &filename) {
    mapping = m;
    mipmap = GetTexture(filename);

}
```

412 ImageTexture
414 ImageTexture::GetTexture()
417 MIPMap
181 Spectrum
405 TextureMapping2D

⟨*ImageTexture Private Data*⟩≡

```
MIPMap<T> *mipmap;
TextureMapping2D *mapping;
```

11.5.1 Texture caching

Because image maps are memory-intensive and because the user may reuse a texture many times within a scene, lrt maintains a hash table of image maps, so that they are only loaded into memory once even if they are used in more than one ImageTexture.

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T> MIPMap<T> *ImageTexture<T>::GetTexture(
        const string &filename) {
    static map<string, MIPMap<T> *> textures;
    if (textures.find(filename) != textures.end())
        return textures[filename];
    int width, height;
    Spectrum *texels = ReadImage(filename, &width, &height);
    MIPMap<T> *ret = NULL;
    if (texels) {
        ⟨Convert texels to type T and create MIPMap⟩
    }
    else {
        ⟨Create zero-valued MIPMap⟩
    }
    textures[filename] = ret;
    return ret;
}
```

ImageTexture	412
ImageTexture::convert()	415
MIPMap	417
Spectrum	181

Because the image loading routines always return an array of Spectrum values for the texels, it is necessary to convert these Spectrum values to the particular type T of texel that this MIPMap is storing (e.g. Float) if the type of T isn't Spectrum. The per-texel conversion is handled by the utility routine ImageTexture::convert(). Note that this conversion is wasted work in the common case where the MIPMap is storing Spectrum values, but the flexibility it gives us is worth this relatively small cost in efficiency.[1]

⟨*Convert texels to type* T *and create* MIPMap⟩≡

```
T *convertedTexels = new T[width*height];
for (int i = 0; i < width*height; ++i)
    convert(texels[i], &convertedTexels[i]);
ret = new MIPMap<T>(width, height, convertedTexels);
delete[] texels;
delete[] convertedTexels;
```

Per-texel conversion is done using C++ function overloading. For every type to which we would like to be able to convert Spectrum values, a separate convert() function must be provided. In the loop over texels above, C++'s function overloading mechanism will select the appropriate instance of convert() based on the destination type. Unfortunately, it is not possible to return the converted value from the function, since C++ doesn't support overloading by return type.

⟨*ImageTexture Private Methods*⟩+≡

```
static void convert(const Spectrum &from, Spectrum *to) {
    *to = from;
}
static void convert(const Spectrum &from, Float *to) {
    *to = from.y();
}
```

[1] Additional C++ template trickery could ensure that this step was skipped when T is type Spectrum, if the cost of this unnecessary work was unacceptable.

If the texture file wasn't found or was unreadable, an image map with a single zero-valued sample is created so that the renderer can continue and generate some image of the scene without needing to abort execution. The ReadImage function will print a warning message to the screen in this case.

⟨*Create zero-valued* MIPMap⟩≡

```
T *oneVal = new T[1];
oneVal[0] = 0.;
ret = new MIPMap<T>(1, 1, oneVal);
delete[] oneVal;
```

The ImageTexture evaluation routine does the usual texture coordinate computation and then hands the image map lookup to the MIPMap. The MIPMap does the difficult image filtering work for anti-aliasing using the partial derivatives of the texture coordinates.

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T>
T ImageTexture<T>::Evaluate(
        const DifferentialGeometry &dg) const {
    Float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    return mipmap->Lookup(s, t, dsdx, dtdx, dsdy, dtdy);
}
```

58	DifferentialGeometry
412	ImageTexture
413	ImageTexture::mapping
413	ImageTexture::mipmap
417	MIPMap
427	MIPMap::Lookup()
181	Spectrum
185	Spectrum::y()
405	TextureMapping2D::Map()

11.5.2 MIP Maps

As always, if the image function that is reconstructed from the point samples has higher frequency detail than can be represented by the texture sampling rate, aliasing will be present in the final image. Any frequencies higher than the Nyquist limit must be removed by pre-filtering before the function is evaluated. Figure 11.7 shows the basic problem we face: an image texture has texels that are samples at a fixed frequency. The filter region is given by its (s,t) center point and offsets to the estimated texture coordinate locations for the adjacent image samples. Because these offsets are estimates of the texture sampling rate, it is necessary to remove any frequencies higher than twice the distance to the adjacent samples in order to satisfy the Nyquist criterion.

The texture sampling and reconstruction process has a few key differences from the process discussed in Chapter 7, however. First, it's inexpensive to get the value of a sample–only the cost of an array lookup is necessary (as opposed to having to trace a ray). Second, because the texture image function is fully defined by the set of samples and there is no mystery about what its highest frequency could be, there is no problem regarding the function's behavior between samples. These differences make it possible to remove detail from the texture before sampling, thus eliminating aliasing due to image texture in final images.

However, the texture sampling rate will in general change from pixel to pixel–it is spatially variant. The sampling rate is determined by scene geometry and its orientation, the texture coordinate mapping function, and the camera projection and image sampling rate. Because the sampling rate is not fixed, texture filtering algorithms need to be able to filter over arbitrary regions of texture samples efficiently.

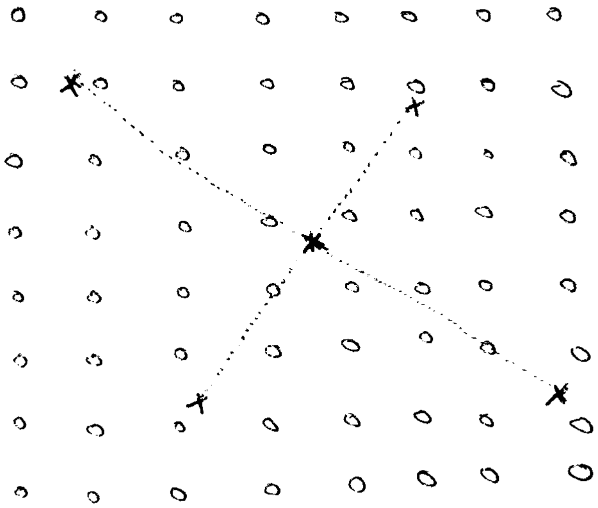


Figure 11.7: may need to look at many texels to filter a texture over a large area...

Figure 11.8: image sequence showing that texture filtering makes a big difference

Figure 11.9: A few levels of an image pyramid...

The MIPMap class implements two methods for efficient texture filtering with spatially varying filter widths. The first, trilinear interpolation, is fast and easy to implement, and has been widely used for texture filtering in graphics hardware. The second, elliptically weighted averaging, is slower and more complex, but returns extremely high quality results.

To limit the potential number of texels that need to be accessed, both of these filtering methods use an *image pyramid* of increasingly lower-resolution pre-filtered versions of the original image to accelerate their operation[2]. The original image texels are at the bottom level of the pyramid, and the image at each level is half the resolution of the the previous level, up to the top level which has a single texel representing the average of all of the texels in the original image. Figure 11.9 shows a few levels of the image pyramid for the **XXX** texture on the **XXX** scene. This collection of images only needs 1/3 more memory than storing the most detailed level alone, and can be used to quickly find filtered values over large regions of the original image.

MIPMap is a template class, and is parameterized by the data type of the image texels. lrt creates MIPMaps of both Spectrum and Float images; Float MIP maps are used for representing directional distributions of intensity from goniometric light sources, for example. The MIPMap requires that the type T support just a few basic operations, including addition and multiplication by a scalar.

```
⟨MIPMap Declarations⟩≡
  template <class T> class MIPMap {
  public:
      ⟨MIPMap Public Methods⟩
  private:
      ⟨MIPMap Private Methods⟩
      ⟨MIPMap Private Data⟩
  };
```

In the constructor, the MIPMap copies the image data provided by the caller, possibly resizes the image to ensure that its resolution is a power of two in each direction, and initializes a lookup table used by the elliptically weighted average

[2] The name “mipmap” comes from the Latin “multum in parvo”, which means “many things in the same place”, a nod to the image pyramid.

filtering method in Section 11.5.4.

⟨*MIPMap Method Definitions*⟩≡

```
template <class T>
MIPMap<T>::MIPMap(int sres, int tres, const T *img) {
    T *resampledImage = NULL;
    if (!IsPowerOf2(sres) || !IsPowerOf2(tres)) {
        ⟨Resample image to power-of-two resolution⟩
    }
    ⟨Initialize levels of MIPMap from image⟩
    if (resampledImage) delete[] resampledImage;
    ⟨Initialize EWA filter weights if needed⟩
}
```

Implementation of an image pyramid is substantially easier if the resolution of the original image is an exact power of two in each direction; this ensures that there is a straightforward relationship between the level of the pyramid and the number of texels at that level. If the user has provided an image where the resolution in one or both of the dimensions is not a power of two, then the `MIPMap` constructor starts by increasing the image resolution up to the next power of two greater than the original resolution before constructing the pyramid.

`IsPowerOf2()`	678
`MIPMap`	417
`RoundUpPow2()`	678

Image magnification in this manner involves more application of the sampling and reconstruction theory from Chapter 7: we have an image function that has been sampled at one sampling rate, and we'd like to reconstruct a continuous image function from the original samples to resample at a new set of sample positions. Because this represents an increase in the sampling rate from the original rate, we don't have to worry about introducing aliasing due to undersampling high frequency components in this step–we only need to reconstruct and directly resample the new function. Figure 11.10 illustrates this task in 1D.

The `MIPMap` will use a separable reconstruction filter for this task; recall from Section 7.7 that separable filters can be written as the product of one-dimensional filters: $f(x,y) = f(x)f(y)$. One advantage of using a separable filter is that if we are using one to resample an image from one resolution (s,t) to another (s',t'), then we can implement the resampling as two one-dimensional resampling steps, first resampling in s to create an image of resolution (s',t) and then resampling that image to create the final image of resolution (s',t'). Resampling the image via two 1D steps in this manner simplifies implementation and makes the number of texels accessed for each texel in the final image a linear function of the filter width, rather than a quadratic one.

⟨*Resample image to power-of-two resolution*⟩≡

```
int sPow2 = RoundUpPow2(sres), tPow2 = RoundUpPow2(tres);
⟨Resample image in s direction⟩
⟨Resample image in t direction⟩
sres = sPow2;
tres = tPow2;
```

Reconstructing the original image function and sampling it at a new texel's position is mathematically equivalent to centering the reconstruction filter kernel at the new texel's position and weighting the nearby texels in the original image appropriately (see Figure 11.11.) Thus, each new texel is a weighted average of a small

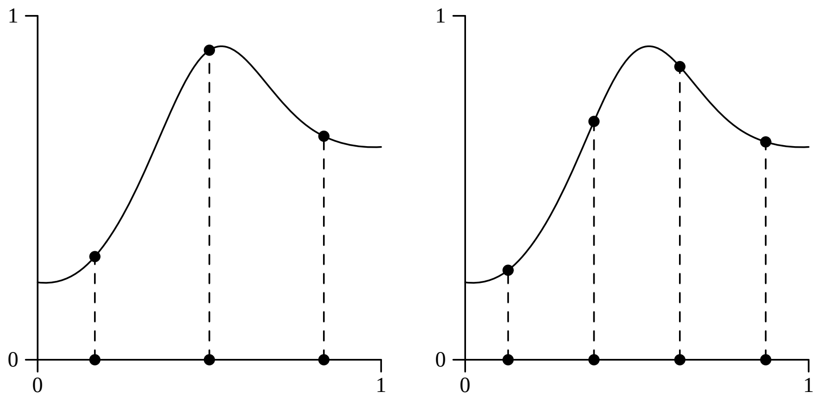


Figure 11.10: To increase an image's resolution to be a power of two, the `MIPMap` performs two 1D resampling steps with a separable reconstruction filter. On the left is a 1D function reconstructed from three samples, denoted by dots. To represent the same image function with more samples, we just need to reconstruct the continuous function and evaluate it at the new positions, shown as dots in the right image.

number of texels in the original image.

The `MIPMap::resampleWeights()` method determines which original texels contribute to each new texel and the values of contribution weights. It returns the values in an array of `ResampleWeight` structures for all of the texels in a 1D row or column of the image. Because this information is the same for all rows of the image when we are resampling in *s* and all columns when we are resampling in *t*, it's more efficient to compute it once for each of the two passes and then reuse it many times for each one. Given these weights, the image is first magnified in the *s* direction, turning the original image with resolution `(sres,tres)` into an image with resolution `(sPow2,tres)`, which is stored in `resampledImage`. We allocate enough space in `resampledImage` to hold the final zoomed image, so we don't have to do two large allocations.

420 MIPMap::resampleWeights()
420 ResampleWeight

well, why is the resampledImage sPow2 * tPow2 then, when the above line says it should be sPow2*tres?

⟨*Resample image in s direction*⟩≡

```
ResampleWeight *sWeights = resampleWeights(sres, sPow2);
resampledImage = new T[sPow2 * tPow2];
⟨Apply sWeights to zoom in s direction⟩
delete[] sWeights;
```

For the filter function we will use below, no more than four of the original texels will contribute to each new texel after zooming, so `ResampleWeight` only needs to hold four weights. Because the four texels are contiguous, we only store the offset to the first one.

⟨*MIPMap Private Data*⟩≡

```
struct ResampleWeight {
    int firstTexel;
    Float weight[4];
};
```

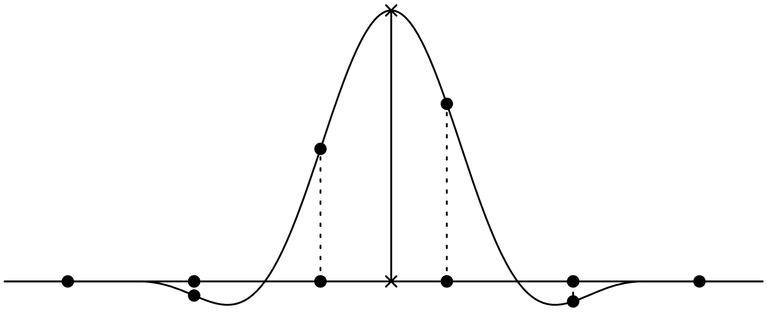


Figure 11.11: Given an image described by evenly-spaced point samples, denoted here by dots, we'd like to reconstruct the image function at an arbitrary point, denoted here by an "x". The value at this point can be computed by centering the reconstruction filter kernel at the new point and computing the weighted average of the nearby samples using the filter's value for the offsets to those samples.

Assert()	659
MIPMap::Lanczos()	420

```
⟨MIPMap Private Methods⟩+≡
  ResampleWeight *resampleWeights(int oldres, int newres) {
      Assert(newres >= oldres);
      ResampleWeight *wt = new ResampleWeight[newres];
      Float filterwidth = 2.f;
      for (int i = 0; i < newres; ++i) {
          ⟨Compute image resampling weights for ith texel⟩
      }
      return wt;
  }
```

Just as it was important to distinguish between discrete and continuous pixel coordinates in Chapter 7, the same issues need to be addressed with texel coordinates here. We will use the same conventions as described in Section 7.2.9 here. For each new texel, this function starts by computing its position's continuous coordinates in terms of the old texel coordinates. This value is stored in `center`, as it is the center of the reconstruction filter for the new texel. Next, it needs to find the offset to the first texel that contributes to the new texel. This is a slightly tricky calculation–after subtracting the filter width to find the start of the filter's non-zero range, it is necessary to add an extra 0.5 offset to the continuous coordinate before computing the floor to find the discrete coordinate. Figure 11.12 illustrates why this offset is needed.

Starting from this first contributing texel, this function loops over four texels, computing each one's offset to the center of the filter kernel and the corresponding filter weight. The reconstruction filter function used to compute the weights, `MIPMap::Lanczos()`, is equivalent to the one in `SincFilter::Sinc1D()`.

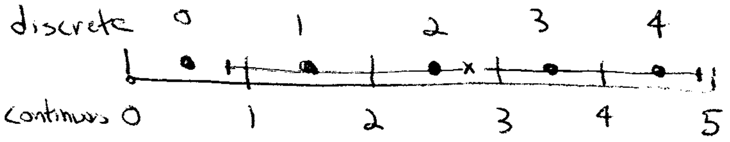


Figure 11.12: The computation to find the first texel inside a reconstruction filter's support is slightly tricky. Consider a filter centered around continuous coordinate 2.75 with width 2, as shown here. The filter's support covers the range $[0.75, 4.75]$, though texel zero is outside the filter's support: adding 0.5 to the lower end before taking the floor to find the discrete texel gives the correct starting texel, number one.

⟨*Compute image resampling weights for* `i`*th texel*⟩≡

```
  Float center = (i + .5f) * oldres / newres;
  wt[i].firstTexel = Floor2Int((center - filterwidth) + 0.5f);
  for (int j = 0; j < 4; ++j) {
      Float pos = wt[i].firstTexel + j + .5f;
      wt[i].weight[j] = Lanczos((pos - center) / filterwidth);
  }
  ⟨Normalize filter weights for texel resampling⟩
```

420 ResampleWeight::firstTexel
420 ResampleWeight::weight

Depending on the filter function used, the four filter weights may not sum to one. Therefore, to ensure that the resampled image won't be any brighter or darker than the original image, the weights are normalized here.

⟨*Normalize filter weights for texel resampling*⟩≡

```
  Float invSumWts = 1.f / (wt[i].weight[0] + wt[i].weight[1] +
      wt[i].weight[2] + wt[i].weight[3]);
  for (int j = 0; j < 4; ++j)
      wt[i].weight[j] *= invSumWts;
```

Once the weights have been computed, it's easy to apply them to compute the zoomed texels. For each of the `tres` horizontal scanlines in the original image, we make a pass across the `sPow2` texels in the *s*-zoomed image using the precomputed weights to compute their values.

⟨*Apply* `sWeights` *to zoom in s direction*⟩≡

```
  for (int t = 0; t < tres; ++t) {
      for (int s = 0; s < sPow2; ++s) {
          ⟨Compute texel (s,t) in s-zoomed image⟩
      }
  }
```

The `MIPMap` uses the convention that any out-of-bounds texel coordinate should be remapped to the range of valid coordinates by taking the modulus of the value with the overall resolution in its dimension, thus repeating the image. It is necessary to handle this case here as well, since some of the texels needed will be off the edges of the image sample array.

⟨*Compute texel* (s,t) *in s-zoomed image*⟩≡

```
resampledImage[t*sPow2+s] = 0.;
for (int j = 0; j < 4; ++j) {
    int origS = Mod(sWeights[s].firstTexel + j, sres);
    resampledImage[t*sPow2+s] += sWeights[s].weight[j] *
        img[t*sres + origS];
}
```

The process for resampling in the *t* direction is almost the same as for *s*, so we won't include the implementation here.

Once we have an image with resolutions that are powers of two, the levels of the MIP map can be initialized, starting from the bottom. Each one will filter the texels from the previous level. Because image maps use a fair amount of memory, and because 8-20 texels are typically used per image texture lookup to compute a filtered value, it's worth carefully considering how the texels are laid out in memory, since reducing cache misses while accessing the texture map can noticeably improve the renderer's performance for certain scenes. Because both of the two texture filtering methods implemented in this section access a set of texels in a rectangular region of the image map each time a lookup is performed, the `MIPMap` uses the `BlockedArray` template class to store the 2D arrays of texel values, rather than using a standard C++ array. The `BlockedArray` reorders the array values in memory in a way that improves cache coherence when the values are accessed with these kinds of rectangular patterns; it is described in Appendix A.3.5.

`BlockedArray`	672
`Log2Int()`	678
`Mod()`	677
`esampleWeight::firstTexel`	420
`ResampleWeight::weight`	420

⟨*Initialize levels of MIPMap from image*⟩≡

```
nLevels = 1 + Log2Int(max(sres, tres));
pyramid = new BlockedArray<T> *[nLevels];
⟨Initialize most detailed level of MIPMap⟩
for (int i = 1; i < nLevels; ++i) {
    ⟨Initialize ith MIPMap level from i − 1st level⟩
}
```

⟨*MIPMap Private Data*⟩+≡

```
BlockedArray<T> **pyramid;
int nLevels;
```

The base level of the MIP map, which holds the original data (or the resampled data, if it didn't originally have power of two resolutions), is easily initialized by the default `BlockedArray` constructor.

⟨*Initialize most detailed level of MIPMap*⟩≡

```
pyramid[0] = new BlockedArray<T>(sres, tres, img);
```

Before showing how the rest of the levels are initialized, we will first define a texel access function that will be used during that process. `MIPMap::texel()` returns a reference to the texel value for the given discrete integer-valued texel position. As described above, if an out-of-range texel coordinate is passed in, this method effectively repeats the texture over the entire 2D texture coordinate domain by taking the modulus of the coordinate with respect to the texture size. Other reasonable ways to handle this condition include clamping the texel coordinates to the valid range, or returning a constant value for out of range coordinates. **should we provide these options? Would be so simple.**

⟨*MIPMap Method Definitions*⟩+≡

```
template <class T>
const T &MIPMap<T>::texel(int level, int s, int t) const {
    const BlockedArray<T> &l = *pyramid[level];
    s = Mod(s, l.uSize());
    t = Mod(t, l.vSize());
    return lPYWEB_NO_USE (s, t);
}
```

For non-square images, the resolution in one direction must be clamped to one for the upper levels of the image pyramid, where there is still down-sampling to do in the larger of the two resolutions. This is handled by the `max()` call below.

⟨*Initialize ith MIPMap level from i* − 1*st level*⟩≡

```
int sRes = max(1, pyramid[i-1]->uSize()/2);
int tRes = max(1, pyramid[i-1]->vSize()/2);
pyramid[i] = new BlockedArray<T>(sRes, tRes);
⟨Filter four texels from finer level of pyramid⟩
```

Here the `MIPMap` just uses a simple box filter to average four texels from the previous level to find the value the current texel. While using the Lanczos filter here would give a slightly better result for this computation, the box filter is fine in practice. **Why not do the right thing? fine in practice is a cop-out.**

Though not shown above, there are actually two versions of the `MIPMap::texel()` method, the other returning a `T &` instead of a `const T &`, thus making it possible to use a `MIPMap::texel()` call on the left hand side of an assignment expression as is done here.

672 BlockedArray
673 BlockedArray::uSize()
673 BlockedArray::vSize()
417 MIPMap

⟨*Filter four texels from finer level of pyramid*⟩≡

```
for (int t = 0; t < tRes; ++t)
    for (int s = 0; s < sRes; ++s)
        texel(i, s, t) = .25f * (
                                 texel(i-1, 2*s, 2*t) +
                                 texel(i-1, 2*s+1, 2*t) +
                                 texel(i-1, 2*s, 2*t+1) +
                                 texel(i-1, 2*s+1, 2*t));
```

11.5.3 Isotropic Triangle Filter

The first of the two `MIPMap::Lookup()` methods uses a triangle filter over the texture samples to remove high frequencies. Although this filter function does not give high-quality results, it can be implemented very efficiently. In addition to the (s,t) coordinates of the evaluation point, the user passes this method a filter width for the lookup, giving the extent of the region of the texture to filter across. This method filters over a square region in texture space, so the width should be conservatively chosen to avoid aliasing in both the s and t direction. Filtering techniques like this one that do not support a filter extent that is non-square or non-axis-aligned are known as *isotropic*. The primary disadvantage of isotropic filtering algorithms is that textures viewed at an oblique angle will appear substantially blurry, since the sampling rate along one axis will be very different from the sampling rate along the other.

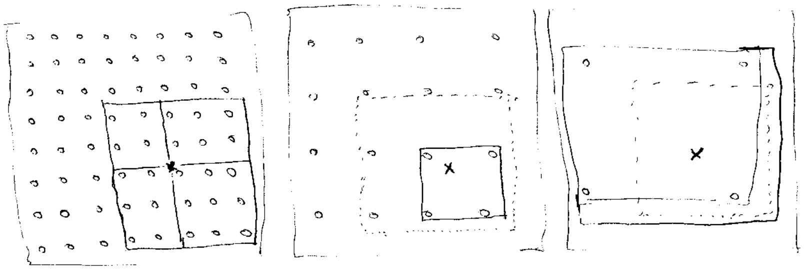


Figure 11.13: choosing a mipmap level for the triangle filter...

Because filtering over many texels for wide filter widths would be inefficient, this method chooses a MIP map level from the pyramid such that the filter region at that level would cover four texels at that level; Figure 11.13 shows the idea.

⟨*MIPMap Method Definitions*⟩+≡

```
template <class T>
T MIPMap<T>::Lookup(Float s, Float t, Float width) const {
    ⟨Compute MIPMap level for trilinear filtering⟩
    ⟨Perform trilinear interpolation at appropriate MIPMap level⟩
}
```

```
Log2()  677
MIPMap  417
```

Since the resolutions of the levels of the pyramid are all powers of two, the resolution of level l is $2^{\text{nLevels}-1-l}$. Therefore, to find the level with a texel spacing width w requires solving the equation:

$$\frac{1}{w} = 2^{\text{nLevels}-1-l}$$

for l. In general this will be a floating-point value between two MIP map levels.

⟨*Compute MIPMap level for trilinear filtering*⟩≡

```
Float level = nLevels - 1 + Log2(max(width, 1e-8f));
```

As shown by Figure 11.13, applying a triangle filter to the four texels around the sample point will either filter over too small a region or too large a region (except for very carefully-selected filter widths). The implementation here applies the triangle filter at both of these levels and blends between them according to how close `level` is to each of them. This helps hide the transitions from one MIP map level to the next at nearby pixels in the final image. While applying a triangle filter to four texels at two levels in this manner doesn't give exactly the same result as applying it to the original highest-resolution texels, the difference isn't too bad in practice and the efficiency of thies approach is worth this penalty. The elliptically weighted average filtering in the next section should be used when texture quality is very important.

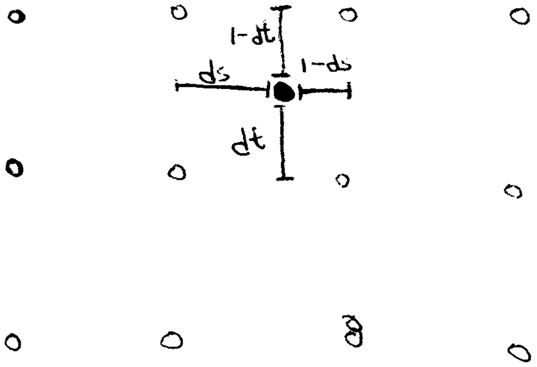


Figure 11.14: To comptue the value of the image texture function at an arbitrary (s,t) position, `MIPMap::triangle()` finds the four texels around (s,t) and weights them according to a triangle filter based on their distance to (s,t). One way to implement this is as a series of linear interpolations, as shown here: first, the two texels below (s,t) are linearly inteprolated to find a value at $(s,0)$, and the two texels above it are interpolated to find $(s,1)$. Then, $(s,0)$ and $(s,1)$ are linearly interpolated again to find the value at (s,t).

423 MIPMap::texel()
426 MIPMap::triangle()

⟨*Perform trilinear interpolation at appropriate MIPMap level*⟩≡

```
if (level < 0)
    return triangle(0, s, t);
else if (level >= nLevels - 1)
    return texel(nLevels-1, 0, 0);
else {
    int iLevel = Floor2Int(level);
    Float delta = level - iLevel;
    return (1.f-delta) * triangle(iLevel, s, t) +
        delta * triangle(iLevel+1, s, t);
}
```

Given floating-point texture coordinates in $[0,1]^2$, the `MIPMap::triangle()` routine uses a triangle filter to interpolate between the four texels that surround the sample point, as shown in Figure 11.14. This method first scales the coordinates by the texture resolution at the given mipmap level in each direction, turning them into continuous texel coordinates. Because these are continuous coordinates, but the texels in the image map are defined at discrete texture coordinates, it's important to carefully convert into a common representation. Here, we will do all of our work in discrete coordinates, mapping the continuous texel coordinates to discrete space.

For example, consider the 1D case with a continuous texture coordinate of 2.4: this coordinate is a distance of 0.1 below the discrete texel coordinate 2 (which corresponds to a continuous coordinate of 2.5), and is 0.9 above the discrete coordinate 1 (continuous coordinate 1.5). Thus, if we subtract 0.5 from the continuous coordinate 2.4, giving 1.9, we can correctly compute the correct distances to the discrete coordinates 1 and 2 by subtracting coordinates.

After computing the distances in s and t to the texel at the lower left of the given coordinates, `ds` and `dt`, `MIPMap::triangle()` computes weights for the four texels and computes the filtered value. Recall that the triangle filter is

$$f(x,y) = (1 - |x|)(1 - |y|);$$

the appropriate weights follow directly. Notice the similarity between this computation and`BilerpTexture::Evaluate()`.

⟨*MIPMap Method Definitions*⟩+≡

```
template <class T>
T MIPMap<T>::triangle(int level, Float s, Float t) const {
    level = Clamp(level, 0, nLevels-1);
    s = s * pyramid[level]->uSize() - 0.5f;
    t = t * pyramid[level]->vSize() - 0.5f;
    int s0 = Floor2Int(s), t0 = Floor2Int(t);
    Float ds = s - s0, dt = t - t0;
    return (1.-ds)*(1.-dt) * texel(level, s0, t0) +
        (1.-ds)*dt * texel(level, s0, t0+1) +
        ds*(1.-dt) * texel(level, s0+1, t0) +
        ds*dt * texel(level, s0+1, t0+1);
}
```

`BlockedArray::uSize()`	673
`BlockedArray::vSize()`	673
`Clamp()`	677
`MIPMap`	417
`MIPMap::texel()`	423

11.5.4 Elliptically Weighted Average

The elliptically weighted average (EWA) algorithm fits an ellipse to the two axes in texture space given by the texture coordinate differentials and then filters the texture with a Gaussian filter function (see Figure 11.15). It is widely regarded as one of the best texture filtering algorithms in graphics and has been carefully derived from the basic principles of sampling theory. Unlike the triangle filter in the previous section, it can filter over arbitrarily-orented regions of the texture, with some flexibility of different filter extents in different directions. This type of filter is known as *isotropic*. This capability greatly improves the quality of its results, since it can properly adapt to different sampling rates along the two image axes.

We won't show its full derivation here, though we do note that it is distinguished by being a *unified resampling filter*: it simultaneously computes the result of a Gaussian filtered texture function convolved with a Gaussian reconstruction filter in image space. This is contrast to many other texture filtering methods that ignore the effect of the image space filter, or equivalently assume that it is a box. Even if a Gaussian isn't being used for filtering the samples for the image being rendered, taking some account of the spatial variation of the image filter improves the results, assuming that the filter being used is somewhat similar in shape to the Gaussian, as the Mitchell and windowed sinc filters are.

⟨*MIPMap Method Definitions*⟩+≡

```
template <class T>
T MIPMap<T>::Lookup(Float s, Float t, Float ds0, Float dt0,
        Float ds1, Float dt1) const {
    ⟨Compute ellipse minor and major axes⟩
    ⟨Clamp ellipse eccentricity if too large⟩
    ⟨Choose level of detail for EWA lookup⟩
    ⟨Do EWA filtering at appropriate level⟩
}
```

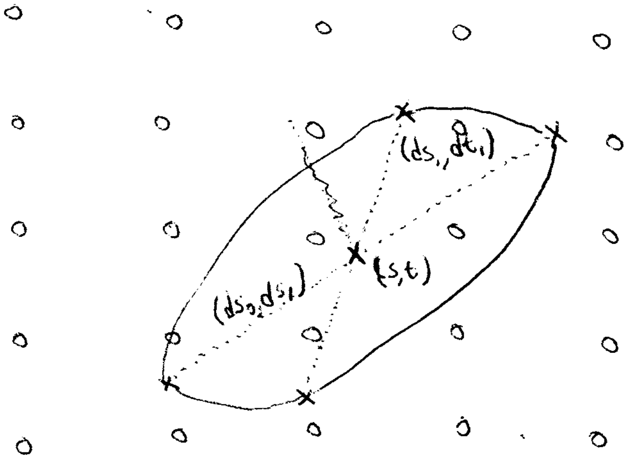


Figure 11.15: The EWA filter applies a Gaussian filter to the texels in an elliptical area around the evaluation point. The extent of the ellipse is such that its edge passes through the positions of the adjacent texture samples as estimated by the texture coordinate partial derivatives.

417 MIPMap

The screen-space partial derivatives of the texture coordinates are the axes of the ellipse. This method starts out by determining which of the two axes is the major axis (the longer of the two) and which is the minor, swapping them if needed so that `ds0,dt0` is the major axis. The length of the minor axis will be used shortly to select a MIP map level to use.

⟨*Compute ellipse minor and major axes*⟩≡

```
if (ds0*ds0 + dt0*dt0 < ds1*ds1 + dt1*dt1) {
    swap(ds0, ds1);
    swap(dt0, dt1);
}
Float majorLength = sqrtf(ds0*ds0 + dt0*dt0);
Float minorLength = sqrtf(ds1*ds1 + dt1*dt1);
```

Next the *eccentricity* of the ellipse is computed–this is the ratio of the length of the major axis to the length of the minor axis. A large eccentricity indicates a very long and skinny ellipse. Because this method filters texels from a MIP map level chosen based on the length of the minor axis, highly eccentric ellipses mean that a large number of texels need to be filtered. To avoid this expense (and to ensure that any EWA lookup takes a bounded amount of time), the length of the minor axis may be increased to limit the eccentricity. The result may be an overly-blurred result for such regions, though this usually isn't noticeable when it happens.

Why isn't the max eccentricity a user-settable parameter? I hate magic constants.

⟨Clamp ellipse eccentricity if too large⟩≡

```
const Float maxEccentricity = 8.f;
if (minorLength * maxEccentricity < majorLength) {
    Float scale = majorLength / (minorLength * maxEccentricity);
    ds1 *= scale;
    dt1 *= scale;
    minorLength *= scale;
}
```

Like the triangle filter, the EWA filter uses the image pyramid to reduce the number of texels to be filtered for a particular texture lookup. It chooses a MIP map level such the minor axis of the ellipse has a total width of five texels at that level. Given the limited eccentricity of the ellipse due to the clamping above, the total number of texels use will be bounded.

Given the length of the minor axis, the computation to find the appropriate pyramid level is the same as was used for the triangle filter. Here, however, filtering is only done at one level of the pyramid, rather than blending between two filtered results. There is some danger of seeing the transition points between MIP map levels when filtering is only done on one level, though this is much less easily seen with the EWA filter than with the triangle filter.

```
      Log2Int()  678
         MIPMap  417
MIPMap::texel()  423
```

⟨Choose level of detail for EWA lookup⟩≡

```
int lod = max(0, nLevels - 1 + Log2Int(minorLength));
```

If the appropriate level is beyond the top of the pyramid, this method can immediately return the average texture value from the top level without doing any filtering. Otherwise the `EWA()` method actually applies the filter.

⟨Do EWA filtering at appropriate level⟩≡

```
if (lod >= nLevels-1)
    return texel(nLevels-1, 0, 0);
else
    return EWA(s, t, ds0, dt0, ds1, dt1, lod);
```

⟨MIPMap Method Definitions⟩+≡

```
template <class T>
T MIPMap<T>::EWA(Float s, Float t, Float ds0, Float dt0,
        Float ds1, Float dt1, int level) const {
    ⟨Convert EWA coordinates to appropriate scale for level⟩
    ⟨Compute ellipse coefficients to bound EWA filter region⟩
    ⟨Compute the ellipse's (s,t) bounding box in texture space⟩
    ⟨Scan over ellipse bound and compute quadratic equation⟩
}
```

The `MIPMap::EWA()` method first converts from texture coordinates in $[0,1]$ to coordinates and differentials in terms of the resolution of the chosen MIP map level. It also subtracts 0.5 from the continuous position coordinate to align the sample point with the discrete texel coordinates, as was done in `MIPMap::triangle()`.

⟨*Convert EWA coordinates to appropriate scale for level*⟩≡

```
s = s * pyramid[level]->uSize() - 0.5f;
t = t * pyramid[level]->vSize() - 0.5f;
ds0 *= pyramid[level]->uSize();
dt0 *= pyramid[level]->vSize();
ds1 *= pyramid[level]->uSize();
dt1 *= pyramid[level]->vSize();
```

Next it is necessary to compute the coefficients of the implicit equation for the ellipse with axes (ds0,dt0) and (ds1,dt1) and centered at the origin. Placing the ellipse at the origin rather than at (s,t) simplifies the implicit equation and the computation of its coefficients, and can be easily corrected for when the equation is evaluated later. The general form of the implicit equation for all points (s,t) inside such an ellipse is

$$e(s,t) = As^2 + Bst + Ct^2 < F,$$

though it is more computationally efficient to divide through by F and express this as

$$e(s,t) = \frac{A}{F}s^2 + \frac{B}{F}st + \frac{C}{F}t^2 = A's^2 + B'st + C't^2 < 1.$$

We will not derive the equations that give the values of the coefficients, though the interested reader can algebraically verify their correctness.[3]

673 BlockedArray::uSize()
673 BlockedArray::vSize()

⟨*Compute ellipse coefficients to bound EWA filter region*⟩≡

```
Float A = dt0*dt0 + dt1*dt1 + 1;
Float B = -2.f * (ds0*dt0 + ds1*dt1);
Float C = ds0*ds0 + ds1*ds1 + 1;
Float invF = 1.f / (A*C - B*B*0.25f);
A *= invF;
B *= invF;
C *= invF;
```

The next step is to find the axis-aligned bounding box in discrete integer texel coordinates of the texels that are potentially inside the ellipse. The EWA algorithm loops over all of these candidate texels, filtering the contributions of those that are in fact inside the ellipse. The bounding box is found by determining the minimum and maximum value that the ellipse takes in the s and t directions. These extrema can be calculated by finding the partial derivatives $\partial e(s,t)/\partial s$ and $\partial e(s,t)/\partial t$, finding their solutions for $s = 0$ and $t = 0$, and adding the offset to the ellipse center. For brevity, we will not include the derivation for these expressions here.

⟨*Compute the ellipse's (s,t) bounding box in texture space*⟩≡

```
Float det = -B*B + 4.f*A*C;
Float invDet = 1.f / det;
Float uSqrt = sqrtf(det * C), vSqrt = sqrtf(A * det);
int s0 = Ceil2Int (s - 2.f * invDet * uSqrt);
int s1 = Floor2Int(s + 2.f * invDet * uSqrt);
int t0 = Ceil2Int (t - 2.f * invDet * vSqrt);
int t1 = Floor2Int(t + 2.f * invDet * vSqrt);
```

[3] Heckbert's thesis has the original derivation (Heckbert 1989, p. 80). A and C have an extra term of 1 added to them to ensure that the ellipse is a minimum of one texel separation wide. This ensures that the ellipse will not fall between the texels when magnifying at the most detailed level.

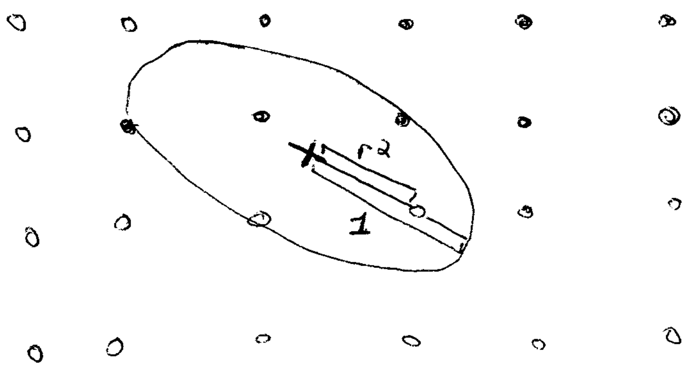


Figure 11.16: finding the r^2 ellipse value for the EWA filter table lookup

Now that the bounding box is known, the EWA algorithm loops over the texels, transforming each one to the coordinate system where the texture lookup point (s,t) is at the origin. It then evaluates the ellipse equation to see if the texel is inside the ellipse (see Figure 11.16.) The weight of each inside texel is computed with a Gaussian centered at the middle of the ellipse. The final filtered value returned is a weighted sum over texels (s',t') inside the ellipse, where f is the Gaussian filter function:

$$\frac{\sum f(s'-s,t'-t)t(s',t')}{\sum f(s'-s,t'-t)}.$$

⟨*Scan over ellipse bound and compute quadratic equation*⟩≡

```
T num(0.);
Float den = 0;
for (int it = t0; it <= t1; ++it) {
    Float tt = it - t;
    for (int is = s0; is <= s1; ++is) {
        Float ss = is - s;
        ⟨Compute squared radius and filter texel if inside ellipse⟩
    }
}
return num / den;
```

A nice feature of the implicit equation is that its value at a particular texel is the squared ratio of the distance from the center of the ellipse to the texel to the distance from the center of the ellipse to the ellipse boundary along the line through that texel (see Figure 11.16. This value is used to index into a precomputed look-up table of Gaussian filter function values.

⟨*Compute squared radius and filter texel if inside ellipse*⟩≡

```
Float r2 = A*ss*ss + B*ss*tt + C*tt*tt;
if (r2 < 1.) {
    Float weight = weightLut[min(Float2Int(r2 * WEIGHT_LUT_SIZE),
        WEIGHT_LUT_SIZE-1)];
    num += texel(level, is, it) * weight;
    den += weight;
}
```

The lookup table is initialized the first time a MIPMap is constructed. Because it will be indexed with squared distances from the filter center r^2, each entry stores a value $e^{-\alpha r}$, rather than $e^{-\alpha r^2}$.

⟨*MIPMap Private Data*⟩+≡

```
#define WEIGHT_LUT_SIZE 128
static Float *weightLut;
```

⟨*Initialize EWA filter weights if needed*⟩≡

```
if (!weightLut) {
    weightLut = (Float *)AllocAligned(WEIGHT_LUT_SIZE *
        sizeof(Float));
    for (int i = 0; i < WEIGHT_LUT_SIZE; ++i) {
        Float alpha = 2;
        Float r2 = float(i) / float(WEIGHT_LUT_SIZE - 1);
        weightLut[i] = expf(-alpha * r2);
    }
}
```

667 AllocAligned()
410 IdentityMapping3D
417 MIPMap
423 MIPMap::texel()
410 TextureMapping3D

11.6 Solid and Procedural Texturing

ALL OF THESE SHOULD HAVE EXAMPLE RENDERINGS – can we use the quadrics RIB file from CS348b circa 2001?

Once one starts to think of (s,t) texture coordinates as quantities that can be computed in a number of ways–not just from the parametric coordinates of the surface, the next step is to consider textures themselves as functions that can be computed in many ways, not necessarily by filtering an image map. Given this generalization to *procedural texturing*, it's natural to consider texture functions defined over a three-dimensional domain (often called *solid textures*) rather than just 2D (s,t). The nice thing about solid textures is that all objects have a natural three-dimensional texture mapping–the object-space position. This is a substantial advantage for texturing objects that don't have a natural two-dimensional parameterization (e.g. triangle meshes and implicit surfaces), and for objects that have a distorted parameterization (e.g. the poles of a sphere.) In preparation for this, Section 11.3.5 defined a general TextureMapping3D interface to compute 3D texture coordinates as well as an IdentityMapping3D implementation.

The problem that solid textures introduce is texture representation; a three-dimensional bitmap takes up a fair amount of storage space, and is much harder to acquire than a two-dimensional texture map. Therefore, procedural texturing came into being around the same time as solid texturing–the idea that short programs could be used to generate texture values at arbitrary positions on surfaces in the scene.

A simple example of this idea is a procedural sine wave. If we wanted to use a sine wave for bump-mapping (for example, to simulate waves in water), it would be inefficient and inaccurate to precompute values of the function at a grid of points and then store them in an image map. Instead, it makes much more sense to evaluate the `sin` function at points on the surface as needed.

If we can describe a three-dimensional function that describes the colors of wood-grain in a solid block of wood, for instance, then we can generate images of complex objects that appear to be carved from wood. Over the years, procedural texturing has grown in application considerably as techniques have been developed to describe more and more complex surfaces procedurally.

Procedural texturing has a number of other interesting implications. First, it can be used to reduce overall memory requirements for rendering, by avoiding the storage of large, high-resolution texture maps. In addition, procedural shading gives the promise of potentially infinite detail; as the viewer approaches an object, the texturing function is evaluated at the points being shaded, which naturally leads to the right amount of detail being visible. In contrast, image texture maps typically become blurry when the viewer is too close to them. On the other hand, procedural textures can be much more difficult to control than image maps.

Shape	63
Spectrum	181
Texture	394
TextureMapping2D	405

Another difficulty with procedural textures is anti-aliasing. Procedural textures are often expensive to evaluate, and point samples don't fully characterize the behavior of function (as they do with image maps). Because we would like to remove high-frequency information in the texture function before we take samples from it, we need to be aware of the frequency content of the various steps we take along the way so we can avoid introducing high frequencies. Though this sounds daunting, there are a handful of techniques that work well to handle this.

Here we will first introduce some very simple procedural textures, then discuss basic tools for introducing more complex variations on them, and then implement a number of more complex procedural textures.

11.6.1 UV texture

Our first procedural texture converts converts the surface's (u,v) coordinates into the first two components of a `Spectrum`. This is especially useful when debugging the parameterization of a new `Shape`.

⟨*Texture Class Declarations*⟩+≡
```
class UVTexture : public Texture<Spectrum> {
public:
    ⟨UVTexture Public Methods⟩
private:
    TextureMapping2D *mapping;
};
```

⟨*Texture Method Definitions*⟩+≡

```
Spectrum UVTexture::Evaluate(
        const DifferentialGeometry &dg) const {
    Float u, v, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &u, &v, &dsdx, &dtdx, &dsdy, &dtdy);
    Float cs[COLOR_SAMPLES];
    memset(cs, 0, COLOR_SAMPLES * sizeof(Float));
    cs[0] = u;
    cs[1] = v;
    return Spectrum(cs);
}
```

11.6.2 Checkerboard

The checkerboard is the canonical procedural texture. The (s,t) texture coordinates are used to break up parameter space into square regions which are shaded with alternating patterns. Rather than just supporting checkerboards that switch between two fixed colors, we allow the user to pass in two texture maps to color the alternating regions. The canonical checkerboard is obtained by passing two ConstantTextures.

⟨*Texture Class Declarations*⟩+≡

```
template <class T> class Checkerboard2D : public Texture<T> {
public:
    ⟨Checkerboard2D Public Methods⟩
private:
    ⟨Checkerboard2D Private Data⟩
};
```

For simplicity, the frequency of the check function is 1 in (s,t) space. This can always be changed with an appropriate scale of the (s,t) coordinates by the TextureMapping2D.

⟨*Checkerboard2D Public Methods*⟩≡

```
Checkerboard2D(TextureMapping2D *m, Texture<T> *c1,
        Texture<T> *c2, const string &aa) {
    mapping = m;
    tex1 = c1;
    tex2 = c2;
    ⟨Select anti-aliasing method for Checkerboard2D⟩
}
```

⟨*Checkerboard2D Private Data*⟩≡

```
Texture<T> *tex1, *tex2;
TextureMapping2D *mapping;
```

The checkerboard is a good procedural texture for demonstrating trade-offs among various general anti-aliasing approaches for procedural textures. Therefore, we will implement a number of them, selectable via a string passed to the constructor. The image sequence in Figure 11.8 shows the reults of these various anti-aliasing strategies.

180	COLOR_SAMPLES
395	ConstantTexture
58	DifferentialGeometry
181	Spectrum
394	Texture
405	TextureMapping2D
405	TextureMapping2D::Map()
432	UVTexture
432	UVTexture::mapping

⟨*Select anti-aliasing method for* `Checkerboard2D`⟩≡

```
if (aa == "none") aaMethod = NONE;
else if (aa == "supersample") aaMethod = SUPERSAMPLE;
else if (aa == "closedform") aaMethod = CLOSEDFORM;
else {
    Warning("Anti-aliasing mode \"%s\" not understood "
        "by Checkerboard2D, defaulting to \"supersample\"", aa.c_str());
    aaMethod = SUPERSAMPLE;
}
```

⟨*Checkerboard2D Private Data*⟩+≡

```
enum { NONE, SUPERSAMPLE, CLOSEDFORM } aaMethod;
```

The evaluating routine does the usual texture coordinate and differential computation and then uses the appropriate fragment to compute an anti-aliased checkerboard value (or not, if point-sampling has been selected).

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T>
T Checkerboard2D<T>::Evaluate(
        const DifferentialGeometry &dg) const {
    Float s, t, dsdx, dtdx, dsdy, dtdy;
    mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
    if (aaMethod == CLOSEDFORM) {
        ⟨Compute closed form box-filtered Checkerboard2D value⟩
    }
    else if (aaMethod == SUPERSAMPLE) {
        ⟨Supersample Checkerboard2D⟩
    }
    ⟨Point sample Checkerboard2D⟩
}
```

`Checkerboard2D`	433
`Checkerboard2D::mapping`	433
`Checkerboard2D::tex1`	433
`Checkerboard2D::tex2`	433
`DifferentialGeometry`	58
`Texture`	394
`Texture::Evaluate()`	395
`TextureMapping2D`	405
`TextureMapping2D::Map()`	405

The simplest case is to ignore anti-aliasing and just point-sample the checkerboard texture at the point being shaded. For this case, after getting the (s,t) texture coordinates from the `TextureMapping2D`, we compute the integer checkerboard coordinates for that (s,t) position and see if this has odd or even parity to determine which of the two texture maps to evaluate.

⟨*Point sample* `Checkerboard2D`⟩≡

```
if ((Floor2Int(s) + Floor2Int(t)) % 2 == 0)
    return tex1->Evaluate(dg);
return tex2->Evaluate(dg);
```

XXX make clear that “filter region” is a misnomer.

Given how bad aliasing can be in a point-sampled checkerboard texture, we will invest some effort to anti-alias it properly. The easiest case happens when the entire filter region lies inside a single check (Figure 11.17). In this case, we simply need to determine which of the check types we are inside and evaluate that one. As long as the `Texture` inside that check does appropriate anti-aliasing itself, the result for this case will be anti-aliased.

Figure 11.17: the easy case for filtering the checkerboard.

⟨*Compute closed form box-filtered* `Checkerboard2D` *value*⟩≡
  ⟨*Evaluate single check if filter is entirely inside one of them*⟩
  ⟨*Apply box-filter to checkerboard region*⟩

We can easily see if the entire filter region is inside a single check by computing its (s,t) bounding box and seeing if all of it maps to a single integer (s,t) checker coordinate.

⟨*Evaluate single check if filter is entirely inside one of them*⟩≡

```
Float ds = .5f * max(fabsf(dsdx), fabsf(dsdy));
Float dt = .5f * max(fabsf(dtdx), fabsf(dtdy));
Float s0 = s - ds, s1 = s + ds;
Float t0 = t - dt, t1 = t + dt;
if (Floor2Int(s0) == Floor2Int(s1) &&
    Floor2Int(t0) == Floor2Int(t1)) {
    ⟨Point sample Checkerboard2D⟩
}
```

Otherwise, we can approximate the filtered value by first computing a floating point value that indicates what fraction of the filter region covers each of the two check types. We are effectively computing the average of the 2D step function that takes on the value 0 where we are in `tex1` and 1 when we are in `tex2`, over the filter region. The left side of Figure 11.18 shows a graph of the checkerboard function $c(x)$, defined as:

$$c(x) = \begin{cases} 0 & : \lfloor x \rfloor \text{is odd} \\ 1 & : \text{otherwise} \end{cases}$$

Given the average value value, we can blend between the two sub-textures, according to what fraction of the filter region each one is visible for.

We can use the integral of the 1D checkerboard function $c(x)$ to compute the average value of the function over some extent. Inspection of the graph reveals that

$$\int_0^x c(x)\mathrm{d}x = \lfloor x/2 \rfloor + 2\max(x/2 - \lfloor x/2 \rfloor - .5, 0).$$

To compute the average value of the step function in one dimension, we separately compute the integral of the checkerboard in each 1D direction in order to compute its average value over the filter region. Figure 11.18 shows graphs of the 1D step function (left) and its integral (right).

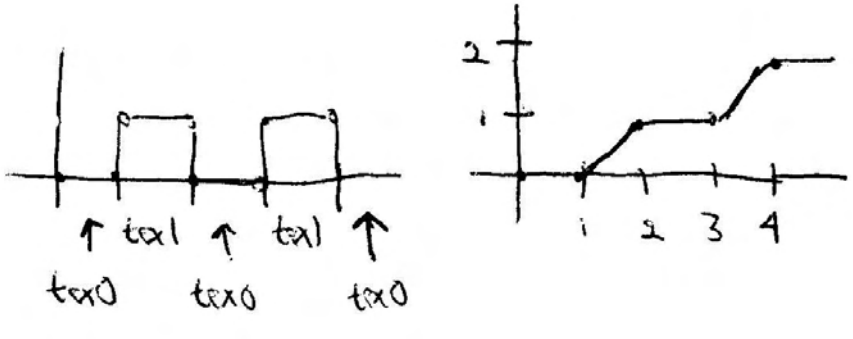


Figure 11.18: left, the 1D step function used to define the checkerboard texture $c(x)$. right, a graph of the value of the integral $\int_0^x c(x)\mathrm{d}x$.

XXX need to carefully explain `area2` computation XXX
I agree –Humper

⟨*Apply box-filter to checkerboard region*⟩≡

Checkerboard2D::tex1 433
Texture::Evaluate() 395

```
#define BUMPINT(x) \
    (Floor2Int((x)/2) + 2.f * max((x/2)-Floor2Int(x/2) - .5f, 0.f))
Float sint = (BUMPINT(s1) - BUMPINT(s0)) / (2. * ds);
Float tint = (BUMPINT(t1) - BUMPINT(t0)) / (2. * dt);
Float area2 = sint + tint - 2.f * sint * tint;
if (ds > 1.f || dt > 1.f)
    area2 = .5f;
return (1.f - area2) * tex1->Evaluate(dg) +
    area2 * tex2->Evaluate(dg);
```

The final checkerboard anti-aliasing method we show is classic supersampling. We will evaluate the checkerboard at a set of random stratified positions around the (s,t) point in texture space, jittered so that they roughly cover the filter area.

An extra point in favor of supersampling is that the box filter approach described previously is actually not completely correct. Specifically, it assumes that we can compute the correct overall anti-aliased result by determining how much of the filter covers each of the two types of checker and then evaluating both of them, blending between their results appropriately. The problem with this assumption is that each of the sub-textures will evaluate and anti-alias itself as if it were completely visible throughout all of the filter region.

Figure 11.19 shows a case where this is an incorrect assumption. There, we have a checkerboard texture where each of the sub-textures is also a checkerboard. The textures are all configured such that the result of the main checkerboard texture is a solid grey color–all of the black parts of the sub-textures are completely hidden. However, if the sub-textures filter themselves with the incorrect assumption described above, then some of the black will “leak in” to the final computed result. This is a somewhat contrived worst-case, and the box filter approach does work correctly for sub-textures that are just a constant value, for example. However, the super-sampling approach we will implement here does not suffer from this problem.

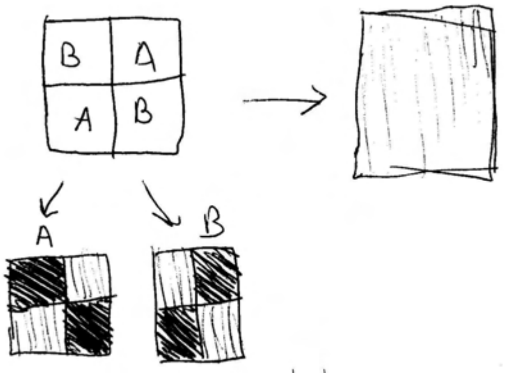


Figure 11.19: The checkerboard with nefarious sub-textures can cause us trouble!

We take a fixed number of stratified samples, N_SAMPLES in the s and t directions. For each one, we initialize a DifferentialGeometry object for the sample position and then point-sample the checkerboard texture function.

⟨*Supersample* Checkerboard2D⟩≡

```
#define N_SAMPLES 4
Float samples[2*N_SAMPLES*N_SAMPLES];
StratifiedSample2D(samples, N_SAMPLES, N_SAMPLES);
T value = 0.;
Float filterSum = 0.;
for (int i = 0; i < N_SAMPLES*N_SAMPLES; ++i) {
    ⟨Compute new differential geometry for supersample location⟩
    ⟨Compute (s,t) for supersample and evaluate sub-texture⟩
}
return value / filterSum;
```

58 DifferentialGeometry
400 DifferentialGeometry::dudx
400 DifferentialGeometry::dudy
400 DifferentialGeometry::dvdx
400 DifferentialGeometry::dvdy
378 DifferentialGeometry::Shift()

This next paragraph is basically opaque to me. What the hell is going on?

We choose texture samples over the rough pixel-separation-sized texture-space filter area; the DifferentialGeometry::Shift() method takes care of computing an approximation to the appropriate DifferentialGeometry. We then scale down the screen-space differentials for the shifted point, so that any anti-aliasing done by the sub-texture will be over an appropriately reduced area.

⟨*Compute new differential geometry for supersample location*⟩≡

```
DifferentialGeometry dgs;
Float dx = samples[2*i]   - 0.5f;
Float dy = samples[2*i+1] - 0.5f;
dg.Shift(dx, dy, &dgs);
dgs.dudx /= N_SAMPLES;
dgs.dudy /= N_SAMPLES;
dgs.dvdx /= N_SAMPLES;
dgs.dvdy /= N_SAMPLES;
```

Finally, we weight the sample values with a Gaussian filter and accumulate the result from the appropriate sub-texture into value.

⟨*Compute* (s,t) *for supersample and evaluate sub-texture*⟩≡

```
Float ss, ts, dsdxs, dtdxs, dsdys, dtdys;
mapping->Map(dgs, &ss, &ts, &dsdxs, &dtdxs, &dsdys, &dtdys);
Float wt = expf(-2.f * (dx*dx + dy*dy));
filterSum += wt;
if ((Floor2Int(ss) + Floor2Int(ts)) % 2 == 0)
    value += wt * tex1->Evaluate(dgs);
else
    value += wt * tex2->Evaluate(dgs);
```

Figure 11.20 shows these three anti-aliasing approaches for the checkerboard in practice.

11.6.3 Solid Checkerboard

The `Checkerboard2D` class from the previous section wraps a checkerboard pattern *around* the object in parameter space. We can also define a solid checkerboard pattern based on three-dimensional texture coordinates. This way, the object appears carved out of 3D checker cubes. Like the 2D variant, we provide two texture functions to choose between. Note that these two textures need not be solid textures themselves; we are merely choosing between them based on the 3D position of the hit point.

Checkerboard2D	433
Checkerboard2D::tex1	433
Checkerboard2D::tex2	433
Texture	394
Texture::Evaluate()	395
TextureMapping2D::Map()	405
TextureMapping3D	410

⟨*Texture Class Declarations*⟩+≡

```
template <class T> class Checkerboard3D : public Texture<T> {
public:
    ⟨Checkerboard3D Public Methods⟩
private:
    ⟨Checkerboard3D Private Data⟩
};
```

⟨*Checkerboard3D Public Methods*⟩≡

```
Checkerboard3D(TextureMapping3D *m, Texture<T> *c1,
        Texture<T> *c2) {
    mapping = m;
    tex1 = c1;
    tex2 = c2;
}
```

⟨*Checkerboard3D Private Data*⟩≡

```
Texture<T> *tex1, *tex2;
TextureMapping3D *mapping;
```

Ignoring anti-aliasing, the basic computation to see if we are inside a 3D checker region is

```
((Floor2Int(P.x) + Floor2Int(P.y) + Floor2Int(P.z)) % 2
== 0).
```

Here we will just use the same basic supersampling approach as we used in the 2D checkerboard. The code with the implementation is elided here, however, since it

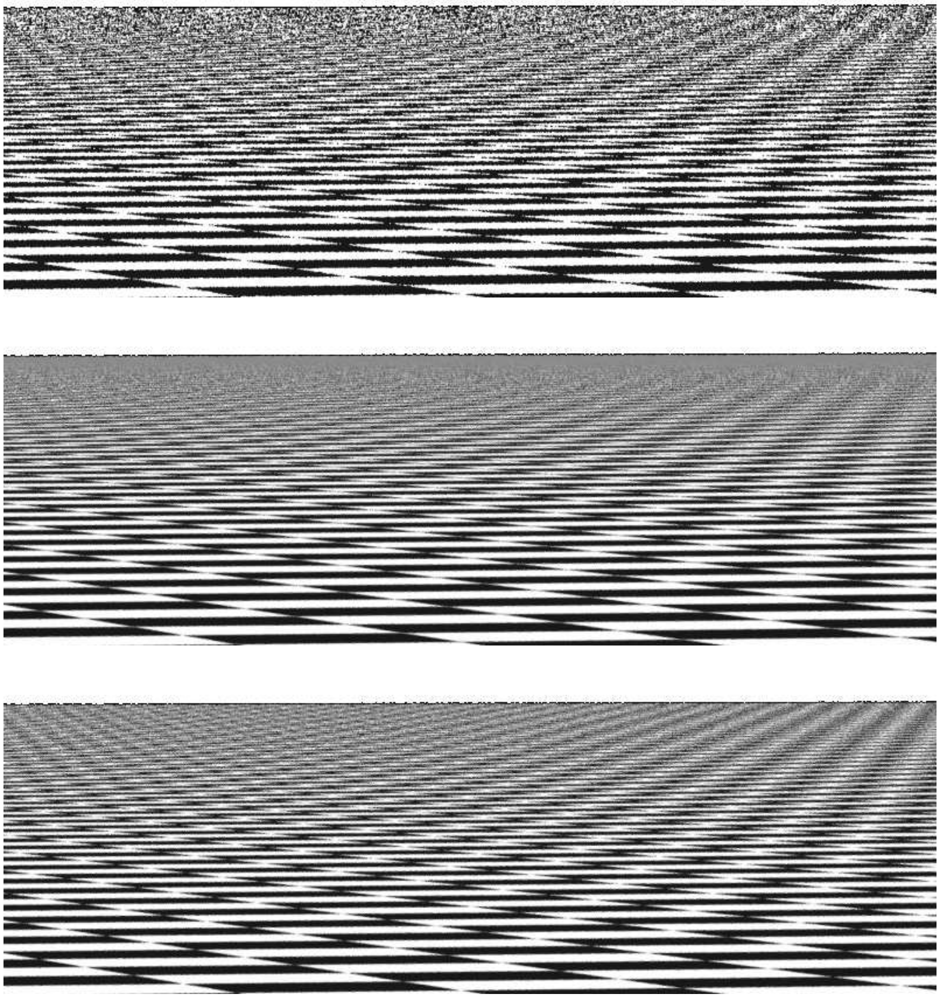


Figure 11.20: Comparisons of the three approaches for anti-aliasing in procedural textures, applied to the checkerboard texture; all images were rendered with one sample per pixel. In the top image, no effort has been made to remove high frequency variation from the texture function, so there are severe artifacts in the image. The middle image shows the approach based on computing the filter region in texture space and averaging the texture function over that area. On the bottom is an image where the checkerboard function was super-sampled 16 times in texture space at each shading point. Both the area-averaging and the supersampling approaches give substantially better results than the top image. In this example, supersampling gives the best results, since the averaging approach has blurred out the checkerboard pattern sooner than was needed, due to over-estimating the filter region. However, in general, the averaging approach will guarantee that there is no aliasing due to the texture, which is desirable.

is substantially the same as in the 2D case, just with the inside-check test modified as above.

```
⟨Texture Template Method Definitions⟩+≡
  template <class T>
  T Checkerboard3D<T>::Evaluate(
          const DifferentialGeometry &dg) const {
      ⟨Supersample Checkerboard3D⟩
  }
```

Example rendering!

11.7 Noise

In order to write solid textures for complex surface appearances, it is helpful to be able to introduce some controlled variation to the process. Consider a wood floor made of individual planks; each plank's color is likely to be slightly different than the others. Or consider a windswept lake; we might want to have waves of similar amplitude across the entire like, but we don't want them to be the same (as they would be if they were constructed from a collection of sine waves.)

These sorts of problems are typically solved using what is known as a *noise function*. In general, noise functions used in graphics are smoothly-varying functions taking $\mathbb{R}^n \to [-1,1]$, for at least $n = 1,2,3$, but without obviously repeating patterns. One of the most crucial properties of a noise function is that they are often bandlimited, with a maximum frequency of roughly 1, which makes it possible to control their frequency content so that they do not introduce frequencies higher than allowed by the Nyquist limit.

Checkerboard3D 438
DifferentialGeometry 58

Many of the noise functions that have been developed are built on the idea of an integer lattice over $\mathbb{R}^3$. Some value is associated with each integer (x,y,z) position in space. Then, given an arbitrary position in space, the eight adjoining lattice values are found. These lattice values are then interpolated to compute the noise value at the particular point. This can be generalized or restricted to more or fewer dimensions d, where the number of lattice points is 2^d.

A simple example of this is *value noise*. Pseudo-random numbers between -1 and 1 are associated with each lattice point, and actual noise values are computed with trilinear interpolation or with a more complex spline interpolant, which can give a smoother result by avoiding derivative discontinuities when moving from one lattice cell to another.

For such a noise function, given an integer (x,y,z) lattice point, we must be able to efficiently compute its parameter value. Because it is infeasible to store values for all possible (x,y,z) points, some compact representation is needed. One option is to use a hash function, where the coordinates are hashed and then used to look up parameters from a fixed-size table of precomputed pseudo-random parameter values. Another option is to use a table of values where the offset into the table is computed with a hash based on x, y, and z.

11.7.1 Perlin Noise

Here we will implement a noise function introduced by Ken Perlin; as such, it is known as *Perlin noise*. It has a value of zero at all (x,y,z) integer lattice points.

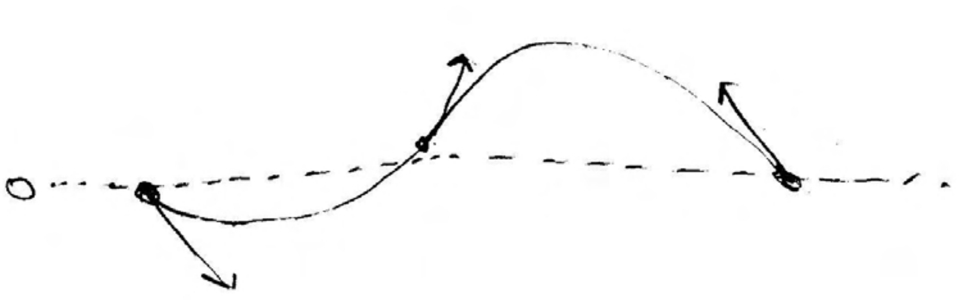


Figure 11.21: The Perlin noise function, shown here in one dimension, is computed by generating a smooth function that is zero but with a given derivative at integer lattice points. The derivatives are used to compute a smooth interpolating curve.

Its variation comes from varying gradient vectors at each lattice point that guide the interpolation of a function in between the points. This noise function has many of the desired characteristics of a noise function described above, is reasonably computationally efficient and is easy to implement. See Figure 11.22 for a graph of Perlin noise.

To evaluate the noise function, we first need to find the eight gradient vectors for the cell the (x,y,z) point is in. Then we just need to do the 3D interpolation.

⟨*Texture Method Definitions*⟩+≡

```
Float Noise(Float x, Float y, Float z) {
    ⟨Compute noise cell coordinates and offsets⟩
    ⟨Compute gradient weights⟩
    ⟨Compute trilinear interpolation of weights⟩
}
```

We first compute the integer coordinates of the cell that the given point lies inside and the fractional offsets of the point from the lower cell corner.

⟨*Compute noise cell coordinates and offsets*⟩≡

```
int ix = Floor2Int(x);
int iy = Floor2Int(y);
int iz = Floor2Int(z);
Float dx = x - ix, dy = y - iy, dz = z - iz;
```

We then compute eight weight values, one for each corner of the cell. Conceptually, each integer lattice point has a random gradient vector associated with it; the weight for any point inside the cell is computed by computing the dot product of the vector from the point to the lower corner of the cell with the dot product of the gradient vector.

This is pretty unclear; need a figure ;)

XXX draw a figure for this, make it more clear XXX

The gradient vectors do not need to be represented explicltly. All of the gradients we will use will only have values −1, 0, or 1 in their coordinates, so that the dot

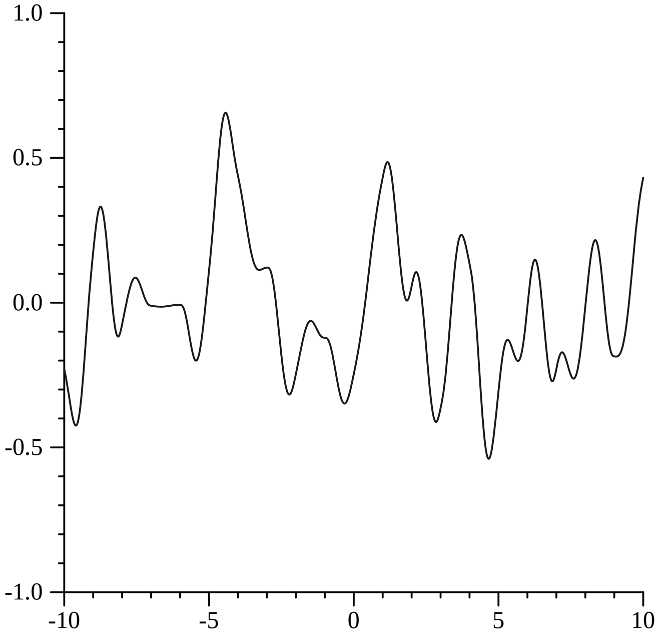


Figure 11.22: Graph of a noise function; important qualities to note include that it is smoothly varying, doesn't have unexpected high frequencies, and ranges between -1 and 1.

Grad() 443
NOISE_PERM_SIZE 443

products reduce to addition of some (possibly negated) components of the vector[4]. The Grad() function handles this computation.

XXX can use and since is power of 2 size... XXX

⟨*Compute gradient weights*⟩≡

```
ix &= (NOISE_PERM_SIZE-1);
iy &= (NOISE_PERM_SIZE-1);
iz &= (NOISE_PERM_SIZE-1);
Float w000 = Grad(ix,   iy,   iz,   dx,   dy,   dz);
Float w100 = Grad(ix+1, iy,   iz,   dx-1, dy,   dz);
Float w010 = Grad(ix,   iy+1, iz,   dx,   dy-1, dz);
Float w110 = Grad(ix+1, iy+1, iz,   dx-1, dy-1, dz);
Float w001 = Grad(ix,   iy,   iz+1, dx,   dy,   dz-1);
Float w101 = Grad(ix+1, iy,   iz+1, dx-1, dy,   dz-1);
Float w011 = Grad(ix,   iy+1, iz+1, dx,   dy-1, dz-1);
Float w111 = Grad(ix+1, iy+1, iz+1, dx-1, dy-1, dz-1);
```

Given an integer lattice point, we use a permutation table to quickly compute a random offset value between 0 and NOISE_PERM_SIZE. We then take the low-order bits of this to determine which gradient vector to use for the point.

In a pre-process, we fill an array of size NOISE_PERM_SIZE with numbers from 0 to NOISE_PERM_SIZE-1 and then randomly permute its elements. We then make an array of size 2*NOISE_PERM_SIZE that holds the resulting permuted table twice in succession. The second copy of the table makes lookups slightly more efficient.

[4] The original formulation of Perlin noise also had a precomputed table of pseudo-random gradient directions, though Perlin has more recently suggested that the randomness from the permutation table is enough to remove regularity from the noise function.

By doing three nested permutations in this way, we avoid any regularity that might be present if we used `NoisePerm[ix+iy+iz]`, where we'd get the same result if `ix` and `iy` were interchanged, etc. By replicating the table twice, we avoid the need to compute modulus values after lookups, like

```
(NoisePerm[ix]+iy) % NOISE_PERM_SIZE
```

Then to compute the gradient value, we...

⟨*Texture Method Definitions*⟩+≡

```
inline Float Grad(int x, int y, int z, Float dx,
        Float dy, Float dz) {
    int h = NoisePerm[NoisePerm[NoisePerm[x]+y]+z];
    h &= 15;
    Float u = h<8 || h==12 || h==13 ? dx : dy;
    Float v = h<4 || h==12 || h==13 ? dy : dz;
    return ((h&1) ? -u : u) + ((h&2) ? -v : v);
}
```

⟨*Perlin Noise Data*⟩≡

```
#define NOISE_PERM_SIZE 256
static int NoisePerm[2 * NOISE_PERM_SIZE] = {
    151, 160, 137, 91, 90, 15, 131, 13, 201, 95, 96,
    53, 194, 233, 7, 225, 140, 36, 103, 30, 69, 142,
    ⟨Noise permutation table⟩
};
```

677 `Lerp()`

Given these eight weights, we want to trilinearly interpolate between them at the point. Rather than interpolating with `dx`, `dy`, and `dz` directly, though, we run each of these values through a smoothing function. This ensures that the noise function has first and second derivative continuity as we move from lattice cell to lattice cell.

⟨*Texture Method Definitions*⟩+≡

```
inline Float NoiseWeight(Float t) {
    Float t3 = t*t*t;
    Float t4 = t3*t;
    return 6.f*t4*t - 15.f*t4 + 10.f*t3;
}
```

⟨*Compute trilinear interpolation of weights*⟩≡

```
Float wx = NoiseWeight(dx);
Float wy = NoiseWeight(dy);
Float wz = NoiseWeight(dz);
Float x00 = Lerp(wx, w000, w100);
Float x10 = Lerp(wx, w010, w110);
Float x01 = Lerp(wx, w001, w101);
Float x11 = Lerp(wx, w011, w111);
Float y0 = Lerp(wy, x00, x10);
Float y1 = Lerp(wy, x01, x11);
return Lerp(wz, y0, y1);
```

For convenience, we’ll also provide a method that takes a `Point` directly.

⟨*Texture Method Definitions*⟩+≡

```
Float Noise(const Point &P) {
    return Noise(P.x, P.y, P.z);
}
```

11.7.2 Random Polka Dots

⟨*Texture Class Declarations*⟩+≡

```
template <class T> class PolkaDots : public Texture<T> {
public:
    ⟨PolkaDots Public Methods⟩
private:
    ⟨PolkaDots Private Data⟩
};
```

To show a basic use of the noise function, we’ll write a polka-dot texture. This texture divides (s,t) texture space into rectangular cells. Each cell has a 50% chance of having a dot inside of it, where the dot is randomly placed inside the cell.

`PolkaDots` takes the usual 2D mapping function, as well as two `Texture`s, one for the regions of the surface outside of the dots and one for the regions inside.

DifferentialGeometry	58
Point	33
Texture	394
Texture::Evaluate()	395
TextureMapping2D	405
TextureMapping2D::Map()	405

⟨*PolkaDots Public Methods*⟩+≡

```
PolkaDots(TextureMapping2D *m, Texture<T> *c1, Texture<T> *c2) {
    mapping = m;
    outsideDot = c1;
    insideDot = c2;
}
```

⟨*PolkaDots Private Data*⟩≡

```
Texture<T> *outsideDot, *insideDot;
TextureMapping2D *mapping;
```

The evaluation function is pretty straightforward. We start by taking the (s,t) texture coordinates and computing integer `sCell` and `tCell` values, which give us the coordinates of the cell that we’re in. (See Figure 11.23.)

⟨*Texture Template Method Definitions*⟩+≡

```
template <class T>
T PolkaDots<T>::Evaluate(const DifferentialGeometry &dg) const {
    ⟨Compute cell incides for dots⟩
    ⟨Return insideDot result if point is inside dot⟩
    return outsideDot->Evaluate(dg);
}
```

⟨*Compute cell incides for dots*⟩≡

```
Float s, t, dsdx, dtdx, dsdy, dtdy;
mapping->Map(dg, &s, &t, &dsdx, &dtdx, &dsdy, &dtdy);
int sCell = Floor2Int(s + .5f), tCell = Floor2Int(t + .5f);
```

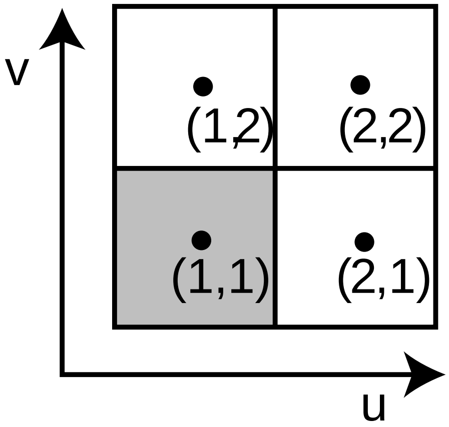


Figure 11.23:

Once we know the cell indices, we need to decide if there is a polka dot in the cell. Obviously, this computation needs to be consistent, so that for all times that this routine runs for points in a particular cell, it always returns the same result. On the other hand, we'd like the result to not be regular. Enter noise: we evaluate the noise function at a position that is the same for all points inside this cell—`sCell+.5, tCell+.5`. If this is greater than zero, we decide that there is a dot in the cell and continue processing.

441 Noise()
444 PolkaDots::insideDot
395 Texture::Evaluate()

Recall that out noise function always returns zero at integer (x,y,z) coordinates, so we don't want to just evaluate it at `sCell, tCell`. Although the 3D noise function would actually be evaluating noise at `sCell, tCell, .5`, slices through noise with integer values for any of the are not as good as with all of them offset.

If there is a dot in the cell, we use the same trick to randomly shift the center of the dot around; we compute a new dot position using noise to offset it from the center of the cell.

Finally, we just need to decide if the (s,t) coordinates are within distance `radius` of the shifted center. We compute their squared distance to the center and compare it to the squared radius.

We will not consider anti-aliasing of the polka dots texture here; an exercise at the end of the chapter outlines how this might be done.

⟨*Return* `insideDot` *result if point is inside dot*⟩≡

```
if (Noise(sCell+.5, tCell+.5) > 0) {
    Float radius = .35f;
    Float maxShift = 0.5f - radius;
    Float sCenter = sCell + maxShift *
        Noise(sCell + 1.5f, tCell + 2.8f);
    Float tCenter = tCell + maxShift *
        Noise(sCell + 4.5f, tCell + 9.8f);
    Float ds = fabsf(s - sCenter), dt = fabsf(t - tCenter);
    if (ds*ds + dt*dt < radius*radius)
        return insideDot->Evaluate(dg);
}
```

This texture, like all procedural textures in this chapter, is an *implicit texture*; in other words, the texture function is written to be able to describe the texture at any particular point being shaded–because it does so in a way such that it squares

11.7.3 FBm

One of the most useful applications of noise is to compute patterns via *spectral synthesis*, where a complex function $f_s(s)$ is defined by a sum of contributions from another function $f(x)$:

$$f_s(x) = \sum_i w_i f(s_i x),$$

for a set of weight values w_i and parameter scale values s_i. If the base function $f(x)$ has a well-defined frequency content (e.g. is a sine or cosine function, or a noise function), then each term $f(s_i x)$ also has a well-defined frequency content–the larger the value of s_i, the higher frequencies result. Each term of the sum is weighted by a weight value w_i, so that the result is a sum of contributions of various frequencies, with different frequency ranges weighted differently.

Typically, the scales s_i are chosen in a geometric progression such that $s_i = 2s_{i-1}$ and the weights are $w_i = 1/2w_{i-1}$. The result is that as higher-frequency variation is added to the function, it has relatively less influence on the overall shape of $f_s(x)$. Each additional term is called an *octave* of noise, borrowing a term from music, since it has twice the frequency content of the previous one. When this scheme is used with Perlin noise, the result is often referred to as "FBm", which stands for "Fractional Brownian motion", a term from fractal geometry that describes a particular type of random process that varies in a similar manner.

Point 33
Vector 27

FBm is a very useful building block for building procedural textures because it gives us a function with more complex variation than plain noise, while still being easy to compute and still having well-defined overall frequency content. The utility function `FBm()` below implements the FBm function. Figure 11.24 shows a graph of it. In addition to taking the point to evaluate the function at and its partial derivatives for anti-aliasing computation, it takes an `omega` parameter, which ranges from zero to one and controls the smoothness of the marble pattern by controlling the falloff of contributions at higher frequencies (values around 0.5 work well), and `octaves`, which gives a maximum number of octaves of noise to use to compute the sum.

```
⟨Texture Method Definitions⟩+≡
  Float FBm(const Point &P, const Vector &dpdx, const Vector &dpdy,
          Float omega, int maxOctaves) {
      ⟨Compute number of octaves for anti-aliased FBm⟩
      ⟨Compute sum of octaves of noise for FBm⟩
      return sum;
  }
```

To anti-alias the FBm function, we will use a technique called *clamping*. The idea is that when we are computing a value based on a sum of components, each with known frequency content, we should stop adding in components that would have frequencies beyond the Nyquist limit and instead add their average value to the sum. Because the average value of `Noise()` is zero, all that we need to do is to

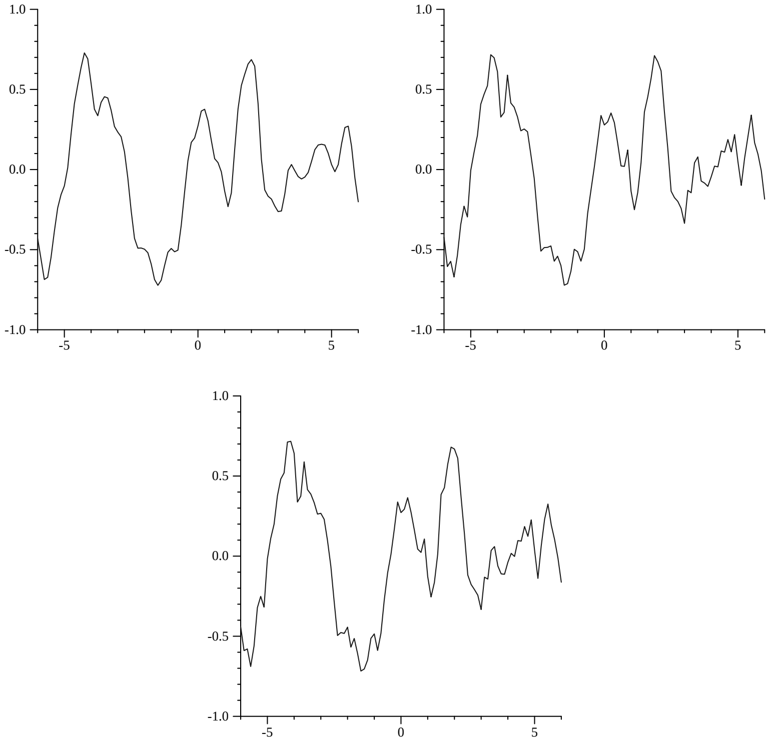


Figure 11.24: Graphs of the `FBm()` function for 2, 4, and 6 octaves of noise. Notice how as more levels of noise are added, the graph has progressively more detail, though its overall shape remains roughly the same.

compute the number of octaves such that none of the terms have excessively high frequencies.

`Noise()` (and thus the first term of $f_s(x)$ as well) has a maximum frequency content of roughly $\omega = 1$. Each subsequent term represents a doubling of frequency content. Therefore, we would like find the appropriate number of terms n such that if the sampling rate in noise space is s, we have

$$2^n \times s = 2\omega = 1.$$

Thus, we have

$$\begin{aligned} 2^{n-1} &= 1/s \\ n-1 &= \log(1/s) \\ n = 1 - 1/2\log(s^2)&. \end{aligned}$$

We can compute the squared sampling rate s^2 by finding the maximum of the length of the differentials $\partial \mathrm{p}/\partial x$ and $\partial \mathrm{p}/\partial y$.

⟨*Compute number of octaves for anti-aliased FBm*⟩≡

```
Float s2 = max(dpdx.LengthSquared(), dpdy.LengthSquared());
Float foctaves = min((Float)maxOctaves, 1.f - .5f * Log2(s2));
int octaves = Floor2Int(foctaves);
```

Clamp()	677
Log2()	677
Noise()	441
Vector::LengthSquared()	30

We then compute the given integer number of octaves up to the Nyquist limit and fade in the last octave, according to the fractional part of `foctaves`. This ensures that successive octaves of noise fade in gradually, rather than appearing abruptly, which can cause visually-noticeable artifacts at the transitions. We increase the frequency between octaves by 1.99, rather than by a factor of two, in order to reduce the impact of the fact that the noise function is zero at integer lattice points. This breaks up that regularity across octaves of noise, **XXX**.

⟨*Compute sum of octaves of noise for FBm*⟩≡

```
Float sum = 0., lambda = 1., o = 1.;
for (int i = 0; i < octaves; ++i) {
    sum += o * Noise(lambda * P);
    lambda *= 1.99f;
    o *= omega;
}
Float partialOctave = foctaves - octaves;
sum += o * SmoothStep(.3, .7, partialOctave) * Noise(lambda * P);
```

The `SmoothStep()` function it takes a minimum and maximum value and a point at which to evaluate the a smooth interpolating function. If the point is below the minimum, zero is returned, and if it's above the maximum, one is returned. Otherwise it smoothly interpolates between zero and one.

⟨*Global Inline Functions*⟩+≡

```
inline Float SmoothStep(Float min, Float max, Float value) {
    Float v = Clamp((value - min) / (max - min), 0., 1.);
    return -2.f * v * v * v + 3.f * v * v;
}
```

11.7.4 Bumpy

FBm is useful by itself as a source of random variation for bump-mapping. The `BumpyTexture` is a `Float`-valued texture that uses FBm to compute bump map offsets.

⟨*Texture Class Declarations*⟩+≡

```
class BumpyTexture : public Texture<Float> {
public:
    ⟨BumpyTexture Public Methods⟩
private:
    ⟨BumpyTexture Private Data⟩
};
```

⟨*BumpyTexture Public Methods*⟩+≡

```
BumpyTexture(int oct, Float roughness, TextureMapping3D *map) {
    omega = roughness;
    octaves = oct;
    mapping = map;
}
```

⟨*BumpyTexture Private Data*⟩≡

```
int octaves;
Float omega;
TextureMapping3D *mapping;
```

⟨*Texture Method Definitions*⟩+≡

```
Float BumpyTexture::Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point P = mapping->Map(dg, &dpdx, &dpdy);
    return FBm(P, dpdx, dpdy, omega, octaves);
}
```

58	DifferentialGeometry
446	FBm()
450	MarbleTexture::mapping
450	MarbleTexture::octaves
450	MarbleTexture::omega
33	Point
181	Spectrum
394	Texture
410	TextureMapping3D
410	TextureMapping3D::Map()
27	Vector

11.7.5 Marble

⟨*Texture Class Declarations*⟩+≡

```
class MarbleTexture : public Texture<Spectrum> {
public:
    ⟨MarbleTexture Public Methods⟩
private:
    ⟨MarbleTexture Private Data⟩
};
```

⟨*MarbleTexture Public Methods*⟩+≡

```
MarbleTexture(int oct, Float roughness, TextureMapping3D *map) {
    omega = roughness;
    octaves = oct;
    mapping = map;
}
```

```
⟨MarbleTexture Private Data⟩≡
  int octaves;
  Float omega;
  TextureMapping3D *mapping;
```

```
⟨Texture Method Definitions⟩+≡
  Spectrum MarbleTexture::Evaluate(const DifferentialGeometry &dg) const {
      Vector dpdx, dpdy;
      Point P = mapping->Map(dg, &dpdx, &dpdy);

      Float scale = .05; // XXX fold into xform...
      Float variation = .3;
      Float marble = scale * P.y + variation * FBm(scale * P, scale * dpdx, scale
      Float t = fabsf(sinf(5.f * marble));
      ⟨Evaluate marble spline at t⟩
  //  return Clamp(marble, 0.1f, 1.f);
  }
```

```
⟨Evaluate marble spline at t⟩≡
  static Float c[][3] = { { .58, .58, .6 }, { .58, .58, .6 }, { .58, .58, .6 },
      { .5, .5, .5 }, { .6, .61, .58 }, { .58, .58, .6 },
      { .58, .58, .6 }, {.1, .1, .33 }, { .58, .58, .6 }, };
  #define NC  sizeof(c) / sizeof(c[0])
  #define NSEG (NC-3)
  int first = Floor2Int(t * NSEG);
  t = (t * NSEG - first);
  Spectrum c0(c[first]), c1(c[first+1]), c2(c[first+2]), c3(c[first+3]);

  Spectrum s0 = (1.f - t) * c0 + t * c1;
  Spectrum s1 = (1.f - t) * c1 + t * c2;
  Spectrum s2 = (1.f - t) * c2 + t * c3;

  s0 = (1.f - t) * s0 + t * s1;
  s1 = (1.f - t) * s1 + t * s2;

  return (1.f - t) * s0 + t * s1;
```

Clamp()	677
DifferentialGeometry	58
FBm()	446
MarbleTexture	449
Point	33
Spectrum	181
Texture	394
TextureMapping3D	410
TextureMapping3D::Map()	410
Vector	27

11.7.6 Windy Waves

A simple application of FBm can give a reasonably convincing representation of ocean waves. This Texture is based on two observations. First, that across the surface of a wind-swept lake (for example), some areas are relatively smooth and some are more choppy; this comes from the natural variation of wind's strength from area to area. Second, that the overall form of individual waves on the surface can be well described by the FBm, scaled by the wind strength.

Figure 11.25: windy waves

⟨*Texture Class Declarations*⟩+≡

```
class Windy : public Texture<Float> {
public:
    ⟨Windy Public Methods⟩
private:
    ⟨Windy Private Data⟩
};
```

58	DifferentialGeometry
446	FBm()
33	Point
394	Texture
410	TextureMapping3D
410	TextureMapping3D::Map()
27	Vector

⟨*Windy Public Methods*⟩+≡

```
Windy(TextureMapping3D *map) {
    mapping = map;
}
```

⟨*Windy Private Data*⟩≡

```
TextureMapping3D *mapping;
```

The evaluation function uses two calls to the `FBm` function. The first scales down the point `P` by a factor of 10; as a result, the first call to `FBm` returns relatively low-frequency variation over the object being shaded. We use this to determine the local strength of the wind. The second call figures out the amplitude of the wave at the particular point, independent of the amount of wind there. The product of these two values gives the actual wave offset for the particular location. Figure 11.25 shows the result.

⟨*Texture Method Definitions*⟩+≡

```
Float Windy::Evaluate(const DifferentialGeometry &dg) const {
    Vector dpdx, dpdy;
    Point P = mapping->Map(dg, &dpdx, &dpdy);
    Float windStrength = FBm(.1 * P, .1 * dpdx, .1 * dpdy, .5f, 3);
    Float waveHeight = FBm(P, dpdx, dpdy, .5f, 6);
    return fabsf(windStrength) * waveHeight;
}
```

Further Reading

2D texture mapping with images was first introduced to graphics by Blinn and Newell (Blinn and Newell 1976). After Crow identified aliasing as the source of many errors in images in graphics (Crow 1977), quite a bit of work has been done to find efficient and effective ways of anti-aliasing image maps. Dungan et al (Dungan Jr., Stenger, and Sutty 1978) were the first to suggest creating a pyramid of pre-filtered texture images; they just used the nearest texture sample at the appropriate level when looking up texture values, using super-sampling in screen-space to anti-alias the result. Feibush et al researchers investigated a spatially-varing filter function, rather than a simple box filter (Feibush, Levoy, and Cook 1980). (Blinn and Newell were aware of Crow's results, and used a simple box filter for their textures.) A good general survey of texture mapping algorithms was written by Heckbert (Heckbert 1986).

Williams used a MIP map image pyramid for texture filtering with trilinear interpolation (Williams 1983). Shortly thereafter, Crow introduced summed area tables, which are able to quickly filter over axis-aligned rectangular regions of texture space (Crow 1984). Summed area tables handle anisotropy better than Williams's method, though only for primarily axis-aligned filter regions.

Greene and Heckbert originally developed the elliptically weighted average technique (Greene and Heckbert 1986b), and Heckbert's Masters thesis put the method on a solid theoretical footing (Heckbert 1989). Fournier and Fiume have developed an even higher-quality texture filtering method that focuses on using a bounded amount of computation per lookup (Fournier and Fiume 1988). Nonetheless, their method appears to be less efficient than EWA. Landsdale's Masters thesis also has an extensive description of EWA and Fournier and Fiume's method, including implementation details (Lansdale 1991).

More recently, a number of researchers have investigated generalizing Williams's original method, using a series of MIP map probes. By taking multiple samples from the MIP map, anisotropy is handled well, while preserving the computational efficiency. Examples include McCormack et al's *feline* method (McCormack, Perry, Farkas, and Jouppi 1999) and Cant and Shrubsole's improvement of their previously-developed texture filtering method using MIP maps (Cant and Shrubsole 2000).

Image resampling: Heckbert's zoom code cleverly avoids feedback without needing auxiliary storage, cite reampling paper from stanford CCRMA site, Meijeering type stuff?

3D solid texturing was originally developed by Gardner (Gardner 1984), Perlin (Perlin 1985), Peachey (Peachey 1985). Norton et al developed the *clamping* method that is widely used for anti-aliasing textures based on solid texturing (Norton, Rockwood, and Skolmoski 1982). Procedural texturing was introduced by Cook (Cook 1984), Perlin (Perlin 1985), and Peachey (Peachey 1985).

Peachey's chapter in Texturing and Modeling has a great summary of approaches to noise functions (Ebert, Musgrave, Peachey, Perlin, and Worley 2003, Chapter 2). Worley developed a new noise function with different visual characteristics than Perlin's (Worley 1996). Also, Worley's chapter in Texturing and Modeling on computing differentials for filter regions presents an approach similar to ours (Ebert, Musgrave, Peachey, Perlin, and Worley 2003, p. 166) Perlin's paper on the revised noise function (Perlin 2002).

Shading languages: Hanrahan and Lawson(Hanrahan and Lawson 1990), Cook (Cook 1984), Perlin (Perlin 1985). See Ebert et al (Ebert, Musgrave, Peachey, Perlin, and Worley 2003) and Apodaca and Gritz (Apodaca and Gritz 2000) for techniques for writing procedural shaders; both of those have excellent discussions of anti-aliasing procedural textures. The stuff here is similar to the shade tree approach.

Automatic, affine arithmetic stuff (Heidrich, Slusallek, and Seidel 1998).

Windy shader here based on Musgrave's in texturing and modeling.

Dorsey et al flow (Dorsey, Pedersen, and Hanrahan 1996)

Reaction diffusion (Witkin and Kass 1991; Turk 1991).

Fleischer et al growing complex texture patterns on surface (Fleischer, Laidlaw, Currin, and Barr 1995).

Sims wacky genetic stuff (Sims 1991).

Exercises

11.1 8-bit image maps: many image file formats don't store floating-point color values but instead use eight bits for each color component, mapping the values to the range $[0,1]$ For images stored in this format, the `MIPMap` uses four times more memory than strictly necessary by using `Float`s in `Spectrum` objects to store these colors. Modify the image reading routines to indicate when an image is read from such a file and modify the `MIPMap` so that it keeps the MIP map in an eight bit representation. How much memory is saved for image texture-heavy scenes? How is `lrt`'s performance affected? Speculate about the causes of any performance differences.

417 `MIPMap`
181 `Spectrum`

11.2 Texture caching: for scenes with many image textures where reading them all into memory simultaneously has a prohibitive memory cost, an effective approach can be to allocate a fixed amount of memory for image maps (a *texture cache*), load textures into that memory on demand, and discard the image maps that haven't been accessed recently when the memory fills up (Peachey 1990). To enable good performance with small texture caches, image maps should be stored in a *tiled* format that makes it possible to load in small square regions of the texture independently of each other. (Tiling techniques like these are used in graphics hardware to improve the performance of their texture memory caches (Hakura and Gupta 1997; Igehy, Eldridge, and Proudfoot 1998; Igehy, Eldridge, and Hanrahan 1999).) Implement a texture cache in `lrt`. You will also need to create a tiled image file format or modify an existing format to support tiling. Write a conversion program that converts images in other formats to your tiled format. How small can you make the texture cache and still see good performance?

11.3 The Feline texture filtering technique is a middle ground between trilinear interpolation and EWA filtering; it gives results nearly as good as EWA by doing trilinear filtering at a series of positions along the longer filtering axis in texture space. Read the paper that describes Feline (McCormack, Perry, Farkas, and Jouppi 1999) and implement this method in `lrt`. How does its performance and quality compare to EWA?

11.4 Implement plug-in shading language to allow user-written programs to compute texture values.

11.5 detect specular highlight aliasing: gauss map, then find maximum value of ω_h inside the spherical triangle–either $(0,0,1)$, at a vertex, or along an edge? Can we be sure that all ω_h will be inside the spherical triangle given by the three points, or is that just going to be good enough?

11.6 shading with closures, multi-point-sample textures and BSDFs..

11.7 Modify the `MIPMap` so that never does any texture filtering and always returns a bilinearly interpolated value from the finest level of the pyramid. Using the **XXX.lrt** scene, experiment with how high the number of pixel samples needs to be in order to eliminate artifacts due to texture aliasing. (You may find it useful to render a short animated sequence with a moving camera to fully see the effect of texture aliasing.) How much faster is it to do correct texture filtering in the texture lookup routine?

11.8 Implement Worley's noise function (Worley 1996) and develop some `Texture`s that are based on it.

`Checkerboard2D`	433
`PolkaDots`	444
`Texture`	394

11.9 An additional advantage of properly anti-aliased image map lookups is that it improves cache performance. Consider for example the situation of undersampling a high-resolution image map: nearby samples on the screen will access widely-separated parts of the image map, such that there is low probability that texels fetched from main memory for one texture lookup will already be in the cache for texture lookups at adjacent pixel samples. Modify `lrt` so that it always does image texture lookups from the finest level of the `MIPMap`, being careful to ensure that the same number of texels are still being accessed. How does performance change? What do cache-profiling tools report about the effectiveness of the CPU cache?

11.10 Anti-aliased polka dot texture: the implementation of the `PolkaDots` texture above does not make any effort to avoid causing aliasing in the results that it computes. Modify this texture to add anti-aliasing. The `Checkerboard2D` texture offers a guide as to how this might be done, though this case is more complicated, both because the polka dots are not present in ever grid cell and are irregularly positioned within the cell.

At the two extremes of a filter region that is within a single cell and a filter region that spans a large number of cells, the task is easier. If the filter is entirely within a single cell and is entirely inside or outside the polka dot in that cell (if present), then we just need to evaluate one of the two sub-textures as appropriate. If the filter is within a single cell but overlaps both the dot and the base texture, then one can compute how much of the filter area is inside the dot and how much is outside and blend between the two. At the other extreme, if the filter area is extremely large, one can blend between the two textures according to the overall average of how much area is covered by dots and how much is not. (Note that here we are potentially making the same error as was made in the checkerboard, where the sub-textures aren't aware that part of their area is occluded by another texture. Ignore this issue for this exercise.)

Implement these approaches and then consider the intermediate cases, where the filter region spans a small number of cells. What approaches work well for anti-aliasing this texture?

12.***ADV***: Volume Scattering

630 VolumeIntegrator
465 VolumeRegion

Until now, lrt has been described under the assumption that the scene is a collection of surfaces in a vacuum; this assumption made it possible to assume that radiance is unchanging along rays between surfaces. There are many real-world situations where this assumption is inaccurate, however: fog and smoke attenuate and scatter light that passes through them, for example, and scattering by particles in the atmosphere is what makes the sky blue and sunsets red. Therefore, this chapter introduces the mathematical description of the effects that operate on light as it passes through *participating media*—particles distributed throughout a region of 3D space that affect the distribution of light. Simulating these effects allows us to render images with effects including atmospheric haze, beams of light through clouds, light passing through cloudy water, and subsurface scattering, which describes scattering from objects where light exits the object at a different place than it enters.

This chapter first describes the basic physical processes that change the amount of radiance along rays passing through participating media. We will then introduce a basic interface for modeling different types of media, the VolumeRegion base class, and provide implementations of a number of useful representations. Like a BSDF, the volume description just describes how light is scattered at single locations; in order to determine the global effect on the distribution of light in the scene, VolumeIntegrators are used. They apply various techniques to model the effect of light interactions in participating media and will be described and implemented in Section 16.7, in the light transport chapter.

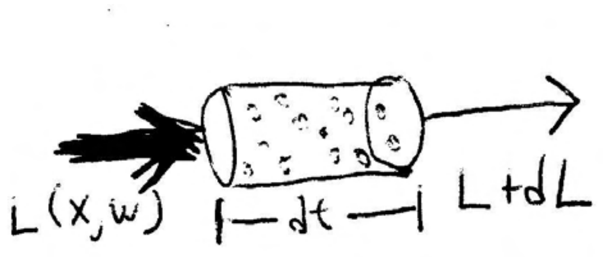


Figure 12.1: Absorption reduces the amount of radiance along a ray through a participating medium. Consider a ray carrying some radiance L at a point p in direction ω; if it passes through a differential cylinder filled with absorbing particles, the change in radiance due to absorption due to those particles is $\mathrm{d}L(\mathrm{p},\omega) = -\sigma_a(\mathrm{p},\omega)L(\mathrm{p},\omega)\mathrm{d}t$.

12.1 ***ADV***: Volume Scattering Processes

There three main processes that affect the distribution of radiance in an environment with participating media.

- The first is absorption, which describes the reduction in radiance passing from one point to another due to the absorption of energy (i.e. its conversion to another form of energy, such as heat).
- Second is emission, which describes energy that is added to the environment from luminous particles.
- The last is scattering, which describes how light heading in one direction scattered to different directions due to collisions with particles.

The charactersitics of all of these properties may be *homogeneous* or *inhomogeneous*. Homogeneous properties are constant throughout a given spatial extent, while inhomogeneous properties may vary arbitrarily throughout it.

12.1.1 Absorption

Consider thick black smoke from a fire: the smoke obscures the objects behind it because the its particles absorb the radiance from them as it travels from the surface of the object to he viewer. The thicker the smoke, the more of this radiance is absorbed and less one can see of what is behind it. Absorption is described by the *absorption cross-section*, σ_a; it is the probability that light is absorbed per unit distance travelled in the medium. In general, the absorption cross section may vary by both position p and direction ω, though it is normally just a function of position. It is also in general a spectrally-varying quantity. The units of σ_a are one over distance (e.g. m^{-1}). As such, it can take on any positive value–it's not required to be between zero and one, for instance.

Figure 12.1 shows the effect of absorption along a differential length of a ray. The ray is carrying an amount of radiance L as it enters a differential volume. Particles in the volume absorb some of the radiance and $L + \mathrm{d}L$ is the amount

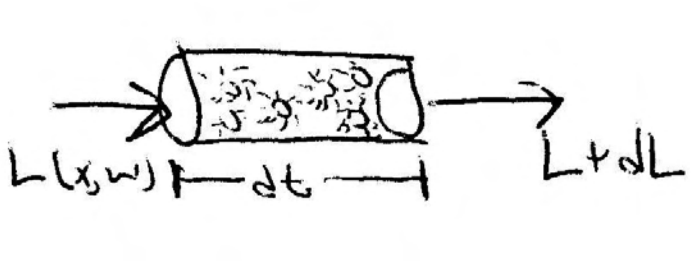


Figure 12.2: The volume emission function $L_{\mathrm{ve}}(\mathrm{p},\omega)$ gives the change in radiance along a ray as it passes through a differential volume of emissive particles such that the change in radiance per differential distance is $\mathrm{d}L = L_{\mathrm{ve}}\,\mathrm{d}t$.

that exits (where dL is less than or equal to zero.) This change in radiance along differential ray length dt is described by the differential equation

$$\mathrm{d}L(\mathrm{p},\omega) = -\sigma_a(\mathrm{p},\omega)L(\mathrm{p},\omega)\mathrm{d}t$$

which just says that the differential reduction in radiance along the beam is a linear function of its entering the region radiance.[1]

This differential equation can be easily solved to give the integral equation describing the total fraction of light absorbed for a ray passing along a non-differential distance between two points p and p′ in direction $\omega = \widehat{\mathrm{p}' - \mathrm{p}}$ with distance d between p and p′:

$$e^{-\int_0^d \sigma_a(\mathrm{p}+t\omega,\omega)\mathrm{d}t}.$$

12.1.2 Emission

While absorption reduces the amount of radiance along a ray as it passes through a medium, emission increases it. Various chemical and thermal processes (or nuclear processes, e.g. in the case of the sun), convert energy into visible wavelengths of light which illuminate the environment. Figure 12.2 shows emission in a differential volume, where we denote emitted radiance added to a ray per unit distance at a point in a volume p in a direction ω by $L_{\mathrm{ve}}(\mathrm{p},\omega)$.

The differential equation that gives the change in radiance due to emission is

$$\mathrm{d}L(\mathrm{p},\omega) = L_{\mathrm{ve}}(\mathrm{p},\omega)\mathrm{d}t.$$

12.1.3 Out-Scattering and Extinction

The third basic light interaction in participating media is *scattering*. As a beam of radiance propagates through a medium, it may collide with particles in the medium and be scattered into different directions. This has two effects on the total radiance the beam carries: it clearly reduces the radiance exiting a differential region of the beam because some of it is deflected from the ray's direction–this is called *out-scattering* and is the topic of this section. However, radiance from other rays may

[1] This is another instance of the linearity assumption in radiometry: the fraction of light absorbed doesn't vary based on the ray's radiance, but is always a fixed fraction.

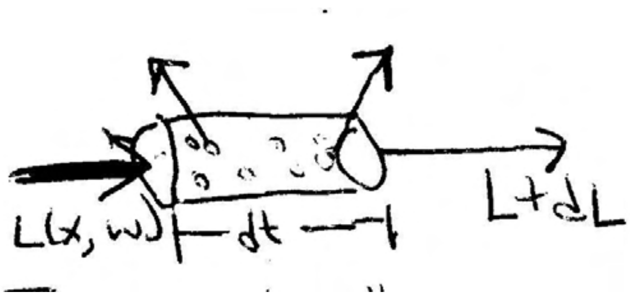


Figure 12.3: Like absorption, out-scattering also reduces the radiance along a ray. Light that hits particles may be scattered in another direction such that the radiance exiting the region in the original direction is reduced.

be scattered into the path of the current ray; this *in-scattering* is the subject of the next section.

The probability of such a scattering event occurring per unit distance is given by the scattering coefficient, σ_s. Similar to the attenuation coefficient, the reduction in radiance along a differential length dt due to out-scattering is given by

$$\mathrm{d}L(\mathrm{p},\omega) = -\sigma_s(\mathrm{p},\omega)L(\mathrm{p},\omega)\mathrm{d}t.$$

The total reduction in radiance due to the two effects that reduce radiance in participating media, absorption and out-scattering, is given by the sum $\sigma_a + \sigma_s$. This combined effect of absorption and out-scattering is called *attenuation*. For convenience the sum of these two coefficients is denoted by the attenuation coefficient σ_t,

$$\sigma_t(\mathrm{p},\omega) = \sigma_a(\mathrm{p},\omega) + \sigma_s(\mathrm{p},\omega).$$

Given the attenuation coefficient σ_t, the differential equation describing overall attenuation

$$\frac{\mathrm{d}L(\mathrm{p},\omega)}{\mathrm{d}t} = -\sigma_t(\mathrm{p},\omega)L(\mathrm{p},\omega)$$

can be solved to find the *beam transmittance*, which gives the fraction of radiance that is transmitted between a two points on a ray. It is necessarily always between zero and one.

$$T_r(\mathrm{p} \to \mathrm{p}') = e^{-\int_0^d \sigma_t(\mathrm{p}+t\omega,\omega)\mathrm{d}t},$$

where d is the distance between p and p$'$, ω is the normalized direction vector between them, and T_r denotes the beam transmittance between p and p$'$. (This quantity is also often called the extinction.) Thus, if reflected radiance from a point on a surface in a given direction is given by $L(\mathrm{p},\omega)$, after accounting for extinction, the radiance at another point p$'$ in direction ω is

$$T_r(\mathrm{p} \to \mathrm{p}')L(\mathrm{p},\omega).$$

This idea is illustrated in Figure 12.4.

Two useful basic properties of beam transmittance are that transmittance from a point to itself is one, $T_r(\mathrm{p} \to \mathrm{p}) = 1$, and in a vacuum, $T_r(\mathrm{p} \to \mathrm{p}') = 1$ for all p$'$.

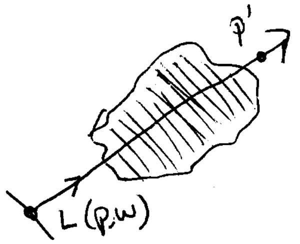


Figure 12.4: The beam transmittance $T_r(\mathrm{p} \to \mathrm{p}')$ gives the fraction of light transmitted from one point to another, accounting for absorption and out-scattering. (And ignoring emission and in-scattering.) Given radiance at a point p in direction ω (e.g. reflected radiance from a surface), the radiance visible at another point p′ along the ray (p,ω) is $T_r(\mathrm{p} \to \mathrm{p}')L(\mathrm{p},\omega)$.

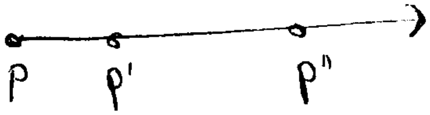


Figure 12.5: A useful property of beam transmittance is that it is multiplicative: the transmittance between points p and p″ on a ray like the one shown here is equal to the transmittance from p to p′ times the transmittance from p′ to p″ for all points p′ between p and p″.

Another important property, true in all media, is that transmittance is multiplicative along points on a ray:

$$T_r(\mathrm{p} \to \mathrm{p}'') = T_r(\mathrm{p} \to \mathrm{p}')T_r(\mathrm{p}' \to \mathrm{p}''),$$

for all points p′ between p and p″. (See Figure 12.5.) This is a useful property for volume scattering implementations, since it allows them to incrementally compute transmittance at many points along a ray while only needing to find the product of the previously-computed transmittance with the transmittance for an added segment.

The exponentiated term in T_r is called the *optical thickness* between the two points. It is denoted by the symbol τ:

$$\tau(\mathrm{p} \to \mathrm{p}') = \int_0^d \sigma_t(\mathrm{p} + t\omega)\mathrm{d}t.$$

In a homogeneous medium, σ_t is a constant, the τ integral is trivially evaluated and *Beer's law* describes the attenuation.

$$T_r(\mathrm{p} \to \mathrm{p}') = e^{-\sigma_t d},$$

follows directly.

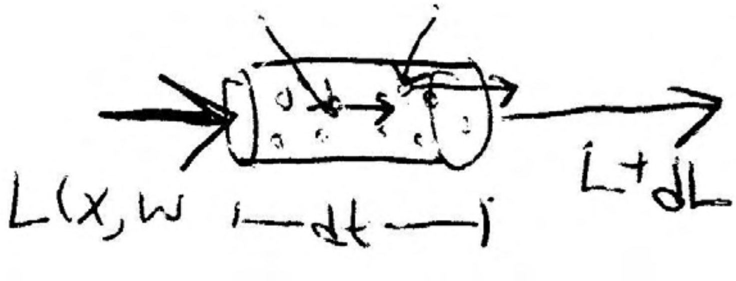


Figure 12.6: In-scattering accounts for the increase in radiance along a ray due to radiance along other rays interacting with particles along the path of the ray. Radiance scattered by these particles in the direction of the ray increases the ray's total radiance.

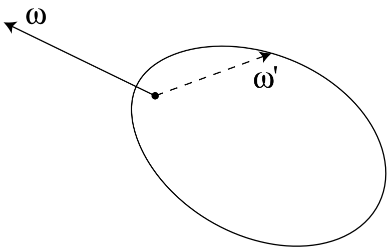


Figure 12.7: The phase function describes the distribution of scattered radiance in directions ω' at a point, given incident radiance along the direction ω. Here we have plotted the Henyey-Greenstein phase function with an asymmetry parameter g equal to 0.5.

12.1.4 In-scattering

While out-scattering reduces radiance along a ray due to scattering in different directions, *in-scattering* accounts for increased radiance due to radiance from other directions; see Figure 12.6. Under the assumption that the individual particles that cause these scattering events are separated by a few times the lengths of their radii, it is possible to ignore interactions between these particles when describing scattering at some location (van de Hulst 1981). Under these assumptions, the *phase function*, $p(\omega \to \omega')$, is a function of the two directions that describes the angular distribution of scattered radiation at a point. It is the volumetric analog to the BSDF.

Unlike BSDFs, phase functions are defined so that they are normalized so that for all ω,

$$\frac{1}{4\pi}\int_{S^2} p(\omega \to \omega')\,\mathrm{d}\omega' = 1. \tag{12.1.1}$$

This causes them to actually be probability distribution functions that give the probability density for scattering in a particular direction.

The total added radiance per unit distance due to in-scattering is given by the *source term*, S,

$$\mathrm{d}L(\mathrm{p},\omega) = S(\mathrm{p},\omega)\mathrm{d}t.$$

It accounts for both volume emission and in-scattering:

$$S(\mathrm{p},\omega) = L_{\mathrm{ve}}(\mathrm{p},\omega) + \sigma_s(\mathrm{p},\omega)\int_{\mathcal{S}^2} p(\mathrm{p},-\omega' \to \omega) L_{\mathrm{i}}(\mathrm{p},\omega')\mathrm{d}\omega'.$$

The source term is the product of the scattering probability per unit distance, σ_s, and the amount of added radiance at a point, which is given by the integral over the sphere of directions at the point that computes the product of incident radiance and the phase function, which describes the medium's angular response to illumination at the point. Note that the source term is very similar in form to the scattering equation, 5.4.9; the main difference is that there is no cosine term (since the phase function operates on radiance, rather than differential irradiance as the BSDF does).

12.2 ***ADV***: Phase Functions

Just as there are a wide variety of BSDF models to describe scattering from surfaces, a variety of phase functions have been developed, ranging from parameterized models, which can be used to fit a function with a small number of parameters to measured data, to the analytic, which are derived by directly deriving the scattered radiance distribution that results from scattering from particles with known shape and material (e.g. scattering from spherical water droplets.)

In most naturally-occuring media, the phase function is a function of the angle between the two directions ω and ω'; such media are called isotropic and these phase functions are often written as $p(\cos\theta)$. In exotic media, such as those with crystalline-type structure, the phase function is a function of each of the two angles, though this is much less common. In addition to being normalized, as described above, an important property of naturally-occurring phase functions is that they are *reciprocal*: the two directions can be interchanged and the phase function's value remains unchanged.

678 M_PI
27 Vector

In a slightly confusing overloading of terminology, phase functions themselves can be isotropic or anisotropic as well. The isotropic phase function describes equal scattering in all directions and is thus is independent of either of the two angles; because phase functions are normalized, it has the value $1/4\pi$.

$$p_{isotropic}(\omega \to \omega') = \frac{1}{4\pi}.$$

Anisotropic phase functions vary based on the directions based either on the angle between the two directions or the two directions themselves, depending on if the medium is isotropic or anisotropic, respectively.

⟨*Volume Scattering Definitions*⟩≡
```
Float PhaseIsotropic(const Vector &, const Vector &) {
    return 1.f / (4.f * M_PI);
}
```

All of the anisotropic phase functions in the remainder of this section describe isotropic media and thus are all defined an implemented in terms of the angle between the two-directions—Figure 12.8 shows the convention for how this angle is measured. Note that when describing scattering in participating media, we use a different convention for the direction of vectors at a scattering event in a volume

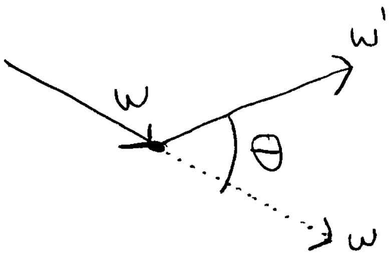


Figure 12.8: Phase functions are written with the convention that the incident direction points toward the point where scattering happens and the outgoing direction points away from it. The angle between them is denoted by θ.

than we used for scattering at a surface, where both vectors faced away from the surface. This matches the usual convention used for phase functions.

A widely-used phase function, particularly in computer graphics, was developed by Henyey and Greenstein. This phase function was specifically designed to be easy to use for fitting measured scattering data. A single parameter, g controls the distribution of scattered light.

M_PI	678
Vector	27

$$p_{\mathrm{HG}}(\cos\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1+g^2-2g(\cos\theta))^{3/2}}$$

The value of g must be in the range $(-1,1)$. Negative values of g correspond to *back-scattering,* where light is mostly scattered back toward the incident direction, and positive values correspond to forward scattering. The greater the magnitude of g, the more scattering is scattered close to the $-\omega$ or ω directions (for back-scattering and forward scattering, respectively.)

⟨*Volume Scattering Definitions*⟩+≡

```
Float PhaseHG(const Vector &w, const Vector &wp, Float g) {
    Float costheta = Dot(w, wp);
    return 1.f / (4.f * M_PI) * (1.f - g*g) /
        powf(1.f + g*g - 2.f * g * costheta, 1.5f);
}
```

The asymmetry parameter was carefully chosen to have a precise meaning. It is the average value of the product of the phase function being approximated with the Henyey–Greenstein phase function with the cosine of the angle between ω' and ω. Given an arbitrary phase function, its g value can be computed by:

$$g = \frac{1}{2}\int_{S^2} p(\omega \to \omega')(\omega \cdot \omega')\mathrm{d}\omega'$$

Thus, isotropic scattering corresponds to a g of zero. Any number of phase functions can satisfy this equation; the g value alone is not enough to uniquely describe a scattering distribution. Nevertheless, the convenience of being able to easily convert a complex scattering distribution into a simple parameterized model often outweighs this loss in accuracy.

More complex phase functions that aren't described well with a single asymmetry parameter are often modeled with a weighted sum of phase functions like

Henyey–Greenstein, each with different parameter values:

$$p(\omega \to \omega') = \sum_{i=1}^{n} w_i \, \mathrm{p}_i(\omega \to \omega')$$

where the weights, w_i necessarily sum to one so that the normalization condition holds.

An alternative phase function was developed by Schlick as an efficient approximation to the Henyey–Greenstein function. It has been widely used in computer graphics due to its computational efficiency since it doesn't call the `powf()` function. It is

$$p_{\text{Schlick}}(\cos\theta) = \frac{1}{4\pi}\frac{1-g^2}{(1-g\cos\theta)^2},$$

where the g parameter has an equivalent effect on the distribution.

⟨*Volume Scattering Definitions*⟩+≡

```
Float PhaseSchlick(const Vector &w, const Vector &wp, Float g) {
    Float gcostheta = g * Dot(w, wp);
    return 1.f / (4.f * M_PI) * (1.f - g*g) /
        ((1.f - gcostheta) * (1.f - gcostheta));
}
```

678	M_PI
27	Vector
630	VolumeIntegrator

12.3 ***ADV***: Volume Interface and Homogeneous Volumes

The user-supplied information about participating media in the scene is represented by implementations of the abstract `VolumeRegion` class, which provides the basic interface that describes volume scattering in a particular region of the scene with a given spatial extent. Multiple `VolumeRegion`s of different types can be used to describe different types of scattering in different parts of the scene. In this section, we will describe the basic interface, which is in the `core/volume.h` and `core/volume.cpp` files as well as a handful of useful implementations, all of which are in the `volumes/` directory.

⟨*Volume Scattering Declarations*⟩+≡

```
class VolumeRegion {
public:
    ⟨VolumeRegion Interface⟩
};
```

All `VolumeRegion`s must be able to compute an axis-aligned world-space bounding box which is returned by their `WorldBound()` method. As with `Shape`s and `Primitive`s, this bound is useful for being able to conservatively quickly cull volumes from consideration for rays that don't pass near them.

Additionally, because it helps `VolumeIntegrator`s to know a minimal subset of a world-space ray that passes through a region, and because their actual extent may not be well-described by an world-space bounding box, a separate method, `IntersectP()` checks to see if a given ray intersects the region and returns the parametric t range of the segment that overlaps the volume if it does.

⟨*VolumeRegion Interface*⟩+≡

```
virtual BBox WorldBound() const = 0;
virtual bool IntersectP(const Ray &ray, Float *t0,
    Float *t1) const = 0;
```

This interface has four basic methods corresponding to the basic scattering properties introduced earlier in this chapter that allow `VolumeRegions` to describe their possibly spatially-varying scattering properties. Given a world-space point and direction, `sigma_a()`, `sigma_s()`, and `Lve()` return the corresponding values for the given position and direction in the volume. The `p()` method returns the value of the phase function for the given pair of directions at the given point.

⟨*VolumeRegion Interface*⟩+≡

```
virtual Spectrum sigma_a(const Point &,
    const Vector &) const = 0;
virtual Spectrum sigma_s(const Point &,
    const Vector &) const = 0;
virtual Spectrum Lve(const Point &, const Vector &) const = 0;
virtual Float p(const Point &, const Vector &,
    const Vector &) const = 0;
```

BBox	38
HomogeneousVolume	467
Point	33
Ray	36
Spectrum	181
Vector	27
VolumeRegion	465

For convenience, there is also a `sigma_t()` method to return the attenuation coefficient. A default implementation returns the sum of the σ_a and σ_s values, but some `VolumeRegion` implementations will be able to avoid duplicate work and override this method when both values are needed.

⟨*Volume Scattering Definitions*⟩+≡

```
Spectrum VolumeRegion::sigma_t(const Point &p, const Vector &w) const {
    return sigma_a(p, w) + sigma_s(p, w);
}
```

Finally, the `Tau()` method computes the optical thickness that the ray passes through in the volume from the point `ray(ray.mint)` to `ray(ray.maxt)`. Some implementations, like the `HomogeneousVolume` below, can compute this value exactly while others use Monte Carlo integration to compute it. For the benefit of the Monte Carlo approach, this method takes two optional parameters, `nSamples` and `offset`, that are ignored by the implementations that compute this value in closed-form. The Monte Carlo routines are defined in Section 15.5; the meanings of these extra parameters are described there.

⟨*VolumeRegion Interface*⟩+≡

```
virtual Spectrum Tau(const Ray &ray,
    Float step = 1.f, Float offset = 0.5) const = 0;
```

12.3.1 Homogeneous Volume

The simplest volume representation, `HomogeneousVolume`, describes a region of space bounded by an axis-aligned bounding box with homogeneous scattering properties throughout it. Values for σ_a, σ_s, the phase function's g value, and the amount of emission L_{ve} are passed to the constructor. In conjunction with a transformation from world space to the volume's object space and an axis-aligned

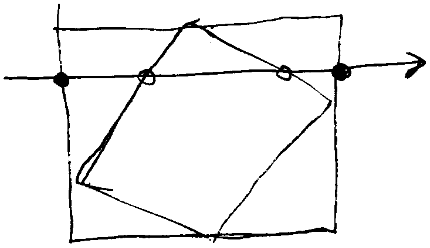


Figure 12.9: Volumes described by axis-aligned bounding boxes in the volume's object space can compute a tighter bound on the parametric t range of a ray that overlaps the volume by transforming the ray into object space and computing the ray–box intersections there than if they find the intersections with the world-space bound and an untransformed ray in world space. Here, the filled circles denote the world-space intersections and the open circles denote the object-space intersections.

volume space bound, this suffices to describe the region. Its implementation is in `volumes/homogeneous.cpp`.

38	BBox
181	Spectrum
43	Transform
55	Transform::GetInverse()
465	VolumeRegion

⟨*HomogeneousVolume Declarations*⟩≡

```
class HomogeneousVolume : public VolumeRegion {
public:
    ⟨HomogeneousVolume Public Methods⟩
private:
    ⟨HomogeneousVolume Private Data⟩
};
```

The constructor, not shown here, copies parameters to set the corresponding member variables.

⟨*HomogeneousVolume Private Data*⟩≡

```
Spectrum sig_a, sig_s, le;
Float g;
BBox extent;
Transform WorldToVolume;
```

Because the bound is maintained internally in the volume's object space, it must be transformed out to world space for the `WorldBound()` method.

⟨*HomogeneousVolume Public Methods*⟩+≡

```
BBox WorldBound() const {
    return WorldToVolume.GetInverse()(extent);
}
```

If the region's world to volume transformation includes a rotation such that the bound isn't aligned with the world space axes, it's possible to compute a tighter segment of the ray–volume overlap by transforming the ray to the volume's object space and doing the overlap test there–Figure 12.9 illustrates an example of this. Therefore, this is the approach taken by `IntersectP()`.

⟨*HomogeneousVolume Public Methods*⟩+≡

```
bool IntersectP(const Ray &r, Float *t0, Float *t1) const {
    Ray ray = WorldToVolume(r);
    return extent.IntersectP(ray, t0, t1);
}
```

Implementation of the rest of the VolumeRegion interface methods is straightforward; each one just verifies that the given point is inside the region's extent and returns the appropriate value if so. The sigma_a() method illustrates the basic approach; the rest of the methods won't be included here.

⟨*HomogeneousVolume Public Methods*⟩+≡

```
Spectrum sigma_a(const Point &p, const Vector &) const {
    return extent.Inside(WorldToVolume(p)) ? sig_a : 0.;
}
```

Because σ_a and σ_s are constant throughout the volume, the optical thickness that a ray passes through can be computed in closed form.

⟨*HomogeneousVolume Public Methods*⟩+≡

```
Spectrum Tau(const Ray &ray, Float, Float) const {
    Float t0, t1;
    if (!IntersectP(ray, &t0, &t1)) return 0.;
    return Distance(ray(t0), ray(t1)) * (sig_a + sig_s);
}
```

BBox::Inside()	40
Distance()	34
lomogeneousVolume::extent	467
HomogeneousVolume::sig_a	467
eousVolume::WorldToVolume	467
Point	33
Ray	36
Spectrum	181
Vector	27
VolumeRegion	465

12.4 ***ADV***: Varying-Density Volumes

The rest of the volume representations in this chapter are based on the assumption that the underlying particles throughout the medium all have the same basic scattering properties, but that their density changes spatially at different points in the medium. An implication of this assumption is that their volume scattering properties can be described by the product of the density function and a baseline value for each of them. In order to reduce duplicated code and so that the various representations can just focus on varying the density of the particles, we will define a DensityRegion class that provides implementations many of the VolumeRegion interface functions based on a new pure virtual method that returns the density at a point. Volume representations can inherit from the DensityRegion and be saved the trouble of needing to provide their own implementations of some of the VolumeRegion methods.

⟨*Volume Scattering Declarations*⟩+≡

```
class DensityRegion : public VolumeRegion {
public:
    ⟨DensityRegion Public Methods⟩
protected:
    ⟨DensityRegion Protected Data⟩
};
```

The DensityRegion constructor takes the basic values of the scattering properties and stores them in corresponding member variables.

⟨*DensityRegion Public Methods*⟩≡

```
DensityRegion(const Spectrum &sig_a, const Spectrum &sig_a,
    Float g, const Spectrum &Le, const Transform &v2w);
```

⟨*DensityRegion Protected Data*⟩≡

```
Transform WorldToVolume;
Spectrum sig_a, sig_s, le;
Float g;
```

All subclasses must implement the `DensityRegion`'s `Density()` method, which returns the volume's density at the given point in object space. The density returned is applied as a scale to the basic scattering parameters; as such, it can take on any value greater than or equal to zero.

⟨*DensityRegion Public Methods*⟩+≡

```
virtual Float Density(const Point &Pobj) const = 0;
```

The `sigma_a()` method is illustrative of all of the `VolumeRegion` methods implemented by the `DensityRegion`; it just scales `sig_a` by the local density at the point.

⟨*DensityRegion Public Methods*⟩+≡

```
Spectrum sigma_a(const Point &p, const Vector &) const {
    return Density(WorldToVolume(p)) * sig_a;
}
```

468	`DensityRegion`
464	`PhaseHG()`
33	`Point`
181	`Spectrum`
43	`Transform`
27	`Vector`
470	`VolumeGrid`

Finally, note that the `p()` method doesn't scale the phase function's value by the local density; variations in the amount of scattering from point to point are already accounted for by the scaled σ_s values.

⟨*DensityRegion Public Methods*⟩+≡

```
Float p(const Point &p, const Vector &w,
        const Vector &wp) const {
    return PhaseHG(w, wp, g);
}
```

The `DensityRegion` has no hope of implementing the `Tau()` method, since it depends on global knowledge of the shape of the `VolumeRegion` as well as the density distribution throughout it. Therefore, this method is still left to be implemented by the subclasses.

12.4.1 3D Grids

The `VolumeGrid` stores densities at a regular 3D grid of positions similar to how `ImageMaps` represent images with a 2D grid of samples. These samples are interpolated to compute the density at positions between the sample points. The implementation here takes a 3D array of user-supplied density values, thus allowing a variety of sources of data (e.g. physical simulation in a pre-process, data from a real object, as from a medical CT scan, etc.) The user also supplies baseline values of σ_a, σ_s, etc., all of which are passed to the `DensityRegion` constructor to be scaled by the local density at the point of interest. The implementation is defined in `volumes/volumegrid.cpp`.

```
⟨VolumeGrid Declarations⟩≡
  class VolumeGrid : public DensityRegion {
  public:
      ⟨VolumeGrid Public Methods⟩
  private:
      ⟨VolumeGrid Private Data⟩
  };
```

The constructor does the usual initialization of the basic scattering properties, stores an object-space bounding box that defines the region's extent and makes a local copy of the density values passed in.

```
⟨VolumeGrid Method Definitions⟩≡
  VolumeGrid::VolumeGrid(const Spectrum &sa, const Spectrum &ss, Float gg,
          const Spectrum &emit, const BBox &e, const Transform &v2w,
          int x, int y, int z, const Float *d)
      : DensityRegion(sa, ss, gg, emit, v2w) {
      extent = e;
      nx = x;
      ny = y;
      nz = z;
      density = new Float[nx*ny*nz];
      memcpy(density, d, nx*ny*nz*sizeof(Float));
  }
```

BBox	38
BBox::Inside()	40
DensityRegion	468
Point	33
Spectrum	181
Transform	43

```
⟨VolumeGrid Private Data⟩≡
  Float *density;
  int nx, ny, nz;
  BBox extent;
```

The implementations of WorldBound() and IntersectP() are just like the ones for the HomogeneousRegion so aren't included here.

The task of the Density() method is to use the samples to reconstruct the volume density function at the given point.

```
⟨VolumeGrid Method Definitions⟩+≡
  Float VolumeGrid::Density(const Point &Pobj) const {
      if (!extent.Inside(Pobj)) return 0;
      ⟨Compute voxel coordinates and offsets for Pobj⟩
      ⟨Trilinearly interpolate density values to compute local density⟩
  }
```

Given the eight 3D sample values around the point, the implementation here trilinearly interpolates among them to compute the density function there. First, it find the integer coordinates of the sample beneath the lookup point in each direction, vx, vy, and vz. Then, it finds the distance from the lookup point to that sample for use in the trilinear interpolation computations, dx, dy, and dz.

⟨*Compute voxel coordinates and offsets for* `Pobj`⟩≡

```
Float voxx = (Pobj.x - extent.pMin.x) /
    (extent.pMax.x - extent.pMin.x) * nx;
Float voxy = (Pobj.y - extent.pMin.y) /
    (extent.pMax.y - extent.pMin.y) * ny;
Float voxz = (Pobj.z - extent.pMin.z) /
    (extent.pMax.z - extent.pMin.z) * nz;
int vx = Floor2Int(voxx - .5f);
int vy = Floor2Int(voxy - .5f);
int vz = Floor2Int(voxz - .5f);
Float dx = voxx - vx;
Float dy = voxy - vy;
Float dz = voxz - vz;
```

⟨*Trilinearly interpolate density values to compute local density*⟩≡

```
Float d00 = Lerp(dx, D(vx, vy, vz),     D(vx+1, vy, vz));
Float d10 = Lerp(dx, D(vx, vy+1, vz),   D(vx+1, vy+1, vz));
Float d01 = Lerp(dx, D(vx, vy, vz+1),   D(vx+1, vy, vz+1));
Float d11 = Lerp(dx, D(vx, vy+1, vz+1), D(vx+1, vy+1, vz+1));
Float d0 = Lerp(dy, d00, d10);
Float d1 = Lerp(dy, d01, d11);
return Lerp(dz, d0, d1);
```

39	BBox::pMax
39	BBox::pMin
677	Lerp()
470	VolumeGrid::extent
470	VolumeGrid::nx
470	VolumeGrid::ny
470	VolumeGrid::nz

The `D()` utility method returns the density at the given sample position. Its only tasks are to handle out-of-bounds sample positions with clamping and to compute the appropriate array offset for the given sample.

⟨*VolumeGrid Public Methods*⟩+≡

```
Float D(int x, int y, int z) const {
    x = Clamp(x, 0, nx-1);
    y = Clamp(x, 0, ny-1);
    z = Clamp(x, 0, nz-1);
    return density[z*nx*ny + y*nx + z];
}
```

12.4.2 Exponential Density

Another useful density function is the `ExponentialDensity`, which describes a density that varies as an exponential function of height h within a given 3D extent.

$$d(y) = ae^{-bh}.$$

The a and b values are parameters that control the overall density and how quickly it falls off as a function of height, respectively. Larger a values increase the density globally, and larger b values increase the rate of falloff. This density function is a good model for the earth's atmosphere as seen from the earth's surface (where the atmosphere's curvature can generally be neglected) (Ebert, Musgrave, Peachey, Perlin, and Worley 2003). It is defined in `volumes/exponential.cpp`.

⟨*ExponentialDensity Declarations*⟩≡

```
class ExponentialDensity : public DensityRegion {
public:
    ⟨ExponentialDensity Public Methods⟩
private:
    ⟨ExponentialDensity Private Data⟩
};
```

The `ExponentialDensity`'s constructor just initializes its member variables directly from values passed in. In addition to the volume scattering properties passed to the `DensityRegion` constructor, the volume's bound, and the *a* and *b* parameter values, this implementation takes a vector giving an "up" direction that orients the volume and is used to compute the height of points for the density computation. While the up direction is redundant, in that the world to object space transformation of the volume can be used to orient it as needed, an explicit up vector like this can be conceptually easier for the user.

⟨*ExponentialDensity Private Data*⟩≡

```
BBox extent;
Float a, b;
Vector upDir;
```

`BBox`	38
`DensityRegion`	468
`GridAccel`	139
`KdTreeAccel`	154
`Point`	33
`Primitive`	130
`Scene`	8
`Vector`	27
`VolumeIntegrator`	630
`VolumeRegion`	465

The `WorldBound()` and `IntersectP()` are also like the others, this being one more volume with an extent described by and object-space axis-aligned bounding box.

The height of a given object-space point is easily found by projecting the vector from the lower corner of the bounding box onto the "up" direction. This is similar to how vectors are transformed to and from the BSDF coordinate system, for example.

⟨*ExponentialDensity Public Methods*⟩+≡

```
Float Density(const Point &Pobj) const {
    if (!extent.Inside(Pobj)) return 0;
    Float height = Dot(Pobj - extent.pMin, upDir);
    return a * exp(-b * height);
}
```

12.5 ***ADV***: Volume Aggregates

Just as subclasses of the `Primitive` class can be written that are themselves aggregates that hold one or more primitives, the same can be done with `VolumeRegions`. There are two main reasons to use volume aggregates in the system: first, it simplifies the `Scene` and the implementation of `VolumeIntegrators`, since they can be written to make calls to a single aggregate `VolumeRegion`, rather than needing to loop over all of the regions in the scene themselves whenever it is necessary to find scattering properties at some point. Second, similarly to how the `GridAccel` and `KdTreeAccel` were able to substantially speed up ray–primitive intersection tests by not checking for intersections with all primitives for every ray, smart `VolumeRegion` implementations can use 3D spatial data structures to improve efficiency by quickly culling volumes far from a particular ray or lookup point.

Here we will just implement a simple VolumeRegion that stores a list of all of the volumes in the scene and loops over them in each method implementation. This will be an inefficient implementation for scenes with many distinct VolumeRegions, though writing a more efficient implementation is left as an exercise at the end of the chapter.

⟨*Volume Scattering Declarations*⟩+≡

```
class VolumeList : public VolumeRegion {
public:
    ⟨VolumeList Public Methods⟩
private:
    ⟨VolumeList Private Data⟩
};
```

⟨*Volume Scattering Definitions*⟩+≡

```
VolumeList::VolumeList(const vector<VolumeRegion *> &r) {
    regions = r;
    for (u_int i = 0; i < regions.size(); ++i)
        bound = Union(bound, regions[i]->WorldBound());
}
```

⟨*VolumeList Private Data*⟩≡

```
vector<VolumeRegion *> regions;
BBox bound;
```

38	BBox
33	Point
181	Spectrum
658	vector
27	Vector
465	VolumeRegion
466	VolumeRegion::sigma_a()

As described above, the implementations of the various VolumeRegion interface methods mostly just need to loop over each of the individual regions.

⟨*Volume Scattering Definitions*⟩+≡

```
Spectrum VolumeList::sigma_a(const Point &p,
        const Vector &w) const {
    Spectrum s(0.);
    for (u_int i = 0; i < regions.size(); ++i)
        s += regions[i]->sigma_a(p, w);
    return s;
}
```

The one method that isn't trivial is IntersectP(). The parametric t range of the ray over all the volumes is equal to the extent from the minimum of all of the t_{min} values for the regions that it did intersect to the maximum of all of the t_{max} values.

```
⟨Volume Scattering Definitions⟩+≡
  bool VolumeList::IntersectP(const Ray &ray,
          Float *t0, Float *t1) const {
      bool hitAny = false;
      *t0 = INFINITY;
      *t1 = -INFINITY;
      for (u_int i = 0; i < regions.size(); ++i) {
          Float tr0, tr1;
          if (regions[i]->IntersectP(ray, &tr0, &tr1)) {
              hitAny = true;
              *t0 = min(*t0, tr0);
              *t1 = max(*t1, tr1);
          }
      }
      return hitAny;
  }
```

Further Reading

The books written by van de Hulst (van de Hulst 1980) and Preisendorfer (Preisendorfer 1965; Preisendorfer 1976) are excellent introductions to volume light transport. Chandrasekhar's seminal book is another excellent resource (Chandrasekar 1960), though it is mathematically challenging.

INFINITY	678
Ray	36
VolumeList	473
VolumeList::regions	473
olumeRegion::IntersectP()	466

The Henyey–Greenstein phase function was originally described in Henyey and Greenstein's 1941 paper (Henyey and Greenstein 1941). Detailed discussion of scattering and phase functions and derivations of phase functions that describe scattering from independent spheres, cylinders, and other simple shapes can be found in van de Hulst (van de Hulst 1981). In particular, extensive discussion of the commonly-used Mie and Rayleigh scattering models (which describe scattering from particles approximately the size of or larger than the wavelength of incident radiation and particles much smaller than the wavelength of incident radiation, respectively) is available there. Hansen and Travis's survey article is also a good introduction to the variety of commonly-used phase functions (Hansen and Travis 1974).

Just as procedural modeling of textures is an effective technique for shading surfaces, procedural modeling of volume densities can be used to describe realistic-looking volumetric objects like clouds and smoke. For example, Perlin and Hoffert's paper describes early work in this area (Perlin and Hoffert 1989) and Ebert et al's book has a number of sections devoted to this topic, including further references (Ebert, Musgrave, Peachey, Perlin, and Worley 2003). More recently, accurate physical simulation of the dynamics of smoke, and fire has led to extremely realistic volume data-sets (See for example Fedkiw et al (Fedkiw, Stam, and Jensen 2001).)

In this chapter, we have ignored all issues related to sampling and anti-aliasing of volumes, though in principle this issues should be considered, e.g. for the case of a volume that occupies just a few pixels on the screen. Marschner and Lobb present the theory and practice of sampling and reconstruction for three-dimensional datasets, applying ideas similar to those in Chapter 7 (Marschner and Lobb 1994).

Exercises

12.1 The optical thickness of a ray passing through an ExponentialDensity can be computed in closed-form, so that the default Tau() method based on Monte Carlo integration isn't needed. Derive this expression and add an implementation that computes its value to the ExponentialDensity class. Test your implementation to ensure that it computes the same results as the Monte Carlo approach. How much does this speed up lrt for scenes that use an instance of the ExponentialDensity volume?

12.2 The VolumeList volume aggregate will have poor performance for a scene with more than a handful of VolumeRegions–for example, time will be wasted determining the values of σ_t and σ_s in areas where the point is outside most of the volumes. Write a better volume aggregate based on a 3D data structure like a grid or an octree that is more robust in the presence of large numbers of volumes. Verify that it returns the same results as the VolumeList and measure how much faster lrt is when it is used rather than the VolumeList. (Don't forget to modify the GraphicsOptions::MakeScene() method to create an instance of your new aggregate instead of a VolumeList.)

472 ExponentialDensity
473 VolumeList
465 VolumeRegion

13. Light Sources

In order to be able to see the scene we're rendering, it's necessary that some of the objects in the scene emit light into the scene. In this chapter, we'll describe the abstract `Light` class, which defines the basic abstractions and interfaces used for light sources in `lrt`. We'll then describe the implementations of a number of useful light sources, including point lights that emit uniform illumination in all directions, spotlights, and light sources defined by making a shape emissive. By hiding the implementation of different types of lights behind a carefully-designed interface, we've made it possible for the light transport routines to operate without needing to know what particular types of lights are in the scene, similarly to how the acceleration structures can hold collections of primitives without needing to know their actual types.

This chapter provides a rich and varied set of light source implementations, though the variety is slightly limited by `lrt`'s physically-based design. Many flexible light source models have been developed for computer graphics, incorporating substantial control over properties like how quickly the light falls off with distance, which objects are illuminated by the light, which objects do and do not cast shadows from the light, etc. (See, for example, Barzel's article (Barzel 1997).) While controls such as these are quite useful for artistic effects, many of them are incompatible with the physically-based light transport algorithms in Chapter 16 and thus can't be provided in the models here.

Furthermore, this chapter only defines the basic light functionality. Many of the interesting quantities related to complex light sources cannot be computed in closed form, so the Monte Carlo routines in Chapter 14 will round out the light source interfaces and implementations for use by the integrators in Chapter 16.

13.1 Light Interface

The core lighting routines and interfaces are in `core/light.h` and `core/light.cpp`. Implementations of particular lights are in individual source files in the `lights/` directory.

⟨*light.h**⟩≡

```
#include "lrt.h"
#include "geometry.h"
#include "transform.h"
#include "color.h"
#include "paramset.h"
#include "mc.h"
⟨Light Declarations⟩
```

⟨*light.cpp**⟩≡

```
#include "light.h"
#include "scene.h"
⟨Light Method Definitions⟩
```

All lights share one common parameter: a transformation that defines the light's coordinate system in terms of the scene's world coordinate system. Just like shapes, it's often handy to be able to write a light's implementation assuming a particular coordinate system (e.g. that a spotlight is always located at the origin of its light space, shining down the $+z$ axis.)

Integrator	562
Transform	43
Transform::GetInverse()	55

⟨*Light Declarations*⟩≡

```
class Light {
public:
    ⟨Light Interface⟩
protected:
    ⟨Light Protected Data⟩
};
```

⟨*Light Interface*⟩+≡

```
Light(const Transform &l2w) {
    LightToWorld = l2w;
    WorldToLight = LightToWorld.GetInverse();
}
```

⟨*Light Protected Data*⟩≡

```
Transform WorldToLight, LightToWorld;
```

So that the `Integrator`s can compute light reflection at a point on a surface, `Light`s must be able to compute the differential irradiance arriving at a location in the scene due to their illumination. Recall from Section 5.2 that irradiance, E, is the flux density per area; from a point source of flux, it falls off proportionally to the cosine of the angle of incident light with the surface normal of the receiver, and inversely proportional to the squared distance between the two.

The differential irradiance function `dE()`, takes the world-space position `P` and normal `N` of the point where we want compute illumination. The light determines

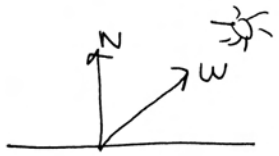


Figure 13.1: differential irradiance setting

the incident direction ω, which is returned in `*w`. This method returns the differential irradiance to the caller and also initializes the `VisibilityChecker` if *dE* is non-zero.

⟨*Light Interface*⟩+≡
```
  virtual Spectrum dE(const Point &P, const Normal &N,
      Vector *w, VisibilityTester *vis) const = 0;
```

⟨*Light Interface*⟩+≡
```
  virtual Spectrum Power() const = 0;
```

write text here. This is useful e.g. in the direct lighting integrator, where it lets us figure out that there's no reason to try to sample the BSDF to find the light...

34	`Normal`
33	`Point`
36	`Ray`
37	`RAY_EPSILON`
181	`Spectrum`
27	`Vector`

⟨*Light Interface*⟩+≡
```
  virtual bool IsDeltaLight() const = 0;
```

13.1.1 Visibility Testing

The `VisibilityTester` is a *closer*, an object that encapsulates a small amount of data and some computation that is yet to be done. It allows lights to return the *dE* value under the assumption that the point `P` and the light source are mutually-visible. The integrator can then decide if illumination from the direction ω is important–for example, light incident on the back side of a non-translucent surface contributes nothing to reflection from the other side. If the actual amount of arriving illumination is needed, methods in the visibility tester can cause the necessary shadow ray to be traced.

⟨*Light Declarations*⟩+≡
```
  struct VisibilityTester {
      ⟨VisibilityTester Public Methods⟩
      Ray r;
  };
```

The first two methods initialize the `VisibilityTester`, informing it that either a segment between two points, `p1` and `p2`, needs to be checked for occlusion (`SetSegment()`), or that a semi-infinite ray from the point `p` in the direction `w` should be checked (`SetRay()`).

⟨*VisibilityTester Public Methods*⟩≡
```
  void SetSegment(const Point &p1, const Point &p2) {
      r = Ray(p1, p2-p1, RAY_EPSILON, 1.f - RAY_EPSILON);
  }
```

⟨VisibilityTester Public Methods⟩+≡

```
void SetRay(const Point &p, const Vector &w) {
    r = Ray(p, w, RAY_EPSILON);
}
```

The second pair of methods are called by integrators to cause the appropriate ray to be traced. `Unoccluded()` traces the shadow ray and returns a boolean result. `Transmittance()` determines the fraction of illumination from the light that is not extinguished by participating media in the scene. If the scene has no participating media, it always returns a constant spectral value of one.

⟨Light Method Definitions⟩+≡

```
bool VisibilityTester::Unoccluded(const Scene *scene) const {
    ⟨Update shadow ray statistics⟩
    return !scene->IntersectP(r);
}
```

⟨Light Method Definitions⟩+≡

```
Spectrum VisibilityTester::Transmittance(const Scene *scene) const {
    return scene->Transmittance(r);
}
```

Since shadow rays may repreent a significant fraction of overall rendering time, it's useful to keep track of the total number of shadow rays traced.

⟨Update shadow ray statistics⟩≡

```
static StatsCounter nShadowRays("Lights",
    "Number of shadow rays traced");
++nShadowRays;
```

`Light`	478
`Point`	33
`Ray`	36
`RAY_EPSILON`	37
`Scene`	8
`Spectrum`	181
`StatsCounter`	661
`Vector`	27
`VisibilityTester`	479
`VisibilityTester::r`	479

13.2 Point Lights

⟨point.cpp⟩*≡

```
#include "lrt.h"
#include "light.h"
#include "shape.h"
⟨PointLight Classes⟩
⟨PointLight Method Definitions⟩
```

Now we can present some light source implementations. The point light is one of the most straightforward. The `PointLight` class implements an isotropic point light source that shines the same amount of light in all directions.

⟨PointLight Classes⟩≡

```
class PointLight : public Light {
public:
    ⟨PointLight Public Methods⟩
private:
    ⟨PointLight Private Data⟩
};
```

`PointLight`s are positioned at the origin in light space; to place them elsewhere, the world-to-light transform should be adjusted with an additional translation as appropriate. We precompute the world-space position of the light in the constructor by transforming $(0,0,0)$ from light space to world space. We also precompute the source's total power from its intensity, applying the closed form solution of the integral XXX.

⟨*PointLight Method Definitions*⟩≡

```
PointLight::PointLight(const Transform &light2world,
        const Spectrum &intensity)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
}
```

⟨*PointLight Public Methods*⟩+≡

```
Spectrum Power() const {
    return Intensity * 4.f * M_PI;
}
```

⟨*PointLight Private Data*⟩≡

```
Point lightPos;
Spectrum Intensity;
```

⟨*PointLight Public Methods*⟩+≡

```
bool IsDeltaLight() const { return true; }
```

Point lights are defined in terms of their radiant intensity. For an isotropic point light, the radiant intensity is constant and independent of direction. To compute the differential irradiance, we start by computing the incident direction ω from the shading point to the light and normalizing it. Next we initialize the `VisibilityChecker` to check the segment between the two points and compute the incident differential irradiance.

34	`DistanceSquared()`
30	`Dot()`
478	`Light`
478	`Light::LightToWorld`
678	`M_PI`
34	`Normal`
33	`Point`
480	`PointLight`
181	`Spectrum`
43	`Transform`
27	`Vector`
30	`Vector::Hat()`
479	`VisibilityTester`
479	`VisibilityTester::SetSegment()`

⟨*PointLight Method Definitions*⟩+≡

```
Spectrum PointLight::dE(const Point &P, const Normal &N,
        Vector *w, VisibilityTester *visibility) const {
    *w = (lightPos - P).Hat();
    visibility->SetSegment(P, lightPos);
    return Intensity * fabs(Dot(*w, N)) /
        DistanceSquared(lightPos, P);
}
```

13.2.1 Spot Light

⟨*spot.cpp**⟩≡

```
#include "lrt.h"
#include "light.h"
#include "shape.h"
```

⟨*SpotLight Declarations*⟩
⟨*SpotLight Method Definitions*⟩

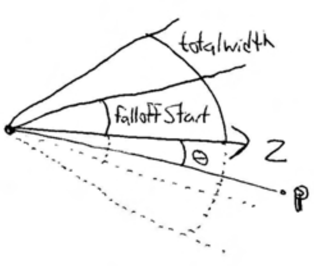


Figure 13.2: Spotlights are defined by two angles, *falloffStart* and *totalWidth*. Objects inside the inner cone of angles, up to *falloffStart* are fully illuminated by the light. The directions between *falloffStart* and *totalWidth* are a transition zone that ramps down from full illumination to no illumination, such that points outside the *totalWidth* cone aren't illuminated at all. The cosine of the angle between the vector to a point *p* and the spotlight axis, θ, can easily be computed with a dot product.

Spot lights are a handy variation on point lights; rather than shining illumination in all directions, they light objects in a cone of directions from their position. For simplicity, we will define the spotlight in the light coordinate system to always be at the position $(0,0,0)$, pointing down the $+z$ axis. To place or orient it elsewhere in the scene, the `Light::WorldToLight` matrix can be set appropriately.

`Light`	478
`Light::LightToWorld`	478
`Point`	33
`Radians()`	677
`Spectrum`	181
`Transform`	43

⟨*SpotLight Declarations*⟩≡

```
class SpotLight : public Light {
public:
    ⟨SpotLight Public Methods⟩
private:
    ⟨SpotLight Private Data⟩
};
```

Two angles are passed in the constructor to set the extent of the `SpotLight`'s cone: the overall angular width of the cone, and the angle at which fall-off from full illumination to no illumination starts; see Figure 13.2. We precompute and store the cosines of these angles in the spotlight object, for efficiency when computing illumination later.

⟨*SpotLight Method Definitions*⟩≡

```
SpotLight::SpotLight(const Transform &light2world,
        const Spectrum &intensity, Float width, Float fall)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    cosTotalWidth = cosf(Radians(width));
    cosFalloffStart = cosf(Radians(fall));
}
```

⟨*SpotLight Private Data*⟩≡

```
Float cosTotalWidth, cosFalloffStart;
Point lightPos;
Spectrum Intensity;
```

The SpotLight::dE() method is almost identical to PointLight::dE(), except that we also call the Falloff() method, which computes the attenuation due to the spotlight cones. This computation is encapsulated in a separate method since other SpotLight methods will need to perform it as well.

To compute the spotlight's strength for a receiving point *p*, we compute the cosine of the angle between the vector from the spotlight origin to the point and the vector alone the center of the spotlight's cone. We compare this to the cosines of the falloff and overall width angles to see where the point lies with respect to the spot light cone. To compute the cosine of the offset angle to a point *p*, we have (see Figure 13.2):

$$\begin{aligned}\cos\theta &= (p - \widehat{(0,0,}0)) \cdot (0,0,1) \\ &= \mathbf{p}_z / \|\mathbf{p}\|\end{aligned}$$

We can trivally determine that points with a cosine greater than the cosine of the falloff angle are inside the cone receiving full illumination, and points with cosine less than the width angle's cosine are completely outside the cone. (Note that the computation is slightly tricky since for $\theta \in [0,2\pi]$, then if $\theta > \theta'$ then $\cos\theta < \cos\theta'$.)

⟨*SpotLight Method Definitions*⟩+≡

```
Spectrum SpotLight::dE(const Point &P, const Normal &N, Vector *w,
        VisibilityTester *visibility) const {
    *w = (lightPos - P).Hat();
    visibility->SetSegment(P, lightPos);
    return Intensity * Falloff(-*w) * fabs(Dot(*w, N)) /
        DistanceSquared(lightPos, P);
}
```

⟨*SpotLight Method Definitions*⟩+≡

```
Float SpotLight::Falloff(const Vector &w) const {
    Vector wl = WorldToLight(w).Hat();
    Float costheta = wl.z;
    if (costheta < cosTotalWidth)
        return 0.;
    if (costheta > cosFalloffStart)
        return 1.;
    ⟨Compute falloff inside spotlight cone⟩
}
```

34	DistanceSquared()
478	Light::WorldToLight
34	Normal
33	Point
181	Spectrum
482	SpotLight
482	SpotLight::cosFalloffStart
482	SpotLight::cosTotalWidth
482	SpotLight::Intensity
482	SpotLight::lightPos
27	Vector
30	Vector::Hat()
479	VisibilityTester
479	VisibilityTester::SetSegment()

For points inside the transition range, we determine how far it is along between the start of falloff and the end, and arbitrarily scale the intensity accordingly.

⟨*Compute falloff inside spotlight cone*⟩≡

```
Float delta = (costheta - cosTotalWidth) /
    (cosFalloffStart - cosTotalWidth);
return delta*delta*delta*delta;
```

We approximate the power of the light by computing the area of the solid angle of directions that is covered by the cone with a spread angle halfway between width and fall. (For the point light, the 4π scale that turns intensity into power comes from the 4π solid angle of the sphere of all directions.)

⟨*SpotLight Public Methods*⟩+≡

```
Spectrum Power() const {
    return Intensity * 2.f * M_PI *
        (1.f - .5 * (cosFalloffStart + cosTotalWidth));
}
```

13.2.2 Texture Projection Light

⟨*projection.cpp**⟩≡

```
#include "lrt.h"
#include "light.h"
#include "shape.h"
#include "mipmap.h"
⟨ProjectionLight Declarations⟩
⟨ProjectionLight Method Definitions⟩
```

Another useful light source acts like a slide projector: it takes a texture map and projects its image out into the scene. We use a projective transformation to project points in the scene onto the light's projection plane; see Figure 13.3. A field of view angle is given with the light so that the constructor can compute an appropriate projection matrix.

Light	478
Light::LightToWorld	478
M_PI	678
MIPMap	417
Point	33
ojectionLight::Intensity	485
rojectionLight::lightPos	485
Spectrum	181
otLight::cosFalloffStart	482
SpotLight::cosTotalWidth	482
SpotLight::Intensity	482
Texture	394
Transform	43

⟨*ProjectionLight Declarations*⟩≡

```
class ProjectionLight : public Light {
public:
    ⟨ProjectionLight Public Methods⟩
private:
    ⟨ProjectionLight Private Data⟩
};
```

⟨*ProjectionLight Method Definitions*⟩≡

```
ProjectionLight::ProjectionLight(const Transform &light2world,
        const Spectrum &intensity, const string &texname,
        Float fov)
    : Light(light2world) {
    ⟨Create ProjectionLight MIP-map⟩
    ⟨Initialize ProjectionLight projection matrix⟩
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
}
```

We could use a `Texture` to represent the light projection distribution, giving the `ProjectionLight` more generality, so that procedural projection patterns could be used, for example. However, having a precise representation of the projection function, as we do by using an image in a `MIPMap` is useful for Monte Carlo sampling the projection distribution.

⟨*Create* `ProjectionLight` *MIP-map*⟩≡

```
int width, height;
Spectrum *texels = ReadImage(texname, &width, &height);
if (texels)
    projectionMap = new MIPMap<Spectrum>(width, height, texels);
else
    projectionMap = NULL;
delete[] texels;
```

⟨*ProjectionLight Private Data*⟩≡

```
MIPMap<Spectrum> *projectionMap;
Point lightPos;
Spectrum Intensity;
```

Similarly to the `PerspectiveCamera`, the `ProjectionLight` computes a projection matrix and the screen-space extent of the projection.

⟨*Initialize* `ProjectionLight` *projection matrix*⟩≡

```
Float aspect = width / height;
if (width > height)  {
    screenX0 = -1.f;
    screenX1 =  1.f;
    screenY0 = -1.f/aspect;
    screenY1 =  1.f/aspect;
}
else {
    screenX0 = -1.f/aspect;
    screenX1 =  1.f/aspect;
    screenY0 = -1.f;
    screenY1 =  1.f;
}
Float opposite = tanf(Radians(fov) / 2.f);
Float tanDiag = opposite * sqrtf(1.f + 1.f/(aspect*aspect));
cosTotalWidth = cosf(atanf(tanDiag));
hither = RAY_EPSILON;
yon = 1e10;
lightProjection = Perspective(fov, hither, yon);
```

417	`MIPMap`
210	`Perspective()`
210	`PerspectiveCamera`
33	`Point`
484	`ProjectionLight`
677	`Radians()`
37	`RAY_EPSILON`
181	`Spectrum`
43	`Transform`

⟨*ProjectionLight Private Data*⟩+≡

```
Transform lightProjection;
Float hither, yon;
Float cosTotalWidth;
Float screenX0, screenX1, screenY0, screenY1;
```

Light the spot light's version, `ProjectionLight::dE()` calls out to a utility method, `Projection()`, to compute how much light is projected in the given direction.

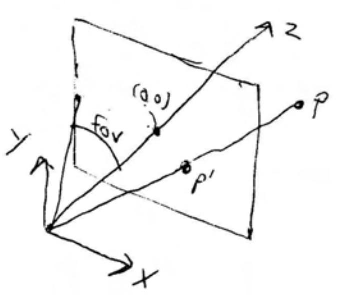


Figure 13.3: The basic setting for projection light sources. A point *p* in the light's coordinate system is projected on to the plane of the image using the light's projection matrix.

⟨*ProjectionLight Method Definitions*⟩+≡

```
Spectrum ProjectionLight::dE(const Point &P, const Normal &N, Vector *w,
        VisibilityTester *visibility) const {
    *w = (lightPos - P).Hat();
    visibility->SetSegment(P, lightPos);
    return Intensity * Projection(-*w) * fabs(Dot(*w, N)) /
        DistanceSquared(lightPos, P);
}
```

Light::WorldToLight	478
Normal	34
Point	33
ProjectionLight	484
ProjectionLight::hither	485
ojectionLight::Intensity	485
rojectionLight::lightPos	485
Spectrum	181
Vector	27
VisibilityTester	479

⟨*ProjectionLight Method Definitions*⟩+≡

```
Spectrum ProjectionLight::Projection(const Vector &w) const {
    Vector wl = WorldToLight(w);
    ⟨Discard directions behind projection light⟩
    ⟨Project point on to projection plane and compute light⟩
}
```

We immediately discard projection points that are behind the hither and plane for the projection. Because the projective transformation has the unfortunate property that it projects points behind the center of projection to points in front of it, is important in particular to discard points with a negative *z* value. Otherwise, given a projected point, we wouldn't be able to know if it was originally behind the light (and not illuminated) or in front of it.

⟨*Discard directions behind projection light*⟩≡

```
if (wl.z < hither) return 0.;
```

After projecting the point to the projection plane, points with coordinate values inside the screen window are inside the projection window. We then offset and scale them to get (s,t) texture coordinates inside $[0,1]^2$ to use when evaluating the projection texture map.

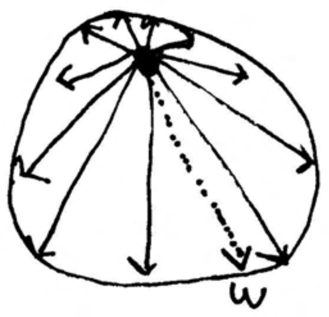


Figure 13.4: An example of a goniometric diagram specifying an outgoing light distribution from a point light source in 2D. The emitted intensity is defined in a fixed set of directions on the unit sphere and the intensity for a given outgoing direction ω is found by interpolating the intensities of the adjacent samples.

⟨*Project point on to projection plane and compute light*⟩≡

```
Point Pl = lightProjection(Point(wl.x, wl.y, wl.z));
if (Pl.x < screenX0 || Pl.x > screenX1 ||
    Pl.y < screenY0 || Pl.y > screenY1) return 0.;
Float s = (Pl.x - screenX0) / (screenX1 - screenX0);
Float t = (Pl.y - screenY0) / (screenY1 - screenY0);
return projectionMap ? projectionMap->Lookup(s, t) : 1.;
```

⟨*ProjectionLight Public Methods*⟩+≡

```
Spectrum Power() const {
    return Intensity * 2.f * M_PI * (1.f - cosTotalWidth);
}
```

678	M_PI
33	Point
485	ProjectionLight::cosTotalWidth
485	ProjectionLight::Intensity
485	ProjectionLight::lightProjection
485	ProjectionLight::projectionMap
485	ProjectionLight::screenX0
485	ProjectionLight::screenX1
485	ProjectionLight::screenY0
485	ProjectionLight::screenY1
181	Spectrum

13.2.3 Goniometric diagram lights

⟨*goniometric.cpp**⟩≡

```
#include "lrt.h"
#include "light.h"
#include "shape.h"
#include "scene.h"
#include "mipmap.h"
⟨GoniometricLight Declarations⟩
⟨GoniometricLight Method Definitions⟩
```

A *goniometric diagram* describes the distribution of luminance from a point light source; widely used in illumination engineering to characterize lights. Here, we'll implement a light source that uses goniometric diagrams encoded in 2D image maps wrapped around the sphere to describe the emission distribution of the light. The implementation is very similar to the point light sources defined previously in this section; we just scale the intensity based on outgoing direction according to the goniometric diagram's values. Figure 13.4 shows an example in two dimensions.

⟨*GoniometricLight Declarations*⟩≡

```
class GoniometricLight : public Light {
public:
    ⟨GoniometricLight Public Methods⟩
private:
    ⟨GoniometricLight Private Data⟩

};
```

⟨*GoniometricLight Method Definitions*⟩+≡

```
GoniometricLight::GoniometricLight(const Transform &light2world,
        const Spectrum &intensity, const string &texname)
    : Light(light2world) {
    lightPos = LightToWorld(Point(0,0,0));
    Intensity = intensity;
    ⟨Create mipmap for GoniometricLight⟩
}
```

⟨*Create* `mipmap` *for* `GoniometricLight`⟩≡

```
int width, height;
Spectrum *texels = ReadImage(texname, &width, &height);
if (texels) {
    Float *ftex = new Float[width*height];
    for (int i = 0; i < width*height; ++i)
        ftex[i] = texels[i].y();
    mipmap = new MIPMap<Float>(width, height, ftex);
    delete[] texels;
    delete[] ftex;
}
else mipmap = NULL;
```

⟨*GoniometricLight Private Data*⟩≡

```
Point lightPos;
Spectrum Intensity;
MIPMap<Float> *mipmap;
```

Goniometric diagrams are usually defined in a coordinate space where the y axis is up, so we'll swap y and z before using the spherical coordiantes functions...

⟨*GoniometricLight Public Methods*⟩+≡

```
Float Scale(const Vector &w) const {
    Vector wp = WorldToLight(w.Hat());
    swap(wp.y, wp.z);
    Float theta = SphericalTheta(wp), phi = SphericalPhi(wp);
    Float s = phi * INV_TWOPI, t = theta * INV_PI;
    return mipmap ? mipmap->Lookup(s, t) : 1.;
}
```

⟨*GoniometricLight Public Methods*⟩+≡

```
Spectrum Power() const {
    return 4.f * M_PI * Intensity;
}
```

`INV_PI`	678
`INV_TWOPI`	678
`Light`	478
`Light::LightToWorld`	478
`Light::WorldToLight`	478
`MIPMap`	417
`MIPMap::Lookup()`	427
`Point`	33
`Spectrum`	181
`SphericalPhi()`	193
`SphericalTheta()`	193
`Transform`	43
`Vector`	27

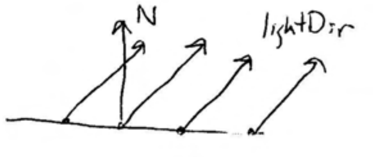


Figure 13.5: All incident light from a distant light source comes in from the same direction...

13.3 Distant Lights

```
⟨distant.cpp*⟩≡
  #include "lrt.h"
  #include "light.h"
  #include "shape.h"
  #include "scene.h"
  ⟨DistantLight Declarations⟩
  ⟨DistantLight Method Definitions⟩
```

```
⟨DistantLight Declarations⟩≡
  class DistantLight : public Light {
  public:
      ⟨DistantLight Public Methods⟩
  private:
      ⟨DistantLight Private Data⟩
  };
```

490	DistantLight::L
490	DistantLight::lightDir
488	GoniometricLight::Intensity
478	Light
478	Light::LightToWorld
678	M_PI
181	Spectrum
43	Transform
27	Vector

Another useful light source type is a *directional light*. It describes an emitter where at every point in space, illumination is arriving from the same direction (see Figure 13.5). Such a light is also called a point light "at infinity", since as a point light becomes progressively farther away, it acts more and more like a directional light. Light sources like the sun (as considered from earth) can be thought of as directional light sources—though they are actually point or area light sources, because they're so far away, the illumination effectively arrives in parallel beams.

```
⟨DistantLight Method Definitions⟩≡
  DistantLight::DistantLight(const Transform &light2world,
          const Spectrum &radiance, const Vector &dir)
      : Light(light2world) {
      lightDir = LightToWorld(dir).Hat();
      L = radiance;
  }
```

Directional lights don't quite fit in with our previous decision to characterize lights in terms of their total power. Interestingly enough, the total power emitted by a directional light is proportional to the area of the scene receiving light. For now, we interpret the power value as the amount of emitted radiance along a ray from the directional light. Later, in Section 15.4, we will revisit this issue to more accurately compute total power from directional lights.

Figure 13.6: The same scene, illuminated by a point light source (top) and an area light source (bottom).

⟨*DistantLight Private Data*⟩≡

```
Vector lightDir;
Spectrum L;
```

Now the method for differential irradiance, which reflects a straightforward application of Equation 5.4.7.

DistantLight	489
Dot()	30
Normal	34
Point	33
Spectrum	181
Vector	27
VisibilityTester	479
isibilityTester::SetRay()	480

⟨*DistantLight Method Definitions*⟩+≡

```
Spectrum DistantLight::dE(const Point &P,
        const Normal &N, Vector *w, VisibilityTester *visibility) const {
    *w = lightDir;
    visibility->SetRay(P, *w);
    return L * fabs(Dot(*w, N));
}
```

XXX fix me XXX. Need to just pass `Scene` into here? Then maybe move this to MC chapter? Or do that basic work here, build on it there?

⟨*DistantLight Public Methods*⟩+≡

```
Spectrum Power() const {
    return L;
}
```

13.4 Area Lights

Area lights are light sources defined by a shape that it emitting light. Because such a light illuminates points in the scene from multiple directions, we can't just consider differential irradiance from a single direction as we have some with light sources so far. In general, computing radiometric quantities related to area lights requires computing integrals over the surface of the light, a task that will be revisited with the Monte Carlo integration techniques of Chapter 14. The reward for this work (and computational expense) is soft shadow penumbra, rather than the hard shadows that are cast by point lights. Figure 13.6 shows a comparison of a scene illuminated by a point light and the same scene illuminated by an area light.

```
⟨area.cpp*⟩≡
  #include "light.h"
  #include "primitive.h"
  ⟨AreaLight Method Definitions⟩
```

```
⟨Light Declarations⟩+≡
  class AreaLight : public Light {
  public:
      ⟨AreaLight Interface⟩
  protected:
      ⟨AreaLight Protected Data⟩
  };
```

In the constructor, we calculate and store the area of the light source when it is defined, because these area calculations may be expensive. Computation methods for surface area are described in Chapter 3.

```
⟨AreaLight Method Definitions⟩≡
  AreaLight::AreaLight(const Transform &light2world,
          const Spectrum &le, const Reference<Shape> &s)
      : Light(light2world) {
      Lemit = le;
      if (s->CanIntersect())
          shape = s;
      else {
          ⟨Create ShapeSet for Shape⟩
      }
      area = shape->Area();
  }
```

478	Light
100	LoopSubdiv
664	Reference
63	Shape
181	Spectrum
43	Transform

```
⟨AreaLight Protected Data⟩≡
  Spectrum Lemit;
  Reference<Shape> shape;
  Float area;
```

Need to explain this Some shapes, like `LoopSubdiv`s, aren't amenable to being used as area lights directly. We need to refine them until we have simple `Shape`s that can all sample themselves. We'll do this refinement here, and store them in a `ShapeSet`, so that the rest of the code here can always just pretend that there's one single shape... The fragment ⟨*Create* `ShapeSet` *for* `Shape`⟩, not included here, handles the details of this.

One ugly thing about this is that we are overloading the meaning of `CanIntersect()`, to include "can we compute surface area and can we sample points on this thing". Need a better name for `CanIntersect()` to reflect this?

Should this go to the shapes chapter? On some level, yes, since it is a `Shape`. On another level, no, since it's more like a private utility thing that `AreaLight`s use...

⟨*Shape Declarations*⟩+≡

```
class ShapeSet : public Shape {
public:
    ⟨ShapeSet Public Methods⟩
private:
    vector<Reference<Shape> > shapes;
    Float area;
    vector<Float> areaCDF;
};
```

We first provide two methods unique to area lights. The first evaluates the area light's emitted radiance, *L*, at a point on the surface of the light for a given direction. We assume that the given point is on the surface of the light. Furthermore, we will use the convention in `lrt` that area lights are *one-sided*–they only illuminate from the side of the surface that the surface normal faces toward; the other side has no emission.

For the basic area lights here, the amount of radiance emitted is the same at all points on the light and the same for all outgoing directions in the hemisphere about the normal. (More generally, emission may vary depending on both of these values.) We can compute emitted irradiance by dividing flux by the surface area (because emission is constant over the surface); dividing this by π, the area of the hemisphere with projected solid angle measure, gives radiance in a particular direction.

`AreaLight::Lemit`	491
`Dot()`	30
`M_PI`	678
`Normal`	34
`Point`	33
`Reference`	664
`Shape`	63
`Spectrum`	181
`vector`	658
`Vector`	27

⟨*AreaLight Interface*⟩+≡

```
virtual Spectrum L(const Point &p, const Normal &N,
        const Vector &w) const {
    return Dot(N, w) > 0 ? Lemit : 0.;
}
```

It's also handy to be able to compute emitted irradiance (often called *radiosity*) at a point on the light. Here we also assume that the point `x` is on the light's surface and that the light's emission doesn't vary by location.

⟨*AreaLight Interface*⟩+≡

```
virtual Spectrum B(const Point &p) const {
    return Lemit * M_PI;
}
```

⟨*AreaLight Interface*⟩+≡

```
Spectrum Power() const {
    return Lemit * area * M_PI;
}
```

We won't provide a differential irradiance method here, but will define one in the Monte Carlo chapter once we have developed its underlying mathematics.

⟨*AreaLight Interface*⟩+≡

```
bool IsDeltaLight() const { return false; }
```

Figure 13.7: Uffizi latlong map

13.5 Infinite Area Lights

```
⟨infinite.cpp*⟩≡
  #include "lrt.h"
  #include "light.h"
  #include "texture.h"
  #include "shape.h"
  #include "scene.h"
  ⟨InfiniteAreaLight Declarations⟩
  ⟨InfiniteAreaLight Method Definitions⟩
```

478 Light

```
⟨InfiniteAreaLight Declarations⟩≡
  class InfiniteAreaLight : public Light {
  public:
      ⟨InfiniteAreaLight Public Methods⟩
  private:
      ⟨InfiniteAreaLight Private Data⟩
  };
```

Another useful kind of light is the infinite area light. This is an area light source at infinity that surrounds the entire scene; one good way to visualize it is as an enormous sphere that casts light into the scene from every direction. One use of infinite area lights is *environment lighting*: given a representation of illumination in some environment, synthetic objects can be lit as if they were in that environment. A widely-used representation for light for this application is the latitude-longitude radiance map; it stores emitted radiance as a function of direction. A lat-long environment map of the Uffizi Gallery in Florence is shown in Figure 13.7; a teapot illuminated by the illumination from this map is shown in Figure 13.8.

Like the other lights, the `InfiniteAreaLight` takes a transformation matrix; here its use is to orient the texture map. We use spherical coordinates to map from directions on the sphere to (θ, ϕ) directions from from there to (u, v) texture coordinates; the transformation describes which direction is "up".

Figure 13.8: Teapot in Uffizi environment

⟨*InfiniteAreaLight Method Definitions*⟩+≡

```
InfiniteAreaLight::InfiniteAreaLight(const Transform &light2world,
        const Spectrum &L, const string &texmap)
    : Light(light2world) {
    radianceMap = NULL;
    if (texmap != "") {
        int width, height;
        Spectrum *texels = ReadImage(texmap, &width, &height);
        if (texels)
            radianceMap = new MIPMap<Spectrum>(width, height, texels);
        delete[] texels;
    }
    Lbase = L;
}
```

⟨*InfiniteAreaLight Private Data*⟩≡

```
Spectrum Lbase;
MIPMap<Spectrum> *radianceMap;
```

CoordinateSystem()	32
InfiniteAreaLight	493
INV_PI	678
INV_TWOPI	678
Light	478
Light::WorldToLight	478
MIPMap	417
MIPMap::Lookup()	427
Spectrum	181
SphericalPhi()	193
SphericalTheta()	193
Transform	43
Vector	27
Vector::Hat()	30

Like directional lights, the total power from the infinite area light is related to the surface area of the scene. Therefore, here we also treat the power as the radiance.

⟨*Compute infinite light radiance for this direction*⟩≡

```
Spectrum L = Lbase;
if (radianceMap != NULL) {
    Vector wh = WorldToLight(w).Hat();
    Vector S, T;
    CoordinateSystem(wh, &S, &T);
    Float s = SphericalPhi(wh) * INV_TWOPI;
    Float t = SphericalTheta(wh) * INV_PI;
    L *= radianceMap->Lookup(s, t);
}
```

Because infinite area lights need to be able to contribute radiance to rays that don't hit any geometry in the scene, we'll add a method to the base `Light` class that returns emitted radiance due to that light along a ray that didn't hit anything in the scene.

XXX how does this change/become better integrated when we have support for volumetric stuff?? XXX

⟨*Light Method Definitions*⟩+≡

```
Spectrum Light::Le(const RayDifferential &) const {
    return Spectrum(0.);
}
```

The `InfiniteAreaLight`'s implementation of this can reuse the fragment from its `dE` method.

⟨*InfiniteAreaLight Method Definitions*⟩+≡

```
Spectrum InfiniteAreaLight::Le(const RayDifferential &r) const {
    Vector w = r.d;
    ⟨Compute infinite light radiance for this direction⟩
    return L;
}
```

⟨*InfiniteAreaLight Public Methods*⟩+≡

```
Spectrum Power() const {
    return Lbase;
}
```

⟨*InfiniteAreaLight Public Methods*⟩+≡

```
bool IsDeltaLight() const { return false; }
```

493	`InfiniteAreaLight`
478	`Light`
35	`Ray::d`
37	`RayDifferential`
181	`Spectrum`
27	`Vector`

Further Reading

Warn developed early models of light sources with non-isotropic emission distributions (Warn 1983). More recently, Barzel has described a highly parameterized model for light sources, including many controls for controlling rate of falloff, the area of space that is illuminated, etc (Barzel 1997). Bjorke has developed flexible controls for controlling illuminaton for artistic effect (Bjorke 01 renderman course notes). (The Barzel and Bjorke approaches are not physically based, however.)

Blinn and Newell first introduced the idea of environment maps and their use for simulating illumination (Blinn and Newell 1976), though they only considered illumination of specular objects. Greene also developed these ideas, considering anti-aliasing and different representations for environment maps (Greene 1986).

Miller and Hoffman first considered using environment maps to illuminate objects with diffuse BRDFs (Miller and Hoffman 1984). Debevec later extended this work (Debevec 1998).

As for efficient ray tracing, lights are special in that we don't care about the geometric details of intersection, just whether or not there is one along a given ray. Beyond the `IntersectP()` stuff we already do, shadow cache, light buffer (Haines and Greenberg 1986), shaft culling (Haines and Wallace 1994).

Pearce points out that shadow cache doesn't work well if scene has fine tessellations; may be better to cache the voxel that held the last occluder, or something similar (Pearce 1991). It can also not be so good if multiple levels of reflection and refraction are present...

Minkowski sum to effectively expand primitives (or bounds of primitives) in scene so that intersecting one ray against primitives tells if any of a collection of rays might have intersected the actual primitives (Lukaszewski 2001). Also simplification envelopes stuff: Cohen et al (Cohen, Varshney, Manocha, Turk, Weber, Agarwal, Brooks Jr., and Wright 1996).

XXX Mention ways of gathering up bundles of rays XXX

Hart et al generalize light shadow cache, find blockers and clip light source geometry against them (Hart, Dutré, and Greenberg 1999).

Exercises

13.1 depth-mapped shadows for lights. Williams (Williams 1978), Reeves et al (Reeves, Salesin, and Cook 1987).

13.2 Another technique that takes advantage of this property of shadow rays is the *shadow cache*. Each light source in the scene keeps a pointer to the last primitive that occluded light from that light source. Subsequent shadow rays are fist checked against this blocker–since the blocking object will often block many shadow rays in a row, this can make it much faster to find the blocker.

13.3 Volumetric ambient light that varies with `x` or `w`

13.4 Through clever algebraic manipulation and precomputation of one more value in the constructor, the `SpotLight::Falloff()` method can be rewritten to compute the exact same result (modulo floating point differences) while using no square root computations and no divides (recall the `Vector::Hat()` uses a square root and a divide.) Derive and implement this optimization. How much is running time improved on a spotlight-heavy scene?

14. Monte Carlo Integration: Basic Concepts

Judicious use of randomness has revolutionized the field of algorithm design. Randomized algorithms fall broadly into two classes: *Las Vegas* and *Monte Carlo*. Las Vegas algorithms are those that use randomness but always give the correct answer. Monte Carlo algorithms, on the other hand, frequently give the wrong answer, but give the right answer *on average*. So, by averaging the results of several runs of a Monte Carlo algorithm (on the same input), we can get a result that is provably very close to the true answer. Rajeev Motwani has written an excellent introduction to the field of randomized algorithms (Motwani and Raghavan 1995).

Monte Carlo Integration[1] is a way of using random sampling to estimate the values of integrals. The key advantage of Monte Carlo integration is that one only needs to be able to evaluate the integrand at an arbitrary points in the domain in order to estimate the value of $\int f(x)\mathrm{d}x$. This not only makes Monte Carlo easy to implement, it also means that we can numerically evaluate integrals that are difficult or impossible to integrate analytically. In contrast to traditional quadrature-based techniques such as the trapezoid, midpoint, or Simpson's rules, Monte Carlo's convergence rate is independent of the dimensionality of the integrand.

Many of the integrals that arise in rendering are difficult or impossible to evaluate directly. For example, to compute the amount of light reflected by a surface at a point (Equation 5.4.9), we must integrate the product of the incident light and the BSDF over the unit sphere. A closed form expression for this product is almost never available, and even if it were performing the integral analytically seems

[1] For brevity, we will refer to Monte Carlo Integration simply as "Monte Carlo".

hopeless. Monte Carlo integration allows us to estimate the reflected radiance simply by simply choosing a set of directions over the sphere, computing incident radiance along them, multiplying by the BSDF's value, and applying a weighting term.

The main disadvantage of Monte Carlo is that if we use n samples to estimate the integral, the algorithm converges as $O(n^{-1/2})$. More simply, to reduce the error by half, we must evaluate four times more samples. In images, artifacts from Monte Carlo sampling manifest themselves as noise—some pixels are much too bright and some are much too dark. Although this can be perceptually less objectionable than aliasing, noise is still distracting, and it is still error. Most of the current research in Monte Carlo is in reducing this error as much as possible.

```
⟨mc.h*⟩≡
  ⟨MC Utility Declarations⟩
  ⟨MC Class Declarations⟩
  ⟨MC Inline Functions⟩
```

```
⟨mc.cpp*⟩≡
  #include "lrt.h"
  #include "geometry.h"
  #include "shape.h"
  #include "mc.h"
  ⟨MC Function Definitions⟩
```

14.1 Background and Probability Review

We will start by defining some basic terms and reviewing basic concepts from probability. We assume that the reader is already familiar with probability at a high-school level; readers needing a more complete introduction to this topic should consult a textbook such as Sheldon Ross's *Introduction to Probability Models* (Ross 2002).

A *random variable X* is a value chosen by some random process. We will always use capital letters to denote random variables. Applying any function to a random variable results in a new random variable $Y = f(X)$. Random variables are always drawn from some domain, which can be either discrete (e.g. a fixed set of possibilities) or continuous (e.g. the real numbers $\mathbb{R}$).

For example, the result of a roll of a die is a discrete random variable sampled from the set of events $X_i = \{1,2,3,4,5,6\}$. Each event has a probability $p_i = 1/6$ and the sum of probabilities $\sum p_i$ is necessarily one. We can take a continuous random variable ξ that is uniformly distributed among the real numbers between zero and one and map it to a discrete random variable, choosing X_i if:

$$\sum_{j=1}^{i-1} p_j < \xi \leq \sum_{j=1}^{i} p_j.$$

For lighting applications, we might want to define a probability of sampling illumination from each of a set of light sources, based on the power Φ_i from each source relative to the total power from all sources.

$$p_i = \frac{\Phi_i}{\sum_j \Phi_j}.$$

Notice that these p_i sum to one, which is always true of probabilities.

The *cumulative distribution function* (CDF) $P(x_i)$ of a random variable is the probability that a value from the variable's domain is less than x_i:

$$P(x) = Pr\{X \leq x\}.$$

For the die example, $P(2) = 1/3$, since two of the six possibilities are less than or equal to 2.

A particularly important random variable is the *uniform random variable*, which we will write as ξ. This variable takes on all values in its domain with equal probability. This particular variable is important for two reasons. First, it is easy to generate a variable with this distribution in software—most runtime libraries have a random number generator that does just that. Second, as we will show later, we can generate samples from arbitrary distributions by first starting with a uniform random variable and applying an appropriate transformation.

Another example of a continuous random variable is one that ranges over the real numbers between 0 and 2 where the probability of it taking on any particular value x is related to the value $2-x$: it is twice as likely for it to take on a value around zero as it is to take one around one, etc. The *probability density function* (PDF) formalizes this idea: it describes the relative probability of a random variable taking on a particular value. The PDF $p(x)$ is just the derivative of the random variable's CDF.

$$p(x) = \frac{dP(x)}{\mathrm{d}x}$$

For uniform random variables, $p(x)$ is a constant.

PDFs are necessarily non-negative and always integrate to one over their domains. For the uniform random variable ξ, $P(x) = x$ and $p(x) = 1$. We will use the notation $x \sim p$ to denote that x is a random variable with the PDF p.

Given an arbitrary interval $[a,b]$ in the domain, the PDF can give the probability that a random variable lies inside the interval.

$$P(x \in [a,b]) = \int_a^b p(x)\mathrm{d}x$$

This follows directly from the first fundamental theorem of calculus and the definition of a PDF.

14.1.1 Expected Values and Variance

The *expected value* $E_p[f(x)]$ of a function f is defined as the average value that f takes, assuming a distribution of values $p(x)$ over its domain.

$$E_p[f(x)] = \int f(x)p(x)\mathrm{d}x \tag{14.1.1}$$

Consider finding the expected value of the cosine function between 0 and π, where p is uniform[2]. Because $p(x)$ must integrate to one over the domain, we have $p(x) =$

[2] When computing expected values with a uniform distribution, we will drop the subscript p

$1/\pi$ and

$$\begin{aligned}
E[\cos(x)] &= \int_0^\pi \frac{\cos x}{\pi}\,\mathrm{d}x \\
&= \frac{1}{\pi}(-\sin\pi + \sin 0) \\
&= 0
\end{aligned}$$

Which is precisely what we expect.

The *variance* of a function is the expected deviation of the function from its expected value:

$$V[f(x)] = E\left[(f(x) - E[f(x)])^2\right] \tag{14.1.2}$$

The expected value and variance have three important properties that follow immediately from their respective definitions:

$$\begin{aligned}
E[af(x)] &= aE[f(x)] \\
E[\sum_i f(X_i)] &= \sum_i E[f(X_i)] \\
V[af(x)] &= a^2V[f(x)]
\end{aligned}$$

These properties, and some simple algebraic manipulation, yield a much simpler expression for the variance:

$$\begin{aligned}
V[f(x)] &= E\left[(f(x) - E[f(x)])^2\right] \\
V[f(x)] &= E\left[f^2(x) - 2f(x)E[f(x)] + E[f(x)]^2\right] \\
V[f(x)] &= E\left[f^2(x)\right] - 2E\left[f(x)E[f(x)]\right] + E\left[E[f(x)]^2\right] \\
V[f(x)] &= E\left[f^2(x)\right] - 2E[f(x)]^2 + E\left[f(x)\right]^2 \\
V[f(x)] &= E\left[f^2(x)\right] - E[f(x)]^2 \qquad (14.1.3)
\end{aligned}$$

So the variance is simply the expected value of the square minus the square of the expected value. One final property of the variance that only holds if the random variables are *independent*:

$$V[\sum_i f(X_I)] = \sum_i V[f(X_i)]$$

14.2 The Monte Carlo Estimator

We can now define the Monte Carlo estimator, which lets us estimate the value of an integral.

Let's suppose that we want to evaluate some difficult one-dimensional integral $I = \int_b^a f(x)dx$. Assuming we have a steady supply of uniform random variables $\xi_i \in [a,b]$, then we can easily estimate this integral as:

$$F_N = \frac{b-a}{N} \sum_{i=1}^{N} f(\xi_i)$$

Why is this a good way to estimate I? It is easy to show that the expected value of this estimator is in fact equal to I. First, note that the PDF $p(x)$ corresponding to our random variable ξ_i must be $\frac{1}{b-a}$, since p must be a constant and integrate to one over the domain $[a,b]$. So, the expected value of our estimator F_N is:

$$\begin{aligned}
E[F_N] &= E\left[\frac{b-a}{N} \sum_{i=1}^{N} f(\xi_i)\right] \\
&= \frac{b-a}{N} \sum_{i=1}^{N} E\left[f(\xi_i)\right] \\
&= \frac{b-a}{N} \sum_{i=1}^{N} \int_a^b f(x)p(x)dx \\
&= \frac{1}{N} \sum_{i=1}^{N} \int_a^b f(x)dx \\
&= \int_a^b f(x)dx \\
&= I
\end{aligned}$$

The restriction to uniform random variables can be relaxed with only a small generalization. This is extremely important, since careful choosing of the PDF from which to draw samples is the primary technique used to reduce variance in Monte Carlo. If the random variables X_i are drawn from some arbitrary PDF $p(x)$, then we use the estimator:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}$$

It is similarly easy to see that this is the right estimator:

$$\begin{aligned}
E[F_N] &= E\left[\frac{1}{N}\sum_{i=1}^{N}\frac{f(X_i)}{p(X_i)}\right] \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b \frac{f(x)}{p(x)}p(x)dx \\
&= \frac{1}{N}\sum_{i=1}^{N}\int_a^b f(x)dx \\
&= \int_a^b f(x)dx \\
&= I
\end{aligned}$$

Extending this estimator to multiple dimensions or complex integration domains is straightforward. N samples X_i are taken from a multi-dimensional (or "joint") PDF and the estimator is applied as usual. This is a key difference between Monte Carlo and traditional deterministic quadrature techniques; The number of samples taken in Monte Carlo is independent of the dimensionality of the integral, while in numerical quadrature the number of samples required is exponential in the dimension.

Showing that the Monte Carlo estimator converges to the right answer is not enough to justify its use; a good rate of convergence is important too. To reason about the rate of convergence of our estimator, we will use a theorem from probability called Chebyshev's inequality. As originally stated, the theorem says that:

$$Pr\{|F - E[F]| \geq k\} \leq \frac{V[F]}{k^2} \tag{14.2.4}$$

To use this inequality for our own purposes, we first write $\delta = \frac{V[f]}{k^2}$, and obtain:

$$Pr\left\{|F - E[F]| \geq \sqrt{\left(\frac{V[F]}{\delta}\right)}\right\} \leq \delta$$

Now, define $Y_i = f(X_i)$. Our estimator F_N is therefore:

$$F_N = \frac{1}{N}\sum_i Y_i$$

Because the Y_i are independent and all have the same variance (which we will call $V[Y]$, we can simplify the expression for $V[F]$:

$$\begin{aligned}
V[F] &= V\left[\frac{1}{N}\sum_i Y_i\right] \\
&= \frac{1}{N^2}V\left[\sum_i Y_i\right] \\
&= \frac{1}{N^2}\sum_i V\left[Y_i\right] \\
&= \frac{1}{N}V\left[Y\right]
\end{aligned}$$

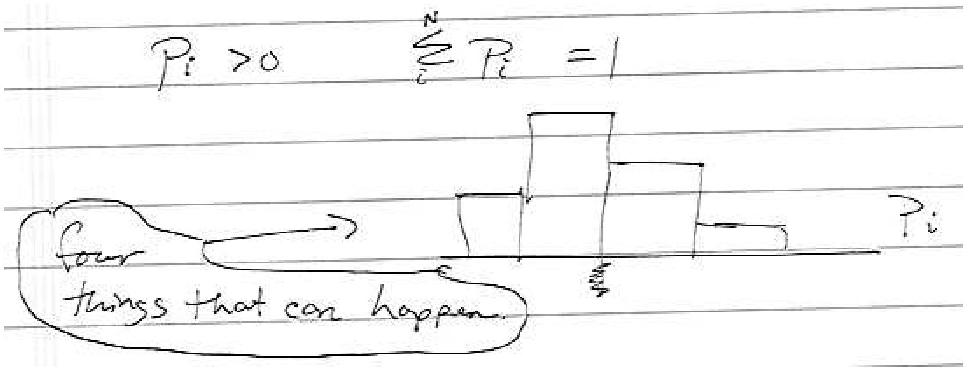


Figure 14.1: A discrete PDF.

Since the root-mean-square error is defined as the square root of the variance, this immediately tells us that the RMS error decreases as $\frac{1}{\sqrt{N}}$. In addition, substituting this result into the Chebyshev inequality, we get:

$$Pr\left\{ |F - E[F]| \geq \frac{1}{\sqrt{N}}\sqrt{\left(\frac{V[Y]}{\delta}\right)} \right\} \leq \delta$$

In other words, if we fix the error confidence δ, the absolute error goes as $O(\sqrt{N})$, just like the RMS error. We can obtain slightly tighter bounds on the absolute error using the central limit theorem; see Veach for details (Veach 1997).

The key insight to all this analysis is that it is independent of the dimensionality of the integral being estimated. Although standard quadrature converges faster than $O(\sqrt{N})$ in one dimension, its performance is exponentially poorer as the dimensionality increases, making Monte Carlo the only practical numerical integration algorithm for high dimensional integrals. We have already encountered some high-dimensional integrals in this book, and in the next chapter we will see that the path tracing formulation of the rendering equation is an *infinite-dimensional* integral!

14.3 The Inversion Method for Sampling Random Variables

In order to properly evaluate equation 14.2, we need to be able to draw a random sample from a given probability distribution. To understand how this process works, first consider a simple, discrete example. Suppose we have a process with four possible outcomes. The probabilities of each of the four outcomes are given by p_1, p_2, p_3, and p_4, respectively, with the requirement that $\sum_{i=1}^{4} p_i = 1$. The corresponding PDF is shown in figure 14.1.

In order to draw a sample from this distribution, we first construct the CDF $P(x)$. In the continuous case, P is simply the indefinite integral of p. In the discrete case, however, we can directly construct the CDF by simply stacking the bars on top of each other, starting at the left. This is shown in figure 14.2.

Notice that the height of the rightmost bar must be one because of the requirement that all probabilities sum to one. Now, to draw a sample from the original distribution, we take a *uniform* random number $\xi \in [0,1]$ and project it onto the CDF from the side. This is illustrated in figure 14.3.

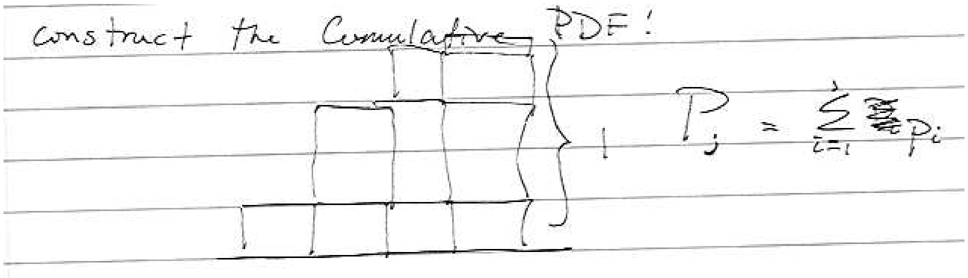


Figure 14.2: A discrete CDF.

Figure 14.3: Drawing the sample from the discrete distribution.

It should be clear that this draws from the correct distribution – the probability of the uniform sample hitting any particular bar is exactly equal to the height of that bar. In order to generalize this technique to continuous distributions, consider what happens as the number of discrete possiblities approaches infinity. The PDF from figure 14.1 becomes a smooth curve, and the CDF from figure 14.2 becomes its integral. The projection process described in the previous paragraph is still the same, although if the function is continuous it has a convenient mathematical interpretation—we simply *invert* the CDF function and evaluate it. This technique is, unsurprisingly, called the *inversion method*.

More precisely, we can draw a sample X_i from an arbitrary PDF $p(x)$ with the following steps:

1. Compute the CDF $P(x) = \int p(x)dx$.
2. Compute the inverse $P^{-1}(x)$.
3. Obtain a uniformly distributed random number ξ.
4. Compute $X_i = P^{-1}(\xi)$.

14.3.1 Example: Power Distribution

As an example of how this procedure works, let's draw samples from a *power distribution*. This will come up again when we are trying to sample the Blinn microfacet model. A power distribution is defined as:

$$p(x) \propto x^n, \text{ or } p(x) = cx^n$$

The first task to tackle is always to normalize the PDF. In most cases, this simply involves computing the value of some proportionality constant (c in this case). This is almost always accomplished by enforcing the constraint that $\int_0^1 p(x)dx = 1$, so:

$$\begin{aligned}
\int_0^1 cx^n dx &= 1 \\
c\left.\frac{x^{n+1}}{n+1}\right|_0^1 &= 1 \\
\frac{c}{n+1} &= 1 \\
c &= n+1
\end{aligned}$$

Therefore, $p(x) = (n+1)x^n$. We can integrate this to get the CDF: $P(x) = x^{n+1}$. Now, inversion is simple: $P^{-1}(x) = \sqrt[n+1]{x}$. Now that we have P^{-1}, it is simple to draw samples from the power distribution:

$$X = \sqrt[n+1]{\xi}$$

A trick that works only for the power distribution is to select $X = \max(\xi_1, \xi_2, \ldots, \xi_{n+1})$. This is a power distribution! To see why, note that $Pr\{X < x\}$ is the probability that *all* the $\xi_i < x$. But the ξ_i are independent, so:

$$Pr\{X < x\} = \prod_{i=1}^{n+1} Pr\{\xi_i < x\} = x^{n+1}$$

Which is exactly the desired CDF[3]. Depending on the speed of your random number generator, this techinque can be faster than the inversion method for small values of n.

14.3.2 Example: Exponential Distribution

In volume rendering, it is frequently useful to draw samples from a distribution $p(x) \propto e^{-cx}$. First, we must normalize this distribution so that it integrates to one:

$$\begin{aligned}
\int_0^\infty ke^{-cx} &= -\left.\frac{k}{c}e^{-cx}\right|_0^\infty \\
&= \frac{k}{c} = 1
\end{aligned}$$

So we know that $k = c$, and our PDF is $p(x) = ce^{-cx}$. Now, we integrate to get $P(x) = -e^{-cx} + C$. Of course, because this is a CDF, we know that[4] $P(\infty) = 1$, so $C = 1$ and the CDF is $P(x) = 1 - e^{-cx}$. This function is straightforward to invert:

$$P^{-1}(x) = -\frac{\ln(1-x)}{c}$$

and we can draw samples thusly: $X = -\frac{\ln(1-\xi)}{c}$. This can be further simplified by making the observation that if ξ is a uniformly distributed random number, so

[3] A similar trick can be used for Gaussian distributions – just take the *average* instead of the maximum. This is a direct consequence of the central limit theorem.

[4] This notion is deliberately sloppy; the more formal requirement is that $\lim_{x\to\infty} P(x) = 1$.

is $1-\xi$, so we can safely replace $1-\xi$ by ξ without losing anything. Therefore, our final sampling strategy is:

$$X = -\frac{\ln(\xi)}{c}$$

14.4 Transforming Between Different Distribution Functions

So far, we have assumed that we started with uniformly distributed random numbers. But suppose we are given random numbers X_i that are already drawn from some PDF $p_X(x)$. Now, if we compute $Y_i = y(X_i)$, what is the new distribution? This might seem like an esoteric question, but we will see that understanding this kind of transformation is critical for drawing samples of multi-dimensional distribution functions.

For simplicity, first assume that $y(x)$ is continuous and non-decreasing (e.g., $y'(x) \geq 0$). Because y is non-decreasing, we know that $Pr\{Y \leq y(x)\} = Pr\{X < x\}$. This means that $P_y(y) = P_x(x)$ where y is shorthand for $y(x)$. Remember that P is the CDF, so we get the PDF by differentiation.

By the chain rule, we get:

$$p_y(y)\frac{dy}{dx} = p_x(x)$$

or

$$p_y(y) = \left(\frac{dy}{dx}\right)^{-1} p_x(x)$$

If $y(x)$ can be non-increasing, this works out to

$$p_y(y) = \left|\frac{dy}{dx}\right|^{-1} p_x(x)$$

How can we use this formula? Suppose that $p_x(x) = 2x$ over the domain $[0,1]$, and let $Y = \sin(X)$. What is the PDF of Y? We know that $\frac{dy}{dx} = \cos(x)$, so:

$$\begin{aligned} p_y(y) &= |\cos(x)|^{-1} p_x(x) \\ &= (\cos(x))^{-1} 2x \\ &= \frac{2\sin^{-1}(y)}{\cos\left(\sin^{-1}(y)\right)} \\ &= \frac{2\sin^{-1}(y)}{\sqrt{1-y^2}} \end{aligned}$$

The alert reader will object that this procedure seems backwards — usually we have some PDF that we want to sample from, not some given transformation. Typically we have X drawn from some $p_x(x)$, and we want Y from some $p_y(y)$. What transformation should we use?

All we need is for the CDF's to be equal, or $P_y(Y) = P_x(X)$. This immediately gives the transformtion:

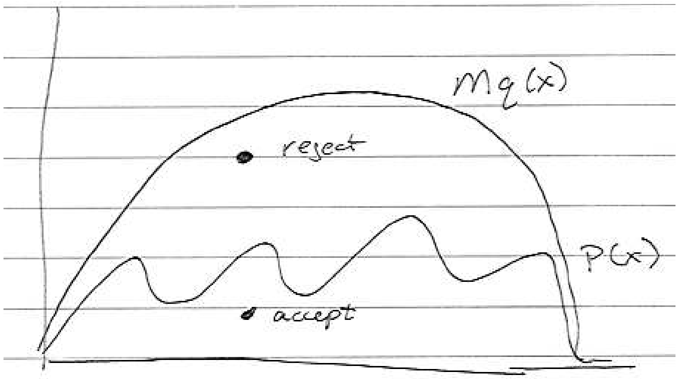


Figure 14.4: Rejection sampling is cool

$$y(x) = P_y^{-1}(P_x(x))$$

This is clearly a generalization of the inversion method, since if X were uniformly distributed, $P_x(x) = x$ and we have the same procedure from the last section.

14.5 The Rejection Method

Before we go too far down this path, let's first look at a simple method for drawing samples from any distribution, in any dimension. The *rejection method* is essentially dart throwing. Assume that we want to draw samples from $p(x)$, but we don't know how to integrate $p(x)$, or we don't know how to invert $P(x)$. Of course, these could always be done numerically, but that is expensive and error-prone.

All is not lost, however. Suppose we have another PDF $q(x)$ that satisfies $p(x) < Mq(x)$ for some M, and we know how to sample from q. The rejection method is then quite simple:

```
loop forever:
    X = q->sample()
    U = uniform->sample()
    if U < p(x) / (Mq(x)) then
         return X
```

What this procedure is doing is choosing a pair of variables (X, ξ). If the point $(X, \xi Mq(X))$ lies under $p(X)$, then the sample X is accepted. Otherwise it is rejected and a new sample pair is chosen. This is illustrated in figure 14.4.

Without going into the math in too much detail, it should be clear that the efficiency of this scheme depends on how tightly $Mq(x)$ bounds $p(x)$. Note that this technique works in any number of dimensions!

14.5.1 Example: Rejection Sampling a Unit Circle

Suppose we want to select a uniformly distributed point inside a unit circle. Using the rejection method, we simply select a random (X, Y) inside the circumscribed square, and return it if it falls inside the circle. This is shown in figure 14.5.

Figure 14.5: Rejection sampling a circle

The function `RejectionSampleDisk()` implements this algorithm. Given two uniform random variables `u1` and `u2`, it returns a point $(\mathtt{x}, \mathtt{y})$ inside the unit circle.

⟨*MC Function Definitions*⟩≡

```
void RejectionSampleDisk( Float u1, Float u2, Float *x, Float *y ) {
    Float sample_x, sample_y;
    do {
        sample_x = 1-2*u1;
        sample_y = 1-2*u2;
    } while (sample_x*sample_x + sample_y*sample_y > 1 );
    *x = sample_x;
    *y = sample_y;
}
```

A similar technique will work for any complex shape as long as it has an inside-outside test.

The efficiency of this technique depends on the percentage of samples we expect to be rejected. In the 2D case, this is easy to compute; it is the area of the circle divided by the area of the square: $\frac{\pi}{4} \approx 78.5\%$. In higher dimensions, the story changes. It is a little known and perhaps counterintuitive fact that the volume of an n-dimensional hypersphere goes to *zero* as n increases. The volume of a unit n-hypersphere is given by (see (Weisstein 1999) for an introduction, and (Wells 1987) for a derivation):

$$V_n = \frac{2\pi^{\frac{n}{2}}}{n\Gamma\left(\frac{n}{2}\right)}$$

Since we are only interested in integral dimensions, we can simplify the gamma function in the above expression:

$$V_n = \begin{cases} \frac{2^{\frac{n+1}{2}}\pi^{\frac{n-1}{2}}}{(n)!!} & \text{n odd}^5 \\ \frac{2\pi^{\frac{n}{2}}}{n(\frac{n}{2}-1)!} & \text{n even} \end{cases}$$

Interestingly, this function peaks at $n = 5$, and then descends back down to zero as n goes to infinity. This tells us that rejection sampling is unlikely to be a good idea in higher dimensions.

A concept that needs clarification at this point is the notion of "uniform" sampling in multiple dimensions. Whenever this is desired, the question must always be asked "Uniform with respect to what?". In the case of the circle, the answer is "uniform with repect to *area*". This means that if we pick any neighborhood inside the circle, the likelihood of choosing a point inside that neighboorhood is exactly equal to the area of the neighboorhood divided by the area of the circle. This is illustrated in figure 14.6.

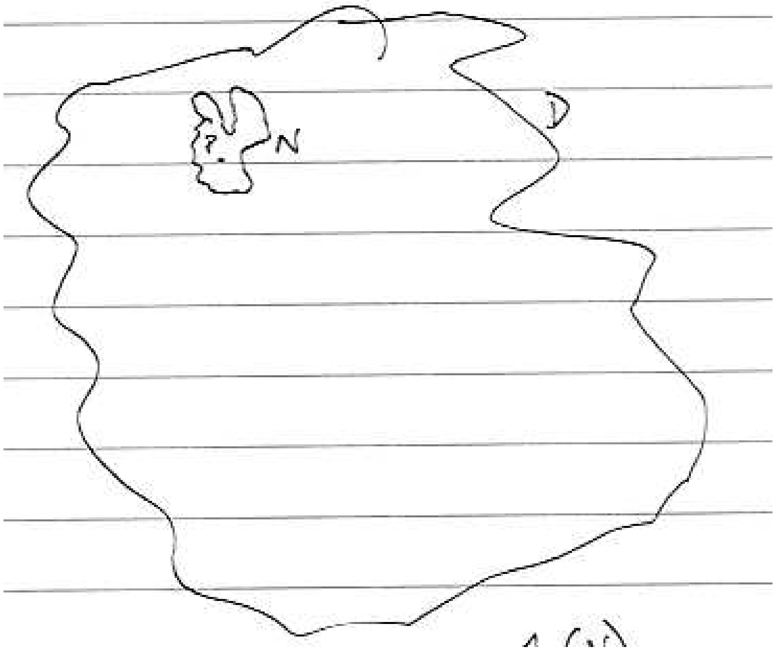


Figure 14.6: Uniform area sampling. $Pr\{P \in N\} = \frac{A(N)}{A(D)}$

14.6 Transformation in Multiple Dimensions

Rejection sampling not only breaks down in higher dimensions, but even in low dimensions there is still some wasted effort. We have already seen the transformation method in one dimension; let's see how multi-dimensional PDF's are affected by transformation.

Suppose we have an n-D random variable $X = (X_1, X_2, \ldots, X_n)$ over some domain $\Omega_x \subset \mathbb{R}^n$ with density function $p_x(x)$. Now let $Y = T(X)$, where $T : \Omega_x \to \Omega_y$ is a bijection. What is the density function of Y?

The argument is a straightforward generalization of the one-dimensional case. Since T is a bijection, we know that

$$Pr\{Y \in T(D)\} = Pr\{X \in D\}$$

for any domain $D \subseteq \Omega_x$.

We can write this in terms of the density functions p_x and p_y:

$$\int_{T(D)} p_y(y)dy = \int_D p_x(x)dx$$

Now, we apply a change of variables to the left hand side of this equation. Since this is a multi-dimensional integral, this requires multiplying by the determinant of the Jacobian matrix J_T:

$$\int_D p_y(T(x))|J_T(x)|dx = \int_D p_x(x)dx$$

But this equation holds for *any* domain D. The only way that this can be true is if the integrands themselves are equal. Therefore,

$$p_y(T(x))|J_T(x)| = p_x(x)$$

or

$$p_y(T(x)) = \frac{p_x(x)}{|J_T(x)|}$$

14.6.1 Example: Polar Coordinates

The polar transformation is given by:

$$\begin{aligned} x &= r\cos\theta \\ y &= r\sin\theta \end{aligned}$$

Suppose we draw samples from some density $p(r,\theta)$. What is the corresponding density $p(x,y)$? The Jacobian of this transformation is:

$$J_T = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial \theta} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial \theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & -r\sin\theta \\ sin\theta & r\cos\theta \end{bmatrix}$$

And the determinant is $r\left(\cos^2\theta + \sin^2\theta\right) = r$. So $p(x,y) = \frac{p(r,\theta)}{r}$. Of course, this is backwards from what we usually want – typically we start with a sampling strategy in Cartesian coordinates and want to transform it to one in polar coordinates. In that case, we would have:

$$p(r,\theta) = r\,p(x,y)$$

14.6.2 Example: Spherical Coordinates

$$\begin{aligned} x &= r\sin\theta\cos\phi \\ y &= r\sin\theta\sin\phi \\ z &= r\cos\theta \end{aligned}$$

The Jacobian of this transformation has determinant $|J_T| = r^2\sin\theta$, so the corresponding density function is:

$$p(r,\theta,\phi) = r^2\sin\theta\, p(x,y,z)$$

The reason this transformation is so important is that we can represent a direction as a point (x,y,z) on the unit sphere. Remember that solid angle is defined as the area of a set of points on the unit sphere. In spherical coordinates, we derived:

$$d\omega = \sin\theta\, d\theta\, d\phi$$

So if we have a density function defined over solid angle, this means that

$$Pr\{\Omega \in D\} = \int_D p(\omega)\, d\omega$$

The density with respect to θ and ϕ can therefore be derived:

$$\begin{aligned} p(\theta,\phi)\, d\theta\, d\phi &= p(\omega)\, d\omega \\ p(\theta,\phi) &= \sin\theta\, p(\omega) \end{aligned}$$

14.7 2D Sampling with Multi-Dimensional Transformation

Suppose we have a 2D joint density function $p(x,y)$ that we wish to draw samples (X,Y) from. To do this we first introduce some basic terminology from conditional probability:

The *marginal density function* $p(x)$ is obtained by simply "integrating out" one of the dimensions:

$$p(x) = \int p(x,y)dy$$

This can be thought of as the density function for X alone.

The *conditional density function* $p(y|x)$ is the density function for Y once we have chosen some particlar X (it is read "p of y given x"):

$$p(y|x) = \frac{p(x,y)}{p(x)}$$

The basic idea for 2D sampling is to first compute the marginal density to isolate one particular variable, and draw a sample from that density using standard 1D inversion techniques. Once that sample is drawn, compute the conditional density function, and draw a sample from that, again using standard 1D techniques.

14.7.1 Example: Uniformly Sampling a Hemisphere

In this example, we want to choose a direction on the hemisphere uniformly with respect to solid angle. Remember, a uniform distribution means that the density function is a constant! So we know that $p(\omega) = c$. In addition, remember that the density function must integrate to one over its domain. Therefore, we have:

$$\int_{H^2} p(\omega)\, d\omega = 1 \Rightarrow c \int_{H^2} d\omega = 1 \Rightarrow c = \frac{1}{2\pi}$$

This tells us that $p(\omega) = \frac{1}{2\pi}$, or $p(\theta,\phi) = \frac{\sin\theta}{2\pi}$ (see the above example on spherical coordinates).

Let's sample θ first. To do this, we need θ's marginal density function:

$$p(\theta) = \int_0^{2\pi} p(\theta,\phi)\, d\phi = \int_0^{2\pi} \frac{\sin\theta}{2\pi} d\phi = \sin\theta$$

Now, we compute the conditional density for ϕ:

$$p(\phi|\theta) = \frac{p(\theta,\phi)}{p(\theta)} = \frac{1}{2\pi}$$

Notice that the density function for ϕ is itself uniform — this should make intuitive sense given the symmetry of the hemisphere. Now, we use the 1D inversion technique to sample each of these PDF's in turn, making sure that the CDF's properly evaluate to 1 at the maximal domain value:

$$\begin{aligned} P(\theta) &= \int \sin\theta d\theta = 1 - cos\theta \\ P(\phi|\theta) &= \int \frac{1}{2\pi} d\phi = \frac{\phi}{2\pi} \end{aligned}$$

Inverting these functions is straightforward, and again noting that we can safely replace $1-\xi$ with ξ since these are both uniformly distributed random numbers over $[0,1]$, we get:

$$\begin{aligned} \theta &= \cos^{-1}(\xi_1) \\ \phi &= 2\pi\xi_2 \end{aligned}$$

Converting these back to Cartesian coordinates, we get the final sampling formulae:

$$\begin{aligned} x &= \sin\theta\cos\phi = \cos(2\pi\xi_2)\sqrt{1-\xi_1^2} \\ y &= \sin\theta\sin\phi = \sin(2\pi\xi_2)\sqrt{1-\xi_1^2} \\ z &= \cos\theta = \xi_1 \end{aligned}$$

This sampling strategy is implemented below. Two uniform random numbers `u1` and `u2` are passed in, and a vector on the hemisphere is returned.

INV_TWOPI	678
M_PI	678
Vector	27

⟨*MC Function Definitions*⟩+≡

```
Vector UniformSampleHemisphere(Float u1, Float u2) {
    Float z = u1;
    Float r = sqrtf(max(0.f, 1.f - z*z));
    Float phi = 2 * M_PI * u2;
    Float x = r * cosf(phi);
    Float y = r * sinf(phi);
    return Vector(x, y, z);
}
```

For each sampling routine we present, there is a corresponding `Weight` function that simply evaluates the PDF. Of course, we need to be careful about which PDF to evaluate — we've already seen that we can transform PDF's around any way we want, and the resulting functions are different! Unless otherwise specified, we will assume that the PDF's are with respect to solid angle. For the hemisphere, the solid angle PDF is a constant $p(\omega) = \frac{1}{2\pi}$.

⟨*MC Function Definitions*⟩+≡

```
Float UniformHemispherePdf(Float theta, Float phi) {
    return INV_TWOPI;
}
```

Sampling the full sphere uniformly over its area is almost exactly the same, and we omit the derivation.

$$\begin{aligned} x &= 2r\sqrt{\xi_1(1-\xi_1)}\cos\phi \\ y &= 2r\sqrt{\xi_1(1-\xi_1)}\sin\phi \\ z &= 1-2\xi_1 \end{aligned}$$

⟨*MC Function Definitions*⟩+≡

```
Vector UniformSampleSphere(Float u1, Float u2) {
    Float z = 1.f - 2.f * u1;
    Float r = sqrtf(max(0.f, 1.f - z*z));
    Float phi = 2.f * M_PI * u2;
    Float x = r * cosf(phi);
    Float y = r * sinf(phi);
    return Vector(x, y, z);
}
```

⟨*MC Function Definitions*⟩+≡

```
Float UniformSpherePdf() {
    return 1.f / (4.f * M_PI);
}
```

14.7.2 Example: Sampling a Unit Disk

Although the disk example seems simpler than the hemisphere, it actually gives many students the most trouble, because it has an (incorrect) intuitive solution. Most students, when asked to choose a random point inside a unit circle, simply give the (wrong) answer:

```
678 M_PI
 27 Vector
```

$$\begin{aligned} r &= \xi_1 \\ \theta &= 2\pi\xi_2 \end{aligned}$$

Although this point is certainly random, and it is inside the circle, it is *not* uniformly distributed; it actually clumps samples near the center of the circle. To see why, let's derive the correct transformation from first principles.

Since we're going to sample uniformly with respect to area, the PDF $p(x,y)$ must be a constant. By normalization, we get $p(x,y) = \frac{1}{\pi}$. If we transform into polar coordinates (see the polar example above), we get $p(r,\theta) = \frac{r}{\pi}$. Now we compute the marginal and conditional densities as before:

$$\begin{aligned} p(r) &= \int_0^{2\pi} p(r,\theta)\, d\theta \\ &= \int_0^{2\pi} \frac{r}{\pi}\, d\theta \\ &= 2\pi\frac{r}{\pi} \\ &= 2r \end{aligned}$$

and

$$\begin{aligned} p(\theta|r) &= \frac{p(r,\theta)}{p(r)} \\ &= \frac{1}{2\pi} \end{aligned}$$

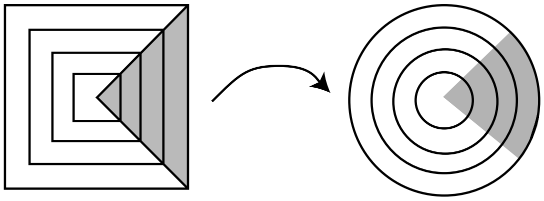


Figure 14.7: The concentric mapping maps squares to circles, giving a less distorted mapping than the first method shown for uniformly sampling points on the unit disk.

NEED A FIGURE FOR THE SIMPLE $r = \sqrt{\xi_1}$ mapping.

Figure 14.8: Please! FIGURE!

As with the hemisphere case, this should make sense because of the symmetry of the circle.

Now we simply integrate and invert:

M_PI 678

$$\begin{aligned} P(r) &= r^2 \\ P^{-1}(r) &= \sqrt{r} \\ P(\theta) &= \frac{\theta}{2\pi} \\ P^{-1}(\theta) &= 2\pi\theta \end{aligned}$$

So sampling a disk can be done thusly:

$$\begin{aligned} r &= \sqrt{\xi_1} \\ \theta &= 2\pi\xi_2 \end{aligned}$$

Notice how taking the square root of ξ_1 tends to spread the samples back towards the edge of the disk, negating the clumping referred to earlier.

⟨*MC Function Definitions*⟩+≡

```
void UniformSampleDisk(Float u1, Float u2, Float *x, Float *y) {
    Float r = sqrtf(u1);
    Float theta = 2.0f * M_PI * u2;
    *x = r * cosf(theta);
    *y = r * sinf(theta);
}
```

Though this mapping solves the problem at hand, it has distortion issues; areas on the unit square are elongated and/or compressed when mapped to the disk. This is unfortunate when we have well-stratified samples on the unit square, since that stratification is lost when we transform to the circle. Peter Shirley has developed a "concentric" mapping from the unit square to the unit circle that avoids this problem. The concentric mapping takes points in the square $[-1,1]^2$ to the unit disk by uniformly mapping concentric squares to concentric circles–see Figure 14.7.

The mapping turns wedges of the square into slices of the disk. For example,

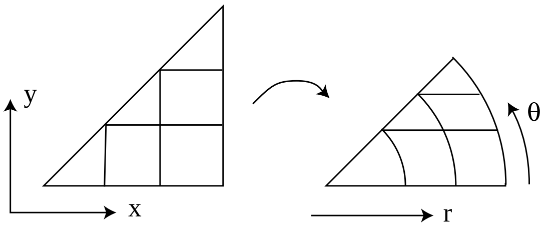


Figure 14.9: Triangular wedges of the square are mapped into (r, θ) pairs in pie-shaped slices of the circle.

points in the shaded area of the square in Figure 14.7 are mapped to (r, θ) by

$$\begin{aligned} r &= x \\ \theta &= \frac{y}{x} \end{aligned}$$

See Figure 14.9. The other four quadrants are handled analogously.

⟨*MC Function Definitions*⟩+≡

```
void ConcentricSampleDisk(Float u1, Float u2,
        Float *dx, Float *dy) {
    Float r, theta;
    ⟨Map uniform random numbers to [−1,1]^2⟩
    ⟨Map square to (r,θ)⟩
    *dx = r*cosf(theta);
    *dy = r*sinf(theta);
}
```

⟨*Map uniform random numbers to* $[-1,1]^2$⟩≡

```
Float sx = 2 * u1 - 1;
Float sy = 2 * u2 - 1;
```

⟨*Map square to* (r,θ)⟩≡

```
⟨Handle degeneracy at the origin⟩
if (sx >= -sy) {
    if (sx > sy) {
        ⟨Handle first region⟩
    }
    else {
        ⟨Handle second region⟩
    }
}
else {
    if (sx <= sy) {
        ⟨Handle third region⟩
    }
    else {
        ⟨Handle fourth region⟩
    }
}
theta *= M_PI / 4.f;
```

⟨*Handle first region*⟩≡

```
r = sx;
if (sy > 0.0)
    theta = sy/r;
else
    theta = 8.0f + sy/r;
```

The remaining cases are analogous and are omitted.

14.7.3 Example: Cosine Weighted Hemisphere Sampling

As we will see later in this chapter, it is often useful to sample from a distribution that is similar to the integrand being estimated. Because we know that the integral in Equation 15.3.1 weights the result by a cosine term, we will generate directions that are more likely to be close to the top of the hemisphere than the bottom, where the cosine term has a small value.

Mathematically, this means that

$$p(\omega) \propto \cos\theta$$

M_PI 678

Converting this to spherical coordinates, we get

$$p(\theta,\phi) \propto \sin\theta\cos\theta$$

Normalizing as usual, we get:

$$\begin{aligned}
\int_{H^2} cp(\theta,\phi)\, d\theta d\phi &= 1 \\
\int_0^{2\pi}\int_0^{\frac{\pi}{2}} c\cos\theta\sin\theta d\theta d\phi &= 1 \\
c2\pi\int_0^1 u du &= 1 \\
c &= \frac{1}{\pi}
\end{aligned}$$

so

$$p(\theta,\phi) = \frac{1}{\pi}\cos\theta\sin\theta$$

We could compute the marginal and conditional densities as before, but instead we’re going to use a trick knows as *Malley’s method* to generate these cosine-weighted points. The idea behind Malley’s method is that if we choose points uniformly from the unit disk and then generate directions by projecting the points on the disk up to the hemisphere above it, the resulting distribution of directions will be a cosine distribution—see Figure 14.10.

Why does this work? Let (r,ϕ) be the polar coordinates of the point chosen on the disk (note that we’re using ϕ instead of the usual θ here). From our calculations before, we know that the joint density $p(r,\phi) = \frac{r}{\pi}$ represents the point sampled on the disk.

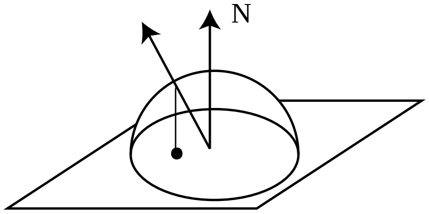


Figure 14.10: Malley's method: to sample direction vectors from a cosine-weighted distribution, uniformly sample points on the unit disk and project them up to the unit sphere.

Now, we map this to the hemisphere. The vertical projection gives $\sin\theta = r$, which is easily seen from the diagram in Figure 14.10. To complete the $(r,\phi) \rightarrow (\sin\theta,\phi)$ transformation, we need the determinant of the Jacobian:

$$|J_T| = \begin{vmatrix} \cos\theta & 0 \\ 0 & 1 \end{vmatrix} = \cos\theta$$

Therefore, $p(\theta,\phi) = |J_T|p(r,\phi) = \cos\theta\frac{r}{\pi} = \frac{1}{\pi}\cos\theta\sin\theta$, which is exactly what we wanted! We have used the transformation method to *prove* that Malley's method generates directions with a cosine-weighted distribution.

Note that this technique works regardless of the method used to sample points from the circle, so we can use Shirley's concentric mapping just as well as the simpler $(r,\theta) = (\sqrt{\xi_1}, 2\pi\xi_2)$ method.

515	ConcentricSampleDisk()
678	INV_PI
27	Vector

⟨*MC Utility Declarations*⟩+≡

```
inline Vector CosineSampleHemisphere(Float u1, Float u2) {
    Vector ret;
    ConcentricSampleDisk(u1, u2, &ret.x, &ret.y);
    ret.z = sqrtf(max(0.f, 1.f - ret.x*ret.x - ret.y*ret.y));
    return ret;
}
```

Remember that all of our weighting functions are with respect to solid angle, not spherical coordinates, so we simply return a weight of $\frac{\cos\theta}{\pi}$.

⟨*MC Utility Declarations*⟩+≡

```
inline Float CosineHemispherePdf(Float theta, Float phi) {
    return cosf(theta) * INV_PI;
}
```

14.7.4 Example: Sampling a triangle

Our final example will show how to uniformly sample a trianlge. Although this might seem like a simple task, it turns out to be more complex than the ones we've seen so far. To simplify the problem, we will assume we are sampling an isoceles right triangle of area $\frac{1}{2}$. Because the output of our sampling routine will be barycentric coordinates, our technique works for any triangle despite this simplificiation. Figure 14.7.4 shows the shape to be sampled.

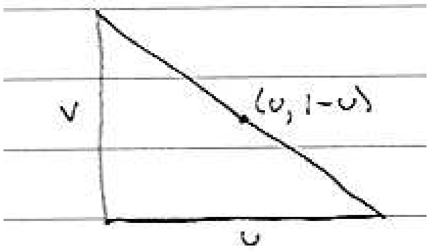


Figure 14.11: Sampling an isoceles right triangle. Note that the equation of the hypotenuse is $v = 1 - u$.

Since we are sampling with respect to area, we know that the PDF $p(x,y)$ must be a constant equal to the reciprocal of the shape's area (work it out yourself if you're not sure — $\int_\Delta c\,dx\,dy = 1$), so $p(u,v) = 2$.

First, we compute the marginal density $p(u)$:

$$\begin{aligned} p(u) &= \int_0^{1-u} p(u,v)\,dv \\ &= 2\int_0^{1-u} dv \\ &= 2(1-u) \end{aligned}$$

and the conditionaly density $p(v|u)$:

$$\begin{aligned} p(v|u) &= \frac{p(u,v)}{p(u)} \\ &= \frac{2}{2(1-u)} \\ &= \frac{1}{1-u} \end{aligned}$$

The CDF's are, as always, gotten by integration:

$$\begin{aligned} P(u) &= \int p(u)\,du = 2u - u^2 \\ P(v) &= \int p(v|u)\,dv = \frac{v}{1-u} \end{aligned}$$

And now, we invert these functions:

$$\begin{aligned} P(u) &= \xi_1 \\ 2u - u^2 &= \xi_1 \\ u^2 - 2u + \xi_1 &= 0 \\ u &= 1 \pm \sqrt{1-\xi_1} \\ &= 1 \pm \sqrt{\xi_1} && \text{remember, if } \xi \text{ is uniform, so is } 1-\xi \\ &= 1 - \sqrt{\xi_1} && \text{because } u \in [0,1] \end{aligned}$$

We do the same for $P(v)$:

$$\begin{aligned}
P(v) &= \xi_2 \\
\frac{v}{1-u} &= \xi_2 \\
v &= \xi_2(1-u) \\
&= \xi_2(1-(1-\sqrt{\xi_1})) \\
&= \xi_2\sqrt{\xi_1}
\end{aligned}$$

So our final strategy is:

$$\begin{aligned}
u &= 1-\sqrt{\xi_1} \\
v &= \xi_2\sqrt{\xi_1}
\end{aligned}$$

Notice that the two variables in this case are *not* independent!

⟨*MC Function Definitions*⟩+≡

```
void UniformSampleTriangle(Float u1, Float u2, Float *u, Float *v) {
    *u = 1 - sqrtf(u1);
    *v = u2 * sqrtf(u1);
}
```

We won’t provide a weight function for this sampling strategy since the proper weight will depend on the triangle’s area.

15. Monte Carlo Integration II: Variance Reduction

Before we go further into techniques for sampling from various distributions, we should first explain why we are doing this at all! Equation 14.2 tells us that we can use *any* distribution, so what makes one distribution more appropriate than another?

It turns out that it is advantageous to choose a sampling distribution that is "similar" to the integrand. This technique is called *importance sampling*. In Monte Carlo sampling, "advantageous" usually means "leads to reduced variance". Importance sampling is not the only technique for reducing variance; the use of expected values, jittering, stratification, and control variates are all examples of techniques designed to reduce variance. We will cover some of these techniques in this chapter; see Veach for a more complete treatment **??**.

The battle against variance is the basis of most of the work in optimizing Monte Carlo. Variance in Monte Carlo ray tracing manifests itself as noise in the image. As we showed earlier, Monte Carlo's convergence rate makes it is necessary to quadruple the number of samples in order to reduce the variance by half. Of course, this increases the runtime of the estimation procedure proportionally to the number of samples.

The *efficiency* of an estimator F is defined as:

$$\varepsilon[F] = \frac{1}{V[F]T[F]}$$

where $V[F]$ is the variance, and $T[F]$ is the running time. According to this metric, F_1 is more efficient than F_2 if it takes less time to produce the same variance, or if

Figure 15.1: The function $h(x)$, bounded above by the constant m

.

it produces less variance in the same amount of time.

15.1 Analytic Integration Techniques

The techniques in this section all involve analytically integrating a function that is similar to the integrand. These techniques turn out to be some of the most powerful for computer graphics, but unfortunately they are not always applicable.

15.1.1 The use of expected values

One of the most obvious techniques for reducing variance is to reduce the dimensionality of the problem by integrating out one of the dimensions. For example, if we start with the estimator

$$F = \frac{f(X,Y)}{p(X,Y)}$$

we can replace it with $F^* = \frac{f^*(X)}{p(X)}$, where

$$\begin{aligned} f^*(x) &= \int f(x,y)\,dy \\ p(x) &= \int p(x,y)\,dy \end{aligned}$$

In order to do this, we need to be able to integrate both f and p with respect to y. We also need to be able to draw samples from the marginal density $p(x)$, but we can always do that by simply drawing samples from $p(x,y)$ (which we assume we could already do because of the existence of the estimator F) and throwing away the y coordinate.

As an example of this technique, assume we want to compute the integral $I = \int_a^b h(x)\,dx$, and we know that $h(x) \leq m \forall x \in [a,b]$, as shown in figure 15.1.

The simplest technique for estimating this integral is just to throw darts at the rectangle and count the number that fall under the curve. Formally,

$$f(x,y) = \begin{cases} 1 & 0 \leq y \leq h(x) \\ 0 & \text{otherwise} \end{cases}$$

In this case, $p(x,y) = \frac{1}{m(b-a)}$, since our darts are uniformly distributed with respect to area. Now we estimate $I = \int f(x,y)\,dx\,dy$ with N points:

$$\hat{I} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i,y_i)}{p(x_i,y_i)} = \frac{m(b-a)}{N} \sum_{i=1}^{N} f(x_i,y_i)$$

Now, we apply the expected values technique described above:

$$\begin{aligned} f^*(x) &= \int f(x,y)\,dy = \int_0^{h(x)} dy = h(x) \\ p(x) &= \int_0^m p(x,y)\,dy = \frac{1}{b-a} \end{aligned}$$

And our new estimator is:

$$\hat{I}^* = \frac{1}{N}\sum_{i=1} N\frac{f^*(x_i)}{p(x_i)} = \frac{b-a}{N}\sum_{i=1} N f(x_i)$$

But this is the standard Monte Carlo estimator from the previous chapter! We have reduced rejection sampling to Monte Carlo estimation through the use of expected values.

This technique is called "the use of expected values" because F^* is the conditional expected value of F. This is defined as:

$$\begin{aligned} E_Y[G] &= \int g(x,y)\,p(y|x)\,dy \\ &= \frac{\int g(x,y)\,p(x,y)\,dy}{\int p(x,y)\,dy} \end{aligned}$$

It is easy to see that F^* meets this definition:

$$\begin{aligned} F^* &= E_Y\left[\frac{f(x,y)}{p(x,y)}\right] \\ &= \int \frac{f(x,y)}{p(x,y)}p(y|x)\,dy \\ &= \int \frac{f(x,y)}{p(x,y)}\frac{p(x,y)}{\int p(x,y')\,dy'}\,dy \\ &= \frac{f(x)}{p(x)} \end{aligned}$$

This fact makes it easy to analyze the effect this technique has on variance. To do so, we will make use of a simple theorem that separates multi-dimensional variance:

THEOREM: $V[G] = E_xV_yG + V_xE_yG$

PROOF:

$$\begin{aligned} E_xV_yG + V_xE_yG &= E_x\left\{E_y\left[G^2\right] - \left[E_YG\right]^2\right\} + E_x\left[E_yG\right]^2 - \left\{E_xE_yG\right\}^2 \\ &= E_xE_y\left[G^2\right] - \left[E_xE_yG\right]^2 \\ &= V[G] \end{aligned}$$

therefore, we can write $V[F] = E_xV_yF + V_xE_yF$. but $E_yF = F^*$, so we have $V[F] = E_xV_yF + V[F^*]$, or

$$V[F] - V[F^*] = E_xV_yF$$

Notice that the right hand term of this equation is necessarily non-negative because the variance of a random variable is always non-negative. This proves that the variance of F^* cannot be more than the variance of F. Although this fact is true in general, our choice of example also serves as a proof that rejection sampling is less efficient than standard Monte Carlo sampling, which should be intuitively clear.

15.1.2 Importance Sampling

Recall that we do not always have to choose our samples uniformly; we can use any $p(x)$ that we like. Importance sampling is based on the observation that th estimator will converge more quickly if the samples are taken from a distribution $p(x)$ that is similar to the function $f(x)$ in the integrand. The basic idea is that by concentrating work where the value of the integrand is relatively high, the estimate is generated more efficiently.

We will motivate this technique informally. Suppose we're trying to use Monte Carlo techniques to evaluate some integral $I = \int f(x)dx$. Since we can choose any sampling distribution we want, let's choose $p(x) \propto f(x)$, or $p(x) = cf(x)$. As always, we must normalize this distribution function. It is trivial to show that normalization forces $c = \frac{1}{\int f(x)dx} = \frac{1}{I}$. But this requires us to know the value of I, which is what we were trying to estimate in the first place!

Nonetheless, if we *could* sample from this distribution, each estimate would be $Y_i = \frac{f(X_i)}{p(X_i)} = \frac{1}{c}$. Since c is a constant, each estimate has the exact same value, and the variance is zero! Of course, this is ludicrous since we wouldn't bother using Monte Carlo if we could integrate f directly. However, it should be clear that the closer $p(x)$ is to a multiple of $f(x)$, the lower the variance will be.

In many cases, the integrand is the product of more than one function. It is frequently inconvenient to construct a PDF that is similar to the product, but even building one that is similar to one of the multiplicands can be very helpful. This will be a common strategy in the next chapter, where we will be trying to estimate integrals that multiply lighting, visibility, BRDF's, and cosine terms.

In practice, importance sampling is one of the most frequently used variance reduction techniques, since it can easily be applied to very complex functions. It will be the variance reduction technique of choice in `lrt`, and a variety of techniques for sampling from distributions that are similar to common functions in graphics are presented later in this chapter.

15.1.3 Multiple Importance Sampling

The idea of combining samples from two different distributions to estimate the integral of a product of functions was first introduced to graphics by Eric Veach and Leo Guibas; many of the facts presented in this section are proved and carefully analyzed in their paper (Veach and Guibas 1995) and in Veach's doctoral dissertaion (Veach 1997).

Monte Carlo gives us tools to estimate an integral of the form $\int f(x)\,dx$. However, we are frequently faced with integrals that are the product of two or more functions: $\int f(x)g(x)\,dx$. If we have an importance sampling strategy for $f(x)$ *and* a strategy for $g(x)$, which should we use? In general it is very difficult to directly combine these strategies to compute a probability density function that is proportional to the product $f(x)g(x)$. Rather than choosing between the two sampling strategies, multiple importance sampling gives us the tools needed to use *both* sampling strategies separately and combine their answers in an intelligent way.

The idea behind multiple importance sampling is that when estimating an integral of the form $\int f(x)g(x)\,dx$ we should draw samples from both distributions.

Then, each sample is instead weighted by

$$\frac{1}{N}\left(\sum_{n_f}\frac{f(x_i)g(x_i)\hat{w}_f(x_i)}{p_f(x_i)}+\sum_{n_g}\frac{f(x_i)g(x_i)\hat{w}_g(x_i)}{p_g(x_i)}\right),$$

where n_f is the number of samples taken from f's importance sampling method, n_g is the number of samples taken from g's, $N = n_f + n_g$, and $\hat{w}_f$ and $\hat{w}_g$ are special weighting functions that take into account *all* of the different ways that a sample x_i could have been generated, rather than just the particular one that was used.

A good choice for this weighting function is the *balance heuristic*.

$$\hat{w}_s(x) = \frac{n_s p_s(x)}{\sum_i n_i p_i(x)}$$

The balance heuristic is a provably good way to weight samples to reduce variance. We provide an implementation of this function for the specific case of two functions f and g; we will not need a more general case in `lrt`.

⟨*MC Inline Functions*⟩≡

```
inline Float BalanceHeuristic(int nf, Float fPdf, int ng, Float gPdf) {
    return (nf * fPdf) / (nf * fPdf + ng * gPdf);
}
```

In practice, the *power heuristic* often reduces variance even further. For some exponent β, the power heuristic is:

$$\hat{w}_s(x) = \frac{(n_s p_s(x))^\beta}{\sum_i (n_i p_i(x))^\beta}$$

Veach determined empirically that $\beta = 2$ is a good value; we have $\beta = 2$ hard-coded into the implementation.

⟨*MC Inline Functions*⟩+≡

```
inline Float PowerHeuristic(int nf, Float fPdf, int ng, Float gPdf) {
    Float f = nf * fPdf, g = ng * gPdf;
    return (f*f) / (f*f + g*g);
}
```

Intuitively, multiple importance sampling reduces variance because it reduces the "surprise factor" that results when one of the sampling terms is large while the other is small. If our sampling strategy for f gives a large contribution while g is near-zero at the same point, we waste a lot of effort sampling at locations where the integrand is small.

Consider our primary objective of evaluating lighting integrals of the form

$$L_o(x,\omega_o) = \int L_i(x,\omega_i)\, f_r(x,\omega_i,\omega_o) \cos\theta \, d\omega_i$$

Further consider the example of glossy reflections of area light sources. Now imagine two separate cases: a large area light reflected in a near-perfect mirror, and a small area light reflected in a dull glossy reflector.

If we were to only perform regular importance sampling according to L_i or f_r, one of these two cases would perform very poorly. Consider the mirror BRDF case

GET ERIC'S PERMISSION

Figure 15.2: Reprint Eric's MIS example with the spheres

when we only importance sampling the lighting. This means that we will choose a point on the area light and evaluate the integral. But since the BRDF is almost a mirror, it will be very near zero at all reflection directions except the mirror direction. This means that almost all of the points we choose on the area light will have almost zero contribution, and we should expect the variance to be quite high. However, the small area light reflected in a dull mirror would look excellent, since the BRDF is large at all directions pointing towards the light source.

If, on the other hand, we sampled the BRDF, we would see exactly the opposite problem. The mirror BRDF would look excellent, since all of the sampled directions had high contribution. However, the dull BRDF will look terrible, since most of the sampled directions will miss the light source and contribute nothing.

With multiple importance sampling, we can use both sampling strategies and combine them effectively. We will first draw a sample from both the lighting and the BRDF. For the lighting sample, we determine the probability that the corresponding direction would have been chosen by the BRDF. Similarly, for the BRDF sample, we determine the probability that the corresponding point would have been chosen by the light (this is the reason for the extra `Weight` methods for both lights and reflection functions).

Now we can combine the two samples in the right way! If the light says "zero probability; the BRDF's direction misses me entirely", we use the light's sample (making the dull BRDF picture look good). If the BRDF says "negligible probability; the light's point is not in the mirror direction", then we use the BRDF sample (making the mirror BRDF picture look good). If the lighting says "fairly high probability" and the BRDF says "fairly low probability", then we weight the two samples accordingly.

Example renderings illustrating this intuitive argument are shown in figure 15.2.

15.2 Careful Sample Placement

In addition to simplifying the problem through analytic integration techniques, we can also reduce variance by carefully placing samples so as to capture "important" features of the integrand (or, more accurately, to be less likely to miss important features). These techniques are complimentary to analytic integration techniques, and are therefore also used extensively in `lrt`.

15.2.1 Stratified Sampling

Stratified sampling was introduced in section 7.4, and we now have the tools to formally motivate its use. Stratified sampling works by subdividing the domain Ω into n non-overlapping regions $\Omega_1, \Omega_2, \ldots, \Omega_n$. Each region is called a *stratum*, and they must completely cover the original domain; that is,

$$\cup_{i=1}^{n} \Omega_i = \Omega$$

To draw samples from Ω, we will draw n_i samples from each Ω_i, according to densities p_i. A simple example is super-sampling a pixel. With stratified sampling, we divide the pixel into a $k \times k$ grid, and draw a smaple from each grid cell. This is better than taking k^2 random samples, since the sample locations are less likely to clump.

Within a single stratum Ω_i, we have an estimate

$$F_i = \frac{1}{n_i} \sum_{i=1}^{n_i} f(X_{ij})$$

where X_{ij} is the j'th sample drawn from density p_i. Then our overall estimate becomes $F = \sum_{i=1}^{n} v_i F_i$, where v_i is the fractional volume of stratum i ($v_i \in (0,1]$).

The true average value of the integrand in stratum i is

$$\mu_i = E\left[f(X_{ij})\right] = \frac{1}{v_i} \int_{\Omega_i} f(x)\,dx$$

and the variance in this stratum is

$$\sigma_i^2 = \frac{1}{v_1} \int_{\Omega_i} (f(x) - \mu_i)^2\,dx$$

So, with n_i samples in the stratum, the variance of the per-stratum estimator is $\frac{\sigma_i^2}{n_i}$. This shows that the variance of the overall estimator is

$$\begin{aligned} V[F] &= V\left[\sum v_i F_i\right] \\ &= \sum V\left[v_i F_i\right] \\ &= \sum v_i^2 V\left[F_i\right] \\ &= \sum \frac{v_i^2 \sigma_i^2}{n_i} \end{aligned}$$

If we make the reasonable assumption that the number of samples n_i is proportional to the volume v_i, then we have $n_i = v_i N$, and the variance of the total estimator is

$$V\left[F_N\right] = \frac{1}{N} \sum v_i \sigma_i^2$$

How can we compare this to the variance without stratification? It turns out that this is simple with a little trick. Choosing an unstratified sample is equivalent to choosing a random *stratum I* according to the discrete probability distribution defined by the volumes v_i, and then choosing a random sample X in Ω_I. In this sense, X is chosen *conditionally* on I, and we can use the conditional expectation theorem from the last section:

$$\begin{aligned} V[F] &= E_x V_i F + V_x E_i F \\ &= \frac{1}{N}\left[\sum v_i \sigma_i^2 + \sum v_i (\mu_i - Q)\right] \end{aligned}$$

Where Q is the mean of f over the whole domain Ω. There are two things to notice about this formula. First, we know that the right hand sum must be non-negative, since variance is always non-negative. Second, this proves that stratified sampling can never reduce variance. In fact, stratification is always a win as long as the right hand sum is not exactly zero, which only happens when the function f has the same mean over each stratum Ω_i. In fact, we would like to maximize the right hand sum, so it is in our interest to make the strata have as unequal means as possible.

This explains why *compact* strata are desirable if you don't know anything about the function f. If the strata are wide, they will contain more variation, and will have μ_i closer to the true mean Q. This also explains why the Shirley square-to-circle mapping (Figure 14.7) is better than the straightforward mapping (Figure 14.8), since the straightforward mapping has less and less compact strata away from the center.

Stratified sampling has a downside, however; it suffers from the same "curse of dimensionality" as standard numeric quadrature. Full stratification in D dimensions with S strata per dimension requires S^D samples, which quickly becomes prohibitive.

15.2.2 Jittered Sampling

There are two main ways to characterize a random point-distribution process. The most straightforward is to compute the mean density per unit area generated by the process. More sophisticated analysis is possible using the *autocorrelation function*. This function tells us, given a point at (x,y), the likelihood of finding another point at $(x+u, y+v)$:

$$A(u,v) = E\left[\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} s(x,y)s(x+u,y+v)\,dx\,dy\right]$$

where s is the sampling function (e.g., a sum of δ-functions), and $E[\ldots]$ is the average over all such sampling functions.

We know from chapter 7 that sampling and reconstruction using a uniform pattern results in copies of the spectrum, which might overlap if the sampling rate is not sufficient. What can we say about random sampling? In order to answer this question, we would need to now the Fourier transform of $s(x,y)$. Unfortunately, this is undefined, as s is a random process.

Instead, we can define a related quantity called the *spectral density* of $s(x,y)$. The spectral density is the energy per frequency *per unit area*. It turns out that this is the Fourier transform of the autocorrelation function. Although this is pretty complicated, spectral density is actually very easy to approximate. We can simply form lots of instances of the sampling function $s(x,y)$, and average their Fourier spectra. A plot of the spectral density of a Poisson (e.g. purely random) process is shown in figure 15.3.

These spectra generally consist of a δ-function at the origin, surrounded by a sea of noise. Because the noise is present at all frequencies, it is "white" noise, and convolving with a Poisson distribution results in noisy, unstructured aliasing. A Poisson disk distribution, on the other hand, has a much nicer spectral density, shown in figure 15.4.

Notice that the Poisson disk distribution also has a δ-function surrounded by a sea of noise, but there is a gap between the spike and the noise. Because the

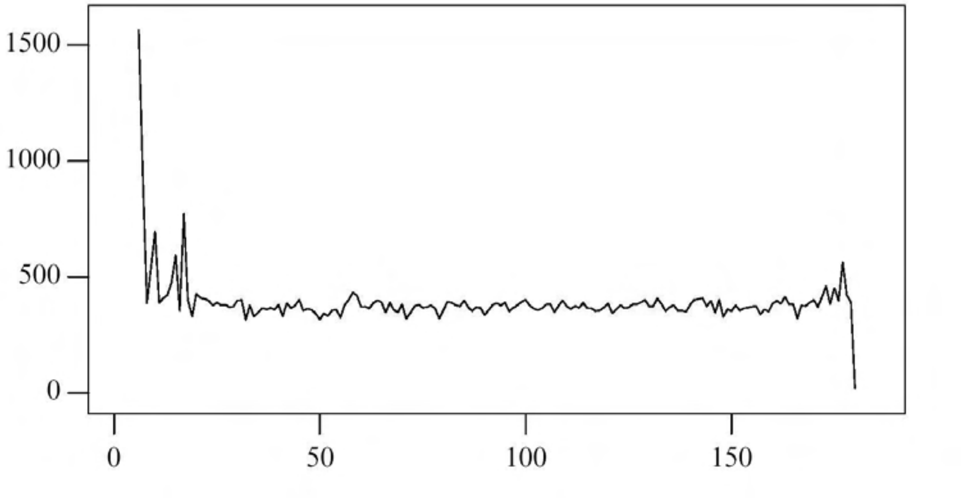


Figure 15.3: The spectral density of a Poisson process. Should either compute this ourselves or get Don’s permission to re-use these figures.

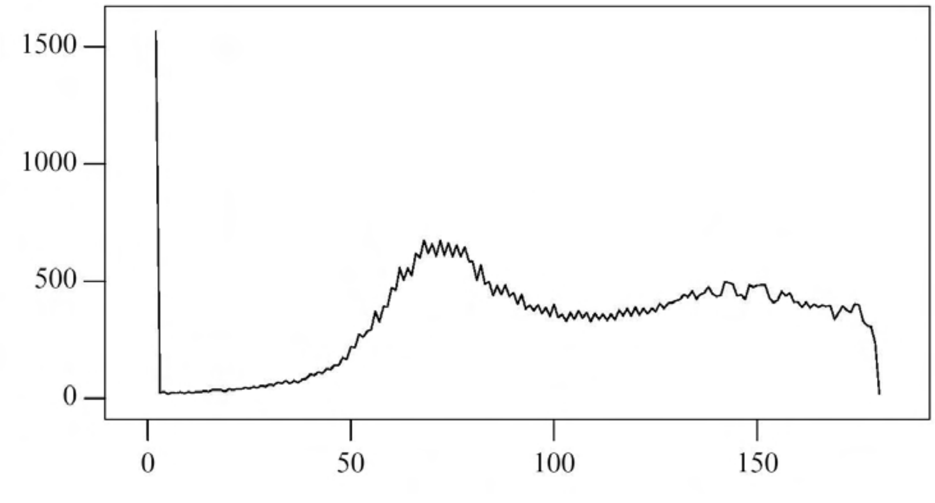


Figure 15.4: The spectral density of a Poisson disk process.

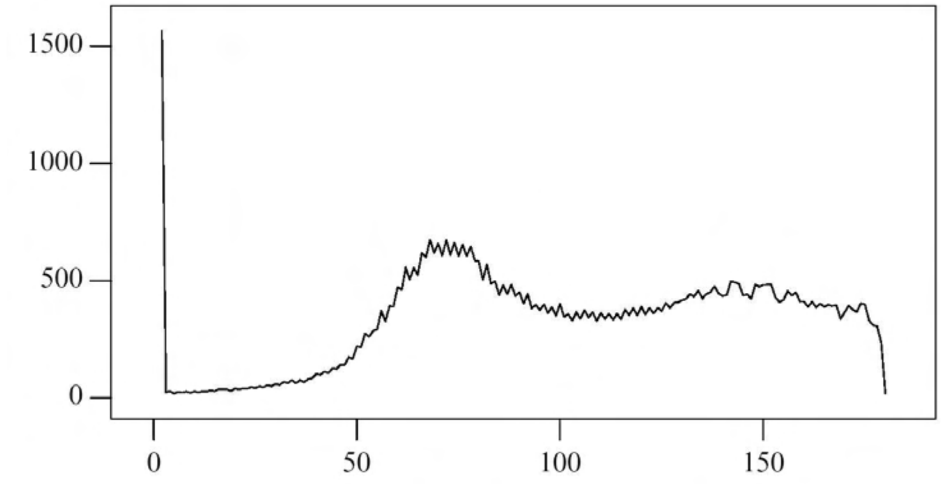


Figure 15.5: The Fourier transform a jittered point distribution.

noise is only present at high frequencies, this is “blue” noise. When using a blue-noise process to sample a function, the overlapping copies of the signal spectrum laid down during reconstruction are pushed out away from the origin. This gives the center copy of the spectrum some “breathing room”, which can help avoid or lessen aliasing. In fact, if the spectrum of the signal fit entirely in the gap between the δ-function and the noise, we could perfectly reconstruct the original signal! Even if we can’t, the noise will all be high-frequency, which is just what we want. It has been observed that the photoreceptors in a monkey eye are arranged in a Poisson-disk distribution (Yellot 1983).

What, then, can we say about jittered sampling? It is easy to compute the Fourier transform of a jittered sampling process, as shown in figure 15.5. The jitter spectrum is also a blue-noise spectrum, although there is more low-frequency noise present than in the Poisson disk case. Although this makes jittered patterns inferior to Poisson disk patterns for sampling, they are still substantially better than purely random processes, and they are very cheap to compute.

In addition, we can analytically determine the spectral density of a jittered distribution. We will do so in one dimension; the extension to multiple dimensions is straightforward. We write the jittered distribution as

$$s(x) = \sum_{n=-\infty}^{\infty} \delta\left(x - x_n\right)$$

where $x_n = nT + j_n$, T is the sampling frequency and j_n is the jitter amount. The random variable j_n is drawn from some probability distribution $j(x)$. The key insight is that $j(x)$ can be anything, but it’s just a function, so it has a Fourier transform $J(\omega)$.

It can be shown **(we should either find the derivation of this or cite something)** that the spectral density $S(\omega)$ of this jitter process is:

$$S(\omega) = \frac{1}{T}\left[1 - J^2(\omega)\right] + \frac{2\pi}{T^2}|J(\omega)|^2 \sum_{n=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi n}{T}\right)$$

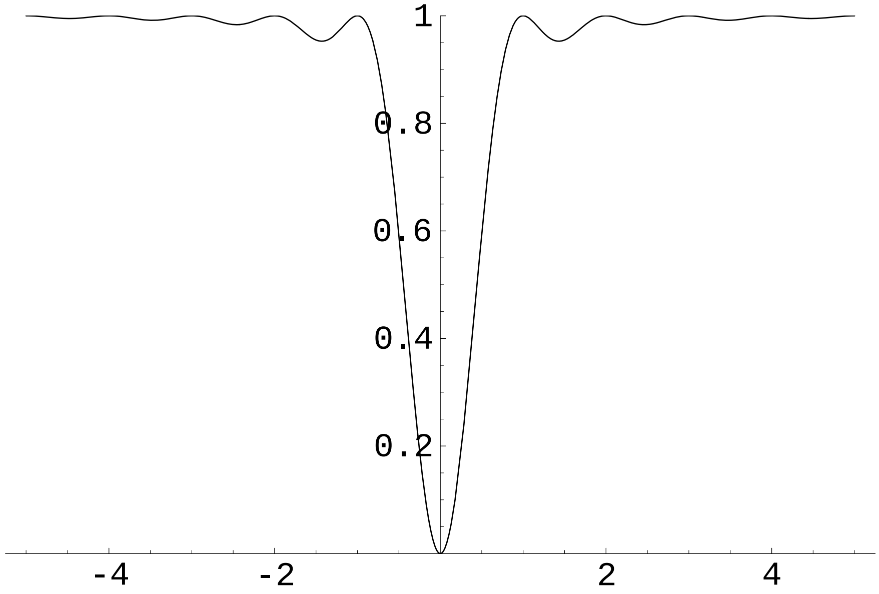


Figure 15.6: $1 - \mathrm{sinc}^2(\omega)$

This looks complicated, but what if we restrict ourselves to the common case where j_n is uniformly distributed? Then $j(x)$ is just a box function, and $J(\omega) = \mathrm{sinc}(\omega)$. But $\mathrm{sinc}(\omega)$ is zero at all integers except $\omega = 0$, so the right hand summation is just a single scaled δ-function, and the spectral density reduces to the simple formula:

$$S(\omega) = \frac{1}{T}\left[1 - \mathrm{sinc}^2(\omega)\right] + k\delta(\omega)$$

This function is plotted in figure 15.6. It is clear that this is a blue-noise like distribution with very little energy near the δ-function at the origin. This demonstrates that jittered sampling produces a reasonable spectrum for sampling and reconstruction, and pushes aliasing error into high frequency noise where it is less perceptually objectionable.

15.2.3 Quasi Monte Carlo Methods

How should this be different from what's in chapter 7? Maybe that stuff should move here?

15.3 Sampling Reflection Functions

We will now show how to use importance sampling to sample BSDFs (this can be used to compute integrals of the reflection functions from Chapter 9, for example.) Given some point on a surface, we often wish to compute the reflection integral that gives outgoing radiance in a direction ω_o.

$$L_o(x, \omega_o) = \int_{\mathcal{H}^2(\mathbf{n})} f_r(x, \omega_i \to \omega_i)\, L_i(\omega_i) |\cos\theta_i| \mathrm{d}\omega_i. \qquad (15.3.1)$$

Our task here is to define probability densities that do a good job of matching the BSDF term of the integrand. Because it's difficult to know when all of the terms will simultaneously have high values, we'll concentrate on strategies for sampling each one of them. Later, we'll show how combining the samples together from multiple strategies works gives excellent results.

First, we need to provide a reasonable default sampling method. If we know nothing about the BDRF, then the best thing to do is to sample the upper hemisphere according to a cosine distribution, because of the cosine term in Equation 15.3.1.

In addition to drawing a sample, all sampling routines will also return the value of the probability distribution for the sample chosen. This is necessary in order to properly evaluate the estimator $\frac{f(X_i)}{p(X_i)}$.

Recall that Malley's method gives a cosine distribution, so we can just use this directly.

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum BxDF::Sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2, Float *pdf) const {
    ⟨Cosine-sample the hemisphere, flipping the direction if necessary⟩
    *pdf = Pdf(wi, *wo);
    return f(wi, *wo);
}
```

BRDFToBTDF	336
BxDF	334
BxDF::f()	335
INV_PI	678
Lambertian	351
Spectrum	181
Vector	27

update text here to match code

We might need to flip the direction returned by Malley's method to make sure that it lies on the same side of the surface as the incoming direction. Because the coordinate system of the surface has $(0,0,1)$ as the normal, doing this test is a simple matter of comparing the signs of the z coordinate of ω_i and ω_o.

⟨*Cosine-sample the hemisphere, flipping the direction if necessary*⟩≡

```
*wo = CosineSampleHemisphere(u1, u2);
if (wi.z < 0.) *wo = -*wo;
```

Rather than evaluating the PDF directly inside the sampling routine, PDF evaluation will always be relegated to a `Weight` routine. This allows the PDF for an arbitrary direction (e.g. one not necessarily generated by the sampling routine itself) to be evaluated. The need for this will become apparent when we discuss multiple importance sampling.

To actually evaluate the PDF (which we showed earlier was $p(\omega) = \frac{\cos\theta}{\pi}$, note that we can simply return $N\dot{\omega}_o$, which simplifies to simply the z coordinate of ω_o since $N = (0,0,1)$:

⟨*BxDF Method Definitions*⟩+≡

```
Float BxDF::Pdf(const Vector &wi, const Vector &wo) const {
    return fabsf(wo.z) * INV_PI;
}
```

This sampling method is a fine one for Lambertian reflectors (where the BRDF is a constant), so we won't override the method for `Lambertian` or `OrenNayar` `BxDF`s. For the wrapper class `BRDFToBTDF`, we simply sample the BRDF but negate the outgoing direction:

⟨*BxDF Method Definitions*⟩+≡

```
Float BRDFToBTDF::Pdf(const Vector &wi,
                      const Vector &wo) const {
    return brdf->Pdf(wi, -wo);
}
```

15.3.1 Sampling the Blinn microfacet distribution

More complex BxDFs to sample are those based on microfacet distribution functions (See Section 9.4). There, the BxDF is a product of three main terms, D, G, and F, which is then divided by two cosine terms. Here we will describe how to importance sample the D part of the Blinn model; trying to develop a sampling method that accounted for all of the terms simultaneously would be difficult, and it's the D term that accounts for most of the variation in the BxDF's value.

All `MicrofacetDistribution`s must implement sampling and weighting functions, each with the same signature as the corresponding `BxDF` function.

⟨*MicrofacetDistribution Interface*⟩+≡

```
virtual void Sample_f(const Vector &wi, Vector *wo,
    Float u1, Float u2) const = 0;
virtual Float Pdf(const Vector &wi, const Vector &wo) const = 0;
```

`Microfacet` `BxDF`s, then, just forward on the sampling and weight requests to their distribution function.

336	`BRDFToBTDF`
334	`BxDF`
532	`BxDF::Pdf()`
357	`Microfacet`
357	`MicrofacetDistribution`
181	`Spectrum`
27	`Vector`

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum Microfacet::Sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2, Float *pdf) const {
    distribution->Sample_f(wi, wo, u1, u2);
    if (wi.z * wo->z < 0.f) return Spectrum(0.f);
    *pdf = distribution->Pdf(wi, *wo);
    return f(wi, *wo);
}
```

⟨*BxDF Method Definitions*⟩+≡

```
Float Microfacet::Pdf(const Vector &wi,
        const Vector &wo) const {
    if (wi.z * wo.z < 0.f) return 0.f;
    return distribution->Pdf(wi, wo);
}
```

Recall that Blinn's microfacet distribution function is $D = (n+1)(\cos\theta_H)^n$, where $\cos\theta_H = (N \cdot H)$. Because the value of ϕ doesn't affect D, the PDF $p_h(\theta,\phi)$ is separable into $p_h(\theta)$ and $p_h(\phi)$. $p_h(\phi)$ is constant, with a value of $1/(2\pi)$.

As usual, to sample θ_H, we must first normalize it, so that $\int_0^{\pi/2} p(\theta)\, d\omega = 1$.

$$\begin{aligned}
1 &= c\int_0^{\pi/2} (n+1)\cos^n\theta_H \sin\theta_H d\theta_H \\
&= c(-\cos^{n+1}\frac{\pi}{2} + \cos^{n+1} 0) \\
c &= 2\pi
\end{aligned}$$

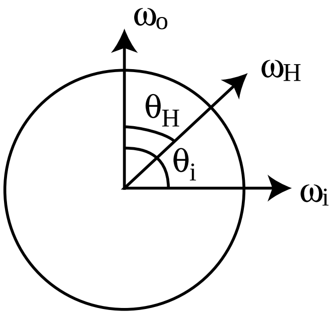


Figure 15.7: The adjustment for change of variable from sampling from the half-angle distribution to sampling from the indicent direction distribution can be derived with an observation about the relative angles involved.

Thus, our PDF $p_h(\theta)$ is $(n+1)\cos^n\theta_H$. To sample from the distribution given a uniform random number ξ, we solve:

$$\begin{aligned}\xi &= \int_0^\theta (n+1)\cos^n\theta_H \sin\theta_H d\theta_H \\ \xi &= \cos^{n+1} 0 - \cos^{n+1}\theta \\ \sqrt[n+1]{1-\xi} &= \cos\theta\end{aligned}$$

Since ξ is a uniform random number, so is $1-\xi$, so we can simplify this to $\theta = \cos^{-1}\left(\sqrt[n+1]{\xi_1}\right)$. Since the value ϕ doesn't affect the value of D, we sample it uniformly: $\phi = 2\pi\xi_2$.

We're not quite done yet, however. Microfacet distributions always give the distribution of normals around the *half-angle vector*, but the reflection integral is with respect to the *incoming vector*. This is a subtle point which many students miss when implementing importance sampling of microfacet distributions.

The fix is simple: we must simply change variables from θ_H to θ_i. This requires multiplying by the Jacobian $\partial\omega_i/\partial\omega_H$:

$$\begin{aligned}\int_{\mathcal{H}^2(\mathbf{n})} f_r(\omega_i,\omega_r)L(\omega_i)\cos\theta_i d\omega_H \frac{\partial\omega_i}{\partial\omega_H} &= \int_{\mathcal{H}^2(\mathbf{n})} f_r(\omega_i,\omega_r)L(\omega_i)\cos\theta_i d\omega_i \\ &= L_o(x,\omega_r)\end{aligned}$$

This Jacobian can be easily computed with a simple geometric construction. Consider the spherical coordinate system oriented about ω_o (see Figure 15.7). The differential solid angles $d\omega_i$ and $d\omega_H$ are $\sin\theta_i d\theta_i d\phi_i$ and $\sin\theta_H d\theta_H d\phi_H$, respectively. Therefore,

$$\frac{d\omega_i}{d\omega_H} = \frac{\sin\theta_i d\theta_i d\phi_i}{\sin\theta_H d\theta_H d\phi_H}$$

But ω_i is computed by reflecting ω_o about ω_H! This immediately gives $\theta_i = 2\theta_H$

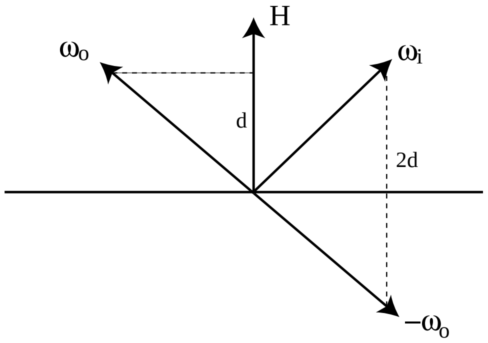


Figure 15.8: The reflection of a direction ω_o about the direction H can be computed by first taking the offset $-\omega_o$ from the origin, giving the vector beneath the surface. We then add two times the distance d, which is given by the projection of ω_o onto H (which is given by their dot product) to give us the direction ω_i above the surface.

and $\phi_i = \phi_H$. Thus,

$$\begin{aligned}
\frac{d\omega_i}{d\omega_H} &= \frac{\sin 2\theta_H 2d\theta_H d\phi_H}{\sin\theta_H d\theta_H d\phi_H} \\
&= \frac{4\cos\theta_H \sin\theta_H}{\sin\theta_H} \\
&= 4\cos\theta_H \\
&= 4(\omega_i \cdot H) = 4(\omega_o \cdot H)
\end{aligned}$$

359	Blinn
678	M_PI
192	SphericalDirection()
27	Vector

Therefore, the PDF after transformation is $p(\theta) = p_h(\theta)4(\omega_i \cdot H)$.

After all that work, the sampling function is actually quite straightforward. We sample a $\cos\theta$ and a ϕ value and convert them to a direction vector using spherical angles; we want to compute a H vector as the vector with that offset from the normal direction.

⟨BxDF Method Definitions⟩+≡

```
void Blinn::Sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2) const {
    Float costheta = powf(u1, 1.f / (exponent+1));
    Float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
    Float phi = u2 * 2.f * M_PI;
    ⟨Compute sampled half-angle vector H⟩
    ⟨Compute incident direction by reflecting about H⟩
}
```

But because `lrt` transforms the normal to $(0,0,1)$ in the reflection coordinate system, we can almost use the computed spherical direction directly. The only additional detail to handle is that if ω_i is in the other hemisphere than the normal, then the half-angle vector needs to be in that hemisphere.

⟨Compute sampled half-angle vector H⟩≡

```
Vector H = SphericalDirection(sintheta, costheta, phi);
if (wi.z < 0.f) H = -H;
```

All that's left to do in the last line of code is to apply the formula for reflection of a vector about another vector; see Figure 15.8.

⟨*Compute incident direction by reflecting about H*⟩≡

```
*wo = -wi + 2.f * Dot(wi, H) * H;
```

The weighting function is also straightforward; we simply evaluate the half-angle PDF, and multiply by the Jacobian we just computed.

⟨*BxDF Method Definitions*⟩+≡

```
Float Blinn::Pdf(const Vector &wi, const Vector &wo) const {
    Vector H = (wi + wo).Hat();
    Float costheta = fabsf(H.z);
    return ((exponent + 1.f) * powf(costheta, exponent)) /
        (4.f * Dot(wo, H));
}
```

15.3.2 Sampling the anisotropic microfacet model

Ashikhmin and Shirley give the equations for sampling their anisotropic Phong BRDF model; see (Ashikhmin and Shirley 2002; Ashikhmin and Shirley 2000). We will simply restate their results here; interested readers should consult the original publications for details and derivations.

Anisotropic	361
Blinn	359
Dot()	30
Vector	27

First, we use the PDF from section 9.4.4 to choose a half-angle vector in the first quadrant of the hemisphere (that is, spherical angles in the range $(\theta,\phi) \in \left[0,\frac{\pi}{2}\right] \times \left[0,\frac{\pi}{2}\right]$. This is done using the following formulae:

$$\phi = \arctan\left(\sqrt{\frac{e_x+1}{e_y+1}}\tan\left(\frac{\pi\xi_1}{2}\right)\right) \qquad (15.3.2)$$

$$\theta = \cos^{-1}\left(\xi_2^{(e_x\cos^2\phi+e_y\sin^2\phi+1)^{-1}}\right) \qquad (15.3.3)$$

⟨*BxDF Method Definitions*⟩+≡

```
void Anisotropic::Sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2) const {
    Float phi, costheta;
    ⟨Sample from first quadrant and remap to hemisphere⟩
    Float sintheta = sqrtf(max(0.f, 1.f - costheta*costheta));
    ⟨Compute sampled half-angle vector H⟩
    ⟨Compute incident direction by reflecting about H⟩
}
```

To sample from the first quadrant, we simply check whether ξ_1 lies in $[0,.25)$, $[.25,.5)$, $[.5,.75)$, or $[.75,1)$. Then we remap it to $[0,1]$, sample using Equations 15.3.2 and 15.3.3, and add 0, $\pi/2$, π, or $3\pi/2$ to ϕ as required.

⟨*Sample from first quadrant and remap to hemisphere*⟩≡

```
if (u1 < .25f) {
    sampleFirstQuadrant(4.f * u1, u2, &phi, &costheta);
} else if (u1 < .5f) {
    u1 = 4.f * (.5f - u1);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi = M_PI - phi;
} else if (u1 < .75f) {
    u1 = 4.f * (u1 - .5f);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi += M_PI;
} else {
    u1 = 4.f * (1.f - u1);
    sampleFirstQuadrant(u1, u2, &phi, &costheta);
    phi = 2.f * M_PI - phi;
}
```

The `sampleFirstQuadrant` method is a direct implementation of the Equations 15.3.2 and 15.3.3.

⟨*BxDF Method Definitions*⟩+≡

```
void Anisotropic::sampleFirstQuadrant(Float u1, Float u2,
        Float *phi, Float *costheta) const {
    *phi = atanf(sqrtf((ex+1)*(ey+1)) * tanf(M_PI * u1 * 0.5f));
    Float cosphi = cosf(*phi), sinphi = sinf(*phi);
    *costheta = powf(u2, 1.f/(ex * cosphi * cosphi +
        ey * sinphi * sinphi + 1));
}
```

361	`Anisotropic`
365	`FresnelBlend`
678	`M_PI`
357	`MicrofacetDistribution::D()`
27	`Vector`

Finally, the weight of this sampling method is straightforward; we simply return the distribution itself, making sure to account for the change of variables required to convert from the half-angle distribution to the incident angle distribution just as in `Blinn::Weight`.

⟨*BxDF Method Definitions*⟩+≡

```
Float Anisotropic::Pdf(const Vector &wi, const Vector &wo) const {
    if (wi.z * wo.z < 0.) return 0;
    Vector H = (wi + wo).Hat();
    return D(H) / (4.f * Dot(wo, H));
}
```

15.3.3 FresnelBlend

Is this the best sampling strategy for this BxDF?

The `FresnelBlend` class is just a mixture of a diffuse and glossy term. With equal probability, we sample one or the other. We use ξ_1 to choose which to sample, and remap ξ_1 to be between 0 and 1 before doing the sampling.

⟨*BxDF Method Definitions*⟩+≡

```
Spectrum FresnelBlend::Sample_f(const Vector &wi, Vector *wo,
        Float u1, Float u2, Float *pdf) const {
    if (u1 < .5) {
        u1 = 2.f * u1;
        ⟨Cosine-sample the hemisphere, flipping the direction if necessary⟩
    }
    else {
        u1 = 2.f * (u1 - .5f);
        distribution->Sample_f(wi, wo, u1, u2);
        if (wi.z * wo->z < 0.f) return Spectrum(0.f);
    }
    *pdf = Pdf(wi, *wo);
    return f(wi, *wo);
}
```

The weight for this sampling strategy is simple; it is just an average of the diffuse and glossy weights.

⟨*BxDF Method Definitions*⟩+≡

```
Float FresnelBlend::Pdf(const Vector &wi, const Vector &wo) const {
    if (wi.z * wo.z < 0.f) return 0.f;
    return .5f * fabsf(wo.z) * INV_PI +
        .5f * distribution->Pdf(wi, wo);
}
```

BxDF	334
FresnelBlend	365
INV_PI	678
Spectrum	181
Vector	27

15.3.4 Reflectance

We will now show how the Monte Carlo sampling routines can be used to estimate the reflectance integrals (defined in Section 9.1.1) for arbitrary BSDFs. Recall that the hemispherical-directional reflectance is the fraction of light coming from a given direction that is reflected in *any* direction, and is given by:

$$\rho_{dh}(\omega) = \frac{1}{\pi}\int_{\mathcal{H}^2(\mathbf{n})} f_r(\omega, \omega')\, d\omega'.$$

To estimate its value for a particular BxDF, we take four samples of the estimator. The number of samples is fixed to avoid allocating memory in this routine, which can be called fairly frequently.

⟨BxDF Method Definitions⟩+≡

```
Spectrum BxDF::rho(const Vector &w, int nSamples,
        Float *samples) const {
    if (!samples) {
        samples = (Float *)alloca(2 * nSamples * sizeof(Float));
        LatinHypercube(samples, nSamples, 2);
    }
    Spectrum r = 0.;
    for (int i = 0; i < nSamples; ++i) {
        ⟨Estimate one term of ρdh⟩
    }
    return r / nSamples;
}
```

Actually evaluating the estimator is straightforward; we simply sample the reflection function, and divide it by the returned weight. Notice that we also weight the reflection function by the cosine of the incoming angle; we do this because

⟨Estimate one term of ρ_{dh}*⟩*≡

```
Vector wi;
Float pdf;
Spectrum f = Sample_f(w, &wi, samples[2*i], samples[2*i+1], &pdf);
if (!f.Black() && pdf > 0.) r += f * fabsf(wi.z) / pdf;
```

334	BxDF
251	LatinHypercube()
678	M_PI
181	Spectrum
27	Vector

The hemispherical-hemispherical reflectance can be estimated similarly. Given

$$\rho_{hh} = \frac{1}{\pi}\int_{\mathcal{H}^2(\mathbf{n})}\int_{\mathcal{H}^2(\mathbf{n})} f_r(\omega_i, \omega_r)\, d\omega_i d\omega_r,$$

to estimate a term of ρ_{hh}, we need to sample two vectors, ω_i and ω_o. We first sample ω_o uniformly over the hemisphere; because our BxDF sampling routine expects the outgoing ray to be passed in, we need to sample one using a different approach. Fortunately, uniform sampling over the hemisphere works well for this.

We then sample the other direction with the BxDF::Sample_f() routine. We then just compute the estimate by multiplying the function's value by the two weights.

⟨BxDF Method Definitions⟩+≡

```
Spectrum BxDF::rho(int nSamples, Float *samples) const {
    if (!samples) {
        samples = (Float *)alloca(4 * nSamples * sizeof(Float));
        LatinHypercube(samples, nSamples, 4);
    }
    Spectrum r = 0.;
    for (int i = 0; i < nSamples; ++i) {
        ⟨Estimate one term of ρhh⟩
    }
    return r / (M_PI*nSamples);
}
```

⟨*Estimate one term of* ρ_{hh}⟩≡

```
Vector wo, wi;
wo = UniformSampleHemisphere(samples[4*i], samples[4*i+1]);
Float pdf_o = 2.f * M_PI, pdf_i;
Spectrum f = Sample_f(wo, &wi, samples[4*i+2], samples[4*i+3], &pdf_i);
if (pdf_i > 0. && !f.Black())
    r += f * fabsf(wi.z * wo.z) / (pdf_o * pdf_i);
```

15.3.5 Sampling BSDFs

Now that we have defined methods to sample individual BxDFs, we define the overall sampling method for the BSDF class. Here we have one or more individual BxDFs that we know how to sample individually, but where we want to sample the BSDF that results from the bunch of them together. Our simple solution is to randomly pick among the BxDFs, with an equal probability of choosing each one. We then use the chosen BxDF's BxDF::Sample_f() method to sample the actual direction.

The incoming and outgoing directions passed to this routine are in *world* coordinates. Because the BxDF sampling methods operate in a local coordinate system, those directions need to be transformed to and from world space as necessary.

BSDF	370
BSDF::WorldToLocal()	371
BxDF	334
BxDF::Sample_f()	335
BxDFType	334
M_PI	678
Spectrum	181
Spectrum::Black()	182
UniformSampleHemisphere()	512
Vector	27

⟨*BSDF Method Definitions*⟩+≡

```
Spectrum BSDF::Sample_f(const Vector &wiW, Vector *woW,
        Float u1, Float u2, Float u3, Float *pdf, BxDFType flags,
        BxDFType *sampledType) const {
    Vector wi = WorldToLocal(wiW);
    ⟨Choose which BxDF to sample⟩
    ⟨Sample chosen BxDF⟩
    ⟨Handle specular BxDF⟩
    if (matchingComps > 1) *pdf /= matchingComps;
    return f;
}
```

We first determine which BxDF we're going to sample. This isn't trivial, since the caller passes in certain flags that the chosen BxDF must match. This way, the caller can choose to only sample diffuse BxDF's if that is appropriate.

⟨*Choose which* BxDF *to sample*⟩≡

```
int nComps = NumComponents(flags);
const int which = min(Floor2Int(u3 * nComps), nComps-1);
BxDF *bxdf = NULL;
int comp = which;
int matchingComps = 0;
for (int i = 0; i < nBxDFs; ++i) {
    if (MatchesFlags(bxdfs[i], flags)) {
        if (comp-- == 0)
            bxdf = bxdfs[i];
        ++matchingComps;
    }
}
if (!bxdf) return Spectrum(0.f);
```

Once the BxDF is chosen, we call its corresponding Sample_f() method. Recall that these methods expect and return vectors in the BxDF's local coordinate system, so we transform the returned vector back into world coordinates immediately.

⟨*Sample chosen BxDF*⟩≡

```
Vector wo;
Spectrum f = bxdf->Sample_f(wi, &wo, u1, u2, pdf);
if (f.Black()) return f;
if (sampledType) *sampledType = bxdf->type;
*woW = LocalToWorld(wo);
```

⟨*Handle specular BxDF*⟩≡

```
if (!(bxdf->type & BSDF_SPECULAR)) {
    for (int i = 0; i < nBxDFs; ++i) {
        if (i != which && MatchesFlags(bxdfs[i], flags)) {
            *pdf += bxdfs[i]->Pdf(wi, wo);
            f += bxdfs[i]->f(wi, wo);
        }
    }
}
```

To compute the weight for the chosen sample, the obvious (but wrong) thing to do would be to simply call the BxDF::Pdf() method of the BxDF we chose. Although this makes intuitive sense, we must consider the true probability distribution of the BSDF, since that is what we sampled from. Because we sampled each BxDF with equal probability, we should equally weight the values of their BxDF::Pdf() methods to compute the overall weight for the BSDF.

370	BSDF
371	BSDF::bxdfs
372	BSDF::LocalToWorld()
371	BSDF::MatchesFlags()
371	BSDF::NumComponents()
371	BSDF::WorldToLocal()
334	BSDF_SPECULAR
334	BxDF
335	BxDF::f()
532	BxDF::Pdf()
335	BxDF::Sample_f()
334	BxDF::type
181	Spectrum
182	Spectrum::Black()
27	Vector

⟨*BSDF Method Definitions*⟩+≡

```
Float BSDF::Pdf(const Vector &wiW, const Vector &woW) const {
    if (nBxDFs == 0.) return 0.;
    Vector wi = WorldToLocal(wiW), wo = WorldToLocal(woW);
    Float pdf = 0.f;
    for (int i = 0; i < nBxDFs; ++i)
        pdf += bxdfs[i]->Pdf(wi, wo);
    return pdf / nBxDFs;
}
```

15.3.6 Specular reflection and transmission

I think this text needs some cleanup — I think it's right but perhaps could be made less informal if I knew exactly how this stuff all fit together with the new zen of light transport.

Dirac delta functions present a problem for this framework, since they involve infinities and singularities. In the real world, this is not a problem because true perfect reflection or transmission is physically impossible. However, it is often convenient to model surfaces as containing perfect reflection properties, so we need to deal with this issue.

Specular reflection or transmission is generally special cased by the particular light transport algorithm, since it is impossible to properly sample from a delta function (see the next chapter for details). In brief, these algorithms will tend to simply compute the reflected or transmitted radiance directly without explicit use of Monte Carlo methods. The reason this technique fits in well with Monte Carlo integration is that the function *and* its weight will both be a scaled Dirac delta function, and those delta functions will cancel out when computing a Monte Carlo estimator. So even though the division by the distribution is never performed explicitly, all the math works out in the end.

The only remaining piece is to return zero as the weight for any other vectors, so that the rest of the Monte Carlo framework knows not to count contributions from delta functions when the contribution will be zero.

⟨*SpecularReflection Public Methods*⟩+≡

```
Float Pdf(const Vector &wi, const Vector &wo) const {
    return 0.;
}
```

⟨*SpecularTransmission Public Methods*⟩+≡

```
Float Pdf(const Vector &wi, const Vector &wo) const {
    return 0.;
}
```

Normal	34
Point	33
Spectrum	181
Vector	27
VisibilityTester	479

15.4 Sampling Light Sources

We move now from sampling reflection distributions to sampling radiance from light sources. For consistency, we will use the same interface for light sources involving singularities (such as point and directional light sources) and true area light sources.

15.4.1 Basic Interface

Each light supports a `Light::Sample_L()` method that returns the incident radiance due to that light at some given point `P`. The interface can optionally supply the surface normal at `P` in order to achieve smarter sampling of points on an area light (for example, we could only choose points that are visible from the given orientation).

Note that these all return densities with respect to solid angle on the sphere. This is really important.

XXX what about transmission type issues here, though? XXX

⟨*Light Interface*⟩+≡

```
virtual Spectrum Sample_L(const Point &P, Float u1,
    Float u2, Vector *wo, Float *pdf,
    VisibilityTester *, bool *deltaLight = NULL) const = 0;
virtual Spectrum Sample_L(const Point &P, const Normal &N, Float u1, Float u2,
    Vector *wo, Float *pdf, VisibilityTester *, bool *deltaLight = NULL) const;
```

We can also sample the radiance along an arbitrary ray for shooting energy from the light source. This is necessary to support light transport algorithms that consider rays from the light source, such as photon mapping (Section 16.6) and bidirectional path tracing (Section 16.4).

⟨*Light Interface*⟩+≡

```
virtual Spectrum Sample_L(const Scene *scene, Float u1, Float u2,
        Float u3, Float u4, Ray *ray, Float *pdf,
        bool *deltaLight = NULL) const = 0;
```

The default implementations of the normal-augmented weighting and sampling functions simply ignore their extra argument and use the simple point-based versions instead.

⟨*Light Interface*⟩+≡

```
virtual Float Pdf(const Point &P, const Normal &,
        const Vector &w) const {
    return Pdf(P, w);
}
```

⟨*Light Method Definitions*⟩+≡

```
Spectrum Light::Sample_L(const Point &P, const Normal &, Float u1, Float u2,
        Vector *wo, Float *pdf, VisibilityTester *visibility,
        bool *deltaLight) const {
    return Sample_L(P, u1, u2, wo, pdf, visibility, deltaLight);
}
```

478	`Light`
34	`Normal`
33	`Point`
36	`Ray`
8	`Scene`
181	`Spectrum`
27	`Vector`
479	`VisibilityTester`

15.4.2 Lights with Singularities

In this section, we present the sampling routines for light sources that have no physical counterpart. One could argue that such lights do not belong in a physically-based rendering system, but they are such useful abstractions both for computational and noise-reduction reasons that we choose to include them in `lrt`.

Some care needs to be taken in implementing these functions, though. Just as with sampling perfect specular reflection, these light sources have distributions that involve delta functions. For example, point lights in `lrt` are defined in terms of radiant intensity, not radiance. However, we can safely ignore this effect, since the delta function will appear in the numerator and denominator of any Monte Carlo estimator, and will therefore cancel itself out. Just bear in mind that the values returned here are actually intensities and not radiance, but the math all works out in the end.

again, some re-wording of the above might be in order.

Point Lights

Sampling a point light is straightforward. We simply construct a vector from the light's position to the supplied sample point, and weight it according to the r^2 falloff of intensity. So notice that the same intensity value is returned for each invocation of this function, but the intensity will be divided by r^2 because of the Monte Carlo weighting.

Actually, we end up returning radiance intensity rather than radiance here and for other point lights. Need to explain that.

⟨*PointLight Method Definitions*⟩+≡
```
Spectrum PointLight::Sample_L(const Point &P, Float u1, Float u2,
        Vector *wo, Float *pdf, VisibilityTester *visibility,
        bool *deltaLight) const {
    *wo = (lightPos - P).Hat();
    *pdf = 1.f;
    if (deltaLight) *deltaLight = true;
    visibility->SetSegment(P, lightPos);
    return Intensity / DistanceSquared(lightPos, P);
}
```

If the `PointLight::Weight` function is called directly, we simply return zero. Because the point light always generates rays that originate in the same location, the odds of any other sampling strategy generating the same ray are zero. This way, if a point light is used to approximate a very small light source, the direct lighting from the source will always be 100% of the final contribution, instead of being mixed in with a reflected ray that cannot hit an infinitessimally small light source.

⟨*PointLight Method Definitions*⟩+≡
```
Float PointLight::Pdf(const Point &, const Vector &) const {
    return 0.;
}
```

`DistanceSquared()`	34
`Point`	33
`PointLight`	480
`PointLight::Intensity`	481
`PointLight::lightPos`	481
`Spectrum`	181
`SpotLight`	482
`SpotLight::Falloff()`	483
`SpotLight::Intensity`	482
`Vector`	27
`Vector::Hat()`	30
`VisibilityTester`	479
`lityTester::SetSegment()`	479

Spotlights

`Spotlight` is similar; just need to comupte outgoing intensity using the spotlight's falloff function rather than returning the same value every time. Note we're still working with radiant intensity (instead of true radiance) by ignoring the delta functions that would eventually cancel out anyway. Plus, for sampling a direction for shooting, we can be clever and only sample directions in the spotlight cone.

⟨*SpotLight Method Definitions*⟩+≡
```
Spectrum SpotLight::Sample_L(const Point &P, Float u1, Float u2,
        Vector *wo, Float *pdf, VisibilityTester *visibility,
        bool *deltaLight) const {
    *wo = (lightPos - P).Hat();
    *pdf = 1.f;
    if (deltaLight) *deltaLight = true;
    visibility->SetSegment(P, lightPos);
    return Intensity * Falloff(-*wo) /
        DistanceSquared(lightPos, P);
}
```

For generating an outgoing ray from a spotlight, we would like to sample a direction θ uniformly over the cone of directions around the center direction ω_c up to that maximum angle.

$$\begin{aligned} 1 &= c\int_0^{\theta_{max}} 1 \sin\theta \, d\theta \\ &= c(-\cos\theta_{max} + 1) \end{aligned}$$

So $p(\theta) = c = 1/(1 - \cos\theta_{max})$ and the weighting function $w(\theta) = 1 - \cos\theta_{max}$.

To sample a particular offset angle,

$$\begin{aligned}
\xi &= \frac{1}{(1-\cos\theta_{max})}\int_0^{\theta'} \sin\theta d\theta \\
\xi(1-\cos\theta_{max}) &= 1-\cos\theta' \\
\cos\theta' &= 1-\xi(1-\cos\theta_{max}) \\
\theta' &= \arccos(1-\xi(1-\cos\theta_{max}))
\end{aligned}$$

Actually $\cos\theta'$ is what we want anyway for spherical angles centered around ω_c.

⟨*MC Function Definitions*⟩+≡

```
Vector UniformSampleCone(Float u1, Float u2, Float costhetamax) {
    ⟨Uniformly sample θ and φ in cone⟩
    return Vector(cosf(phi) * sintheta, sinf(phi) * sintheta, costheta);
}
```

⟨*Uniformly sample* θ *and* φ *in cone*⟩≡

```
Float costheta = Lerp(u1, costhetamax, 1.f);
Float sintheta = sqrtf(1.f - costheta*costheta);
Float phi = u2 * 2.f * M_PI;
```

Of course, the weighting for the chosen cone is simply the inverse of the cone's solid angle.

⟨*MC Function Definitions*⟩+≡

```
Float UniformConePdf(Float cosThetaMax) {
    return 1.f / (2.f * M_PI * (1.f - cosThetaMax));
}
```

677	Lerp()
478	Light::LightToWorld
678	M_PI
36	Ray
8	Scene
181	Spectrum
482	SpotLight
482	SpotLight::Intensity
27	Vector

Now, we're ready to sample a ray coming out of the spotlight. We just choose a random ray inside the spotlight's cone, using the functions we just defined. Note that a more sophisticated scheme could be employed if we knew in advance what the falloff function was, since we would like to send rays only where the intensity of the spotlight is high. Since we don't know the falloff function in advance, this is impractical.

⟨*SpotLight Method Definitions*⟩+≡

```
Spectrum SpotLight::Sample_L(const Scene *scene, Float u1, Float u2,
        Float u3, Float u4, Ray *ray, Float *pdf,
        bool *deltaLight) const {
    ray->o = lightPos;
    Vector v = UniformSampleCone(u1, u2, cosTotalWidth);
    ray->d = LightToWorld(v);
    *pdf = UniformConePdf(cosTotalWidth);
    if (deltaLight) *deltaLight = true;
    return Intensity * Falloff(ray->d);
}
```

Projection lights and Goniometric lights

`Projectionlights` and `Goniometriclights` are essentially the same as `Spotlights`; we simply use the `ProjectionLight::Projection` or the `GoniometricLight::Scale` function instead of the Spotlight::Falloff function to evaluate intensity. The sampling functions are almost identical to spot lights and are therefore omitted.

Directional light

Sampling a directional light from a given point is trivial; we just return the directional light's vector and radiance.

⟨*DistantLight Method Definitions*⟩+≡

```
Spectrum DistantLight::Sample_L(const Point &P, Float u1, Float u2,
        Vector *wo, Float *pdf, VisibilityTester *visibility,
        bool *deltaLight) const {
    *wo = lightDir;
    *pdf = 1.f;
    if (deltaLight) *deltaLight = true;
    visibility->SetRay(P, *wo);
    return L;
}
```

Sampling a *ray* from the light's distribution is more interesting. Our task here is to choose a ray whose direction is the same as the light's direction, but which intersects the scene at a random location. To do this, we construct a disk which has the same radius as the scene's bounding sphere, and whose normal is oriented with the light's direction. We then choose a random point on this disk, using the ConcentricSampleDisk function.

Once this point has been chosen, we simply displace the point along the light's direction by the scene's bounding sphere radius, and use the new point as the origin of our light ray. This construction ensures that the ray origin lies outside the bounding sphere of the scene, and also that it will intersect the bounding sphere when shot.

DistantLight	489
M_PI	678
Point	33
Ray	36
Scene	8
Spectrum	181
Vector	27
VisibilityTester	479
isibilityTester::SetRay()	480

⟨*DistantLight Method Definitions*⟩+≡

```
Spectrum DistantLight::Sample_L(const Scene *scene,
        Float u1, Float u2, Float u3, Float u4,
        Ray *ray, Float *pdf, bool *deltaLight) const {
    ⟨Choose point on disk oriented toward infinite light direction⟩
    ⟨Set ray origin and direction for infinite light ray⟩
    if (deltaLight) *deltaLight = true;
    *pdf = 1.f / (M_PI * worldRadius * worldRadius);
    return L;
}
```

Choosing the point on the oriented disk is a simple application of vector algebra.

⟨*Choose point on disk oriented toward infinite light direction*⟩≡

```
Point worldCenter;
Float worldRadius;
scene->WorldBound().BoundingSphere(&worldCenter, &worldRadius);
Vector v1, v2;
CoordinateSystem(lightDir, &v1, &v2);
Float d1, d2;
ConcentricSampleDisk(u1, u2, &d1, &d2);
Point Pdisk = worldCenter + worldRadius * (d1 * v1 + d2 * v2);
```

And finally, we just offset the point along the light direction and initialize the ray.

⟨*Set ray origin and direction for infinite light ray*⟩≡

```
ray->o = Pdisk + worldRadius * lightDir;
ray->d = -lightDir;
```

Like all other lights involving singularities, we return a weight of zero for all direct queries, since we want multiple importance sampling to always choose our direct light sampling over a sampling of surface scattering.

⟨*DistantLight Method Definitions*⟩+≡

```
Float DistantLight::Pdf(const Point &, const Vector &) const {
    return 0.;
}
```

15.4.3 Infinite Area Lights

`InfiniteAreaLight` is the first example of light with a non-trivial sampling and weighting method. Recall that this light is a sphere that surrounds the entire scene, illuminating geometry from all directions. Sampling this light is fairly easy; we just choose a point on the sphere that is sampled with a cosine distribution.

41	`BBox::BoundingSphere()`
515	`ConcentricSampleDisk()`
32	`CoordinateSystem()`
489	`DistantLight`
493	`InfiniteAreaLight`
495	`InfiniteAreaLight::Le()`
34	`Normal`
33	`Point`
679	`RandomFloat()`
37	`RayDifferential`
14	`Scene::WorldBound()`
181	`Spectrum`
27	`Vector`
479	`VisibilityTester`
480	`VisibilityTester::SetRay()`

⟨*InfiniteAreaLight Method Definitions*⟩+≡

```
Spectrum InfiniteAreaLight::Sample_L(const Point &P,
        const Normal &N, Float u1, Float u2,
        Vector *wo, Float *pdf, VisibilityTester *visibility,
        bool *deltaLight) const {
    ⟨Sample cosine-weighted direction on unit sphere⟩
    ⟨Compute pdf for cosine-weighted infinite light direction⟩
    ⟨Transform direction to world space⟩
    if (deltaLight) *deltaLight = false;
    // XXX yuck
    visibility->SetRay(P, *wo);
    return Le(RayDifferential(P, *wo));
}
```

Choosing a cosine-weighted point on the unit sphere is almost exactly the same as Malley's method from section 14.7.3, except that we randomy select either the upper or lower hemisphere on which to genereate a point. We leave it to the reader to prove that this generates the right distribution (see exercise **??**).

⟨*Sample cosine-weighted direction on unit sphere*⟩≡

```
Float x, y, z;
ConcentricSampleDisk(u1, u2, &x, &y);
z = sqrtf(max(0.f, 1.f - x*x - y*y));
if (RandomFloat() < .5) z *= -1;
*wo = Vector(x, y, z);
```

The weight of this point is computed similarly to `CosineHemisphereWeight()`, except we need to take into account the additional solid angle subtended by the entire sphere.

⟨*Compute* `pdf` *for cosine-weighted infinite light direction*⟩≡

```
*pdf = fabsf(wo->z) * INV_TWOPI;
```

As with sampling BSDFs, we need to transform the chosen point to world coordinates. We do this with an implicit coordinate system constructed around the supplied normal.

⟨*Transform direction to world space*⟩≡

```
Vector v1, v2;
CoordinateSystem(Vector(N).Hat(), &v1, &v2);
*wo = Vector(v1.x * wo->x + v2.x * wo->y + N.x * wo->z,
             v1.y * wo->x + v2.y * wo->y + N.y * wo->z,
             v1.z * wo->x + v2.z * wo->y + N.z * wo->z);
```

Finally, we provide a non-trivial `Weight` function for `InfiniteAreaLight`. This function returns the same value as `InfiniteAreaLight::Sample_L()`, which is just the cosine weight converted to a solid angle measure. Because this function doesn't always return zero, `InfiniteAreaLights` can be used properly with multiple importance sampling.

⟨*InfiniteAreaLight Method Definitions*⟩+≡

```
Float InfiniteAreaLight::Pdf(const Point &, const Normal &N,
        const Vector &w) const {
    return AbsDot(N, w) * INV_TWOPI;
}
```

If no shading normal is supplied, we simply sample the sphere uniformly.

⟨*InfiniteAreaLight Method Definitions*⟩+≡

```
Spectrum InfiniteAreaLight::Sample_L(const Point &P,
        Float u1, Float u2, Vector *wo, Float *pdf,
        VisibilityTester *visibility, bool *deltaLight) const {
    *wo = UniformSampleSphere(u1, u2);
    *pdf = UniformSpherePdf();
    if (deltaLight) *deltaLight = false;
    visibility->SetRay(P, *wo);
    return Le(RayDifferential(P, *wo));
}
```

In the case of uniform sampling, the PDF is simply a constant equal to the surface area of the sphere being sampled, or $\frac{1}{4\pi}$.

⟨*InfiniteAreaLight Method Definitions*⟩+≡

```
Float InfiniteAreaLight::Pdf(const Point &, const Vector &) const {
    return 1.f / (4.f * M_PI);
}
```

Generating a random ray from an infinite sphere is tricky, because we need to ensure that the ray *directions* are themselves uniformly distributed. Li et al. proved that uniformly distributed lines through the volume enclosed by a sphere can be generated by connecting two uniformly chosen points on the surface of the sphere (Li, Wang, Martin, and Bowyer 2003).

Ugh, is that the right weight?

⟨*InfiniteAreaLight Method Definitions*⟩+≡

```
Spectrum InfiniteAreaLight::Sample_L(const Scene *scene,
        Float u1, Float u2, Float u3, Float u4,
        Ray *ray, Float *pdf, bool *deltaLight) const {
    ⟨Choose two points p1 and p2 on scene bounding sphere⟩
    ⟨Construct ray between p1 and p2⟩
    if (deltaLight) *deltaLight = false;
    ⟨Compute InfiniteAreaLight ray weight⟩
    ray->d *= -1.;
    Spectrum L = Le(RayDifferential(*ray));
    ray->d *= -1.;
    return L;
}
```

Of course, the “sphere” here is only an implicit one surrounding the scene. In order to use our sphere sampling routines, we must explicitly find the radius and center of the scene’s bounding sphere, and sample it.

grow the bounding sphere by a little to avoid precision problems at extrema? Suppose the entire scene is a single sphere illuminated by an infinite area light.

⟨*Choose two points* `p1` *and* `p2` *on scene bounding sphere*⟩≡

```
Point worldCenter;
Float worldRadius;
scene->WorldBound().BoundingSphere(&worldCenter, &worldRadius);
Point p1 = worldCenter + worldRadius * UniformSampleSphere(u1, u2);
Point p2 = worldCenter + worldRadius * UniformSampleSphere(u3, u4);
```

Once we have chosen the two points `p1` and `p2` on the scene’s bounding sphere, it is a simple matter to construct a ray between these two points, using `p1` as the origin and `p2` – `p1` as the direction vector.

41	`BBox::BoundingSphere()`
493	`InfiniteAreaLight`
678	`M_PI`
34	`Normal`
33	`Point`
36	`Ray`
37	`RayDifferential`
4	`Scene`
14	`Scene::WorldBound()`
181	`Spectrum`
513	`UniformSampleSphere()`
27	`Vector`
479	`VisibilityTester`

⟨*Construct ray between* `p1` *and* `p2`⟩≡

```
ray->o = p1;
ray->d = (p2-p1).Hat();
```

shorty – explain this please. The densities in the Li paper are different and I’m not sure if they’re relevant or not. This is pretty hairy.

⟨*Compute* `InfiniteAreaLight` *ray weight*⟩≡

```
//*weight = 1.f / ((4 * M_PI * worldRadius * worldRadius) *
//  (4 * M_PI * worldRadius * worldRadius));
*pdf = 1.f / (4.f * worldRadius * worldRadius);
```

why is this zero? needs explanation. no good reason; it just needs to be thought through and implemented.

⟨*InfiniteAreaLight Method Definitions*⟩+≡

```
Spectrum InfiniteAreaLight::dE(const Point &P, const Normal &N,
        Vector *wo, VisibilityTester *visibility) const {
    *wo = Vector(0,0,0);
    return 0.;
}
```

15.4.4 Area Lights

Finally, we present the methods for sampling arbitrary area lights. Recall that in `lrt`, area lights are defined by attaching an emission profile to geometry. Therefore, in order to properly sample such a light source, we must be able to generate samples over the surface of geometry. In order to do this, we will add methods to the `Shape` class to generate random points on their surfaces[1].

As with lights, we provide more than one way to generate these points. The first interface simply returns a uniformly distributed random point on the surface of the shape, along with the surface normal at the chosen point.

⟨*Shape Interface*⟩+≡

```
virtual Point Sample(Float u1, Float u2, Normal *Ns) const {
    Severe("Unimplemented Shape::Sample method called");
    return Point();
}
```

The next interface also allows the sampling routine to consider an additional point not on the surface. This is particularly useful for lighting, since we can pass in the point to be lit and ensure that we only sample the portion of the shape that is visible from that point. The `Sphere` class implements this interface; see below. The default implementation ignores the additional point and calls the default method.

`Normal`	34
`Point`	33
`Shape`	63
`Sphere`	68

⟨*Shape Interface*⟩+≡

```
virtual Point Sample(const Point &P,
        Float u1, Float u2, Normal *Ns) const {
    return Sample(u1, u2, Ns);
}
```

no one overrides this third interface, do they?

⟨*Shape Interface*⟩+≡

```
virtual Point Sample(const Point &P, const Normal &N,
        Float u1, Float u2, Normal *Ns) const {
    return Sample(P, u1, u2, Ns);
}
```

For most shapes, our sampling methods will sample uniformly over the surface of the shape. We will override this with `Sphere` below, but the same `Shape::Pdf()` function can be used for all `Shape`s to compute the reciprocal of the probability density.

again, no one implements the more complex normal-augmented interface.

⟨*Shape Interface*⟩+≡

```
virtual Float Pdf(const Point &P, const Normal &N,
        const Vector &dir) const {
    return Pdf(P, dir);
}
```

[1] Notice that the `Shape::Sample` method is not a C++ “pure virtual” function. This is deliberate; not all shapes will implement a `Sample` method. Some shapes are very difficult to sample (e.g. fractals), and it would be unreasonable to require those shapes to implement a `Sample` method just to be used as geometry. Of course, it will be a runtime error to use a shape without a `Sample` method as an area light.

The `Shape::Weight` function takes a `Point` and a `Vector`, and tells us the probability that the place where that ray intersects the geometry would have been chosen by the `Shape`'s own `Shape::Sample` routine.

⟨*Shape Interface*⟩+≡

```
virtual Float Pdf(const Point &P, const Vector &dir) const {
    ⟨Intersect sample ray with area light geometry⟩
    ⟨Convert light sample weight to solid angle measure⟩
    return pdf;
}
```

Of course, if the ray doesn't intersect the shape at all, the probability is zero, so we check for that first. Otherwise, we get the differential geometry for the corresponding sample point on the light, which will come in handy below.

⟨*Intersect sample ray with area light geometry*⟩≡

```
DifferentialGeometry dgLight;
Ray ray(P, dir);
Float thit;
if (!Intersect(ray, &thit, &dgLight)) return 0.;
ray.maxt = thit;
```

We can now compute the sample weight for this sample. We start by computing the weight with respect to the area measure over the shape. Since `Shape::Sample` chooses samples based on uniform area sampling over the surface, the sample weight is just equal to the shape's area.

However, in light transport, the integrals we are solving are with respect to a solid angle measure. Although it is more mathematically natural to express the shape density in terms of an area measure, it greatly simplifies the caller's job if all sample weights are with respect to solid angle. This is a subtle point, but crucial to understanding this code and the `Integrator` classes in the next chapter.

58	`DifferentialGeometry`
34	`DistanceSquared()`
678	`INFINITY`
562	`Integrator`
34	`Normal`
33	`Point`
36	`Ray`
63	`Shape`
66	`Shape::Intersect()`
27	`Vector`

To convert a density from area to solid angle, we simply multiply by the Jacobian:

$$\frac{\partial \omega_i}{\partial A} = \frac{r^2}{\cos \theta_o}$$

where θ_o is the angle between the ray leaving the light source and the light's surface normal, and r^2 is the distance between the point on the light and the point being shaded.

⟨*Convert light sample weight to solid angle measure*⟩≡

```
Vector dirHat = dir.Hat();
Float pdf = DistanceSquared(P, ray(ray.maxt)) /
    (AbsDot(dgLight.nn, -dirHat) * Area());
if (AbsDot(dgLight.nn, -dirHat) == 0.f) pdf = INFINITY;
```

We also provide a `Weight` function that only considers the point chosen, not the point to be shaded. Obviously, in this case the returned weight must be with respect to an area measure, since there are no implied distances or angles. This method is useful for routines that want to generate rays *originating* on the surface of a light, such as the photon mapping integrator (see section 16.6) or the bidirectional path tracer (see section 16.4).

⟨*Shape Interface*⟩+≡

```
virtual Float Pdf(const Point &Pshape) const {
    return 1.f / Area();
}
```

Sampling Disks

Sampling a `Disk` is just like picking a point on the unit disk, except we account for the value of `phiMax` and we use the value of `Disk::height` for the *z* value.

we don't actually handle phimax here. Is it easy to modify Shirley's mapping, or should we do something else?

⟨*Disk Public Methods*⟩+≡

```
Point Disk::Sample(Float u1, Float u2, Normal *Ns) const {
    Point p;
    ConcentricSampleDisk(u1, u2, &p.x, &p.y);
    p.x *= radius;
    p.y *= radius;
    p.z = height;
    *Ns = ObjectToWorld(Normal(0,0,1)).Hat();
    if (reverseOrientation) *Ns *= -1.f;
    return ObjectToWorld(p);
}
```

`ConcentricSampleDisk()`	515
`Cylinder`	78
`Cylinder::phiMax`	79
`Cylinder::radius`	79
`Cylinder::zmax`	79
`Cylinder::zmin`	79
`Disk`	82
`Normal`	34
`Point`	33
`Shape::ObjectToWorld`	64
`Shape::Pdf()`	551
`Shape::reverseOrientation`	64

Sampling Cylinders

Uniform sampling on cylinders is straightforward; we just pick a height and a ϕ value uniformly. Intuitively, this works because a cylinder is just a rolled-up rectangle.

⟨*Cylinder Public Methods*⟩+≡

```
Point Cylinder::Sample(Float u1, Float u2,
        Normal *Ns) const {
    Float h = Lerp(u1, zmin, zmax);
    Float t = u2 * phiMax;
    Point p = Point(radius * cosf(t), radius * sinf(t), h);
    *Ns = ObjectToWorld(Normal(p.x, p.y, 0.)).Hat();
    if (reverseOrientation) *Ns *= -1.f;
    return ObjectToWorld(p);
}
```

Sampling Triangles

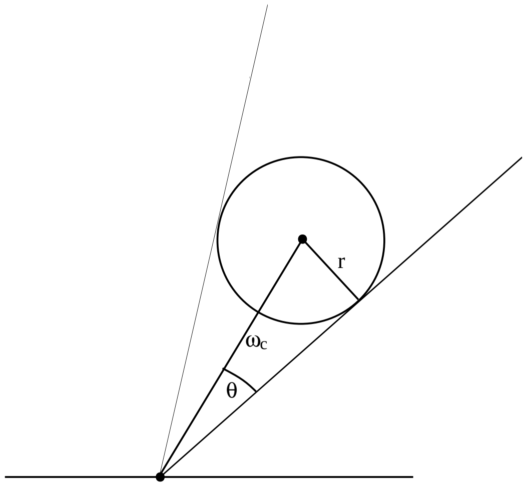


Figure 15.9: To sample points on a spherical light source, we can uniformly sample within the cone of directions around a central vector ω_c with a angular spread of up to θ.

⟨*TriangleMesh Method Definitions*⟩+≡

```
Point Triangle::Sample(Float u1, Float u2,
        Normal *Ns) const {
    Float b1, b2;
    UniformSampleTriangle(u1, u2, &b1, &b2);
    ⟨Get triangle vertices in p1, p2, and p3⟩
    Point p = b1 * p1 + b2 * p2 + (1.f - b1 - b2) * p3;
    Normal n = Normal(Cross(p2-p1, p3-p1));
    *Ns = ObjectToWorld(n).Hat();
    if (reverseOrientation) *Ns *= -1.f;
    return ObjectToWorld(p);
}
```

34	`Normal`
33	`Point`
64	`Shape::ObjectToWorld`
64	`Shape::reverseOrientation`
68	`Sphere`
69	`Sphere::radius`
90	`Triangle`
513	`UniformSampleSphere()`
519	`UniformSampleTriangle()`

Sampling Spheres

handle partial spheres

If we're not given an external point that's being lit, sampling a point on a sphere is extremely simple. We just use the `UniformSampleSphere` method to generate a point on the unit sphere, and scale the point by the sphere's radius.

⟨*Sphere Public Methods*⟩+≡

```
Point Sphere::Sample(Float u1, Float u2, Normal *Ns) const {
    Point P = Point(0,0,0) + radius * UniformSampleSphere(u1, u2);
    *Ns = ObjectToWorld(Normal(P.x, P.y, P.z)).Hat();
    if (reverseOrientation) *Ns *= -1.f;
    return ObjectToWorld(P);
}
```

If we are given a point to be lit, however, we can do better. Uniform sampling give a correct estimate, but we can greatly reduce variance by being careful to not sample points on the sphere that we know aren't visible to the point being shaded (ignoring the back side of the sphere as seen from the shading point). Figure 15.9 shows the basic two-dimensional setting for an alternate approach.

What we will do instead is uniformly sample *directions* over the solid angle subtended by the sphere from the point being shaded. We can generate directions inside this cone of directions by sampling an offset θ from the center vector ω_c, and then sampling a rotation angle ϕ around the vector.

As seen from point being shaded, the sphere subtends an angle of

$$\theta_{max} = \arcsin\left(\frac{r}{|P-c|}\right) = \arccos\sqrt{1-\left(\frac{r}{|P-c|}\right)^2} \qquad (15.4.4)$$

where r is the radius of the sphere and c is its center–see Figure 15.9.

```
⟨Sphere Public Methods⟩+≡
  Point Sphere::Sample(const Point &P,
          Float u1, Float u2, Normal *Ns) const {
      ⟨Compute coordinate system for sphere sampling⟩
      ⟨Sample uniformly on sphere if P is inside it⟩
      ⟨Sample sphere uniformly inside subtended cone⟩
      if (reverseOrientation) *Ns *= -1.f;
  }
```

CoordinateSystem()	32
DistanceSquared()	34
Normal	34
Point	33
Shape::ObjectToWorld	64
hape::reverseOrientation	64
Sphere	68
Sphere::radius	69
Vector	27

We first compute a coordinate system to use for sampling the sphere. The main axis is the vector between the sphere's center and the point being lit.

```
⟨Compute coordinate system for sphere sampling⟩≡
  Point Pcenter = ObjectToWorld(Point(0,0,0));
  Vector wc = (Pcenter - P).Hat();
  Vector wcX, wcY;
  CoordinateSystem(wc, &wcX, &wcY);
```

We must be careful about shading points that lie inside the sphere. If this happens, we simply sample the entire sphere uniformly, since the whole sphere is clearly visible from inside.

```
⟨Sample uniformly on sphere if P is inside it⟩≡
  if (DistanceSquared(P, Pcenter) < radius*radius)
      return Sample(u1, u2, Ns);
```

If the point to be lit is outside, we have the situation from Figure 15.9, and can proceed accordingly. We compute the cosine of the subtended angle θ_{max} using equation 15.4.4. We then generate a random ray inside the subtended cone using the `UniformSampleCone` function, and intersect it with the sphere to get our sample point.

Note that we must be careful about precision errors here. If the generated ray just grazes the edge of the sphere, the `Sphere::Intersect` routine might fail, which would not give us a point. In this case, we arbitrarily choose to return the point on the line between the shading point and the sphere center. Note that this very slightly biases the sampling routine, although the error introduced by this bias is extremely small.

⟨*Sample sphere uniformly inside subtended cone*⟩≡

```
Float cosThetaMax = sqrtf(max(0.f, 1.f - radius*radius /
    DistanceSquared(P, Pcenter)));
DifferentialGeometry dgSphere;
Float thit;
Point Ps;
Ray r(P, UniformSampleCone(u1, u2, cosThetaMax, wcX, wcY, wc));
if (!Intersect(r, &thit, &dgSphere)) Ps = Pcenter - radius * wc; // !@$!$
else                                 Ps = r(thit);
*Ns = Normal(Ps - Pcenter).Hat();
return Ps;
```

To compute the weight for this sampling routine, we must first differentiate between the two sampling strategies for points inside and outside the sphere.

⟨*Sphere Public Methods*⟩+≡

```
Float Sphere::Pdf(const Point &P, const Vector &dir) const {
    Point Pcenter = ObjectToWorld(Point(0,0,0));
    ⟨Return uniform weight if point inside sphere⟩
    ⟨Compute general sphere weight⟩
}
```

58	DifferentialGeometry
34	DistanceSquared()
34	Normal
33	Point
36	Ray
63	Shape
64	Shape::ObjectToWorld
551	Shape::Pdf()
68	Sphere
70	Sphere::Intersect()
69	Sphere::radius
545	UniformSampleCone()
27	Vector

If the shading point was inside the sphere, we used a uniform sampling strategy. In this case, we just hand off the `Weight` call to the parent class, which will take care of the solid angle conversion for us.

⟨*Return uniform weight if point inside sphere*⟩≡

```
if (DistanceSquared(P, Pcenter) < radius*radius)
    return Shape::Pdf(P, dir);
```

In the general case, we simply re-compute the angle subtended by the sphere and call `UniformConeWeight`. Note that no conversion is required here, because `UniformConeWeight` already returns weights with respect to a solid angle measure.

⟨*Compute general sphere weight*⟩≡

```
Float cosThetaMax = sqrtf(max(0.f, 1.f - radius*radius /
    DistanceSquared(P, Pcenter)));
return UniformConePdf(cosThetaMax);
```

Does this deserve its own subsection? Or ifdraft it out entirely?

Would be nice to have a better-distributed random sample to pick the light...

⟨*ShapeSet Public Methods*⟩+≡

```
Point Sample(Float u1, Float u2, Normal *Ns) const {
    Float ls = RandomFloat();
    u_int sn;
    for (sn = 0; sn < shapes.size()-1; ++sn)
        if (ls < areaCDF[sn]) break;
    return shapes[sn]->Sample(u1, u2, Ns);
}
```

15.4.5 Sampling Area Lights

The methods shown above were all for sampling shapes. In order to use these methods for area lights, we must implement the proper `Light::Sample_L()` methods. Fortunately, this is simple; we just delegate the calls to the underlying `Shape` and compute the required vectors and spectrum.

⟨*AreaLight Method Definitions*⟩+≡

```
Spectrum AreaLight::Sample_L(const Point &P,
        const Normal &N, Float u1, Float u2,
        Vector *wo, Float *pdf, VisibilityTester *visibility,
        bool *deltaLight) const {
    Normal Ns;
    Point Ps = shape->Sample(P, N, u1, u2, &Ns);
    *wo = (Ps - P).Hat();
    *pdf = shape->Pdf(P, N, *wo);
    if (deltaLight) *deltaLight = false;
    visibility->SetSegment(P, Ps);
    return L(P, Ns, -*wo);
}
```

```
Float AreaLight::Pdf(const Point &P, const Normal &N,
        const Vector &w) const {
    return shape->Pdf(P, N, w);
}
```

AreaLight	491
Normal	34
Point	33
Shape	63
Shape::Pdf()	551
Spectrum	181
Vector	27
VisibilityTester	479
lityTester::SetSegment()	479

The other `Light::Sample_L()` and `Light::Pdf()` interfaces are similar and omitted.

⟨*AreaLight Method Definitions*⟩+≡

```
Spectrum AreaLight::dE(const Point &P, const Normal &N, Vector *wo,
        VisibilityTester *visibility) const {
    Normal Ns;
    Point Ps = shape->Sample(P, N, RandomFloat(), RandomFloat(), &Ns);
    *wo = (Ps - P).Hat();
    visibility->SetSegment(P, Ps);
    return L(P, Ns, -*wo) * AbsDot(N, *wo) /
        shape->Pdf(P, N, *wo);
}
```

15.5 Sampling Volume Scattering

Add a `VolumeRegion::sample_phase()` method

Needed this here for the label. Either don't refer to it, or write the section.

Maybe note that this is a low-dimensional integral (1D) of a smooth function, so MC really isn't the best way to go. Maybe do some quadrature rule, or use a fixed step size or ... ?

⟨*Volume Scattering Definitions*⟩+≡

```
Spectrum DensityRegion::Tau(const Ray &r, Float stepSize, Float offset) const {
    Float t0, t1;
    Float length = r.d.Length();
    Ray rn(r.o, r.d / length, r.mint * length, r.maxt * length);
    if (!IntersectP(rn, &t0, &t1)) return 0.;
    Spectrum tau(0.);
    t0 += offset * stepSize;
    while (t0 < t1) {
        tau += sigma_t(rn(t0), -rn.d);
        t0 += stepSize;
    }
    return tau * stepSize;
}
```

We will wrap up by defining sampling methods for atmospheric scattering, as described in Chapter 12.

Beer's law says that $e^{-\alpha x}$ describes how much unattenuated light remains in a beam after travelling some distance x through a medium. Say that we have traced a ray through a scene and it has hit an object at a distance d. We then might want to randomly sample a point along the ray according to how much light remains; we'd like to focus our sampling on the parts where the light energy is strongest. First, we need to transform the exponential function into a valid PDF:

468	DensityRegion
36	Ray
181	Spectrum
466	VolumeRegion::IntersectP()
466	VolumeRegion::sigma_t()

$$
\begin{aligned}
1 &= c\int_0^d e^{-\alpha x}\mathrm{d}x \\
&= -\frac{c}{\alpha}(e^{-\alpha d}-1) \\
&= \frac{c}{\alpha}(1-e^{-\alpha d})
\end{aligned}
$$

So

$$c = \alpha/(1-e^{-\alpha d}).$$

Following similar steps, we can now determine how to sample a distance d' given a uniform random number ξ:

$$
\begin{aligned}
\xi &= \frac{\alpha}{1-e^{-\alpha d}}\int_0^{d'} e^{-\alpha x}\mathrm{d}x \\
&= \frac{1-e^{-\alpha d'}}{1-e^{-\alpha d}} \\
\xi(1-e^{-\alpha d}) &= 1-e^{-\alpha d'} \\
e^{-\alpha d'} &= 1-\xi(1-e^{-\alpha d}) \\
-\alpha d' &= \log(1-\xi(1-e^{-\alpha d})) \\
d' &= -\frac{\log(1-\xi(1-e^{-\alpha d}))}{\alpha}
\end{aligned}
$$

To sample Henyey-Greenstein, it's just:

$$\cos\theta = -\frac{1}{|2g|}\left(1+g^2-(\frac{1-g^2}{1-g+2g\xi})^2\right)$$

If $g \neq 0$, otherwise $\cos\theta = 1 - 2\xi$

XXX put it all together, show how you sample that, then sample ϕ, make a little coordinate system and you're off....

$\langle$*Foo*$\rangle\equiv$

```
double evalHG(double g, double costheta) {
    return (1 - g*g) / powf(1 + g*g - 2*g*costheta, 1.5);
}
```

$\langle$*Foo*$\rangle+\equiv$

```
double sampleHG(double g, double u, double *pdf) {
    if (fabsf(g) < 1e-5) {
    *pdf = 1.;
    return 1.f - u * 2.;
    }
    double cost = -1.f / (2.0 * g) * (1 + g*g - sqr((1 - g*g)/(1-g+2*g*u)));
    *pdf = evalHG(g, -cost);
    return cost;
}
```

15.6 Russian Roulette

XXX just introduce RR more directly: say "here is the algorithm, here is how the weighting works, and the result is unbiased... XXX

$$v' = \begin{cases} v/p & \xi < p \\ 0 & \text{otherwise} \end{cases}$$

Expected value is then

$$(1-p)\cdot 0 + p \cdot v/p = v.$$

For the first problem, we will apply a Monte Carlo technique known as *Russian roulette*. Recall that we defined a discrete probability density function over the lights in the scene for the direct lighting integrator in Section 16.1. Here, we will in a similar manner define a probability for sampling each of the terms of the infinite sum. For example, we might define the probability of sampling the ith term as

$$p_i = \frac{1}{4^{i-1}}.$$

Along the same lines as the direct lighting example, when we randomly decided to go ahead and sample the ith term according to the probability p_i, we would need to weight it's estimate by $1/p_i$ to make the estimate unbiased.

To turn this approach into an algorithm that still doesn't require us to loop over an infinite number of terms, we will incrementally decide whether to sample the ith term only if we also decided to sample the $i-1$st term. Once we decide not to sample a particular term, we don't sample any of the subsequent ones. This approach works so long as the probability of sampling each term is a non-increasing sequence. For example, for the probabilities p_i above, we equivalently have

$$\begin{aligned} p_1 &= 1 \\ p_i &= p_c^i p_{i-1} \end{aligned}$$

where p_c^i, the probability that sampling continues after the *i*th term, is $1/4$.

There is almost total freedom in how the continuation probabilities p_c^i are selected: we're free to use any information we'd like to set them so long as the weight is updated appropriately when we decide to continue. However, poorly chosen Russian roulette weights can substantially increase variance: consider if we immediately applied Russian roulette to all of the camera rays with a continuation probability of .01: we'd only trace 1% of the eye rays, weighting each of them by $1/.01 = 100$. The resulting image would numerically be just as correct as if we hadn't applied Russian roulette, though visually the result would be terrible: mostly black pixels with a few very bright ones. One of the exercises at the end of this chapter discusses this problem further and describes a technique called *efficiency optimized Russian roulette* that tries to set Russian roulette weights in a way that minimizes variance.

Splitting too!!!

Further Reading

Spanier and Gelbard (Spanier and Gelbard 1969)

Kalos and Whitlock (Kalos and Whitlock 1986)

Fishman (Fishman 1996). Liu book (Liu 2001).

Cook et al (Cook, Porter, and Carpenter 1984; Cook 1986).

Shirley thesis (Shirley 1990a)

Shirley et al on light source sampling (Shirley, Wang, and Zimmerman 1996).

Shirley square to disk mapping (Shirley and Chiu 1997)

Shirley's article has a number of recipies for warping uniform random numbers to the surfaces of various shapes (Shirley 1992).

Arvo and Kirk (Arvo and Kirk 1990)

Veach thesis (Veach 1997), includes multiple importance sampling stuff (Veach and Guibas 1995).

Keller on QMC stuff (Keller 1996), cite other stuff here as well

Dutre GI compendum

Monte Carlo/Quasi Monte Carlo website `www.mcqmc.org`.

16. Light Transport

202 Camera
563 SurfaceIntegrator
630 VolumeIntegrator

This chapter brings together the ray tracing algorithms, radiometric concepts, and Monte Carlo sampling algorithms of the previous chapters to implement a set of *integrators* that compute radiance along rays in the scene. For example, these radiance values are necessary to compute image formation in the camera Integrators are so-named because the are responsible for evaluating the integral equation called the rendering equation that describes the equilibrium distribution of radiance in an environment. As the `Camera` generates rays, they are handed off to the `SurfaceIntegrator` and the `VolumeIntegrator` that the user selected; together they are responsible for doing appropriate shading and lighting computations to compute the radiance scattered back along the ray, accounting for light reflected from the first surface visible along the ray and light attenuated and scattered by participating media along the ray, respectively.

Because the rendering equation can only be solved in closed-form for the simplest of scenes, it's generally necessary to apply a suitable numerical integration technique to approximate its solution. This has been an active area of research in rendering, and many solution methods have been proposed. In this chapter, we will provide implementations of a number of different integrators based on Monte Carlo integration that represent a broad selection of major approaches to the problem. Due to basic decisions made in `lrt`'s design, we do not have any methods based on finite-element algorithms ("radiosity"), which is the other major approach to solving the rendering equation. See the further reading section for more information about this method.

The basic integrator interfaces are defined in `core/transport.h` and some utility functions used by integrators are in `core/transport.cpp`. The implementations of the various integrators are in the `integrators/` directory.

⟨*transport.h**⟩≡

```
#include "lrt.h"
#include "primitive.h"
#include "color.h"
#include "light.h"
#include "reflection.h"
#include "sampling.h"
#include "material.h"
⟨Integrator Declarations⟩
```

⟨*transport.cpp**⟩≡

```
#include "transport.h"
#include "scene.h"
⟨Integrator Method Definitions⟩
⟨Integrator Utility Functions⟩
```

Both surface and volume integrators inherit from the `Integrator` abstract base class which defines the common interface that both of them must implement.

⟨*Integrator Declarations*⟩≡

```
class Integrator {
public:
    ⟨Integrator Interface⟩
};
```

`RayDifferential`	37
`Sampler`	237
`Scene`	8
`Spectrum`	181

The key method that all integrators must implement is `Integrator::L()`, which returns the radiance along the ray. The parameters are the following:

1. `scene`: a pointer to the `Scene` being rendered. The integrator will query the scene for information about the lights and geometry present, etc.
2. `ray`: the ray along which the radiance should be evaluated.
3. `sample`: a pointer to a `Sample` generated by the `Sampler` for this ray; some integrators will use some of its entries for Monte Carlo sampling.
4. `alpha`: the opacity of the surface that was hit should be set in this output variable; it should be zero if no surface was hit.

The method returns a `Spectrum` that holds the radiance along the ray.

⟨*Integrator Interface*⟩+≡

```
virtual Spectrum L(const Scene *, const RayDifferential &ray,
    const Sample *sample, Float *alpha) const = 0;
```

Optionally, the integrator may implement the `Preprocess()` method. It is called after the `Scene` has been fully initialized, and gives the integrator a chance to do scene-dependent computation, such as allocating additional data structures that are dependent on the number of lights in the scene, or pre-computing a rough representation of the distribution of radiance in the scene. If there isn't any work like this to be done, then this method can be left unimplemented.

⟨*Integrator Interface*⟩+≡

```
virtual void Preprocess(const Scene *scene) {
}
```

If the integrator would like the `Sampler` to generate sample patterns in the `Sample` for it to use, it should override the `RequestSamples()` method and call back to `Sample::Add1D()` and `Sample::Add2D()` methods, as described in Section 7.3.1.

⟨*Integrator Interface*⟩+≡

```
virtual void RequestSamples(Sample *sample, const Scene *scene) {
}
```

The `SurfaceIntegrator` base class doesn't add any new methods beyond those required by the `Integrator`.

⟨*Integrator Declarations*⟩+≡

```
class SurfaceIntegrator : public Integrator {
};
```

16.1 Direct Lighting

⟨*directlighting.cpp**⟩≡

```
#include "lrt.h"
#include "transport.h"
#include "scene.h"
⟨DirectLighting Declarations⟩
⟨DirectLighting Method Definitions⟩
```

562	`Integrator`
237	`Sampler`
8	`Scene`

Before we introduce the light transport equation in its full generality, we will implement an integrator that only accounts for direct lighting–light that has travelled directly from a light source to the point being shaded–and ignores indirect illumination from objects that are not themselves emissive. Starting out with this integrator allows us to focus on some of the key details of direct lighting without worrying about the full light transport equation. Furthermore, some of the routines developed here will be used in subsequent integrators that solve the complete light transport equation.

⟨*DirectLighting Declarations*⟩≡

```
class DirectLighting : public SurfaceIntegrator {
public:
     ⟨DirectLighting Public Methods⟩
private:
     ⟨DirectLighting Private Data⟩
};
```

We support three different strategies for computing direct lighting; all compute an unbiased estimate of reflection from direct lighting at the point being shaded, though they show off different approaches to the problem. An enumerant records which one has been selected.

⟨*DirectLighting Private Data*⟩≡

```
enum LightStrategy { SAMPLE_ALL_UNIFORM, SAMPLE_ONE_UNIFORM,
     SAMPLE_ONE_WEIGHTED } strategy;
```

⟨*DirectLighting Method Definitions*⟩+≡

```
DirectLighting::DirectLighting(const string &st, int md) {
    maxDepth = md;
    rayDepth = 0;
    if (st == "one") strategy = SAMPLE_ONE_UNIFORM;
    else if (st == "all") strategy = SAMPLE_ALL_UNIFORM;
    else if (st == "weighted") strategy = SAMPLE_ONE_WEIGHTED;
    else {
        Warning("Strategy \"%s\" for direct lighting unknown. "
            "Using \"all\".", st.c_str());
        strategy = SAMPLE_ALL_UNIFORM;
    }
    ⟨DirectLighting constructor implementation⟩
}
```

⟨*DirectLighting Public Methods*⟩+≡

```
void RequestSamples(Sample *sample, const Scene *scene) {
    if (strategy == SAMPLE_ALL_UNIFORM) {
        u_int nLights = scene->lights.size();
        lightSampleOffset = sample->Add2D(nLights);
        lightNumOffset = -1;
        bsdfSampleOffset = sample->Add2D(nLights);
        bsdfComponentOffset = sample->Add1D(nLights);
    }
    else {
        lightSampleOffset = sample->Add2D(1);
        lightNumOffset = sample->Add1D(1);
        bsdfSampleOffset = sample->Add2D(1);
        bsdfComponentOffset = sample->Add2D(1);
    }
}
```

DirectLighting	563
nting::SAMPLE_ALL_UNIFORM	563
nting::SAMPLE_ONE_UNIFORM	563
ting::SAMPLE_ONE_WEIGHTED	563
DirectLighting::strategy	563
Sample::Add1D()	240
Sample::Add2D()	240
Scene	8
Scene::lights	9

⟨*DirectLighting Private Data*⟩+≡

```
mutable int rayDepth;
int maxDepth;
int lightSampleOffset, lightNumOffset;
int bsdfSampleOffset, bsdfComponentOffset;
```

Recall the scattering equation from Section 5.4. It says that reflected radiance $L(\mathrm{p},\omega_o)$ from a point p on a surface in direction ω_o is the sum of emitted radiance from the surface at the point plus the integral of incoming radiance over the sphere times the BSDF for each direction times a cosine term. For the `DirectLighting` integrator, we are only interested in incoming radiance directly from light sources, which we denote by $L_{\mathrm{d}}(\mathrm{p},\omega)$.

$$L(\mathrm{p},\omega_o) = L_{\mathrm{e}}(\mathrm{p},\omega_o) + \int_{\mathcal{S}^2} f(\mathrm{p},\omega_o,\omega_i) L_{\mathrm{d}}(\mathrm{p},\omega_i) |\cos\theta_i| \mathrm{d}\omega_i \tag{16.1.1}$$

The value of $L_{\mathrm{e}}(\mathrm{p},\omega_o)$ is easily found by calling the `Intersection::Le()` method at the intersection point. To estimate the integral over the sphere, we will apply Monte Carlo integration.

The basic form of the `DirectLighting::L()` method is similar to `WhittedIntegrator::L()`; the `Scene::Intersect()` method is called to find the first visible surface along the ray, the BSDF at that point is computed etc. We won't include the full implementation of `DirectLighting::L()` here in order to focus on its key fragment, ⟨*Compute direct lighting at hit point*⟩, which estimates the value of the integral that gives the reflected radiance.

⟨*Compute direct lighting for* `DirectLighting` *integrator*⟩≡

```
if (scene->lights.size() > 0) {
    ⟨Apply direct lighting strategy⟩
}
```

The fragment ⟨*Apply direct lighting strategy*⟩ calls one of the functions that implement the three direct lighting approaches–`UniformSampleAllLights()`, `UniformSampleOneLight()`, or `WeightedSampleOneLight()`–depending on the value of the `strategy` member variable.

Consider the term of the direct lighting equation that we're concerned with here:

$$\int_{S^2} f(\mathrm{p}, \omega_o, \omega_i) L_\mathrm{d}(\mathrm{p}, \omega_i) |\cos\theta_i| d\omega_i.$$

This can be broken into a sum over the lights in the scene

9 `Scene::lights`

$$\sum_{i=1}^{\text{lights}} \int_{S^2} f(\mathrm{p}, \omega_o, \omega_i) L_{\mathrm{d(i)}}(\mathrm{p}, \omega_i) |\cos\theta_i| d\omega_i,$$

where $L_{\mathrm{d(i)}}$ denotes incident radiance from the *i*th light and

$$L_\mathrm{d}(\mathrm{p}, \omega_i) = \sum_i L_{\mathrm{d(i)}}(\mathrm{p}, \omega_i).$$

One valid approach is to estimate each term of this sum individually, adding the results together. This is the most basic direct lighting strategy and is implemented in `UniformSampleALlLights()`, which we have implemented as a global function rather than a `DirectLighting` method so that other integrators can use it as well. The `EstimateDirect()` function, which computes a Monte Carlo estimate of one of these terms, will be defined after we have described the other two direct lighting strategies.

⟨*Integrator Utility Functions*⟩≡

```
Spectrum UniformSampleAllLights(const Scene *scene, const Point &p,
        const Normal &n, const Vector &wo, BSDF *bsdf,
        const Sample *sample, int lightSampleOffset,
        int bsdfSampleOffset, int bsdfComponentOffset) {
    Spectrum L(0.);
    for (u_int i = 0; i < scene->lights.size(); ++i) {
        Light *light = scene->lights[i];
        L += EstimateDirect(scene, light, p, n, wo, bsdf,
            sample, lightSampleOffset, bsdfSampleOffset,
            bsdfComponentOffset, i);
    }
    return L;
}
```

In a scene with a large number of lights, we may not want to always compute direct lighting from all of the lights at every point that is shaded. Monte Carlo gives us a way to do this in a way that still computes the correct result in the limit. Consider as an example computing the expected value of the sum of two functions $E[f(x)+g(x)]$. It can be shown that if we randomly evaluate just one of $f(x)$ or $g(x)$ but multiply the result by two, then the expected value of the result will still be $f(x)+g(x)$.

XXX show this properly.

This generates to sums of more terms; here we can estimate direct lighting for only one randomly-chosen light, multiplying the result by the number of lights to compensate. In the first strategy, we effectively used a probability of one for sampling each light, so no additional weighting was necessary there.

⟨*Integrator Utility Functions*⟩+≡

```
Spectrum UniformSampleOneLight(const Scene *scene, const Point &p,
        const Normal &n, const Vector &wo, BSDF *bsdf,
        const Sample *sample, int lightSampleOffset,
        int lightNumOffset, int bsdfSampleOffset,
        int bsdfComponentOffset) {
    ⟨Randomly chose a single light to sample, light⟩
    return (Float)nLights *
        EstimateDirect(scene, light, p, n, wo, bsdf, sample,
            lightSampleOffset, bsdfSampleOffset,
            bsdfComponentOffset, 0);
}
```

BSDF	370
EstimateDirect()	570
Light	478
Normal	34
Point	33
RandomFloat()	679
Sample::oneD	241
Scene	8
Scene::lights	9
Spectrum	181
Vector	27

⟨*Randomly chose a single light to sample,* `light`⟩≡

```
int nLights = int(scene->lights.size());
int lightNum;
if (lightNumOffset != -1)
    lightNum = Floor2Int(sample->oneD[lightNumOffset][0] * nLights);
else
    lightNum = Floor2Int(RandomFloat() * nLights);
lightNum = min(lightNum, nLights-1);
Light *light = scene->lights[lightNum];
```

It's possible to be even more creative in choosing the individual light sampling probabilities. In fact, we're free to set the probabilities any way we like, so long as we weight the result appropriately and there is non-zero probability of sampling any light that contributes to the reflection at the point. The better a job we do at setting the probabilities so that the probability of sampling a light is proportional to the light's contribution to reflection at the point, the more efficient the Monte Carlo estimator will be and the fewer rays will be needed to reach a particular level of variance. (This is just the discrete instance of importance sampling.)

XXX Emphasize issue of handling large numbers of light sources, e.g. in a densely occluded building, not just making the most out of simple situations XXX

Here we'll implement a strategy that tries to adapt the probabilities of sampling each light over the course of rendering the image, increasing the relative probability of sampling lights that have made a large contribution to reflection for previous samples. For example, for a light that is always shadowed, we will reduce the probability of sampling it, increasing the probability of sampling lights that are contributing illumination. So long as the probability of sampling any light never goes to zero so hat we have no chance of detecting when it does contribute illumination at some point, the result will remain unbiased.

We will start with a uniform probability for sampling each of the lights. Whenever a sample is taken from a particular light, a running average of reflected radiance due to that light is updated. By evaluating the importance of each light according to the amount of light reflected rather than the amount of incident light, we also account for the effect of the BSDF in our weighting; if the BSDF is very glossy, a bright light may have much less effect on the image than a dimmer light that is often along the specular reflection direction, for example.

for each weight, store a weight, so that relative weights give relative probability of sampling lights. to make a discrete pdf, sum the weights and divide all by the sum. to make a discrete cdf, take sum of weights up to *i*th one. to choose a light, take a uniform random number,

weight is exponentially decaying average of reflected luminance $\overline{y}$. can be computed incrementally...

$$\overline{y} = (1-\alpha)y_i + \alpha\,\overline{y}_{-1}$$

where α controls rate of decay. **XXX why luminance: perceptually based... XXX**

We'll keep track of both the running average of reflected luminance from each light source as well as running average of reflected luminance from the light sources we sampled. This allows us to determine the relative importance of different lights...

XXX just make this an exercise?

⟨*DirectLighting Private Data*⟩+≡

```
  mutable Float *avgY, *avgYsample, *cdf;
  mutable Float overallAvgY;
```

Until we find a light source that contributes reflected light, `overallAvgY` will be zero. In this case, we just sample a single light with uniform probability. This gives us a reflected luminance value we can use to start updating the running averages with. Otherwise, we choose a light according to its previous contribution and update

⟨Integrator Utility Functions⟩+≡

```
Spectrum WeightedSampleOneLight(const Scene *scene, const Point &p,
        const Normal &n, const Vector &wo, BSDF *bsdf,
        const Sample *sample, int lightSampleOffset,
        int lightNumOffset, int bsdfSampleOffset,
        int bsdfComponentOffset, Float *&avgY,
        Float *&avgYsample, Float *&cdf, Float &overallAvgY) {
    int nLights = int(scene->lights.size());
    ⟨Initialize avgY array if necessary⟩
    Spectrum L(0.);
    if (overallAvgY == 0.) {
        ⟨Sample one light uniformly and initialize luminance arrays⟩
    }
    else {
        ⟨Choose light according to average reflected luminance⟩
        L = EstimateDirect(scene, light, p, n, wo, bsdf,
            sample, lightSampleOffset, bsdfSampleOffset,
            bsdfComponentOffset, 0);
        ⟨Update avgY array with reflected radiance due to light⟩
        L /= lightSampleWeight;
    }
    return L;
}
```

BSDF	370
EstimateDirect()	570
Normal	34
Point	33
Scene	8
Scene::lights	9
Spectrum	181
Spectrum::y()	185
UniformSampleOneLight()	566
Vector	27

We can't allocate space for avgY until the first time the L() method is called; we don't know how many lights are in the scene until then.

XXX do this in Preprocess() method XXX

⟨Initialize avgY array if necessary⟩≡

```
if (!avgY) {
    avgY = new Float[nLights];
    avgYsample = new Float[nLights];
    cdf = new Float[nLights+1];
    for (int i = 0; i < nLights; ++i)
        avgY[i] = avgYsample[i] = 0.;
}
```

To use the relative light weights to select a light source, we first use them to compute a discrete pdf over the light sources. We can then generate a uniform random sample value and use it to search through the cdf to find the appropriate light.

⟨Sample one light uniformly and initialize luminance arrays⟩≡

```
L = UniformSampleOneLight(scene, p, n, wo, bsdf, sample,
    lightSampleOffset, lightNumOffset, bsdfSampleOffset,
    bsdfComponentOffset);
Float luminance = L.y();
overallAvgY = luminance;
for (int i = 0; i < nLights; ++i)
    avgY[i] = luminance;
```

XXX trade-off of wasting time sampling lights that have never done us any good, just to check and see if as we move around the image thing have changed, versus not noticing when the set of important lights changes... emphasize that this a demonstration of the idea, not necessarily the best for all applications... XXX

⟨*Choose* `light` *according to average reflected luminance*⟩≡

```
Float c, lightSampleWeight;
for (int i = 0; i < nLights; ++i)
    avgYsample[i] = max(avgY[i], .1f * overallAvgY);
ComputeStep1dCDF(avgYsample, nLights, &c, cdf);
Float t = SampleStep1d(avgYsample, cdf, c, nLights,
    sample->oneD[lightNumOffset][0], &lightSampleWeight);
int lightNum = min(Float2Int(nLights * t), nLights-1);
Light *light = scene->lights[lightNum];
```

⟨*Update* `avgY` *array with reflected radiance due to light*⟩≡

```
Float luminance = L.y();
avgY[lightNum] =
    ExponentialAverage(avgY[lightNum], luminance, .99f);
overallAvgY =
    ExponentialAverage(overallAvgY, luminance, .999f);
```

478	`Light`
241	`Sample::oneD`
9	`Scene::lights`
185	`Spectrum::y()`

⟨*Global Inline Functions*⟩+≡

```
inline Float ExponentialAverage(Float avg, Float val, Float alpha) {
    return (1.f - alpha) * val + alpha * avg;
}
```

16.1.1 Estimating the direct lighting integral

Having chosen a particular light to estimate direct lighting from, we need to estimate the value of the integral

$$\int_{S^2} f(\mathrm{p}, \omega_o, \omega_i) L_\mathrm{d}(\mathrm{p}, \omega_i) |\cos\theta_i| \mathrm{d}\omega_i$$

for that light. To compute this estimate, we need to sample one or more directions ω_i and apply the Monte Carlo estimator

$$\frac{1}{N} \sum_{j=1}^{N} \frac{f(\mathrm{p}, \omega_o, \omega_j) L_\mathrm{d}(\mathrm{p}, \omega_j) |\cos\theta_j|}{p(\omega_j)}.$$

To reduce variance, we like to use importance sampling to choose directions ω_j. Because both the BSDF and the direct radiance terms are individually complex, it would be difficult in general to find sampling distributions that match their product well. Instead we will use the BSDF's sampling distribution for some of the samples and the light's for the rest. Depending on the characteristics of each of them, one of these two sampling methods may be far more effective than the other. Therefore, we will use multiple importance sampling to further improve the results.

XXX mention that visibility makes direct light part particularly bad? XXX

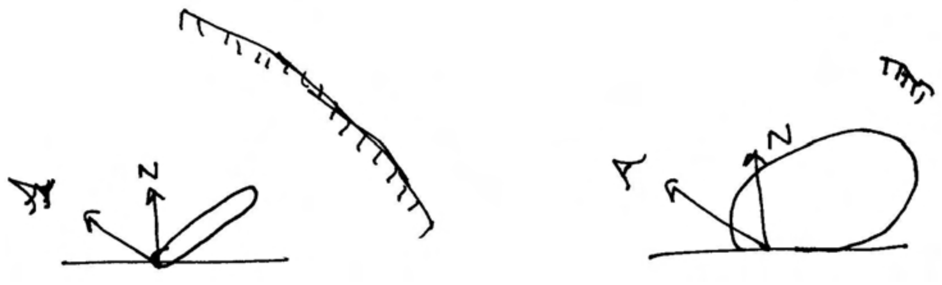


Figure 16.1: sample BSDF vs. sample light..

Figure 16.1 shows two cases where each of the sampling methods is much better than the other. On the left, the BSDF is very specular and the light source is relatively large. Sampling the BSDF will be effective at finding directions where the integrand's value is large, while sampling the light will be less effective: most of the samples will be black since the BSDF is zero for most of the directions to the light source. When the light happens to sample a point in the BSDF's glossy region, there will be a spike in the image because the light will return a low pdf, while the value of the integrand will be relatively large. In effect, we suffer from variance because the sampling distribution didn't match the actual distribution of the function's values very well.

BSDF	370
Light	478
Normal	34
Point	33
Scene	8
Spectrum	181
Vector	27

On the other hand, sometimes sampling the light is the right strategy; on the right side of Figure 16.1, the BSDF is non-zero over many directions and the light is relatively small. It will be far more effective to choose points on the light to compute ω_i, since the BSDF will have trouble finding directions where there is non-zero incident radiance from the light. Similar to the first case, we would see substantial spikes of noise when the BSDF happens to select a direction where the light was visible since the sampling distribution didn't match the overall function well.

By applying multiple importance sampling, we can not only use both of the two sampling methods, but can do so in a way that eliminates the extreme variance that each of the methods is vulnerable to individually, since the weighting terms from MIS reduce this variance substantially.

⟨*Integrator Utility Functions*⟩+≡

```
Spectrum EstimateDirect(const Scene *scene, const Light *light,
        const Point &p, const Normal &n, const Vector &wo,
        BSDF *bsdf, const Sample *sample, int lightSamp,
        int bsdfSamp, int bsdfComponent, u_int sampleNum) {
    Spectrum Ld(0.);
    ⟨Find light and BSDF sample values for direct lighting estimate⟩
    ⟨Sample light source with multiple importance sampling⟩
    ⟨Sample BSDF with multiple importance sampling⟩
    return Ld;
}
```

First we need the values for the various random numbers that we'll be using for Monte Carlo integration. A 2D sample (`ls1`, `ls2`) is needed for sampling the light source, and another 2D sample (`bs1`, `bs2`) for sampling the BSDF, and finally a

1D sample bcs is used to select a BxDF component to sample from the complete BSDF. **(XXXX explain that more carefully...)**

If the integrator calling this routine has requested appropriate samples from the Smapler, we use the corresponding values in the Sample. Otherwise, we get uniform random values from RandomFloat().

⟨*Find light and BSDF sample values for direct lighting estimate*⟩≡

```
Float ls1, ls2, bs1, bs2, bcs;
if (lightSamp != -1 && bsdfSamp != -1 &&
    sampleNum < sample->n2D[lightSamp] &&
    sampleNum < sample->n2D[bsdfSamp]) {
    ls1 = sample->twoD[lightSamp][2*sampleNum];
    ls2 = sample->twoD[lightSamp][2*sampleNum+1];
    bs1 = sample->twoD[bsdfSamp][2*sampleNum];
    bs2 = sample->twoD[bsdfSamp][2*sampleNum+1];
    bcs = sample->oneD[bsdfComponent][sampleNum];
}
else {
    ls1 = RandomFloat();
    ls2 = RandomFloat();
    bs1 = RandomFloat();
    bs2 = RandomFloat();
    bcs = RandomFloat();
}
```

For sampling the light, it's pretty straightforward application of the Monte Carlo sampling routines and the balance heuristic...

XXX explain delta function issues for this XXX

XXX update code so that sample function fills in a PDF variable, then we compute a weight, then compute $f(x)w(x)/p(x)$**... XXX**

373	BSDF::f()
542	Light::Sample_L()
679	RandomFloat()
241	Sample::n2D
241	Sample::oneD
241	Sample::twoD
181	Spectrum
182	Spectrum::Black()
27	Vector
479	VisibilityTester
480	VisibilityTester::Unoccluded()

⟨*Sample light source with multiple importance sampling*⟩≡

```
Vector wi;
Float lightPdf, bsdfPdf;
bool deltaLight;
VisibilityTester visibility;
Spectrum Li = light->Sample_L(p, n,
    ls1, ls2, &wi, &lightPdf, &visibility, &deltaLight);
if (lightPdf > 0. && !Li.Black()) {
    Spectrum f = bsdf->f(wo, wi);
    if (!f.Black() && visibility.Unoccluded(scene)) {
        ⟨Add light's contribution to reflected radiance⟩
    }
}
```

⟨*Add light's contribution to reflected radiance*⟩≡

```
if (deltaLight) {
    Ld += f * AbsDot(wi, n) * Li *
        visibility.Transmittance(scene) /
        lightPdf;
}
else {
    bsdfPdf = bsdf->Pdf(wo, wi);
    Float weight = PowerHeuristic(1, lightPdf, 1, bsdfPdf);
    Ld += f * AbsDot(wi, n) * Li *
        visibility.Transmittance(scene) * weight /
        lightPdf;
}
```

XXX BSDF is only slightly more tricky, where we need a `Ld()` utility method that computes incident radiance from only the given light source; other lights are ignored XXX

Don't do MIS for specular stuff, since other technique has no chance of finding it. Or, in a sense, the implicit delta function in the weight for the specular guy swamps the weight of the non-specular guy.

XXX it's a waste to see if we try to hit the light for a delta light source XXX

XXX make clear we ignore specular components of BSDF throughout all this, here is the only place it has to be done actively. Will generally be tracing rays for that from integrator, wasteful to do here... XXX

XXX if `f` is black, is pdf always zero? Or, is pdf ever zero w/o `f` being black? Simplify code all over the place..

Note that it's not worth bothering if this is a delta light source, since then there is no chance that the direction the BSDF samples will hit the light

BSDF::Sample_f()	540
BSDF_ALL	334
BSDF_SPECULAR	334
BxDFType	334
Light::Pdf()	543
PowerHeuristic()	525
Spectrum	181
Spectrum::Black()	182
yTester::Transmittance()	480

⟨*Sample BSDF with multiple importance sampling*⟩≡

```
if (!light->IsDeltaLight()) {
    BxDFType flags = BxDFType(BSDF_ALL & ~BSDF_SPECULAR);
    Spectrum f = bsdf->Sample_f(wo, &wi,
        bs1, bs2, bcs, &bsdfPdf, flags);
    if (!f.Black() && bsdfPdf > 0.) {
        lightPdf = light->Pdf(p, n, wi);
        if (lightPdf > 0.) {
            ⟨Add light contribution from BSDF sampling⟩
        }
    }
}
```

⟨*Add light contribution from BSDF sampling*⟩≡

```
Float weight = PowerHeuristic(1, bsdfPdf, 1, lightPdf);
Intersection lightIsect;
if (scene->Intersect(Ray(p, wi), &lightIsect) &&
    lightIsect.primitive->GetAreaLight() == light)
    Ld += f * lightIsect.Le(-wi) * AbsDot(wi, n) *
        weight / bsdfPdf;
else if (!light->Le(RayDifferential(p, wi)).Black() && !scene->IntersectP(Ray(p, wi)))
    Ld += f * light->Le(RayDifferential(p, wi)) * AbsDot(wi, n) *
        weight / bsdfPdf;
```

16.2 The Light Transport Equation

The light transport equation (LTE) is the governing equation that describes the equilibrium distribution of radiance in a scene accounting for all of the light scattering among objects in the scene. It gives the total reflected radiance at a point on a surface in a scene in terms of emission from the surface, its BSDF, and the distribution of incident illumination arriving at the point. The key task of the integrator objects in `lrt` is to numerically compute a solution to the LTE to find the radiance along camera rays starting at the film plane. Because radiance along a ray is unchanged as long as the ray doesn't intersect a surface (in a scene without participating media), we just need to find the first surface that a camera ray intersects and solve the LTE to find the outgoing radiance from that surface in the ray's direction in order to compute the radiance arriving at the film. Section 16.7 at the end of this chapter describes the generalizations necessary for scenes that do have participating media.

375	`GeometricPrimitive::GetAreaLight`
131	`Intersection`
132	`Intersection::Le()`
131	`Intersection::primitive`
525	`PowerHeuristic()`
36	`Ray`
37	`RayDifferential`
14	`Scene::Intersect()`

The detail that makes the task of evaluating the LTE difficult is the fact that incident illumination at any particular point is affected by the geometry and scattering properties of all of the other objects in the scene. For example, bright light shining on a deep red object may cause a reddish tint on nearby objects in the scene, or a glass may focus light into caustic patterns on a table top. Rendering algorithms that account for this complexity are often called global illumination algorithms, to differentiate them from local illumination algorithms that don't use a representation of the complete scene description in their shading computations.

In this section, we will first derive the basic LTE and will describe some general approaches for manipulating the equation in ways that make it easier to develop rendering algorithms that compute solutions to it. We will then describe two generalizations of the LTE that make some of its key properties more clear and serve as the foundation for some of the advanced integrators we will implement later in this chapter.

16.2.1 Basic derivation

The light transport equation depends on the basic assumptions we have already made in choosing to use radiometry to describe light–that wave optics effects are unimportant and that the distribution of radiance in the scene is in equilibrium. To be able to compute radiance arriving at the film along the camera ray, we would like to be able to express the outgoing radiance from a point on the first surface

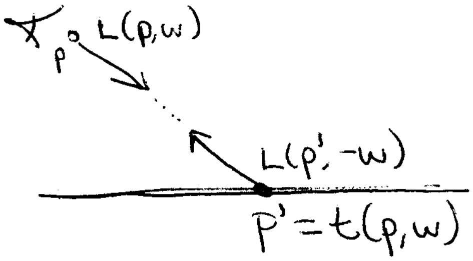


Figure 16.2: Radiance along a ray through free space is unchanged. Therefore, to compute the radiance along a ray from point p in direction ω, we can find the first surface the ray intersects and compute reflected radiance in the direction $-\omega$ there. The trace operator $t(\mathrm{p},\omega)$ gives the point p' on the first surface that the ray (p,ω) intersects.

the camera ray intersects p, in direction ω_o, the direction from the ray's origin to p, which we will denote by $L_\mathrm{o}(\mathrm{p},\omega_o)$. This can be separated into radiance that is directly emitted by the surface if it is an area light source, L_e, and radiance that is scattered by the surface due to incident illumination at p from other objects, L_s. The emitted radiance is a known property of the scene, and the scattered radiance is given by the scattering equation, 5.4.9. Combining these, we have:

$$L_\mathrm{o}(\mathrm{p},\omega_o) = L_\mathrm{e}(\mathrm{p},\omega_o) + \int_{\mathcal{S}^2} f(\mathrm{p},\omega_o,\omega_i)\, L_\mathrm{i}(\mathrm{p},\omega_i)\, |\cos\theta_i| \mathrm{d}\omega_i.$$

Because we have assumed for now that there is no participating media present, radiance is constant along rays through free space and we can relate the incident radiance at p in terms of the outgoing radiance from another point p', as shown by Figure 16.2. If we define the *ray-casting function* $t(\mathrm{p},\omega)$ as a function that computes the first surface point p' intersected by a ray from p in the direction ω, we can write the incident radiance at p in terms of outgoing radiance at p':

$$L_\mathrm{i}(\mathrm{p},\omega_o) = L_\mathrm{o}(t(\mathrm{p},\omega_o),-\omega_o).$$

In case the scene is not closed, we will define the ray casting function as returning a special value Λ if the ray (p,ω) doesn't intersect any object in the scene, such that $L_\mathrm{o}(\Lambda,\omega)$ is zero.

We can now combine these two expressions to find the light transport equation, which gives outgoing radiance point in terms of outgoing radiance at all of the other points in the scene that are visible from p:

$$L(\mathrm{p},\omega_o) = L_\mathrm{e}(\mathrm{p},\omega_o) + \int_{\mathcal{S}^2} f(\mathrm{p},\omega_o,\omega_i)\, L(t(\mathrm{p},\omega_i),-\omega_i)\, |\cos\theta_i| \mathrm{d}\omega_i, \qquad (16.2.2)$$

where for simplicity we have replaced the L_o symbols with L.

16.2.2 Analytic solution

The brevity of the LTE masks the fact that in general it is impossible to solve analytically. The complexity that comes from physically-based BSDF models, complex scene geometry, and the complex visibility relationships between objects that result all conspire to require that a numerical solution technique be used to attack the problem. Fortunately, the combination of ray tracing algorithms and Monte Carlo integration gives a powerful pair of tools that can handle this complexity without needing to impose restrictions on various components of the LTE (e.g. requiring that all BSDFs be Lambertian, or substantially limiting the geometric representations that are supported.)

It is possible to find analytic solutions to the LTE in extremely simple settings. While this is of little help with developing general-purpose rendering algorithms, it can help with debugging the implementations of integrators. If an integrator that is supposed to solve the complete LTE doesn't compute a solution that matches the analytic solution, then clearly there is a bug in the integrator to be fixed. As an example, consider the interior of a sphere where all points on the surface of the sphere have a Lambertian BRDF, $f(\mathrm{p}, \omega_o, \omega_i) = c$, and also emit a constant amount of radiance in all directions. We have

$$L(\mathrm{p}, \omega_o) = L_{\mathrm{e}} + c \int_{\mathcal{H}^2(\mathbf{n})} L(t(\mathrm{p}, \omega_i), -\omega_i) \, |\cos\theta_i| \mathrm{d}\omega_i.$$

The outgoing radiance distribution at any point on the sphere must be the same as at any other point; nothing in the environment as described could introduce any variation among different points. Therefore, the incident radiance distribution must be the same at all points and the cosine weighted integral of incident radiance must be the same everywhere as well. As such, we can replace the radiance terms with constants and simplify, writing the LTE as

$$L = L_{\mathrm{e}} + c\pi L.$$

While we could immediately solve this equation for L, it's interesting to consider successive substitution of the right-hand side into the L term on the right-hand side. If we also replace πc with ρ_{hh}, the reflectance of a Lambertian surface, we have

$$\begin{aligned} L &= L_{\mathrm{e}} + \rho_{hh}(L_{\mathrm{e}} + \rho_{hh}(L_{\mathrm{e}} + \cdots \\ &= \sum_{i=0}^{\infty} L_{\mathrm{e}} \rho_{hh}^i. \end{aligned}$$

In other words, reflected radiance is equal to the emitted radiance at the point plus light that has been scattered by a BSDF once after emission, plus light that has been scattered twice, and so forth.

Because $\rho_{hh} < 1$ due to conservation of energy, the series converges and the reflected radiance at all points in all directions is

$$L = \frac{L_{\mathrm{e}}}{1 - \rho_{hh}}.$$

This process we have just applied–repeatedly substituting the LTE's right hand side into the incident radiance term in the integral–can be productively applied in more general cases. For example, the `DirectLighting` integrator effectively

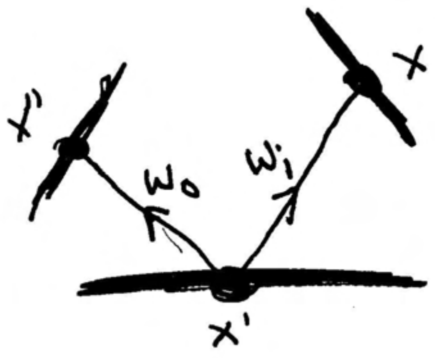


Figure 16.3: The three-point form of the light transport equation converts the integral to be over the domain of points on surfaces in the scene, rather than over directions over the sphere. It is a key transformation for deriving the path integral form of the light transport equation.

computes the result of making a single such substitution,

$$L(\mathrm{p},\omega_o) = L_\mathrm{e}(\mathrm{p},\omega_o) + \int_{\mathcal{S}^2} f(\mathrm{p},\omega_o,\omega_i)$$
$$\left(L_\mathrm{e}(t(\mathrm{p},\omega_i),-\omega_i) + \int_{\mathcal{S}^2} f(\mathrm{p},-\omega_i,\omega')L(t(\mathrm{p},\omega_i),\omega')\,|\cos\theta'|\mathrm{d}\omega'\right)\,|\cos\theta_i|\mathrm{d}\omega_i$$

And then ignores the result of multiply scattered light, setting $L(t(\mathrm{p},\omega_i),\omega') = 0$.

Over the next few pages, we will see how performing successive substitutions in this manner and then regrouping the results expresses the LTE in a more natural way for developing rendering algorithms.

16.2.3 Integral Over Paths

The introduction of the light transport equation to graphics led to a flurry of work in rendering, giving a sound theoretical basis for deriving and evaluating the correctness of rendering algorithms. For instance, the path-tracing algorithm in Section 16.3 below is based on successively evaluating the effect of light that has scattered once, twice, and so forth, using Monte Carlo integration.

One shortcoming of the LTE as expressed in the form of Equation 16.2.2 is that it is an integral equation where the left hand side of the equation appears under the integral on the right hand side. Not only is this somewhat unwieldy, but it naturally leads to light transport algorithms based on successively evaluating the LTE by recursively expanding the equation, constructing paths through the scene starting from the camera and ending at the lights. Using the light transport equation in this way limits the set of sampling techniques that are naturally applied to constructing paths and makes it difficult to incorporate more sophisticated techniques that can solve the LTE more efficiently. For example, ray tracing two paths–one starting from the camera and one starting from a light in the scene–and connecting them up in the middle can be a more effective way of finding important indirect lighting contributions than just creating paths starting from the camera.

Figure 16.4: The $G(\mathrm{p} \leftrightarrow \mathrm{p}')$ term of the three-point light transport equation gathers up a number of geometric factors that affect the amount of light exchanged between two differential areas in the scene.

The *path integral* form of the light transport equation is a more general way of expressing it. It has the form of sums over paths of various numbers of vertices on surfaces in the scene, where the first vertex is on the image plane and the last is on a light source. This form makes it more natural to develop creative ways of generating light transport paths through the scene and easier to apply more general integration techniques, which in turn can lead to more accurate results.

To derive the path integral form, we start with the *three-point form* of the light transport equation, where integral over incident directions ω_i and p is replaced with an integral over points p' in the scene (see Figure 16.3). First, we define outgoing radiance from a point p' to a point p by

$$L(\mathrm{p}' \to \mathrm{p}) = L(\mathrm{p}', \omega),$$

if p' and p are mutually visible and $\omega = \widehat{\mathrm{p} - \mathrm{p}'}$. We can also write the BSDF at p' as

$$f(\mathrm{p}'' \to \mathrm{p}' \to \mathrm{p}) = f_{(}\mathrm{p}', \omega_o, \omega_i),$$

where $\omega_i = \widehat{\mathrm{p}'' - \mathrm{p}'}$ and $\omega_o = \widehat{\mathrm{p} - \mathrm{p}'}$. Substituting these into the light transport equation and applying the term to convert an integral over solid angle into an integral over area, we have

$$L(\mathrm{p}' \to \mathrm{p}) = L_{\mathrm{e}}(\mathrm{p}' \to \mathrm{p}) + \int_A L(\mathrm{p}'' \to \mathrm{p}') f(\mathrm{p}'' \to \mathrm{p}' \to \mathrm{p}) G(\mathrm{p}'' \leftrightarrow \mathrm{p}') \mathrm{d}A(\mathrm{p}''),$$

where A is the surface area of all of the surfaces of the scene and the $G(\mathrm{p}'' \leftrightarrow \mathrm{p}')$ term accounts for $\cos\theta_i$ term in the original integral and the change of variables from integral over solid angle to integral over area. It is:

$$G(\mathrm{p} \leftrightarrow \mathrm{p}') = V(\mathrm{p} \leftrightarrow \mathrm{p}') \frac{|\cos\theta|\,|\cos\theta'|}{\|\,\mathrm{p} - \mathrm{p}'\,\|^2},$$

where $V(\mathrm{p} \leftrightarrow \mathrm{p}')$ is a visibility term that is one if the points are mutually visible and zero otherwise (see Figure 16.4).

We can now start to expand the three-point light transport equation, repeatedly substituting the right hand side of the equation into the $L(\mathrm{p}' \to \mathrm{p}')$ term inside the

Figure 16.5: example path

integral. Here are the first few terms that give incident radiance at a point p_0 from another point p_1, where p_1 is the first point on a surface along the ray from p_0 in direction $\mathrm{p}_1 - \mathrm{p}_0$.

$$\begin{aligned} L(\mathrm{p}_1 \to \mathrm{p}_0) = {} & L_\mathrm{e}(\mathrm{p}_1 \to \mathrm{p}_0) + \\ & \int_A L_\mathrm{e}(\mathrm{p}_2 \to \mathrm{p}_1)\, f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0) G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1) \mathrm{d}A(\mathrm{p}_2) + \\ & \int_{A^2} L_\mathrm{e}(\mathrm{p}_3 \to \mathrm{p}_2)\, f(\mathrm{p}_3 \to \mathrm{p}_2 \to \mathrm{p}_1) G(\mathrm{p}_3 \leftrightarrow \mathrm{p}_2) \\ & \qquad f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0) G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1)\, \mathrm{d}A(\mathrm{p}_3)\, \mathrm{d}A(\mathrm{p}_2) + \cdots \end{aligned}$$

The pattern becomes clear, and we have

$$L(\mathrm{p}_1 \to \mathrm{p}_0) = \sum_{i=1}^{\infty} P(\bar{\mathrm{p}}_i). \qquad (16.2.3)$$

$P(\bar{\mathrm{p}}_i)$ gives the amount of radiance scattered over a path $\bar{\mathrm{p}}$ with $i+1$ vertices,

$$\bar{\mathrm{p}} = \mathrm{p}_0, \mathrm{p}_1, \ldots, \mathrm{p}_\mathrm{i},$$

where p_0 is on the film plane and p_i is on a light source, and

$$\begin{aligned} P(\bar{\mathrm{p}}_i) = {} & \int_{A^{i-1}} L_\mathrm{e}(\mathrm{p}_\mathrm{i} \to \mathrm{p}_{\mathrm{i}-1}) \\ & \left(\prod_{j=1}^{i-1} f(\mathrm{p}_{\mathrm{j}+1} \to \mathrm{p}_\mathrm{j} \to \mathrm{p}_{\mathrm{j}-1}) G(\mathrm{p}_{\mathrm{j}+1} \leftrightarrow \mathrm{p}_\mathrm{j}) \right) \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_\mathrm{i}). \end{aligned}$$

See Figure 16.5.

The product of the BSDF and geometry terms is the *throughput* of the path; it describes the fraction of radiance from the light source that arrives at the camera after all of the scattering at vertices between them. We will denote it by

$$T(\bar{\mathrm{p}}_i) = \prod_{j=1}^{i-1} f(\mathrm{p}_{\mathrm{j}+1} \to \mathrm{p}_\mathrm{j} \to \mathrm{p}_{\mathrm{j}-1}) G(\mathrm{p}_{\mathrm{j}+1} \leftrightarrow \mathrm{p}_\mathrm{j}),$$

so

$$P(\bar{\mathrm{p}}_i) = \int_{A^{i-1}} L_\mathrm{e}(\mathrm{p}_\mathrm{i} \to \mathrm{p}_{\mathrm{i}-1}) T(\bar{\mathrm{p}}_i)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p}_\mathrm{i}).$$

Given Equation 16.2.3 and a particular length *i*, all that we need to do to compute a Monte Carlo estimate the radiance arriving at p_0 due to paths of length *i* is to sample a set of vertices with appropriate density in the scene p_i to generate a path and then to evaluate an estimate of $P(\bar{\mathrm{p}}_i)$ using those vertices. Whether we generate those vertices by starting a path from the camera, starting from the light, starting from both ends, or starting from a point in the middle is a detail that only affects how the weights for the Monte Carlo estimates are computed. We will see how this formulation leads in practice to practical light transport algorithms in the following two sections.

16.2.4 Delta distributions in the integrand

Delta functions may be present in $P(\bar{\mathrm{p}}_i)$ terms due to both BSDF components described by delta distributions as well as certain types of light sources (e.g. point lights and directional lights). These distributions need to be handled explicitly by the light transport algorithm if present. For example, it is impossible to randomly choose an outgoing direction from a point on a surface that would intersect a point light source–instead it is necessary to explicitly choose the single direction from the point to the light source if we want to be able to include its contribution. (The same is true for sampling BSDFs with delta components.) While handling this case introduces some additional complexity to the integrators, it is generally welcome because it reduces the dimensionality of the integral we need to evaluate, turning parts of it into a plain sum.

For example, consider the direct illumination term, $P(\bar{\mathrm{p}}_2)$, in a scene with a single point light source at point $\mathrm{p}_\mathrm{light}$ described by a delta distribution,

$$\begin{aligned} P(\bar{\mathrm{p}}_2) &= \int_A L_\mathrm{e}(\mathrm{p}_2 \to \mathrm{p}_1)\, f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0)\, G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1)\, \mathrm{d}A(\mathrm{p}_2) \\ &= \frac{\delta(\mathrm{p}_\mathrm{light} - \mathrm{p}_2) L_\mathrm{e}(\mathrm{p}_\mathrm{light} \to \mathrm{p}_2)}{p(\mathrm{p}_\mathrm{light})} f(\mathrm{p}_2 \to \mathrm{p}_1 \to \mathrm{p}_0) G(\mathrm{p}_2 \leftrightarrow \mathrm{p}_1). \end{aligned}$$

In other words, p_2 must be the same as the light's position in the scene, (the delta distribution in the numerator cancels out due to an implicit delta distribution in $p(\mathrm{p}_\mathrm{light})$; recall the discussion of sampling delta distributions in Section MC.XXX), and we are left with terms that can be evaluated directly, with no need for Monte Carlo. An analogous situation holds for BSDFs with delta distributions in the path throughput $T(\bar{\mathrm{p}}_i)$; each one eliminates an integral over area from the estimate to be computed.

16.2.5 Partitioning the integrand

Many rendering algorithms have been developed that are particularly good at solving the LTE under some conditions, but don't work well (or at all) under others. For example, the Whitted integrator only handles specular reflection from delta BSDFs and ignores multiply scattered light from diffuse and glossy BSDFs, and the irradiance caching technique described later in this chapter effectively computes scattering form diffuse surfaces but would introduce a large amount of error if used for glossy or specular reflection.

Because we would like to be able to derive correct light transport algorithms that account for all possible modes of scattering without introducing any contributions and without double-counting others, it is important to carefully account for which parts of the LTE a particular solution method accounts for. A nice way of approaching this problem is to is by partition the LTE in various ways. For example, we might expand the sum over paths to

$$L(\mathrm{p}_1 \to \mathrm{p}_0) = P(\bar{\mathrm{p}}_1) + P(\bar{\mathrm{p}}_2) + \sum_{i=3}^{\infty} P(\bar{\mathrm{p}}_i),$$

where the first term is trivially evaluated by computing the emitted radiance at p_1, the second term is solved with an accurate direct lighting solution technique, but the remaining terms in the sum are handled with a faster but less-accurate approach. If the contribution of these terms to the total reflected radiance is relatively small for the scene we're rendering, this may be a reasonable approach to take. We just need to be careful to ignore $P(\bar{\mathrm{p}}_1)$ and $P(\bar{\mathrm{p}}_2)$ with the algorithm that handles $P(\bar{\mathrm{p}}_3)$ and beyond (and similarly with the other parts.)

We may also want to partition individual $P(\bar{\mathrm{p}}_i)$ terms. For example, we might want to split the emission term into emission from small light sources, $L_{\mathrm{e},s}$, and emission from large light sources, $L_{\mathrm{e},l}$, giving us two separate integrals to estimate.

$$\begin{aligned}
P(\bar{\mathrm{p}}_i) &= \int_{A^{i-1}} (L_{\mathrm{e,s}}(\mathrm{p_i} \to \mathrm{p_{i-1}}) + L_{\mathrm{e,l}}(\mathrm{p_i} \to \mathrm{p_{i-1}}))\, T(\bar{\mathrm{p}}_i)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p_i}) \\
&= \int_{A}^{i} L_{\mathrm{e,s}}(\mathrm{p_i} \to \mathrm{p_{i-1}}))\, T(\bar{\mathrm{p}}_i)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p_i}) + \\
&= \int_{A}^{i} L_{\mathrm{e,l}}(\mathrm{p_i} \to \mathrm{p_{i-1}}))\, T(\bar{\mathrm{p}}_i)\, \mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p_i})
\end{aligned}$$

The two integrals can be evaluated independently, possibly using completely different algorithms, or different numbers of samples, selected in a way that handles the different conditions well. As long as the estimate of the $L_{\mathrm{e},s}$ integral ignores any emission from large lights, the estimate of the $L_{\mathrm{e},l}$ integral ignores emission from small lights, and all lights are categorized as either "large" or "small", we will still compute the correct result.

Finally, the BSDF terms can be partitioned as well (in fact, this application was the reason that BSDF categorization with `BxDFType` values was introduced in Section 9.1.) For example, if f_Δ denotes components of the BSDF described by delta distributions and $f_{\neg\Delta}$ denotes the remaining components,

$$\begin{aligned}
P(\bar{\mathrm{p}}_i) = \int_{A^{i-1}} L_{\mathrm{e}}(\mathrm{p_i} \to \mathrm{p_{i-1}}) \prod_{j=1}^{i-1} (f_\Delta(\mathrm{p_{j+1}} \to \mathrm{p_j} \to \mathrm{p_{j-1}}) + \\
f_{\neg\Delta}(\mathrm{p_{j+1}} \to \mathrm{p_j} \to \mathrm{p_{j-1}})) G(\mathrm{p_{j+1}} \leftrightarrow \mathrm{p_j}) \\
\mathrm{d}A(\mathrm{p}_2) \cdots \mathrm{d}A(\mathrm{p_i}).
\end{aligned}$$

Note that because there are $i-1$ BSDF terms in the product, we need to be careful to not only count terms with only f_Δ components or only $f_{\neg\Delta}$ components; we need to handle all of the terms like $f_\Delta f_{\neg\Delta} f_{\neg\Delta}$ as well.

16.2.6 The measurement equation and importance

In light of the path integral form of the LTE, it's useful to go back and formally describe the quantity that is being estimated as we compute pixel values for the image. Doing so will help us be able to formally apply the LTE to a wider set of problems than just computing 2D images (for example, to precomputing scattered radiance at the vertices of a polygonal model, as might be useful for interactive rendering applications.) Furthermore, this leads us to a key theoretical mechanism for understanding particle tracing and the photon mapping algorithm that will be described in Section 16.6.

The *measurement equation* describes the value of an abstract measurement that is found by integrating over some set of rays carrying radiance. For example, when computing the value of a pixel in the image, we want to integrate over rays starting in the neighborhood of the pixel, with contribution weighted by the image reconstruction filter. Ignoring depth of field for now (so that each point on the film plane corresponds to a single outgoing direction from the camera), we can write the pixel's value as an integral over points on the film plane of a weighting function times the incident radiance along the corresponding camera rays.

$$\begin{aligned} I_j &= \int_{A_{\mathrm{film}}} \int_{\mathcal{S}^2} W_{\mathrm{e}}(\mathrm{p}_{\mathrm{film}}, \omega) L_{\mathrm{i}}(\mathrm{p}_{\mathrm{film}}, \omega) |\cos\theta| \, \mathrm{d}A(\mathrm{p}_{\mathrm{film}}) \mathrm{d}\omega \\ &= \int_{A_{\mathrm{film}}} \int_{A} W_{\mathrm{e}}(\mathrm{p}_0 \to \mathrm{p}_1) L(\mathrm{p}_1 \to \mathrm{p}_0) G(\mathrm{p}_0 \leftrightarrow \mathrm{p}_1) \mathrm{d}A(\mathrm{p}_0) \mathrm{d}A(\mathrm{p}_1) \end{aligned}$$

where I_j is the measurement for the jth pixel and paths $\bar{\mathrm{p}}$ start at a point on the film. In this setting, the $W_{\mathrm{e}}(\mathrm{p}_0 \to \mathrm{p}_1)$ term is the product of the filter function around the pixel, f_j and a delta function that selects the appropriate camera ray direction of the sample from p_0, $\omega_{\mathrm{camera}}(\mathrm{p}_0)$:

$$W_{\mathrm{e}}(\mathrm{p}_0 \to \mathrm{p}_1) = f_j(\mathrm{p}_{\mathrm{film}}) \delta(t(\mathrm{p}_{\mathrm{film}}, \omega_{\mathrm{camera}}(\mathrm{p}_{\mathrm{film}})) - \mathrm{p}_1).$$

This formulation may initially seem gratuitously complex, but it leads us to an important insight. If we expand the $P(\bar{\mathrm{p}}_i)$ terms of the LTE sum, we have

$$\begin{aligned} I_j &= \int_{A_{\mathrm{film}}} \int_{A} W_{\mathrm{e}}(\mathrm{p}_0 \to \mathrm{p}_1) L(\mathrm{p}_1 \to \mathrm{p}_0) G(\mathrm{p}_0 \leftrightarrow \mathrm{p}_1) \mathrm{d}A(\mathrm{p}_0) \mathrm{d}A(\mathrm{p}_1) \\ &= \sum_i \int_{A^2} W_{\mathrm{e}}(\mathrm{p}_{\mathrm{film}} \to \mathrm{p}_1) P(\bar{\mathrm{p}}_i) G(\mathrm{p}_0 \leftrightarrow \mathrm{p}_1) \mathrm{d}A(\mathrm{p}_0) \mathrm{d}A(\mathrm{p}_1) \\ &= \sum_i \int_{A^{i+1}} W_{\mathrm{e}}(\mathrm{p}_0 \to \mathrm{p}_1) T(\bar{\mathrm{p}}_i) L_{\mathrm{e}}(\mathrm{p}_{\mathrm{i+1}} \to \mathrm{p}_{\mathrm{i}}) G(\mathrm{p}_0 \leftrightarrow \mathrm{p}_1) \mathrm{d}A(\mathrm{p}_0) \cdots \mathrm{d}A(\mathrm{p}_{\mathrm{i}}). \end{aligned}$$

A nice symmetry between the emitted radiance from light sources, L_{e}, and the contribution of a sample on sample on the film to the pixel measurement, W_{e} has become apparent. The implications of this symmetry are important: it says that we can think of the rendering process in two different ways–light could be emitted from light sources, bounce around the scene, and arrive at a sensor where W_{e} describes its contribution to the measurement. Alternatively, we can think of some quantity being emitted from the sensor, bouncing around the scene, and making a contribution when it hits a light source. Either intuition is equally valid.

The value described by the $W_{\mathrm{e}}(\mathrm{p}_0 \to \mathrm{p}_1)$ term is known as the *importance* for the ray between p_0 and p_1 in the scene. When the measurement equation is used to compute pixel measurements, the importance will often be partially or fully described by delta distributions, as it was in the example above. Many other types of measurement besides image formation can be described by appropriately-constructed importance functions and thus the formalisms described here can be used to show how the integral over paths described by the measurement equation is the integral that must be estimated to compute them. We will make particular use of these ideas when describing the bidirectional path tracing and photon mapping algorithms later in this chapter.

XXX mention adjoint equations and BSDF symmetry issues here, too... XXX

16.3 Path Tracing

Now that we have derived the path integral form of the light transport equation, we'll show how it can be used to derive the *path tracing* light transport algorithm and will present a path tracing integrator. Path tracing was the first general-purpose unbiased Monte Carlo light transport algorithm used in graphics. Kajiya introduced it in the same paper that first described the light transport equation. Path tracing incrementally generates paths of scattering events starting at the eye and ending at light sources in the scene. One was to think of it is as an extension of Whitted's method to include both delta-distribution and non-delta BSDFs and light sources, rather than just accounting for the delta terms.

Although it is slightly easier to derive path tracing directly from the basic light transport equation, we will instead approach it from the path integral form, which helps build understanding of the path integral equation and will make the generalization to *bidirectional path tracing*, where paths are generated starting from the lights as well as from the eye easier to understand.

16.3.1 Overview

Given the path integral form of the LTE, we need to estimate the value of

$$L(\mathrm{p}_1 \to \mathrm{p}_0) = \sum_{i=1}^{\infty} P(\bar{\mathrm{p}}_i)$$

for a given eye ray from p that first intersects the scene at p_1. We have two problems that must be solved in order to compute this estimate:

1. How do we estimate the value of the sum of the infinite number of $P(\bar{\mathrm{p}}_i)$ terms with a finite amount of computation?

2. Given a particular $P(\bar{\mathrm{p}}_i)$ term, how do we generate one or more paths $\bar{\mathrm{p}}$ in order to compute a Monte Carlo estimate of its multi-dimensional integral?

For path tracing, we can take advantage of the fact that for physically-valid scenes, paths with more vertices scatter less light than paths with fewer vertices overall (this isn't necessarily true for any particular pair of paths, just in the aggregate.) This is a natural consequence of conservation of energy in BSDFs. Therefore, we will always estimate the first few terms $P(\bar{\mathrm{p}}_i)$ and will then start to apply

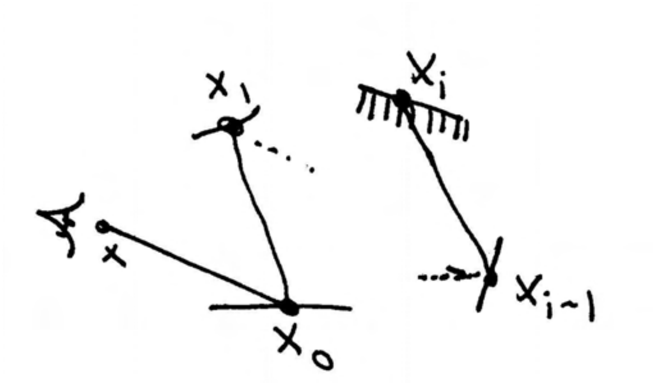


Figure 16.6:

Russian roulette to stop sampling after a finite number of terms, without introducing bias. Recall that Russian roulette allows us to probabilistically stop computing terms in a sum so long as we re-weight the terms that we do compute. For example, if we always computed estimates of $P(\bar{p}_1)$, $P(\bar{p}_2)$, and $P(\bar{p}_3)$ but stopped without computing more terms with probability q, then an unbiased estimate of the sum would be

$$P(\bar{p}_1)+P(\bar{p}_2)+P(\bar{p}_3)+\frac{1}{1-q}\sum_{i=4}^{\infty}P(\bar{p}_i)$$

Using Russian roulette in this way doesn't solve the problem of needing to evaluate an infinite sum, but has pushed it a bit farther out.

If we take this idea a step further and instead randomly consider terminating evaluation of the sum at each term with probability q_i,

$$\frac{1}{1-q_1}\left(P(\bar{p}_1)+\frac{1}{1-q_2}\left(P(\bar{p}_2)+\frac{1}{1-q_3}\left(P(\bar{p}_3)+\cdots\right.\right.\right.$$

We will eventually stop continued evaluation of the sum, yet because for any particular value of i there is greater than zero probability of evaluating the term $P(\bar{p}_i)$ and because it will be weighted appropriately if we do evaluate it, the final result will be an unbiased estimate of the sum.

16.3.2 Path sampling

Given this method for evaluating only a finite number of terms of the infinite sum, we also need a way to estimate the contribution of particular term $P(\bar{p}_i)$. We need $i+1$ vertices to specify the path, where the last vertex p_i is on a light source and the first vertex p_0 is determined by the camera ray's first intersection point (see Figure 16.6.) Looking at the form of $P(\bar{p}_i)$, a multiple integral over surface area of objects in the scene the most natural thing to do is to sample vertices p_i according to the differential area of objects in the scene, such that it's equally probable to sample any particular point on an object in the scene for p_i as any other point. (We don't actually use this approach in the `PathIntegrator` implementation for reasons that will be described below, but this sampling technique could possibly be used to improve the efficiency of our basic implementation and helps to clarify the meaning of the path integral LTE.)

We could define a discrete probability over the n objects in the scene; if each has surface area A_i, then the probability of sampling a vertex on the surface of the ith object should be

$$p_i = \frac{A_i}{\sum_j A_j}.$$

Then, given a method to sample a point on the ith object with uniform probability, the pdf for sampling any particular point on object i is $1/A_i$. Thus, the overall probability density for sampling the point is

$$\frac{A_i}{\sum_j A_j} \frac{1}{A_i}.$$

And all samples $\mathrm{p_i}$ have the same pdf value

$$p_A(\mathrm{p_i}) = \frac{1}{\sum_j A_j}.$$

It's reassuring that they all have the same weight, since our intent was to choose among all points on surfaces in the scene with equal probability.

Given the set of vertices $\mathrm{p_0}, \mathrm{p_1}, \ldots, \mathrm{p_{i-1}}$, we can then sample the last vertex $\mathrm{p_i}$ on a light source in the scene, defining its pdf in the same way. Although we could use the same technique used for sampling path vertices to sample points on lights, this would lead to high variance, since for all of the paths where $\mathrm{p_i}$ wasn't on the surface of an emitter, the path would have zero value. The expected value would still be the correct value of the integral, but convergence would be extremely slow. A better approach is to sample over the areas of only the emitting objects with probabilities updated accordingly. Given a complete path, we have all of the information we need to compute the estimate of $P(\bar{\mathrm{p}})$; it's just a matter of evaluating each of the terms.

It's easy to be more creative about how we set the sampling probabilities with this general approach. For example, if we knew that indirect illumination from a few objects contributed to most of the lighting in the scene, we could assign a higher probability to generating path vertices $\mathrm{p_i}$ on those objects, updating the sample weights appropriately.

There are two interrelated problems with sampling paths in this manner. The first can lead to high variance, while the second can lead to incorrect results. The first problem is that many of the paths will have no contribution if they have pairs of adjacent vertices that are not mutually visible. Consider applying the area sampling method in a complex building model: adjacent vertices in the path will almost always have a wall or two between them, giving no contribution for the path and excessive variance in the estimate.

The second problem is that if the integrand has delta functions in it (e.g. a point light source or a perfectly specular BSDF), this sampling technique will never be able to choose path vertices such that the delta distributions are non-zero. And even if there aren't delta distributions, as the BSDFs become increasingly glossy specular, almost all of the paths will have low contributions since the points in $f(\mathrm{p_{i+1}} \to \mathrm{p_i} \to \mathrm{p_{i-1}})$ will cause the BSDF to have a small or zero value and again we will suffer from high variance.

16.3.3 Path sampling

A solution that solves both of these problems is to construct the path incrementally, starting from p_0. At each vertex, the BSDF is sampled to generate a new direction; the next vertex $\mathrm{p}_{\mathrm{i+1}}$ is found by tracing a ray from p_i in the sampled direction and finding the closest intersection. We are effectively trying to find a path with a large contribution by making a series of choices that locally find directions with important contributions. While one can imagine situations where this approach could be ineffective, it is a generally a good strategy.

Because this approach constructs the path by sampling BSDFs according to solid angle, and because the path integral LTE is an integral over surface area in the scene, we need to apply the correction to convert from the probability density according to solid angle p_ω to a density according to area p_A (recall Section 5.3):

$$p_A = p_\omega \frac{\|\mathrm{p}_\mathrm{i} - \mathrm{p}_{\mathrm{i+1}}\|^2}{|\cos\theta_i|}.$$

This correction causes some of the terms of the geometric term $G(\mathrm{p}_\mathrm{i} \leftrightarrow \mathrm{p}_{\mathrm{i+1}})$ to cancel out of $P(\bar{\mathrm{p}})$. Furthermore, we already know that p_i and $\mathrm{p}_{\mathrm{i+1}}$ must be mutually visible since we traced a ray to find $\mathrm{p}_{\mathrm{i+1}}$, so the visibility term is trivially one. If we still sample the last vertex p_i from some distribution over the surfaces of light sources $p_A(\mathrm{p}_\mathrm{i})$, our estimate is

$$P(\bar{\mathrm{p}}) \approx \frac{L_\mathrm{e}(\mathrm{p}_\mathrm{i} \to \mathrm{p}_{\mathrm{i-1}})}{p_A(\mathrm{p}_\mathrm{i})} \left(\prod_{j=1}^{i-1} \frac{f(\mathrm{p}_{\mathrm{j+1}} \to \mathrm{p}_\mathrm{j} \to \mathrm{p}_{\mathrm{j-1}})|\cos\theta_j|}{p_\omega(\mathrm{p}_{\mathrm{j+1}} - \mathrm{p}_\mathrm{j})} \right).$$

16.3.4 Implementation

Our path tracing implementation computes an estimate of the sum of path contributions $P(\bar{\mathrm{p}})$ using the approach described above. Starting at the first intersection of the camera ray with the scene geometry, p_1, we incrementally sample path vertices by importance sampling the BSDF at the current vertex and tracing a ray to the next vertex. To find the last vertex of a particular path, p_i, which must be on a light source in the scene, we will use the multiple importance sampling-based direct lighting code that was developed for the direct lighting integrator. By using the multiple importance sampling weights instead of $p_A(\mathrm{p}_\mathrm{i})$ to compute the estimate as described above, we will have lower variance in the result for cases where sampling the BSDF would have been a better way to find a point on the light.

Another small difference is that as we are evaluating the estimates of the path contribution terms $P(\bar{\mathrm{p}})$, we will re-use the vertices of the previous path of length $i-1$ (except the vertex on the emitter) as a starting point when constructing the path of length i. This means that we just need to trace one more ray to construct the new path, rather then i rays as we would if we started from scratch. Reusing paths in this manner does introduce correlation among all of the $P(\bar{\mathrm{p}})$ terms in the sum, which slightly reduces the quality of the result. In practice this is more than made up for by the improved overall efficiency due to tracing fewer rays.

```
⟨path.cpp*⟩≡
  #include "lrt.h"
  #include "transport.h"
  #include "scene.h"
  ⟨PathIntegrator Declarations⟩
  ⟨PathIntegrator Method Definitions⟩
```

```
⟨PathIntegrator Declarations⟩≡
  class PathIntegrator : public SurfaceIntegrator {
  public:
      ⟨PathIntegrator Public Methods⟩
  private:
      ⟨PathIntegrator Private Data⟩
  };
```

We will use samples from the `Sampler` for random sampling at the first `SAMPLE_DEPTH` vertices of the path. After the first few bounces, the advantages of well-distributed sample points are greatly reduced and we will just use uniform random numbers. We need light and BSDF samples for multiple importance sampling for the direct lighting calculation at each vertex of the path as well as BSDF samples for sampling directions when generating the outgoing direction for finding the next vertex of the path.

Sample::Add1D()	240
Sample::Add2D()	240
Scene	8
SurfaceIntegrator	563

```
⟨PathIntegrator Method Definitions⟩≡
  void PathIntegrator::RequestSamples(Sample *sample,
          const Scene *scene) {
      for (int i = 0; i < SAMPLE_DEPTH; ++i) {
          lightPositionOffset[i] = sample->Add2D(1);
          lightNumOffset[i] = sample->Add1D(1);
          bsdfDirectionoffset[i] = sample->Add2D(1);
          bsdfComponentOffset[i] = sample->Add1D(1);
          outgoingDirectionOffset[i] = sample->Add2D(1);
          outgoingComponentOffset[i] = sample->Add1D(1);
      }
  }
```

```
⟨PathIntegrator Private Data⟩≡
  #define SAMPLE_DEPTH 3
  int lightPositionOffset[SAMPLE_DEPTH];
  int lightNumOffset[SAMPLE_DEPTH];
  int bsdfDirectionoffset[SAMPLE_DEPTH];
  int bsdfComponentOffset[SAMPLE_DEPTH];
  int outgoingDirectionOffset[SAMPLE_DEPTH];
  int outgoingComponentOffset[SAMPLE_DEPTH];
```

Each time through the `for` loop of the integrator, we find the next vertex of the path by intersecting the current ray with the scene geometry and compute the contribution of the path to the overall radiance value with the direct lighting code. We then choose a new direction from the last vertex of the path to update the ray to be traced the next time through the loop. After a few vertices have been sampled, we start using Russian roulette to randomly terminate the path.

⟨*PathIntegrator Method Definitions*⟩+≡

```
Spectrum PathIntegrator::L(const Scene *scene,
        const RayDifferential &r, const Sample *sample,
        Float *alpha) const {
    ⟨Declare common path integration variables⟩
    for (int pathLength = 0; ; ++pathLength) {
        ⟨Find next vertex of path⟩
        ⟨Possibly add emitted light at path vertex⟩
        ⟨Evaluate BSDF at hit point⟩
        ⟨Sample illumination from lights to find path contribution⟩
        ⟨Sample BSDF to get new path direction⟩
        ⟨Possibly terminate the path⟩
    }
    return L;
}
```

Four variables record the current state of the path. `pathThroughput` holds the product of the BSDF values and cosine terms for the vertices generated so far, divided by their respective sampling pdfs,

$$\prod_{j=1}^{i-1} \frac{f(\mathrm{p_{j+1}} \to \mathrm{p_j} \to \mathrm{p_{j01}})|\cos\theta_j|}{p_\omega(\mathrm{p_{j+1}} - \mathrm{p_j})}.$$

586	`PathIntegrator`
37	`RayDifferential`
8	`Scene`
181	`Spectrum`

Thus, the product of `pathThroughput` with scattered light from direct lighting at the final vertex of the path gives the contribution for that overall path. One advantage of this approach is that we don't need to store the positions and BSDFs of all of the vertices of the path, only the last one. `L` holds the radiance value from the running total of $\sum P(\bar{\mathrm{p}}_i)$ that we are computing, and `ray` holds the next ray to be traced to extend the path one more vertex. Finally, `specularBounce` records if the last outgoing path direction sampled was due to specular reflection; the need to record this will be explained shortly.

⟨*Declare common path integration variables*⟩≡

```
Spectrum pathThroughput = 1., L = 0.;
RayDifferential ray(r);
bool specularBounce = false;
```

Because of the loop invariant that `ray` has been initialized to be the ray to be traced to find the next path vertex, our task here is quite easy. If no intersection is found along the given ray, we stop processing this path. The fragment ⟨*Stop path sampling since no intersection was found*⟩, not included here, does some final cleanup and breaks out of the loop. If we did find an intersection, and if this is the first vertex of the path after the point on the film plane, we also need to initialize the `alpha` output variable and the `maxt` variable of the ray that was originally passed to `PathIntegrator::L()`.

```
⟨Find next vertex of path⟩≡
  Intersection isect;
  if (!scene->Intersect(ray, &isect)) {
      ⟨Stop path sampling since no intersection was found⟩
      break;
  }
  if (pathLength == 0) {
      r.maxt = ray.maxt;
      if (alpha) *alpha = 1.;
  }
```

We only include emission at a vertex that happened to hit an emitting object at the first intersection point since that isn't handled by any other sampling method or if the previous bounce was due to specular reflection. In other words, we need to ignore any emission from vertices in the path that aren't explicitly sampled as part of the direct lighting computation, unless the previous bounce was due to specular reflection, because the direct lighting code ignores any specular components of the BSDF, leaving the integrator to handle them on its own instead.

```
⟨Possibly add emitted light at path vertex⟩≡
  if (pathLength == 0 || specularBounce)
      L += pathThroughput * isect.Le(-ray.d);
```

BSDF::dgShading	370
DifferentialGeometry::p	58
Intersection	131
Intersection::Le()	132
Normal	34
tor::bsdfComponentOffset	586
tor::bsdfDirectionoffset	586
tegrator::lightNumOffset	586
tor::lightPositionOffset	586
Point	33
Ray::d	35
Ray::maxt	36
Scene::Intersect()	14
UniformSampleOneLight()	566
Vector	27

The direct lighting computation uses of the UniformSampleOneLight() function, which gives us an estimate of the reflected radiance from the BSDF at the vertex at the end of the current path. Scaling this value by the running product of the path contribution gives the overall contribution to the total radiance estimate.

```
⟨Sample illumination from lights to find path contribution⟩≡
  const Point &p = bsdf->dgShading.p;
  const Normal &n = bsdf->dgShading.nn;
  Vector wo = -ray.d;
  if (pathLength < SAMPLE_DEPTH)
      L += pathThroughput *
          UniformSampleOneLight(scene, p, n, wo, bsdf, sample,
              lightPositionOffset[pathLength],
              lightNumOffset[pathLength],
              bsdfDirectionoffset[pathLength],
              bsdfComponentOffset[pathLength]);
  else
      L += pathThroughput *
          UniformSampleOneLight(scene, p, n, wo, bsdf, sample);
```

And now we need to sample the BSDF at the end of the current path to get an outgoing direction from this vertex for the next ray to trace. The ⟨*Get random numbers for sampling new direction,* bs1*,* bs2*, and* bcs⟩ fragment, not included here, gets three sample values, either from sample if the current path length is less than SAMPLE_DEPTH or using ‘[RandomFloat()]RandomFloat() otherwise. BSDF::Sample_f() uses the first two, bs1 and bs2 to select a direction from a single BxDF's distribution, and bcs to choose which of potentially multiple BxDFs to sample. We update the path throughput as described earlier and initialize ray with the ray to be traced to find the next vertex the next time we go through the `for` loop.

⟨*Sample BSDF to get new path direction*⟩≡

```
⟨Get random numbers for sampling new direction, bs1, bs2, and bcs⟩
Vector wi;
Float pdf;
BxDFType flags;
Spectrum f = bsdf->Sample_f(wo, &wi, bs1, bs2, bcs,
    &pdf, BSDF_ALL, &flags);
if (f.Black() || pdf == 0.)
    break;
specularBounce = (flags & BSDF_SPECULAR);
pathThroughput *= f * AbsDot(wi, n) / pdf;
ray = RayDifferential(p, wi);
```

Path termination kicks in after a few bounces, with a fixed termination probability for all additional bounces. If the path isn't terminated, we can update `pathThroughput` with the Russian roulette weight and all subsequent $P(\bar{\mathrm{p}})$ terms will be appropriately affected by the weight.

⟨*Possibly terminate the path*⟩≡

```
if (pathLength > 3) {
    Float continueProbability = .5f;
    if (RandomFloat() > continueProbability)
        break;
    pathThroughput /= continueProbability;
}
```

540	BSDF::Sample_f()
334	BSDF_ALL
334	BSDF_SPECULAR
334	BxDFType
679	RandomFloat()
37	RayDifferential
181	Spectrum
182	Spectrum::Black()
27	Vector

16.4 ***ADV***: Bidirectional Path Tracing

XXX incorporate In `lrt`,since we don't explicitly model the camera geometry as part of the scene, there is no way to generate a path from a light to the film without explicitly sampling a point on the film anyway. Though this slightly limits the ways in which we can generate paths $\bar{\mathrm{p}}$, it's usually a minor loss in practice (XXX explain better XXX).

⟨*bidirectional.cpp**⟩≡

```
#include "lrt.h"
#include "transport.h"
#include "scene.h"
#include "mc.h"
⟨Bidirectional Local Declarations⟩
⟨Bidirectional Method Definitions⟩
```

The path tracing algorithm described in the previous section was the first general light transport algorithm in graphics, handling both a wide variety of geometric objects as well as area lights and general BSDF models. Although it works well for many scenes, it can exhibit high variance in the presence of particular tricky lighting conditions. For example, consider the setting shown in Figure 16.7; a light source is illuminating a small area on the ceiling, such that the rest of the room

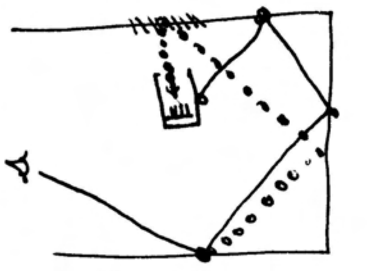


Figure 16.7:

is only illuminated by indirect lighting bouncing from that area. If we only trace paths starting from the eye, we will almost never happen to sample a vertex in the illuminated region before we trace a shadow ray to the light. Most of the paths will have no contribution, while a few of them–the ones that happen to hit the small region on the ceiling–will have a large contribution. The resulting image will have high variance.

Difficult lighting settings like this can be handled more effectively by constructing paths that start from the eye on one end, from the light on the other end, and are connected in the middle with a visibility ray. This *bidirectional path tracing* algorithm is a generalization of the standard path tracing algorithm; for the same amount of computation, it can give substantially lower variance. XXX say something about adjoint algorithms in general XXX

XXX p_0 versus p_1 XXX

The path integral LTE makes it easy to understand how to construct a bidirectional algorithm. As with standard path tracing, the first vertex, p_1, is found by computing the first intersection along the camera ray, and the last vertex is found by sampling a point on a light source in the scene. Here we will label the last vertex as q_1, so that we can construct a path of not-initially-determined length “backward” from the light.

In the basic bidirectional algorithm, we go forward from the eye to create a subpath $p_1, p_2, \ldots, p_i$ and backward from the light to compute a subpath $q_1, q_2, \ldots, q_j$. Each sub-path is usually computed incrementally by sampling the BSDF at the previous vertex, though other sampling approaches can be used in the same way as was described for standard path tracing. (Weights for each vertex are computed in the same manner as well.) In either case, in the end, we have a path

$$\bar{p} = p_1, \ldots, p_i, q_j, \ldots, q_1.$$

We need to trace a shadow ray between p_i and q_j to make sure they are mutually visible; if so, the path carries light from the light to the camera and we can evaluate the path’s contribution directly.

There are three refinements to the basic algorithm that improve its performance in practice. The first two are analogous to improvements made to path tracing.

- First, we will re-use sub-paths: given a path $p_1, \ldots, p_i, q_j, \ldots, q_1$, we will evaluate transport over all of the paths generated by connecting all the various combinations of prefixes of the two paths together. If the two paths have

i and j vertices, respectively, then a total of $i \cdots j$ unique paths can be constructed from them, ranging in length from 2 to $i + j$ vertices long. XXX number of paths of length $n =$ ... XXX. Each such path built this way only requires that a visibility check be performed by tracing a shadow ray between the last vertices of each of the sub-paths.

- The second optimization is to ignore the paths generated in the path-reuse stage that only use one vertex from the light sub-path and instead to use the optimized direct lighting code that we developed for the direct lighting integrator. This gives a lower-variance result than using the vertex on the light sampled for the light sub-path, since it allows us to both use multiple importance sampling with the BSDF and to use stratified sampling patterns.

- The third optimization, left as an exercise, is to use multiple importance sampling to re-weight paths. Recall the example of a light pointed up at the ceiling, indirectly illuminating a room. As described so far, bidirectional path tracing will improve the result substantially by greatly reducing the number of paths with no contribution, since the paths from the light will be effective at finding those light transport routes. However, the image will still suffer from variance due to paths with excessively large contributions, for example from paths from the eye that happened to find the bright spot in the ceiling. We can apply MIS, recognizing that for a path with n vertices, there are actually $n - 1$ ways we could generate a path with that length–e.g. a 4 vertex path could have been built from one eye vertex and three light vertices, two of each kind of vertex, or three eye vertices and one light vertex. Given a particular path sampled in a particular way, we can compute the weights for each of the other ways the path could have been generated and apply the balance heuristic.

563 SurfaceIntegrator

```
⟨Bidirectional Local Declarations⟩+≡
  class BidirIntegrator : public SurfaceIntegrator {
  public:
      ⟨BidirIntegrator Public Methods⟩
  private:
      ⟨BidirIntegrator Private Methods⟩
      ⟨BidirIntegrator Data⟩
  };
```

⟨*Bidirectional Method Definitions*⟩≡

```
void BidirIntegrator::RequestSamples(Sample *sample, const Scene *scene) {
    for (int i = 0; i < MAX_VERTS; ++i) {
        eyeBSDFOffset[i] = sample->Add2D(1);
        eyeBSDFCompOffset[i] = sample->Add1D(1);
        lightBSDFOffset[i] = sample->Add2D(1);
        lightBSDFCompOffset[i] = sample->Add1D(1);
        directLightOffset[i] = sample->Add2D(1);
        directLightNumOffset[i] = sample->Add1D(1);
        directBSDFOffset[i] = sample->Add2D(1);
        directBSDFCompOffset[i]  = sample->Add1D(1);
    }
    lightNumOffset = sample->Add1D(1);
    lightPosOffset = sample->Add2D(1);
    lightDirOffset = sample->Add2D(1);
}
```

⟨*BidirIntegrator Data*⟩≡

```
#define MAX_VERTS 4
int eyeBSDFOffset[MAX_VERTS], eyeBSDFCompOffset[MAX_VERTS];
int lightBSDFOffset[MAX_VERTS], lightBSDFCompOffset[MAX_VERTS];
int directLightOffset[MAX_VERTS], directLightNumOffset[MAX_VERTS];
int directBSDFOffset[MAX_VERTS], directBSDFCompOffset[MAX_VERTS];
int lightNumOffset, lightPosOffset, lightDirOffset;
```

BidirIntegrator	591
RayDifferential	37
Sample::Add1D()	240
Sample::Add2D()	240
Scene	8
Spectrum	181

⟨*Bidirectional Method Definitions*⟩+≡

```
Spectrum BidirIntegrator::L(const Scene *scene,
        const RayDifferential &ray,
        const Sample *sample, Float *alpha) const {
    Spectrum L(0.);
    ⟨Generate eye and light sub-paths⟩
    ⟨Connect bidirectional path prefixes and evaluate throughput⟩
    return L;
}
```

XXX should use `sample` here, etc...

⟨*Generate eye and light sub-paths*⟩≡

```
BidirVertex eyePath[MAX_VERTS], lightPath[MAX_VERTS];
int nEye = generatePath(scene, ray, sample, eyeBSDFOffset,
    eyeBSDFCompOffset, eyePath, MAX_VERTS);
if (nEye == 0) {
    *alpha = 0.;
    return L;
}
*alpha = 1;
⟨Choose light for bidirectional path⟩
⟨Sample ray from light source to start light path⟩
int nLight = generatePath(scene, lightRay, sample, lightBSDFOffset,
    lightBSDFCompOffset, lightPath, MAX_VERTS);
```

⟨*Choose light for bidirectional path*⟩≡

```
int lightNum = Floor2Int(sample->oneD[lightNumOffset][0] *
    scene->lights.size());
lightNum = min(lightNum, (int)scene->lights.size() - 1);
Light *light = scene->lights[lightNum];
Float lightWeight = Float(scene->lights.size());
```

XXX use good sample pattern here XXX

⟨*Sample ray from light source to start light path*⟩≡

```
Ray lightRay;
Float lightSampleWeight;
bool deltaLight;
Float u[4];
u[0] = sample->twoD[lightPosOffset][0];
u[1] = sample->twoD[lightPosOffset][1];
u[2] = sample->twoD[lightDirOffset][0];
u[3] = sample->twoD[lightDirOffset][1];
Spectrum Le = light->Sample_L(scene, u[0], u[1], u[2], u[3],
    &lightRay, &lightSampleWeight, &deltaLight);
Le = lightWeight / lightSampleWeight;
```

⟨*Bidirectional Local Declarations*⟩+≡

```
struct BidirVertex {
    BidirVertex() { bsdfWeight = dAWeight = 0.; rrWeight = 1.;
        flags = BxDFType(0); bsdf = NULL; }
    BSDF *bsdf;
    Point p;
    Normal ng, ns;
    Vector wi, wo;
    Float bsdfWeight, dAWeight, rrWeight;
    BxDFType flags;
};
```

⟨*Bidirectional Method Definitions*⟩+≡

```
int BidirIntegrator::generatePath(const Scene *scene, const Ray &r,
        const Sample *sample, const int *bsdfOffset,
        const int *bsdfCompOffset,
        BidirVertex *vertices, int maxVerts) const {
    int nVerts = 0;
    RayDifferential ray(r.o, r.d);
    while (nVerts < maxVerts) {
        ⟨Find next vertex in path and initialize vertices⟩
        ++nVerts;
        ⟨Possibly terminate bidirectional path sampling⟩
        ⟨Initialize ray for next segment of path⟩
    }
    ⟨Initialize additional values in vertices⟩
    return nVerts;
}
```

⟨*Find next vertex in path and initialize* `vertices`⟩≡

```
Intersection isect;
if (!scene->Intersect(ray, &isect))
    break;
BidirVertex &v = vertices[nVerts];
v.bsdf = isect.GetBSDF(ray); // do before Ns is set!
v.p = isect.dg.p;
v.ng = isect.dg.nn;
v.ns = v.bsdf->dgShading.nn;
v.wi = -ray.d;
```

⟨*Possibly terminate bidirectional path sampling*⟩≡

```
if (nVerts > 2) {
    Float rrProb = .2f;
    if (RandomFloat() > rrProb)
        break;
    v.rrWeight = 1.f / rrProb;
}
```

⟨*Initialize* `ray` *for next segment of path*⟩≡

```
Float u1 = sample->twoD[bsdfOffset[nVerts-1]][0];
Float u2 = sample->twoD[bsdfOffset[nVerts-1]][1];
Float u3 = sample->oneD[bsdfCompOffset[nVerts-1]][0];
Spectrum fr = v.bsdf->Sample_f(v.wi, &v.wo, u1, u2, u3,
     &v.bsdfWeight, BSDF_ALL, &v.flags);
if (fr.Black())
    break;
ray = RayDifferential(v.p, v.wo);
```

Should cache brdf and g terms, etc...

⟨*Initialize additional values in* `vertices`⟩≡

```
for (int i = 0; i < nVerts-1; ++i)
    vertices[i].dAWeight = vertices[i].bsdfWeight *
        AbsDot(-vertices[i].wo, vertices[i+1].ng) /
        DistanceSquared(vertices[i].p, vertices[i+1].p);
```

⟨*Connect bidirectional path prefixes and evaluate throughput*⟩≡

```
Spectrum directWt(1.0);
for (int i = 1; i <= nEye; ++i) {
    ⟨Handle direct lighting for bidirectional integrator⟩
    for (int j = 1; j <= nLight; ++j)
        L += Le * evalPath(scene, eyePath, i, lightPath, j) /
            weightPath(eyePath, i, lightPath, j);
}
```

`AbsDot()`	31
`BidirIntegrator`	591
`irIntegrator::evalPath()`	595
`Integrator::weightPath()`	595
`BidirVertex`	593
`BidirVertex::bsdf`	593
`BidirVertex::ng`	593
`BidirVertex::ns`	593
`BidirVertex::p`	593
`BidirVertex::wi`	593
`BSDF::dgShading`	370
`BSDF::Sample_f()`	540
`BSDF_ALL`	334
`DifferentialGeometry::p`	58
`DistanceSquared()`	34
`Intersection`	131
`Intersection::dg`	131
`RandomFloat()`	679
`Ray`	36
`Ray::d`	35
`Ray::o`	35
`RayDifferential`	37
`Sample::oneD`	241
`Sample::twoD`	241
`Scene`	8
`Scene::Intersect()`	14
`Spectrum`	181
`Spectrum::Black()`	182

⟨*Handle direct lighting for bidirectional integrator*⟩≡

```
directWt /= eyePath[i-1].rrWeight;
L += directWt *
    UniformSampleOneLight(scene, eyePath[i-1].p, eyePath[i-1].ng, eyePath[i-1].wi,
    eyePath[i-1].bsdf, sample, directLightOffset[i-1], directLightNumOffset[i-1],
    directBSDFOffset[i-1], directBSDFCompOffset[i-1]) /
    weightPath(eyePath, i, lightPath, 0);
directWt *= eyePath[i-1].bsdf->f(eyePath[i-1].wi, eyePath[i-1].wo) *
    AbsDot(eyePath[i-1].wo, eyePath[i-1].ng) /
    eyePath[i-1].bsdfWeight;
```

⟨*Bidirectional Method Definitions*⟩+≡

```
Float BidirIntegrator::weightPath(BidirVertex *eye, int nEye,
        BidirVertex *light, int nLight) const {
    return Float(nEye + nLight);
}
```

XXX splatting for caustics, review indexing stuff carefully, etc...

XXX look out for specular stuff when we divide by weights!

⟨*Bidirectional Method Definitions*⟩+≡

```
Spectrum BidirIntegrator::evalPath(const Scene *scene, BidirVertex *eye, int nEye,
        BidirVertex *light, int nLight) const {
    Spectrum L(1.);
    for (int i = 0; i < nEye-1; ++i)
        L *= eye[i].bsdf->f(eye[i].wi, eye[i].wo) *
            AbsDot(eye[i].wo, eye[i].ng) /
            (eye[i].bsdfWeight * eye[i].rrWeight);
    Vector w = light[nLight-1].p - eye[nEye-1].p;
    L *= eye[nEye-1].bsdf->f(eye[nEye-1].wi, w) *
        G(eye[nEye-1], light[nLight-1]) *
        light[nLight-1].bsdf->f(-w, light[nLight-1].wi) /
        (eye[nEye-1].rrWeight * light[nLight-1].rrWeight);
    for (int i = nLight-2; i >= 0; --i)
        L *= light[i].bsdf->f(light[i].wi, light[i].wo) *
            AbsDot(light[i].wo, light[i].ng) /
            (light[i].bsdfWeight * light[i].rrWeight);
    if (L.Black())
        return L;
    if (!visible(scene, eye[nEye-1].p, light[nLight-1].p))
        return 0.;
    return L;
}
```

591	BidirIntegrator
596	BidirIntegrator::visible()
593	BidirVertex
373	BSDF::f()
8	Scene
181	Spectrum
182	Spectrum::Black()
27	Vector

⟨*Bidirectional Method Definitions*⟩+≡

```
Float BidirIntegrator::G(const BidirVertex &v0, const BidirVertex &v1) {
    Vector w = (v1.p - v0.p).Hat();
    return AbsDot(v0.ng, w) * AbsDot(v1.ng, -w) /
        DistanceSquared(v0.p, v1.p);
}
```

```
⟨Bidirectional Method Definitions⟩+≡
  bool BidirIntegrator::visible(const Scene *scene, const Point &P0,
          const Point &P1) {
      Ray ray(P0, P1-P0, RAY_EPSILON, 1.f - RAY_EPSILON);
      return !scene->IntersectP(ray);
  }
```

16.5 Irradiance Caching

Even with bidirectional path tracing, for some scenes it can take a large number of rays (and corresponding compute time) to generate images without objectionable noise. One approach to this problem has been the development of biased approaches to solving the LTE. These approaches generally reuse previously-computed results over multiple estimates, even when the values used don't estimate the precise quantity that needs to be computed (for example, reusing an illumination value from a nearby point under the assumption that illumination is slowly changing.) **XXX make interpolation idea more clear.** Irradiance caching, described in this section, and photon mapping, described in the next, have been two successful biased methods for light transport.

BidirIntegrator	591
BidirVertex	593
DistanceSquared()	34
Point	33
Ray	36
RAY_EPSILON	37
Scene	8
Vector	27
Vector::Hat()	30

By introducing bias, these methods produce images without the high-frequency noise artifacts that unbiased Monte Carlo techniques are prone to. They can often do so using relatively little additional computation compared to basic techniques like Whitted-style ray tracing. This efficiency comes at a price, however: one key characteristic of unbiased Monte Carlo techniques is that variance decreases in a predictable and well-characterized manner as more samples are taken. As such, if an image was computed with an unbiased technique and has no noise, we can be extremely confident that the image correctly represents the lighting in the scene. With a biased solution method, however, error estimates aren't well defined for the approaches that have been developed so far; if the image doesn't have visual artifacts, it still may have substantial error. And given an image with artifacts, increasing the sampling rate with a biased technique doesn't necessarily eliminate artifacts in a predictable way.

One of the first biased Monte Carlo light transport techniques that was developed is *irradiance caching*; it is based on the observation that while direct lighting often changes rapidly from point to point (e.g. consider a hard shadow edge), indirect lighting is usually slowly changing in many environments. Therefore, if we compute an accurate representation of indirect light at a sparse set of sample points in the scene and then interpolate nearby samples to compute an approximate representation of indirect light at any particular point being shaded, we can expect that the error introduced by not recomputing indirect lighting everywhere shouldn't be too bad and can enjoy a substantial computational savings.

There are immediately two issues that must be addressed in the design of an algorithm such as this one:

1. How do we represent and store the indirect lighting distribution after we have computed it at a point?

2. How often do we compute accurate representations of indirect light, and how often do we interpolate from already-existing samples?

In the irradiance caching algorithm, indirect lighting is computed on demand at a subset of the points that are shaded (as opposed to in a pre-process) and stored in a cache. When a point is being shaded, we first search the cache for one or more acceptable nearby samples, using a set of error metrics to be described later to determine if the already-existing samples are acceptable. In order to have a compact representation of indirect light, this algorithm only stores the irradiance at each point, rather than a directionally-varying radiance distribution, thus reducing the light representation to just a single `Spectrum`.

Recall that irradiance arriving at one side of a surface with normal **n** is

$$E(\mathrm{p},\mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L(\mathrm{p},\omega_i)|\cos\theta_i|\mathrm{d}\omega_i.$$

It is in a sense a weighted average of incoming radiance at a point, giving a sense of the aggregate illumination there. The reflection component of scattering equation,

$$L_\mathrm{o}(\mathrm{p},\omega_o) = \int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p},\omega_i)\, f_\mathrm{r}(\mathrm{p},\omega_o,\omega_i)\,|\cos\theta_i|\mathrm{d}\omega_i.$$

If the surface is Lambertian, the BSDF is constant, and we have

$$\begin{aligned} L_\mathrm{o}(\mathrm{p},\omega_o) &= c\int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p},\omega_i)\,|\cos\theta_i|\mathrm{d}\omega_i \\ &= cE(\mathrm{p},\mathbf{n}), \end{aligned}$$

where $E(\mathrm{p},\mathbf{n})$ is the irradiance at the point p around the hemisphere centered around the normal **n**, as defined in Equation 5.2.5. Thus, for perfectly diffuse materials, the irradiance alone is enough information to exactly compute the reflection from the surface due to a particular incident lighting distribution. Thus, another key assumption in the irradiance caching algorithm is that many of the surfaces in the scene are diffuse, or that other approaches will be used for the glossy and specular components of the BSDF.

If the surface is nearly Lambertian (e.g. has the Oren–Nayar BRDF or a glossy surface with a very wide specular lobe), we can instead view the irradiance caching algorithm as making the approximation

$$\begin{aligned} L_\mathrm{o}(\mathrm{p},\omega_o) &\approx \left(\int_{\mathcal{H}^2(\mathbf{n})} f_\mathrm{r}(\mathrm{p},\omega_o,\omega_i)\,\mathrm{d}\omega_i\right)\left(\int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p},\omega_i)\,|\cos\theta_i|\mathrm{d}\omega_i\right) \\ &= (\pi\rho_{dh}(\omega_o))\,E(\mathrm{p},\mathbf{n}) \end{aligned}$$

Where ρ_{dh} is the hemispherical-directional reflectance, introduced in Section 9.1.1. This error in this approximation increases as the variation of either of the two integrands increases. In particular, to the degree that if the incident light distribution is uniform, or the BRDF is Lambertian, there is less error. (For example, consider a perfectly specular surface, where only incident radiance from a single direction contributes to reflected radiance; using the irradiance and the hemispherical-directional reflectance to compute reflection for such a BSDF incorrectly includes the effect of radiance from all the other incident directions.)

```
⟨irradiancecache.cpp*⟩≡
  #include "lrt.h"
  #include "transport.h"
  #include "scene.h"
  #include "mc.h"
  #include "octree.h"
  ⟨IrradianceCache Forward Declarations⟩
  ⟨IrradianceCache Declarations⟩
  ⟨IrradianceCache Method Definitions⟩
```

```
⟨IrradianceCache Declarations⟩≡
  class IrradianceCache : public SurfaceIntegrator {
  public:
      ⟨IrradianceCache Public Methods⟩
  private:
      ⟨IrradianceCache Data⟩
      ⟨IrradianceCache Private Methods⟩
  };
```

SurfaceIntegrator 563

Our implementation takes parameters that give the maximum number of bounces of indirect light and specular reflection that the integrator will account for. (We use basic Whitted-style raytracing to account for perfect specular reflection.) The `maxError` value will be used to control how frequently we reuse irradiance samples versus compute a new sample, and `nSamples` controls how many rays are used to estimate the irradiance integral for each sample we compute. Finally, `specularDepth` tracks the current ray depth as we recursively call the integrator to account for multiply-scattered light.

```
⟨IrradianceCache Method Definitions⟩≡
  IrradianceCache::IrradianceCache(int maxspec, int maxind,
          Float maxerr, int ns, int nf) {
      maxError = maxerr;
      nSamples = ns;
      nFilter = min(nSamples-1, nf);
      maxSpecularDepth = maxspec;
      specularDepth = 0;
  }
```

```
⟨IrradianceCache Data⟩≡
  Float maxError;
  int nSamples, nFilter;
  int maxSpecularDepth;
  mutable int specularDepth;
```

We'll again reuse the direct lighting routines that were defined in the direct lighting integrator, so we will request that the `Sampler` provide us with an appropriate set of sample values to be used for those calls.

XXX do one for each level up to max specular depth?? XXX Need to do this elsewhere, too?

⟨*IrradianceCache Method Definitions*⟩+≡

```
void IrradianceCache::RequestSamples(Sample *sample,
        const Scene *scene) {
    u_int nLights = scene->lights.size();
    lightPositionOffset = sample->Add2D(nLights);
    bsdfDirectionoffset = sample->Add2D(nLights);
    bsdfComponentOffset = sample->Add1D(nLights);
}
```

⟨*IrradianceCache Data*⟩+≡

```
int lightPositionOffset;
int bsdfDirectionoffset, bsdfComponentOffset;
```

We won't include most of the irradiance cache's `L()` method here, but will just focus on its key two fragments, ⟨*Compute direct lighting for irradiance cache*⟩ and ⟨*Compute indirect lighting for irradiance cache*⟩.

For direct lighting, we use one of the functions from Section 16.1 to use multiple importance sampling for the direct lighting estimate. We sample all of the lights for the initial ray intersection and intersections from purely specular reflection to give a good result for points that are directly visible. However, if this is a recursive call to the `IrradianceCache` where we're computing reflected radiance for one of the sample rays that is being used to compute the irradiance estimate, we only sample a single light source for greater efficiency for those rays. Because so many rays are traced for a single irradiance estimate, the overall result is still reasonable.

Note also that we need to be careful to ignore emission from light sources that indirect sample rays happen to intersect directly; this ensures that we don't inadvertently include the effect of direct illumination in the irradiance estimate we are computing, since it is only supposed to account for the effects of indirect lighting.

132	`Intersection::Le()`
598	`IrradianceCache`
598	`IrradianceCache::maxSpecularDept`
598	`IrradianceCache::specularDepth`
240	`Sample::Add1D()`
240	`Sample::Add2D()`
8	`Scene`
9	`Scene::lights`
566	`UniformSampleAllLights()`
27	`Vector`

⟨*Compute direct lighting for irradiance cache*⟩≡

```
L += isect.Le(wo);
L += UniformSampleAllLights(scene, p, n, wo, bsdf, sample,
    lightPositionOffset, bsdfDirectionoffset, bsdfComponentOffset);
```

We also partition the BSDF for the indirect lighting computation. Perfect specular reflection is handled by sampling the BSDF and recursively calling the integrator, just as the `WhittedIntegrator` does. The implementation here uses irradiance caching for both the diffuse and glossy components of the BSDF, thus introducing error for the glossy components. (An exercise at the end of the chapter describes generalizing this integrator to sample from the glossy parts of the BSDF and recursively call the integrator for those rays instead.)

⟨*Compute indirect lighting for irradiance cache*⟩≡

```
if (specularDepth++ < maxSpecularDepth) {
    Vector wi;
    ⟨Trace rays for specular reflection and refraction⟩
}
--specularDepth;
⟨Estimate indirect lighting with irradiance cache⟩
```

We need to handle reflection and transmission individually here, possibly computing two independent irradiance values since the irradiance from the hemisphere

$\mathcal{H}^2(\mathbf{n})$, which is needed for reflective surfaces, and the hemisphere $\mathcal{H}^2(-\mathbf{n})$, which is needed for transmissive surfaces, are completely different. The `IndirectReflectedL()` method handles either case, interpolating a value using the cache or computing a new value if necessary. We will reorient the normal here so that it points in the same hemisphere that the ω_o vector is in; `IndirectReflectedL()` depends on this convention when it generates sample rays over the hemisphere, since it will ensure that all of these rays are in the same hemisphere as `n`.

⟨*Estimate indirect lighting with irradiance cache*⟩≡

```
Normal ng = isect.dg.nn;
if (Dot(wo, ng) < 0.f) ng = -ng;
BxDFType flags = BxDFType(BSDF_REFLECTION | BSDF_DIFFUSE | BSDF_GLOSSY);
L += IndirectReflectedL(p, ng, wo, bsdf, flags, sample, scene);
flags = BxDFType(BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
L += IndirectReflectedL(p, -ng, wo, bsdf, flags, sample, scene);
```

XXX think we need a π factor in final mult, but apparently not? Review rho computations again... Lambertian vs. Oren Nayar, etc.. XXX

If the `InterpolateIrradiance()` method isn't able to find enough nearby irradiance samples of good enough quality, we go ahead and compute a new sample and add it to the cache.

⟨*IrradianceCache Method Definitions*⟩+≡

```
Spectrum IrradianceCache::IndirectReflectedL(const Point &p,
        const Normal &n, const Vector &wo, BSDF *bsdf,
        BxDFType flags, const Sample *sample,
        const Scene *scene) const {
    if (bsdf->NumComponents(flags) == 0) return Spectrum(0.);
    Spectrum E;
    if (!InterpolateIrradiance(scene, p, n, &E)) {
        ⟨Compute irradiance at current point⟩
        ⟨Add computed irradiance value to cache⟩
    }
    return E * bsdf->rho(wo, flags);
}
```

BSDF	370
BSDF::NumComponents()	371
BSDF::rho()	373
BSDF_DIFFUSE	334
BSDF_GLOSSY	334
BSDF_REFLECTION	334
BSDF_TRANSMISSION	334
BxDFType	334
Intersection::dg	131
IrradianceCache	598
:InterpolateIrradiance()	605
Normal	34
Point	33
Scene	8
Spectrum	181
Vector	27

Before describing how irradiance values are stored, looked-up, and interpolated, first we'll discuss how new estimates are computed; this will make more clear how some of the details of reuse work later. We need to estimate the value of the integral

$$E(\mathrm{p},\mathbf{n}) = \int_{\mathcal{H}^2(\mathbf{n})} L_\mathrm{i}(\mathrm{p},\omega_i)\,|\cos\theta_i|\mathrm{d}\omega_i. \tag{16.5.4}$$

Because there is no easy available way to importance sample based on the distribution of incident radiance, we will just use a cosine weighted distribution of directions. We are then faced with the problem of computing the amount of radiance along each one $L_\mathrm{i}(\mathrm{p},\omega_i)$. This is naturally handled with a recursive call to `IrradianceCache::L()` via `Scene::L()`. Because we need to account for indirect illumination at intersection points from these rays, we will again have irradiance cache lookups there and possibly new irradiance estimates to be computed.

We are thus de facto computing each radiance value with a forward path-tracing algorithm, using the irradiance cache for indirect illumination at each vertex. We could alternatively have used standard path tracing without irradiance caching, or even bidirectional path tracing to compute these values. The improved robustness from using bidirectional path tracing (for example, for indirect lighting due to a spotlight shining at a small area on a ceiling) to compute these values would be a good improvement to the basic algorithm we have implemented here.

Before we start tracing rays to compute the irradiance estimate, we initialize a pair of random values to use to scramble the two dimensions of a low-discrepancy point sequence that we will map to cosine-weighted directions over the hemisphere. (See Section 7.5.3 for an explanation of how we randomly scramble point sequences so that we use a different set of sample values each time, while still preserving the good distribution properties of the point set.) We also initialize the `sumInvDists` variable to zero; it accumulates the sum of one over the distance each sample ray travels before intersecting an object; we will use this value later to help estimate how widely reusable the irradiance estimate is.

⟨*Compute irradiance at current point*⟩≡

```
  u_int scramble[2] = { RandomUInt(), RandomUInt() };
  Float sumInvDists = 0.;
  vector<Spectrum> Lirrad;
  for (int i = 0; i < nSamples; ++i) {
       ⟨Trace ray to sample radiance for irradiance estimate⟩
  }
  ⟨Filter radiance values and compute irradiance estimate⟩
```

370	BSDF
598	IrradianceCache::nSamples
586	PathIntegrator
248	RandomUInt()
181	Spectrum
658	vector

To compute the irradiance estimate, we will use the standard Monte Carlo estimator

$$E(\mathrm{p},\mathbf{n}) = \frac{1}{N}\sum_j \frac{L_\mathrm{i}(\mathrm{p},\omega_j)|\cos\theta_j|}{p(\omega_j)}.$$

Because we are generating rays with a cosine weighted distribution, $p(\omega) = \cos\theta/\pi$, so we have

$$\frac{1}{N}\sum_j \frac{L_\mathrm{i}(\mathrm{p},\omega_j)|\cos\theta_j|}{|\cos\theta_j|/\pi} = \frac{\pi}{N}\sum_j L_\mathrm{i}(\mathrm{p},\omega_j).$$

`CosineSampleHemisphere()` returns a direction in the canonical reflection coordinate system, with the normal direction mapped to the $+z$ axis, based on a 2D sample value from a randomized low-discrepancy $(0,2)$-sequence, as introduced in Section 7.5. To get a world-space ray direction, we can use the convenient method `LocalToWorld()` method from the `BSDF` class. Finally, we may need to flip the ray around so that it lies in the same hemisphere as the normal that was passed in. The radiance along the sample ray is computed with path tracing. The fragment that does this computation, ⟨*Do path tracing to compute radiance along ray for estimate*⟩, is not included here, since it is essentially the same as the code in the `PathIntegrator`.

```
⟨Trace ray to sample radiance for irradiance estimate⟩≡
  ⟨Update irradiance statistics for rays traced⟩
  Float u[2];
  Sample02Net(i+1, scramble, u);
  Vector w = CosineSampleHemisphere(u[0], u[1]);
  RayDifferential r(Ray(p, bsdf->LocalToWorld(w), RAY_EPSILON));
  if (Dot(r.d, n) < 0) r.d = -r.d;
  ⟨Do path tracing to compute radiance along ray for estimate⟩
  sumInvDists += 1.f / (r.maxt * r.d.Length());
```

Figure XXX shows an image of the Sponza atrium scene, rendered with the irradiance caching integrator. Something as clearly gone wrong, since the bright splotchy artifacts are not expected given the model and lighting environment. Investigation showed that the unexpectedly bright irradiance estimates have roughly five times as much energy as their neighbors others, but that this variation wasn't due to variance due to too few rays being traced–4,096 were used for the irradiance estimates in the figure, which should have been plenty.

The root cause of the bright irradiance estimates was that the reflectances of surfaces in this model are unexpectedly high–around 0.9, which is higher to the maximum reflectance ever seen in realistic materials, which is roughly 0.8 for bright white paint. Rock surfaces like these would be expected to have a much lower reflectance. Due to the Russian roulette algorithm used for path termination, the path throughput of surviving paths tends to grow larger with successive bounces: for example, assuming diffuse surfaces, cosine-weighted sampling, and Russian roulette with a path termination of 0.5 starting after the third bounce, the path throughput for an i bounce path will be

$$\frac{0.9^i}{0.5^{i-3}}.$$

For a fifteen bounce path, this value is roughly 843. If the randomness in the Russian roulette computations work out so that a handful of paths for a particular estimate are this long and if a number of them are able to see the light source, the estimate has the potential to have a much larger value than a neighboring estimate where paths happened to be terminated sooner.

BSDF::LocalToWorld()	372
CosineSampleHemisphere()	517
Ray	36
Ray::maxt	36
RAY_EPSILON	37
RayDifferential	37
Sample02Net()	262
Vector	27

From a strict Monte Carlo perspective, there is no bug in the code; while the blotches are unsightly, they only reflect variance in the estimator. The bright estimates are no more wrong than the rest of the estimates. This is a slightly unsatisfying argument, as is the claim that the splotches reflect an error in the scene description and that light transport in scenes with high reflectance like this one isn't important to handle well.

One solution to this problem would be to adjust the Russian roulette algorithm, for example by using a lower termination probability. Alternatively, paths could just be terminated without question after a particular number of bounces. Instead, we will solve this problem here by adding a bit more bias to the algorithm. A direct solution to the bright splotches is to simply ignore very bright radiance values from the rays used to compute the irradiance estimate; if one is many times brighter than all of the others, it is likely that it represents a high variance spike. Here, we will optionally ignore the k brightest radiance values, where k is a user-supplied parameter. The application of techniques like these to rendering is often called *filtering*, since they were first applied to filtering pixels in images with very high values due to variance spikes.

Fortunately, there are linear-time partition algorithms that take an unsorted array of *n* values and reorder them such that the *k*th element of the array holds the *k*th largest value, all smaller values are placed before the *k*th entry and all larger values are after it. The C++ standard library's `nth_element()` function does just this.

⟨*Filter radiance values and compute irradiance estimate*⟩≡
```
if (nFilter > 0)
    std::nth_element(&Lirrad[0], &Lirrad[nSamples-nFilter], &Lirrad[nSamples]);
for (int i = 0; i < nSamples - nFilter; ++i)
    E += Lirrad[i];
E *= M_PI / Float(nSamples - nFilter);
```

The bottom image in Figure XXX shows the result of ignoring the 8 brightest radiance values out of the 4,096 used for computing irradiance estimates; the result is substantially more visually acceptable.

So that we can efficiently search for all of the already-computed irradiance estimates around a point in the scene, we use an octree data structure to store the estimates. The `Octree` template class, which is described in Appendix A.5, recursively splits a given bounding box into subregions, refining the current region into eight sub-regions at each level of the tree by dividing the box in half at the mid point of its extent along the *x*, *y*, and *z* axes. Each irradiance estimate has an axis-aligned bounding box associated with it, giving the overall area for which it may be a valid sample. The octree uses the extent of this box as a guide for an appropriate level of refinement at which to store the sample. Later, given a point to lookup nearby irradiance samples for, the octree just needs to traverse the nodes that the point is inside (and there is one such node at each level of the tree) and provide the samples overlapping those nodes to be considered for interpolation (see Figure 16.8).

598 `IrradianceCache::maxError`
598 `IrradianceCache::nSamples`
605 `IrradianceCache::octree`
605 `IrradianceSample`
678 `M_PI`
682 `Octree::Add()`

If the user set the `maxError` parameter to zero, we don't bother storing the sample in the octree and instead recompute the irradiance at each point in the scene. Otherwise, we compute a maximum region of influence that the irradiance estimate could potentially contribute to and provide this to the octree.

⟨*Add computed irradiance value to cache*⟩≡
```
⟨Update statistics for new irradiance sample⟩
if (maxError > 0.) {
    ⟨Compute bounding box of irradiance sample's contribution region⟩
    octree->Add(IrradianceSample(E, p, n, maxDist),
        sampleExtent);
}
```

Having taken a set of radiance samples, we'd like to estimate over how wide an area we can reuse the irradiance at other points without introducing too much error. For example, indirect irradiance at a point in the middle of the ceiling of a room is likely to be changing more slowly as a function of position than it is at the edge of the ceiling where the wall meets it. In general, the more objects that are close to the sample point, the greater potential there is for rapidly-changing irradiance. Therefore, we compute the harmonic mean of the distance each of the sample rays

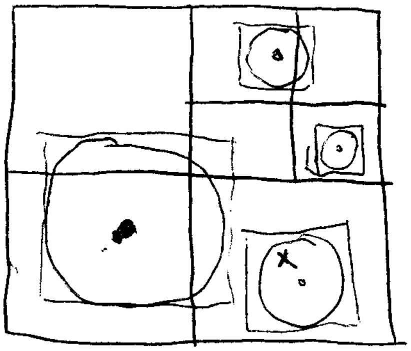


Figure 16.8: Example of irradiance sample storage in 2D (with a quadtree, rather than an octree). Each irradiance sample, denoted by a dot, has a maximum distance over which it potentially can contribute irradiance, denoted here by a circle. Samples are stored in the tree nodes that the overlap, and the tree is refined adaptively so that each sample is stored in a small number of nodes. Given a point at which we want to look up nearby irradiance estimates, here shown with an "x", we just need to traverse the tree nodes that the point overlaps, considering all of the irradiance samples stored in these nodes.

BBox	38
BBox::Expand()	40
Clamp()	677
rradianceCache::nSamples	598
IrradProcess::maxError	606

travelled before intersecting an object,

$$\frac{N}{\sum^N 1/d_i},$$

where d_i is the distance that the ith ray travelled. This will serves as an upper bound on the area of influence for the estimate.

⟨*Compute bounding box of irradiance sample's contribution region*⟩≡

```
static Float minMaxDist = .001 * powf(scene->WorldBound().Volume(), 1./3.);
static Float maxMaxDist = .125 * powf(scene->WorldBound().Volume(), 1./3.);
Float maxDist = nSamples / sumInvDists;
if (minMaxDist > 0.f)
    maxDist = Clamp(maxDist, minMaxDist, maxMaxDist);
maxDist *= maxError;
BBox sampleExtent(p);
sampleExtent.Expand(maxDist);
```

Each irradiance estimate is represented by an instance of the `IrradianceSample` structure, which just holds th relevant pieces of information. Its constructor, not included here, initializes the member variables with the values passed to it.

⟨*IrradianceCache Declarations*⟩+≡

```
struct IrradianceSample {
    ⟨IrradianceSample Constructor⟩
    Spectrum E;
    Normal n;
    Point p;
    Float maxDist;
};
```

The octree is allocated in the `Preprocess()` method since the scene is available to us then and we can find its overall extent, which is needed by the `Octree` constructor. We expand the bound by a small amount in each direction so that the octree can gracefully deal with the fact that some of the irradiance samples and some of the lookup points will be marginally outside the scene bounds due to floating-point error from ray intersection computations.

⟨*IrradianceCache Method Definitions*⟩+≡

```
void IrradianceCache::Preprocess(const Scene *scene) {
    BBox wb = scene->WorldBound();
    Vector delta = .01 * (wb.pMax - wb.pMin);
    wb.pMin -= delta;
    wb.pMax += delta;
    octree = new Octree<IrradianceSample, IrradProcess>(wb);
}
```

⟨*IrradianceCache Data*⟩+≡

```
mutable Octree<IrradianceSample, IrradProcess> *octree;
```

Now we can define the method that attempts t compute an interpolated irradiance value at a point in the scene using cached values. Much of the work is done by the `Octree::Lookup()` method, which traverses the nodes of the octree that the given point is inside and calls a method of the `IrradProcess` object for each `IrradianceSample` in each of these nodes. This `IrradProcess` method decides if each sample is acceptable and accumulates the value of the interpolated result.

38	BBox
39	BBox::pMax
39	BBox::pMin
598	IrradianceCache
606	IrradProcess
608	IrradProcess::GetIrradiance()
607	IrradProcess::Successful()
34	Normal
681	Octree
685	Octree::Lookup()
33	Point
8	Scene
14	Scene::WorldBound()
181	Spectrum
27	Vector

⟨*IrradianceCache Method Definitions*⟩+≡

```
bool IrradianceCache::InterpolateIrradiance(const Scene *scene,
        const Point &p, const Normal &n, Spectrum *E) const {
    if (!octree) return false;
    IrradProcess proc(n, maxError);
    octree->Lookup(p, proc);
    ⟨Update irradiance cache lookup statistics⟩
    if (!proc.Successful()) return false;
    *E = proc.GetIrradiance();
    return true;
}
```

`IrradProcess` stores additional information about the point being shaded that we'll need for deciding whether irradiance samples can be used at that point as well as information about the interpolated value that is being computed. Its constructor initializes `n` and `maxError` with the values passed in and zeros the rest of its members.

```
⟨IrradianceCache Declarations⟩+≡
  struct IrradProcess {
      ⟨IrradProcess Public Methods⟩
      Normal n;
      Float maxError;
      mutable int nFound, samplesChecked;
      mutable Float sumWt;
      mutable Spectrum E;
  };
```

This IrradProcess method is called for each IrradianceSample in an octree node that the lookup point is inside. It is given both the original point p that was passed to Octree::Lookup() above as well as an irradiance sample from the octree. It first performs a series of tests that may reject the sample as not being acceptable. If the sample passes these tests, we compute a value that tries to approximate the error from using the sample at the shading point, which we compare to the user-supplied error limit.

```
⟨IrradianceCache Method Definitions⟩+≡
  void IrradProcess::operator()(const Point &p,
          const IrradianceSample &sample) const {
      ++samplesChecked;
      ⟨Skip irradiance sample if surface normals are too different⟩
      ⟨Skip irradiance sample if it's too far from the sample point⟩
      ⟨Skip irradiance sample if it's in front of point being shaded⟩
      ⟨Compute estimate error term and possibly use sample⟩
  }
```

IrradianceSample	605
rradianceSample::maxDist	605
IrradianceSample::n	605
IrradianceSample::p	605
Normal	34
Point	33
Spectrum	181

If the surface normal of the lookup point and the normal used when computing the irradiance estimate are substantially different (Figure 16.9, right), the hemisphere of directions that determines their irradiance values will be different enough that it's unlikely that the sample will accurately represent the actual irradiance.

```
⟨Skip irradiance sample if surface normals are too different⟩≡
  if (Dot(n, sample.n) < 0.01f)
      return;
```

We next make sure that the point being shaded isn't too far from the sample. This check is redundant given the way we compute the error metric below, but it gives us an early out before the expensive square-root and division operations that we will use there.

```
⟨Skip irradiance sample if it's too far from the sample point⟩≡
  Float d2 = DistanceSquared(p, sample.p);
  if (d2 > sample.maxDist * sample.maxDist)
      return;
```

We next check to see if the irradiance sample is in front of the lookup point (see Figure 16.9, left). If so, the sample might have a very large maxDist value, reflecting an expected slowly-changing indirect irradiance, while the lookup point might be close to a corner or other geometric feature that causes irradiance to actually be changing more quickly there.

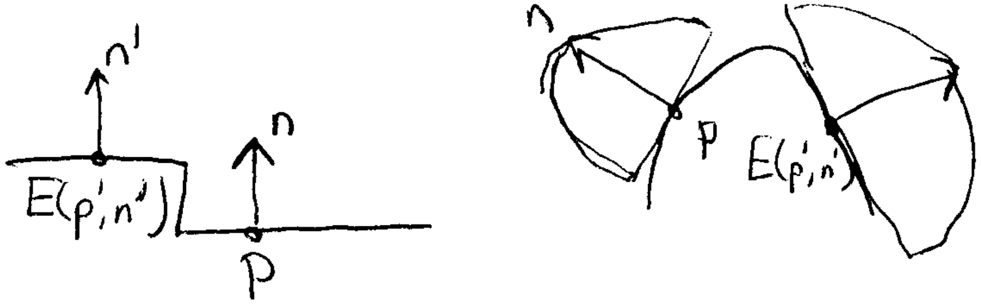


Figure 16.9: Rejection tests for samples in the irradiance cache.

⟨*Skip irradiance sample if it's in front of point being shaded*⟩≡

```
Normal navg = sample.n + n;
if (Dot(p - sample.p, navg) > .01)
    return;
```

If the sample has passed these four tests, we compute a numerical estimate of the expected error from including the sample in our irradiance value. The expression we use to do this is ad-hoc, but it captures the key ideas that erorr should increase with distance between the sample and the point being shaded and should increase as their normal vectors diverge. If this is less than the user-supplied error limit, we compute a weight for this irradiance value, giving it more weight if its error is low, and add the irradiance to a running sum of interpolated irradiances.

605	IrradianceSample::E
605	IrradianceSample::maxDist
605	IrradianceSample::n
605	IrradianceSample::p
606	IrradProcess::E
606	IrradProcess::n
606	IrradProcess::nFound
606	IrradProcess::sumWt
34	Normal

⟨*Compute estimate error term and possibly use sample*⟩≡

```
Float err = sqrtf(d2) / (sample.maxDist * Dot(n, sample.n));
if (err < 1.) {
    ++nFound;
    Float wt = (1.f - err) * (1.f - err);
    E += wt * sample.E;
    sumWt += wt;
}
```

When we are done traversing the octree and processing candidate samples, we need to decide if we have computed an acceptable interpolated irradiance value from the irradiance samples. We will require that at least three acceptable samples were found; ensuring that multiple samples were used helps to smooth out the result.

⟨*IrradProcess Public Methods*⟩+≡

```
bool Successful() {
    return (sumWt > 0. && nFound > 0);
}
```

The final interpolated irradiance value is a weighted sum of the irradiance values of the acceptable estimates,

$$E = \frac{\sum_i w_i E_i}{\sum_i w_i}.$$

```
⟨IrradProcess Public Methods⟩+≡
  Spectrum GetIrradiance() const { return E / sumWt; }
```

16.6 Particle Tracing and Photon Mapping

Photon mapping is another biased technique for solving the LTE. Unlike irradiance caching, it handles both glossy and diffuse reflection well; perfectly specular reflection is handled separately with recursive ray tracing. Photon mapping is one of a family of *particle tracing* algorithms, which are based on the idea of constructing paths from the lights. At each vertex of the path, the amount of incident illumination arriving at the vertex is recorded. After a certain number of these illumination samples have been computed, a data structure that stores a representation of the distribution of light in the scene is built, and at rendering time, this representation is used to compute values of measurements needed to compute the image. Because the particle tracing step is decoupled from computing the measurements, many measurements may be able to reuse the work done for a single particle path, thus leading to more efficient rendering algorithms.

In this section, we will start by introducing a theory of particle tracing algorithms and will discuss the conditions that must be fulfilled by a particle tracing algorithm so that arbitrary measurements an be computed correctly using the particles created by the algorithm. We will then describe an implementation of a photon mapping integrator that uses particles to estimate illumination by interpolating lighting contributions from particles around the point being shaded.

IrradProcess::E	606
IrradProcess::sumWt	606
Spectrum	181

16.6.1 Theoretical basis for particle tracing

Particle tracing algorithms in computer graphics are typically explained in terms of packets of energy being shot from the light sources in the scene that deposit energy at surfaces they intersect before scattering in new directions. This is an intuitive way of thinking about particle tracing, but the intuition that it provides doesn't make it easy to answer basic questions about how propagation and scattering affect the particles. For example, does their contribution fall off with squared distance like flux density? Which $\cos\theta$ terms, if any, affect the particles after they scatter from a surface?

In order to give a solid theoretical basis for particle tracing, we will describe it with a framework introduced by Veach (Veach 1997, Appendix 4.A), which instead interprets the stored particle histories as samples of the scene's equilibrium radiance distribution. Under certain conditions on the distribution and weights of the particles, the particles can be used to compute estimates of nearly any measurements based on the light distribution in the scene. In this framework, it is quite easy to answer questions about the details of particle propagation like the ones above.

A particle tracing algorithm generates a set of N samples of illumination at points p_j, on surfaces in the scene

$$(\mathrm{p}_\mathrm{j}, \omega_j, \alpha_j),$$

where each sample records incident illumination from direction ω_j and has some weight α_j associated with it. We would like to determine the conditions on the weights such that we can use them to compute estimates of arbitrary measurements.

Given an importance function $W_e(p,\omega)$ that describes the measurement to be taken, the condition we would like to fulfill is that the particles should be distributed and weighted such that using them to compute an estimate has the same expected value as the measurement equation for the same importance function:

$$E\left[\frac{1}{N}\sum_{j=1}^{N}\alpha_j W_e(p_j,\omega_j)\right] = \int_{A^2}\int_{\mathcal{S}^2} W_e(p,\omega)L_i(p,\omega)\,dA\,d\omega. \tag{16.6.5}$$

For example, we might want to use the particles to compute the total flux on a wall. Using the definition of flux,

$$\Phi = \int_{A_{\text{wall}}}\int_{\mathcal{H}^2(\mathbf{n})} L_i(p,\omega)\,|\cos\theta|\,dA\,d\omega,$$

the corresponding importance function selects the particles that lie on the wall and arrived from the hemisphere around the normal.

$$W_e(p,\omega) = \left\{ \begin{array}{lcl} 1 & : & \text{p is on wall surface} \\ 0 & : & \text{otherwise} \end{array} \right. \times \max((\omega\cdot\mathbf{n}),0)$$

If the particle weights and distribution is true for arbitrary importance functions, as in Equation 16.6.5, then our flux estimate can be computed directly as is just a sum of the appropriate particle weights multiplied by the $\cos\theta$ term. If we want to estimate flux over a different wall, a subset of the original wall, etc., we only need to recompute the weighted sum with an updated importance function; the particles and weighs can be re-used.

To see how to generate and weight particles that fulfill these conditions, consider the task of evaluating the measurement equation integral

$$\begin{aligned}\int_A\int_{\mathcal{S}^2} W_e(p_0,\omega)L(p_0,\omega)\,d\omega\,dA(p_0) = \\ \int_A\int_A W_e(p_0\to p_1)L(p_1\to p_0)G(p_0\leftrightarrow p_1)\,dA(p_0)\,dA(p_1)\end{aligned}$$

where the importance function that describes the measurement is a black box and thus cannot be used to drive the sampling of the integral at all. We can still compute an estimate of the integral, but must sample a set of points p_0 and p_1 from all of the surfaces in the scene, using some sampling distribution that doesn't depend on W_e (e.g. by uniformly sampling points by surface area). Expanding the LTE in the integrand and applying the standard Monte Carlo estimator for N samples, we can find the estimator for the measurement

$$E\left[\frac{1}{N}\sum_{i=1}^{N} W_e(p_{i,0}\to p_{i,1})\left\{\frac{L(p_{i,1}\to p_{i,0})G(p_{i,0}\leftrightarrow p_{i,1})}{p(p_{i,0})p(p_{i,1})}\right\}\right].$$

We can further expand out the L term into the sum over paths and use the fact that $E[ab] = E[aE[b]]$. For a particular sample, the expected value $E[\frac{L(p_{i,1}\to)}{p_{i,0}}]$ can be written as a finite sum of n_i terms in just the same way that we generated a finite set of weighted path vertices for path tracing. If the probability of continuing the sum after j terms is $q_{i,j}$, then the jth term of the ith sample has contribution

$$\frac{L_e(p_{n_i}\to p_{n_i-1})}{p(p_{n_i})}\prod_{j=1}^{n_i-1}\frac{1}{q_{i,j}}\frac{f(p_{i,j+1}\to p_{i,j}\to p_{i,j-1})G(p_{i,j+1}\leftrightarrow p_{i,j})}{p(p_{i,j})}.$$

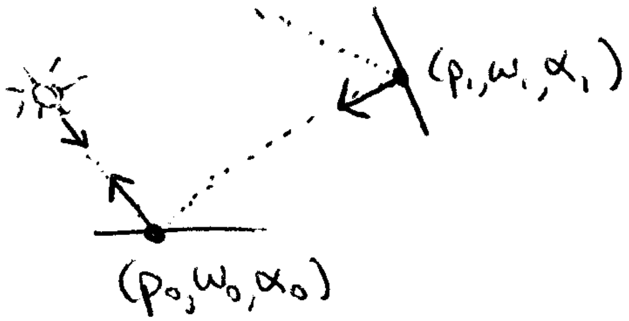


Figure 16.10: photon particle history

Looking back at Equation 16.6.5, we can see that this value gives the appropriate value of the particle weights.

$$\alpha_{i,j} = \frac{L_\mathrm{e}(\mathrm{p}_{\mathrm{n_i}} \to \mathrm{p}_{\mathrm{n_i}-1})}{p(\mathrm{p}_{\mathrm{n_i}})} \prod_{j=1}^{n_i-1} \frac{1}{q_{i,j}} \frac{f(\mathrm{p}_{\mathrm{i,j+1}} \to \mathrm{p}_{\mathrm{i,j}} \to \mathrm{p}_{\mathrm{i,j-1}}) G(\mathrm{p}_{\mathrm{i,j+1}} \leftrightarrow \mathrm{p}_{\mathrm{i,j}})}{p(\mathrm{p}_{\mathrm{i,j}})}.$$

XXX review the $p(\mathrm{p})$ stuff, general indexing, etc... XXXX

Mention joint distribution

$$\int \alpha p(\alpha, \mathrm{p}, \omega) d\alpha = L(\mathrm{p}, \omega)$$

Intuition about interplay between number of particles and their weights...

If we only had a single measurement to make, it would be better if we used information about W_e and could compute the estimate more intelligently, since the general particle tracing approach described here may generate many useless samples if W_e only covers a small subset of the points on scene objects. If we will be computing many measurements, however, the key advantage that particle tracing brings is that we can generate the samples and weights once, and can then reuse them over a large number of measurements, potentially computing results much more efficiently than if the measurements were all computed from scratch.

Note that we have the freedom to generate a set of particles with these weights in all sorts of different ways; although the natural approach is to start from point on lights and incrementally sample paths using the BSDFs at the path vertices, similarly to how the path tracing integrator generates paths (starting here from the light, rather than from the camera), we could generate them with any number of different sampling strategies, so long as there was non-zero probability of generating a particle at any point where W_e was non-zero and the particle weights were computed appropriately for the sampling distribution used.

XXX In tricky settings, can be tough to get enough photons to the part of the scene you're actually looking at... This is the flip side to the problem of finding the small reflected light source on the ceiling when only tracing rays from the eye. XXX

16.6.2 Photon Integrator

The photon mapping integrator traces particles into the scene as described above and interpolates among particles to approximate the incident illumination at shading points. For consistency with other descriptions of the algorithm, we will refer to particles generated for photon mapping as photons. We use a kd-tree data here to store the photons; it allows us to quickly find the photons around the point being shaded. The kd-tree is the *photon map*. Because the kd-tree is decoupled from the scene geometry, this algorithm isn't limited to a particular set of types of geometric representations. (In contrast to using a texture map defined over shapes' (u,v) parameterizations to store illumination, for instance.)

Photon mapping partitions the LTE in a number of different ways that make it easier to adjust the solution quality. For example, particles from the lights are characterized as being one of three types: direct illumination–light that has arrived directly at a surface, without any scattering, caustic illumination–light that has arrived at a non-specular surface after reflecting from one or more specular surfaces, and indirect illumination, which covers all other types of illumination. Thanks to this partitioning, the integrator has a fair amount of flexibility in how it estimates reflected radiance. For instance, it might use the indirect and caustic photons to estimate reflection due to those modes of light transport, but sample the light sources directly for direct illumination. As mentioned previously, this integrator also partitions the BSDF: perfect specular components are always handled with recursive ray tracing, while either the photon maps or further Monte Carlo ray tracing is used for the rest of it.

563 SurfaceIntegrator

⟨*photonmap.cpp**⟩≡

```
#include "lrt.h"
#include "transport.h"
#include "scene.h"
#include "mc.h"
#include "kdtree.h"
#include "sampling.h"
⟨Photonmap Local Declarations⟩
⟨Photonmap Method Definitions⟩
```

⟨*Photonmap Local Declarations*⟩+≡

```
class PhotonIntegrator : public SurfaceIntegrator {
public:
    ⟨PhotonIntegrator Public Methods⟩
private:
    ⟨PhotonIntegrator Private Methods⟩
    ⟨PhotonIntegrator Private Data⟩
};
```

Quite a few different parameters control the operation of the integrator. The user must specify a desired number of photons of each type–caustic, direct, and indirect–to store in each of the three types of photon map. More photons increases the quality of results, but takes more time and memory. Because this integrator interpolates nearby photons to estimate illumination at the shading point, the user can also set how many photons are used for the interpolation. The more that are used, the smoother the illumination estimate will be, since a larger number of photons will be used to reconstruct it. If too many are used, the result will tend to be too blurry, while too few gives a splotchy appearance. Usually 50 to 100 is a good choice. Finally, the integrator can be configured to do a one-bounce sampling of indirect illumination by sampling the BSDF and tracing rays, rather than using the indirect photon map directly. This process, which will be described in more detail later, is known as final gathering.

We won't include the implementation of the `PhotonIntegrator` here, since it just directly initializes its member variables from the parameters passed to it. Its member variables are:

- `nCausticPhotons`, `nDirectPhotons`, and `nIndirectPhotons` give the total number of photons we'd like to store for each category of illumination stored in the photon map.
- `nLookup` gives the total number of photons to try to use for the interpolation step.
- `maxDistSquared` gives the maximum allowed squared distance from the point being shaded to a photon that can be used for the interpolation there. If its value is too large, we will waste time searching for nearby photons, while if it's too small, we may not be able to find `nLookup` nearby photons, leading to an overly-splotchy result.
- `specularDepth` and `maxSpecularDepth` track the current and maximum values of specular reflection, similarly to the Whitted integrator.
- `directWithPhotons` determines whether direct illumination is computed with the photon map or by tracing shadow rays to sample the lights.
- `finalGather` controls if final gathering is used for indirect lighting rather than using the indirect map directly; if true, `gatherSamples` controls the number of samples taken.

⟨*PhotonIntegrator Private Data*⟩≡

```
u_int nCausticPhotons, nIndirectPhotons, nDirectPhotons;
u_int nLookup;
mutable int specularDepth;
int maxSpecularDepth;
Float maxDistSquared;
bool directWithPhotons, finalGather;
int gatherSamples;
int nFilter;
```

XXX here and elsewhere round 2D stuff to power of 2 for LD sampling stuff? XXX

The `PhotonIntegrator::RequestSamples()` method is also quite similar to the others and so is not included in the text; it requests samples for multiple importance sampling for direct lighting, and, if final gathering has been enabled, a well-distributed set of samples for sampling the BSDF for the final gather step.

⟨*PhotonIntegrator Private Data*⟩+≡

```
int lightPositionOffset;
int bsdfDirectionoffset, bsdfComponentOffset;
int gatherSampleOffset, gatherComponentOffset;
```

16.6.3 Building the photon maps

When the integrator's `Preprocess()` method is called, we follow particle paths through the scene until we have accumulated the desired number of particle histories to build the three photon maps. At each intersection of the path with an object, a weighted photon is stored if the map for the corresponding type of illumination is not yet full. `Photon` objects represent such an illumination sample; the contents of this structure will be defined in a few pages.

⟨*Photonmap Method Definitions*⟩+≡

```
void PhotonIntegrator::Preprocess(const Scene *scene) {
    if (scene->lights.size() == 0) return;
    vector<Photon> causticPhotons, directPhotons, indirectPhotons;
    ⟨Initialize photon shooting statistics⟩
    bool causticDone = (nCausticPhotons == 0);
    bool directDone = (nDirectPhotons == 0);
    bool indirectDone = (nIndirectPhotons == 0);
    while (!causticDone || !directDone || !indirectDone) {
        ++nshot;
        ⟨Give up if we're not storing enough photons⟩
        ⟨Trace a photon path and store contribution⟩
    }
}
```

617	`Photon`
611	`PhotonIntegrator`
8	`Scene`
661	`StatsCounter`
658	`vector`

We keep track of the total number of paths generated in `nshot`. One one hand, we may need to shoot many more photons than are stored, for example due to photons that leave the scene without intersecting any objects. On the other hand, each path may contribute multiple `Photon` sample values as it bounces around the scene.

⟨*Initialize photon shooting statistics*⟩≡

```
static StatsCounter nshot("Photon Map",
    "Number of photons shot from lights");
```

If we find that we have generated many paths while storing few to no photons of some types, we eventually give up and exit without creating the corresponding kd-trees. (For example, this might happen if we were trying to populate a caustic map but there weren't any specular objects in the scene to create caustic paths.)

⟨*Give up if we're not storing enough photons*⟩≡

```
if (nshot > 500000 &&
    (unsuccessful(nCausticPhotons, causticPhotons.size(), nshot) ||
     unsuccessful(nDirectPhotons, directPhotons.size(), nshot) ||
     unsuccessful(nIndirectPhotons, indirectPhotons.size(), nshot))) {
    Error("Unable to store enough photons.  Giving up.\n");
    return;
}
```

⟨*PhotonIntegrator Private Methods*⟩≡

```
static bool unsuccessful(int needed, int found, int shot) {
    return (found < needed && (found == 0 || shot / found > 1000));
}
```

To create a new path, we start by sampling a ray from one of the lights in the scene and then follow the path, recording particle intersections as it bounces around the scene. The `alpha` variable is incrementally updated to store the path contribution at each vertex. After path termination, we free the BSDF memory allocated for the path before going on to start the next one.

⟨*Trace a photon path and store contribution*⟩≡

```
⟨Choose 4D sample values for photon⟩
⟨Choose light to shoot photon from⟩
⟨Generate photonRay from light source and initialize alpha⟩
if (!alpha.Black()) {
    ⟨Follow photon path through scene and record intersections⟩
}
BSDF::FreeAll();
```

Since we would like the samples used to generate ray directions to be well-distributed, but we don’t know ahead of time how many paths will need to be generated to get the desired number of particle histories, we’ll use a Halton sequence for these sample values. Recall from Section 7.5 that any number of sequential points starting from the beginning of the Halton sequence have good low-discrepancy properties. Using a Halton sequence here gives a more uniform distribution of photons throughout the scene than uniform random points, for example.

`BSDF`	370
`BSDF::FreeAll()`	374
`Light`	478
`RadicalInverse()`	254
`Scene::lights`	9
`Spectrum::Black()`	182

⟨*Choose 4D sample values for photon*⟩≡

```
Float u[4];
u[0] = RadicalInverse(nshot+1, 2);
u[1] = RadicalInverse(nshot+1, 3);
u[2] = RadicalInverse(nshot+1, 5);
u[3] = RadicalInverse(nshot+1, 7);
```

We choose among the lights in the scene with equal probability, using the Halton point from the next dimension to select among them. As with the direct lighting integrator, one could imagine more effective sampling strategies, such as sampling according to lights’ total power or changing sampling probabilities adaptively based on the contributions of the particles that they emit. If such a strategy was used, we’d just need to update the computation of `lightPdf` accordingly.

⟨*Choose light to shoot photon from*⟩≡

```
int nLights = int(scene->lights.size());
int lightNum = min(Floor2Int(nLights * RadicalInverse(nshot+1, 11)),
    nLights-1);
Light *light = scene->lights[lightNum];
Float lightPdf = 1.f / nLights;
```

We can now sample a ray from the light source and initialize its α value with

$$\frac{L_{\mathrm{e}}(\mathrm{p}_0, \omega_0)}{p(\mathrm{p}_0, \omega_0)},$$

where $p(\mathrm{p}_0, \omega_0)$ is the product of the pdf for sampling this particular light and the pdf for sampling this particular ray leaving the light.

⟨*Generate* `photonRay` *from light source and initialize* `alpha`⟩≡

```
RayDifferential photonRay;
Float pdf;
Spectrum alpha = light->Sample_L(scene, u[0], u[1], u[2], u[3],
    &photonRay, &pdf);
alpha /= pdf * lightPdf;
```

And now we start following the path though the scene, updating α after each scattering event and recording photons at the path vertices. The `specularPath` variable records whether the path we are following has only intersected perfectly specular surfaces after leaving the light source, which indicates that the path is a caustic path.

⟨*Follow photon path through scene and record intersections*⟩≡

```
bool specularPath = false;
Intersection photonIsect;
int nIntersections = 0;
while (scene->Intersect(photonRay, &photonIsect)) {
    ++nIntersections;
    ⟨Handle photon/surface intersection⟩
    ⟨Sample new photon ray direction⟩
    ⟨Possibly terminate photon path⟩
}
```

Given a photon-surface intersection, the first thing we need to do is to update whichever of the three photon maps is the appropriate one for storing this particle's contribution. However, if the photon has hit a perfectly specular surface, there's no need to record it in any photon map, since we don't use photons for computing reflection from specular surfaces at render-time.

XXX hmm, does that break some deep-seated assumptions that density estimation is based on?? XXX

⟨*Handle photon/surface intersection*⟩≡

```
alpha *= scene->Transmittance(photonRay);
Vector wo = -photonRay.d;
BSDF *photonBSDF = photonIsect.GetBSDF(photonRay);
BxDFType specularType = BxDFType(BSDF_REFLECTION |
    BSDF_TRANSMISSION | BSDF_SPECULAR);
bool hasNonSpecular = (photonBSDF->NumComponents() >
    photonBSDF->NumComponents(specularType));
if (hasNonSpecular) {
    ⟨Deposit photon at surface⟩
}
```

If this is the first intersection found after the particle has left the light source, then the photon represents direct illumination. Otherwise, if it has only reflected from specular surfaces before arriving at the current intersection point, it must be a caustic photon. Any other case–a path that only hit non-specular surfaces, or a path that hit a series of specular surfaces before scattering from non-specular

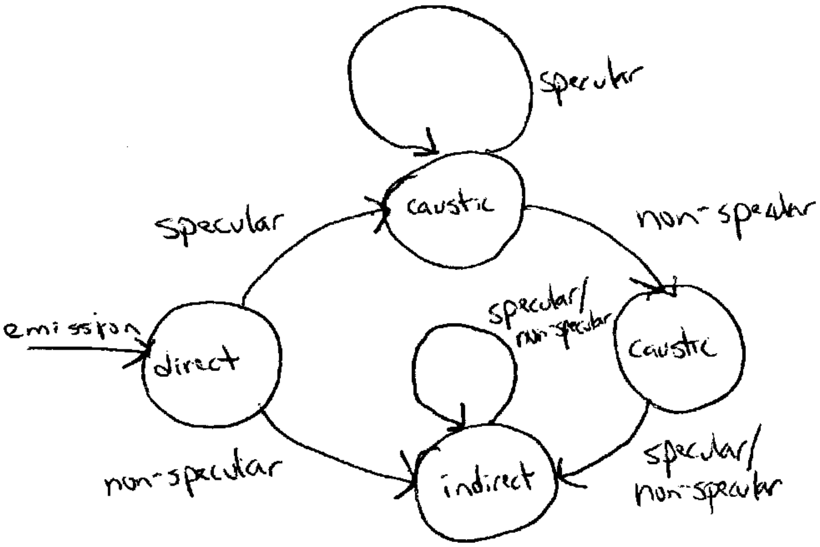


Figure 16.11: photon fsm

DifferentialGeometry::p	58
Intersection::dg	131
Photon	617

surfaces–represents indirect illumination. The finite state machine in Figure 16.11 illustrates these ideas; the nodes of the graph show which of the photon maps should be updated for a photon in that state, and the edges describe whether the photon has scattered from a specular or a non-specular BSDF component at its previous intersection.

⟨*Deposit photon at surface*⟩≡

```
  Photon photon(photonIsect.dg.p, alpha, wo);
  if (nIntersections == 1) {
      ⟨Process direct lighting photon intersection⟩
  }
  else if (specularPath) {
      ⟨Process caustic photon intersection⟩
  }
  else {
      ⟨Process indirect lighting photon intersection⟩
  }
```

The fragments for processing the three types of photon map updates all have equivalent functionality, so we only have the fragment for direct lighting photons here. Until enough photons of a particular type have been found, the corresponding photons are stored in the corresponding `vector` of `Photon`s declared above. When the desired number are available, we go ahead and construct a kd-tree (the `KdTree` template class we use is described in Appendix A.6.) When the tree is built and we have decided to ignore any additional photons of this type that are found while following paths to fill up the unfilled maps, we also record how many paths from the lights needed to be constructed before we found the desired number of photons. This value will be important for the density estimation algorithm used to interpolate among photons around a point being shaded.

⟨*Process direct lighting photon intersection*⟩≡

```
if (!directDone) {
    directPhotons.push_back(photon);
    if (directPhotons.size() == nDirectPhotons) {
        directDone = true;
        nDirectPaths = nshot;
        directMap = new KdTree<Photon, PhotonProcess>(directPhotons);
    }
}
```

⟨*PhotonIntegrator Private Data*⟩+≡

```
int nCausticPaths, nDirectPaths, nIndirectPaths;
mutable KdTree<Photon, PhotonProcess> *causticMap;
mutable KdTree<Photon, PhotonProcess> *directMap;
mutable KdTree<Photon, PhotonProcess> *indirectMap;
```

The `Photon` structure stores just enough information to record a photon's contribution– the position where it hit the surface, its weight, and the direction it arrived from.

⟨*Photonmap Local Declarations*⟩+≡

```
struct Photon {
    ⟨Photon Constructor⟩
    Point p;
    Spectrum alpha;
    Vector wi;
};
```

688	KdTree
623	PhotonProcess
33	Point
181	Spectrum
27	Vector

⟨*Photon Constructor*⟩≡

```
Photon(const Point &pp, const Spectrum &wt, const Vector &w)
    : p(pp), alpha(wt), wi(w) {
}
```

Now that we have recorded the particle's contribution in one of the photon maps, we need to choose a new outgoing direction from the intersection point and update the α value to account for the effect of the BSDF scattering the incident illumination at the surface. Equation 16.6.5 shows how to incrementally update the particle weight after a scattering event: we have some weight αi,j that represents the weight for the *j*th intersection of the *i*th particle history. After a scattering event where a new vertex $\mathrm{p}_{\mathrm{i,j+1}}$ has been sampled, it should be scaled by

$$\frac{1}{q_{i,j+1}} \frac{f(\mathrm{p}_{\mathrm{i,j+1}} \to \mathrm{p}_{\mathrm{i,j}} \to \mathrm{p}_{\mathrm{i,j-1}}) G(\mathrm{p}_{\mathrm{i,j+1}} \leftrightarrow \mathrm{p}_{\mathrm{i,j}})}{p(\mathrm{p}_{\mathrm{i,j+1}})}.$$

As with the path tracing integrator, we'd like to choose the next vertex in the path by sampling the BSDF at the intersection point, rather than directly sampling by area on the scene surfaces. Therefore, we again apply the Jacobian to account for this change in measure, all of the terms in $G(e \leftrightarrow x)$ cept for a single $|\cos\theta|$ cancel out, and the scaling term is

$$\frac{1}{q_{i,j+1}} \frac{f(\mathrm{p}_{\mathrm{i,j+1}} \to \mathrm{p}_{\mathrm{i,j}} \to \mathrm{p}_{\mathrm{i,j-1}}) G(\mathrm{p}_{\mathrm{i,j+1}} \leftrightarrow \mathrm{p}_{\mathrm{i,j}})}{p(\mathrm{p}_{\mathrm{i,j+1}})}.$$

As the very first intersection, the random numbers used for sampling are the next two dimensions of the point in the Halton sequence that was used to start this path, ensuring a good distribution of directions at the first bounces across all of the paths. For subsequent bounces, we just use uniform random numbers, as the advantages of further low discrepancy points matter less as more bounces occur.

```
⟨Sample new photon ray direction⟩≡
  Vector wi;
  Float pdf;
  BxDFType flags;
  ⟨Get random numbers for sampling outgoing photon direction⟩
  Spectrum fr = photonBSDF->Sample_f(wo, &wi, u1, u2, u3,
      &pdf, BSDF_ALL, &flags);
  if (fr.Black() || pdf == 0.f)
      break;
  specularPath = (nIntersections == 1 || specularPath) &&
      bool(flags & BSDF_SPECULAR);
  alpha *= fr * AbsDot(wi, photonBSDF->dgShading.nn) / pdf;
  photonRay = RayDifferential(photonIsect.dg.p, wi);
```

BSDF::dgShading	370
BSDF::Sample_f()	540
BSDF_ALL	334
BSDF_SPECULAR	334
BxDFType	334
DifferentialGeometry::p	58
Intersection::dg	131
RandomFloat()	679
RayDifferential	37
Spectrum	181
Spectrum::Black()	182
Vector	27

As usual, after the first few bounces, Russian roulette either terminates the path or increases its weight to account for the missing contributions of the paths that were terminated.

```
⟨Possibly terminate photon path⟩≡
  if (nIntersections > 3) {
      Float continueProbability = .5f;
      if (RandomFloat() > continueProbability)
          break;
      alpha /= continueProbability;
  }
```

16.6.4 Using the photon map

XXX make clear the changing illumination error problem–e.g. a small wall, etc. XXX

At rendering-time, the photon map is used to compute reflected light at each point being shaded. We'd like to estimate the value of the scattering equation at a point p, which can equivalently (and cumbersomely) be written as an integral over all points on surfaces in the scene

$$\int_{\mathcal{S}^2} L_\mathrm{i}(\mathrm{p},\omega_i)\, f(\mathrm{p},\omega_o,\omega_i)\,|\cos\theta_i| \mathrm{d}\omega_i = \int_A \int_{\mathcal{S}^2} \delta(\mathrm{p}-\mathrm{p}') L_\mathrm{i}(\mathrm{p}',\omega_i)\, f_{(}(\mathrm{p}',\omega_o,\omega_i)\,|\cos\theta_i| \mathrm{d}\omega_i \mathrm{d}A(\mathrm{p}'),$$

and so the function that describes the measurement we need to compute is

$$W_\mathrm{e}(\mathrm{p}',\omega) = \delta(\mathrm{p}'-\mathrm{p})\, f(\mathrm{p},\omega_o,\omega)\,|\cos\theta|.$$

Unfortunately, because there is a delta distribution in W_e, the particle histories that were generated without accounting for this have zero probability of having non-zero contribution when Equation 16.6.5 is used to compute the estimate of the

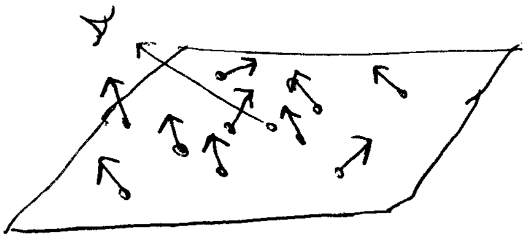


Figure 16.12: photon basic

measurement value (just as we will never be able to choose a direction from a surface that intersects a point light source unless the direction is sampled accounting for this.)

Here is the point at which bias is introduced into the photon mapping algorithm. Under the assumption that the information about illumination at nearby points can be used to construct an estimate of illumination at the shading point, photon mapping interpolates information about illumination at the point being shaded from nearby photons. The more photons there are around the point and the higher their weights, the more light we estimate is illuminating the point. The estimated illumination at the point is used in conjunction with the surface's BSDF to compute the reflected light; Figure 16.12 shows the basic idea.

The error introduced by this interpolation can be difficult to quantify. Storing more photons, so that it isn't necessary to use photons as far away, will almost always improve the results, but in general, error will depend on how quickly the illumination is changing at the point being shaded. Because this interpolation step tends to blur out illumination, high-frequency changes in lighting are least-well reconstructed with photon mapping.

In order to reduce the blurriness in final images from this interpolation, the photon mapping algorithm partitions the integrand and uses photons to evaluate some parts of it, but traces rays to estimate the rest. A number of different partitionings have been proposed; here we will split the BSDF into delta and non-delta components, and will separate incident radiance from the non-delta component into incident direct ($L_{\mathrm{i},d}$), indirect ($L_{\mathrm{i},i}$), and caustic ($L_{\mathrm{i},c}$) illumination:

$$\begin{aligned}\int_{S^2} L_\mathrm{i}(\mathrm{p},\omega_i)\, f(\mathrm{p},\omega_o,\omega_i)\,|\cos\theta_i| \mathrm{d}\omega_i = \\ \int_{S^2} L_\mathrm{i}(\mathrm{p},\omega_i)\, f_\Delta(\mathrm{p},\omega_o,\omega_i)\,|\cos\theta_i| \mathrm{d}\omega_i + \\ \int_{S^2} (L_{\mathrm{i},d}(\mathrm{p},\omega_i) + L_{\mathrm{i},i}(\mathrm{p},\omega_i) + L_{\mathrm{i},c}(\mathrm{p},\omega_i))\, f_{\neg\Delta}(\mathrm{p},\omega_o,\omega_i)\,|\cos\theta_i| \mathrm{d}\omega_i.\end{aligned}$$

We will elide the complete implementation of `PhotonIntegrator::L()`, which is similar in form to the `L()` methods of the other integrators and will focus on the two key fragments that do the direct and indirect lighting computations, respectively. When these fragments run, the specular BSDF components have already been handled in the elided code, so we only need to consider the non-delta reflection case here.

The delta components of the BSDF can be efficiently and accurately handled with recursive ray tracing. For the non-delta components, it is usually worth sampling direct lighting $L_{\mathrm{i},d}$ by tracing shadow rays to the light sources. Photon maps for direct lighting are normally only useful for large area light sources casting smooth shadows and for quickly rendering preview images. A parameter to the integrator selects between these two options. The `LPhoton()` function, which will be described shortly, computes reflected radiance in the direction `wo` for the given BSDF and photon map.

⟨*Compute direct lighting for photon map integrator*⟩≡

```
if (directWithPhotons)
    L += LPhoton(directMap, nDirectPaths, nLookup,
        bsdf, isect, wo, maxDistSquared,
        nFilter);
else
    L += UniformSampleAllLights(scene, p, n, wo, bsdf, sample,
        lightPositionOffset, bsdfDirectionoffset,
        bsdfComponentOffset);
```

`tor::bsdfComponentOffset`	613
`tor::bsdfDirectionoffset`	613
`onIntegrator::causticMap`	617
`tonIntegrator::directMap`	617
`nIntegrator::finalGather`	612
`nIntegrator::indirectMap`	617
`tor::lightPositionOffset`	613
`tegrator::maxDistSquared`	612
`ntegrator::nCausticPaths`	617
`Integrator::nDirectPaths`	617
`hotonIntegrator::nFilter`	612
`tegrator::nIndirectPaths`	617
`hotonIntegrator::nLookup`	612
`UniformSampleAllLights()`	566

We will always use the caustic photon map to account for light paths that have one or more specular bounces before hitting a non-specular surface. These paths are particularly difficult to find when tracing paths starting from the point being shaded; because caustics are focused light, there tend to be enough photons to compute good lighting estimates in areas with caustics.

For some scenes, indirect lighting can be represented well (and computed extremely efficiently) with the photon map. If higher quality is needed, we also provide a method known as *final gathering*. At each shading point, we sample the BSDF and trace rays out into the scene to find incident radiance along those rays. However, at the intersection points of the gather rays, we always use the direct, indirect, and caustic maps to compute the reflection there. Thus, we don't pay any cost for tracing additional rays to shade those points. Because the error from the interpolation of photons there isn't seen directly at the original point being shaded, the final result can have very high quality.

⟨*Compute indirect lighting for photon map integrator*⟩≡

```
L += LPhoton(causticMap, nCausticPaths, nLookup, bsdf,
    isect, wo, maxDistSquared,
    nFilter);
if (finalGather) {
    ⟨Do one-bounce final gather for photon map⟩
}
else
    L += LPhoton(indirectMap, nIndirectPaths, nLookup,
        bsdf, isect, wo, maxDistSquared,
        nFilter);
```

For final gathering, we need to estimate the reflected radiance due to indirect illumination,

$$\int_{S^2} f(\mathrm{p},\omega_o,\omega_i) L_{\mathrm{i},i}(\mathrm{p},\omega_i) |\cos\theta_i| \mathrm{d}\omega_i.$$

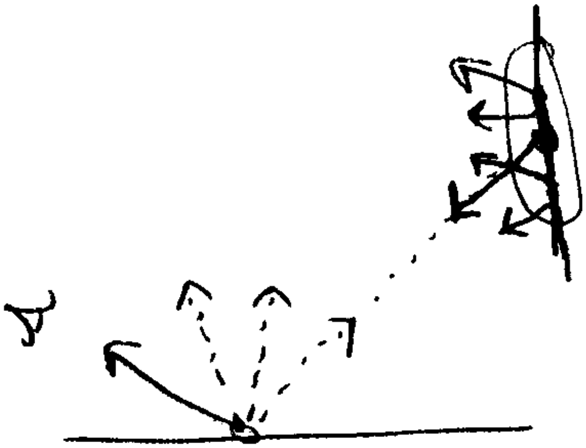


Figure 16.13: photon gathering

In our implementation here, we will always use the BSDF's importance sampling method to choose sampled directions. (The further reading section has pointers to a number of techniques that use the incident directions of the photons that are close to p to derive importance sampling techniques that try to match the distribution $L_{\mathrm{i},i}$; using both of these sampling methods along with multiple importance sampling can substantially improve the efficiency of final gathering.) Our estimator is

131	Intersection
612	PhotonIntegrator::gatherSamples
37	RayDifferential
14	Scene::Intersect()
181	Spectrum

$$\frac{1}{N}\sum_{j=1}^{N}\frac{f(\mathrm{p},\omega_o,\omega_j)L_{\mathrm{i},i}(\mathrm{p},\omega_j)|\cos\theta_j|}{p(\omega_j)}.$$

⟨*Do one-bounce final gather for photon map*⟩≡

```
Spectrum Li(0.);
for (int i = 0; i < gatherSamples; ++i) {
    ⟨Sample random direction for final gather ray⟩
    RayDifferential bounceRay(p, wi);
    Intersection gatherIsect;
    if (scene->Intersect(bounceRay, &gatherIsect)) {
        ⟨Compute reflected radiance due to final gather sample⟩
    }
}
L += Li / gatherSamples;
```

⟨*Sample random direction for final gather ray*⟩≡

```
Vector wi;
Float u1 = sample->twoD[gatherSampleOffset][2*i];
Float u2 = sample->twoD[gatherSampleOffset][2*i+1];
Float u3 = sample->oneD[gatherComponentOffset][i];
Float pdf;
BxDFType flags;
Spectrum fr = bsdf->Sample_f(wo, &wi, u1, u2, u3, &pdf, BSDF_ALL,
    &flags);
if (fr.Black() || pdf == 0.f || (flags & BSDF_SPECULAR)) continue;
```

Having found the first point of intersection of the ray from p in direction ω_j, we need to compute the outgoing radiance at that point in direction $-\omega_j$, which gives the incident radiance at p, $L_\mathrm{i}(\mathrm{p},\omega) = L_\mathrm{o}(t(\mathrm{p},\omega),-\omega)$.

Note we ignore L_e–i.e. the direct illumination term for the point we're shading..

⟨*Compute reflected radiance due to final gather sample*⟩≡

```
BSDF *gatherBSDF = gatherIsect.GetBSDF(bounceRay);
Vector bounceWo = -bounceRay.d;
Spectrum Lindir =
    LPhoton(directMap, nDirectPaths, nLookup,
        gatherBSDF, gatherIsect, bounceWo, maxDistSquared,
        nFilter) +
    LPhoton(indirectMap, nIndirectPaths, nLookup,
        gatherBSDF, gatherIsect, bounceWo, maxDistSquared,
        nFilter) +
    LPhoton(causticMap, nCausticPaths, nLookup,
        gatherBSDF, gatherIsect, bounceWo, maxDistSquared,
        nFilter);
Li += fr * Lindir / pdf;
```

16.6.5 Photon interpolation and density estimation

The `LPhoton()` function has two main tasks: first, it needs to find the `nLookup` photons that are closest to the point where the looking is being done. Second, it needs to use those photons to compute the reflected radiance at the point.

⟨*Photonmap Method Definitions*⟩+≡

```
Spectrum PhotonIntegrator::LPhoton(
        KdTree<Photon, PhotonProcess> *map,
        int nPaths, int nLookup, BSDF *bsdf,
        const Intersection &isect, const Vector &wo,
        Float maxDistSquared, int nFilter) {
    Spectrum L(0.);
    if (!map) return L;
    BxDFType nonSpecular = BxDFType(BSDF_REFLECTION |
        BSDF_TRANSMISSION | BSDF_DIFFUSE | BSDF_GLOSSY);
    if (bsdf->NumComponents(nonSpecular) == 0)
        return L;
    ⟨Initialize PhotonProcess object, proc, for photon map lookups⟩
    ⟨Do photon map lookup⟩
    return L;
}
```

Similar to the octree used for the irradiance cache, the `KdTree` used to store photons takes a parameter to its `Lookup()` method that defines an object that is called back for every item in the search region. `PhotonProcess` handles that for us here, storing information about the nearby photons that have been passed to it using the `ClosePhoton` structure. We use `alloca()` to efficiently allocate space for the array of`ClosePhoton`s, rather than repeatedly calling `new` each time this method is called.

BSDF	370
BSDF::NumComponents()	371
BSDF::Sample_f()	540
BSDF_ALL	334
BSDF_DIFFUSE	334
BSDF_GLOSSY	334
BSDF_REFLECTION	334
BSDF_SPECULAR	334
BSDF_TRANSMISSION	334
BxDFType	334
Intersection	131
Intersection::GetBSDF()	375
KdTree	688
Photon	617
PhotonIntegrator	611
onIntegrator::causticMap	617
tonIntegrator::directMap	617
r::gatherComponentOffset	613
ator::gatherSampleOffset	613
nIntegrator::indirectMap	617
tegrator::maxDistSquared	612
ntegrator::nCausticPaths	617
Integrator::nDirectPaths	617
hotonIntegrator::nFilter	612
tegrator::nIndirectPaths	617
hotonIntegrator::nLookup	612
PhotonProcess	623
Ray::d	35
Sample::oneD	241
Sample::twoD	241
Spectrum	181
Spectrum::Black()	182
Vector	27

⟨*Initialize* `PhotonProcess` *object,* `proc`*, for photon map lookups*⟩≡

```
  PhotonProcess proc(nLookup, isect.dg.p);
  proc.photons = (ClosePhoton *)alloca(nLookup * sizeof(ClosePhoton));
```

⟨*Do photon map lookup*⟩≡

```
  map->Lookup(isect.dg.p, proc, maxDistSquared);
  ⟨Accumulate light from nearby photons⟩
```

In addition to keeping track of the nearby photons `PhotonProcess` also needs to record the position at the lookup point, so that it can determine the distance from photons that are passed to it to the shading point.

⟨*Photonmap Local Declarations*⟩+≡

```
  struct PhotonProcess {
      ⟨PhotonProcess Public Methods⟩
      const Point &p;
      ClosePhoton *photons;
      u_int nLookup;
      mutable u_int foundPhotons;
  };
```

To keep track of a photon close to the lookup point, we only need to store a pointer to it. However, we also cache the squared distance from the photon to the lookup point in order to more quickly be able to discard the farthest away photon when a closer one is found.

58	`DifferentialGeometry::p`
131	`Intersection::dg`
691	`KdTree::Lookup()`
617	`Photon`
612	`PhotonIntegrator::nLookup`
33	`Point`

⟨*Photonmap Local Declarations*⟩+≡

```
  struct ClosePhoton {
      ClosePhoton(const Photon *p, Float md2) {
          photon = p;
          distanceSquared = md2;
      }
      bool operator<(const ClosePhoton &p2) const {
          return distanceSquared < p2.distanceSquared;
      }
      const Photon *photon;
      Float distanceSquared;
  };
```

As the `KdTree` traverses the tree nodes, it will call this callback method of `PhotonProcess`, which decides if this photon is useful for the reflected radiance computation at the intersection point. It adds a reference to it to the `ClosePhoton`s array, `photons`, if so.

```
⟨Photonmap Method Definitions⟩+≡
  void PhotonProcess::operator()(const Photon &photon,
          Float distSquared, Float &maxDistSquared) const {
      if (foundPhotons < nLookup) {
          ⟨Add photon to unordered array of photons⟩
      }
      else {
          ⟨Remove most distant photon from heap and add new photon⟩
      }
  }
```

Until we have found nLookup photons around the point, we just store the nearby photons in an unordered array that is filled in whatever order photons are passed to this callback method. However, once the nLookup+1st photon arrives (if it does), we need to discard the one that is farthest away and only keep the nLookup closest ones. Therefore, after nLookup have arrived, we reorder photons to be a heap, such that the root element is the one with the greatest distance from the lookup point. Recall that a heap data structure can be constructed in linear time and that it can be updated after an item is removed or added in constant time. It is substantially more efficient to organize with a heap than to keep them sorted by distance. The make_heap() function in the C++ standard library reorders a given array into a heap so that the zero-th element is the root of the heap.

ClosePhoton	623
ePhoton::distanceSquared	623
Photon	617
PhotonProcess	623
tonProcess::foundPhotons	623
PhotonProcess::nLookup	623
PhotonProcess::photons	623

Once we have found nLookup photons, we can also decrease the search radius that the KdTree uses as it traverses its nodes; there's no reason to consider any additional photons that are farther away than the farthest one in the heap. KdTree::Lookup() passes a reference to maxDistSquared into this callback method so that we can reduce its value in situations like this. (It will not work as expected, however, if we increase maxDistSquared partway through a search.)

```
⟨Add photon to unordered array of photons⟩≡
  photons[foundPhotons++] = ClosePhoton(&photon, distSquared);
  if (foundPhotons == nLookup) {
      std::make_heap(&photons[0], &photons[nLookup]);
      maxDistSquared = photons[0].distanceSquared;
  }
```

As additional photons come in after we have built the heap, we know that the squared distance to new ones must be less than the maxDistSquared, since the KdTree doesn't call the callback method for items that are further away. Thus, any additional photon must be closer than the root node of the heap, which is the farthest away one we are storing. Therefore, we immediately call the standard library routine that removes the root item of the heap and updates the order of the remaining ones in the heap to reestablish a valid heap, pop_heap(). The new photon is added to the end of the array, which will have been left empty after pop_heap()'s updates, and then push_heap() again rebuilds the heap, accounting for a new element added to the end of it. After all of this, we can again reduce maxDistSquared to whatever the distance is to the new farthest-away photon.

⟨*Remove most distant photon from heap and add new photon*⟩≡

```
std::pop_heap(&photons[0], &photons[nLookup]);
photons[nLookup-1] = ClosePhoton(&photon, distSquared);
std::push_heap(&photons[0], &photons[nLookup]);
maxDistSquared = photons[0].distanceSquared;
```

Once we have found the nearby photons, we need to interpolate them in some manner. Recall from the original description of the particle tracing algorithm that both the local density of the particles and their individual weights together affect their contribution (a greater density of particles in the domain where W_e is nonzero will cause more of them to contribute their weights to the sum, and clearly higher weights of individual particles increases their contribution). A statistical technique called *density estimation* gives us exactly the tools we need to perform this interpolation.

Density estimation constructs a pdf given a set of sample points under the assumption that the samples are distributed according to the overall distribution of some function of interest. Histogramming is a straightforward example of the idea– in 1D, the line is divided into intervals with some width, and one can count how many samples land in each interval and normalize so that the areas of the intervals sum to one. **XXX Figure XXX.**

Kernel methods are a more sophisticated density estimation technique. They generally give better results and smoother pdfs that don't suffer from the discontinuities in histograms. Given a kernel function $k(x)$ that integrates to one,

623 `ClosePhoton`
623 `ClosePhoton::distanceSquared`
623 `PhotonProcess::nLookup`
623 `PhotonProcess::photons`

$$\int_{-\infty}^{\infty} k(x)\mathrm{d}x = 1$$

the kernel estimator for N samples at locations x_i is

$$\hat{p}(x) = \frac{1}{Nh}\sum_{i=1}^{N} k\left(\frac{x-x_i}{h}\right),$$

where h is the window width (also known as the smoothing parameter). Kernel methods can be thought of as placing a series of bumps at observation points, where the sum of the bumps forms a pdf since they individually integrate to one and the sum is normalized. Figure 16.14 shows an example of density estimation in 1D, where a smooth pdf is computed from a set of sample points.

use p for points

The key question with kernel methods is how the window width h is chosen. If it is too wide, the pdf will blur out relevant detail in parts of the domain with many samples, while if it is too narrow, the pdf will be too bumpy in the tails of the distribution where there aren't many samples. Nearest neighbor techniques solve this problem by choosing h adaptively based on local density of samples. Where there are many samples, the width is small, while where there are few samples, the width is large. For example, one approach is to pick a number n and find the distance to the nth nearest sample from the point x and use that distance, $d_k(x)$ for the window width. This is the *generalized nth nearest neighbor estimate*,

$$\hat{p}(x) = \frac{1}{Nd_n(x)}\sum_{i=1}^{N} k\left(\frac{x-x_i}{d_n(x)}\right).$$

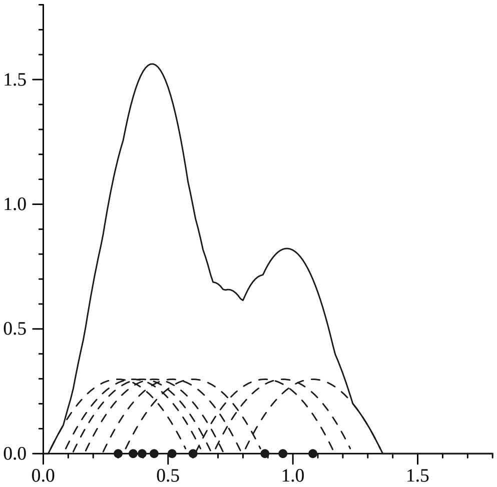


Figure 16.14: 1D example of density estimation, using the Epanechnikov kernel, $k(t) = .75(1 - .2t^2)/\sqrt{5}$, if $t < \sqrt{5}$, 0 otherwise, and a width of 0.1.

In d dimensions, this generalizes to

$$\hat{p}(x) = \frac{1}{N(d_n(x))^d} \sum_{i=1}^{N} k\left(\frac{x - x_i}{d_n(x)}\right).$$

And if we can't find n items, but only find n'

$$\hat{f}(x) = \frac{n'}{n} \frac{1}{N(d_n(x))^d} \sum_{i}^{N} k\left(\frac{x - x_i}{d_n(x)}\right).$$

This is the estimator we will use. We will just use a constant kernel function. In 2D, the constant kernel function that fulfills the normalization requirement is

$$k(x) = \begin{array}{ccl} \frac{1}{\pi} & : & |x| < 1 \\ 0 & : & \text{otherwise} \end{array}$$

Explain why no $\cos\theta$ here.

$$\hat{p}(\mathrm{p}) \sum_i \alpha_i f(\mathrm{p}, \omega_o, \omega_i)$$

$$L(\mathrm{p}, \omega) = \frac{d\Phi(\mathrm{p}, \omega)}{\mathrm{d}A(\mathrm{p}) \cos\theta \mathrm{d}\omega}$$

Derive in terms of using the reconstructed pdf...

Hack to ignore less than 3, but if it's that sparse, it just looks splotchy...

And if it's Lambertian, it's a big performance win to not evaluate the BSDF for each photon but to pull it out of the sum...

```
⟨Accumulate light from nearby photons⟩≡
  if (proc.foundPhotons > 3) {
      ⟨Compute photon scale factor with density estimation⟩
      ⟨Filter photons with high α values⟩
      ⟨Estimate reflected light from photons⟩
  }
```

We just scale uniformly. Can reduce blurriness of results slightly by using a weighting function that gives greater weight to photons the closer they are to the point being shaded...

⟨*Compute photon scale factor with density estimation*⟩≡

```
Float scale = proc.foundPhotons /
    (Float(nLookup) * Float(nPaths) * maxDistSquared * M_PI);
```

Just as filtering out unusually high radiance values from irradiance estimates helped reduce artifacts with the irradiance cache, it is similarly worthwhile to ignore a small number of the photons with the largest alpha values found at the lookup point. The same sort of bright spotches are eliminated by doing so.

In the code below, after `nth_element()` places the brightest photons at the end of the `photons` array, it is also necessary to increase the photon scale-factor to compensate for the ignored photons and reduce `nFound` so that the loop over photons below doesn't consider the ones at the end of the array.

⟨*Filter photons with high* α *values*⟩≡

```
ClosePhoton *photons = proc.photons;
int nFound = proc.foundPhotons;
if (nFilter > 0 && nFound > 2*nFilter) {
    std::nth_element(&photons[0], &photons[nFound-nFilter],
        &photons[nFound], CmpPhotonAlpha());
    scale *= nFound / (nFound - nFilter);
    nFound -= nFilter;
}
```

623	`ClosePhoton`
678	`M_PI`
612	`PhotonIntegrator::nLookup`
623	`PhotonProcess::foundPhotons`
623	`PhotonProcess::photons`

⟨*Photonmap Local Declarations*⟩+≡

```
struct CmpPhotonAlpha {
    bool operator()(const ClosePhoton &p1,
        const ClosePhoton &p2) const {
        return p1.photon->alpha < p2.photon->alpha;
    }
};
```

For purely diffuse BSDFs, it's wasteful to call the `BSDF::f()` method `nFound` times, since it will always return a constant value. Therefore, this method is only called for each photon if glossy components are present; otherwise, the equivalent computation can be done much more efficiently by finding the weighted sum of photon α values and multiplying it by the constant BSDF, found by dividing the surface's reflectance by π.

⟨*Estimate reflected light from photons*⟩≡

```
if (bsdf->NumComponents(BxDFType(BSDF_REFLECTION |
        BSDF_TRANSMISSION | BSDF_GLOSSY)) > 0)
    for (int i = 0; i < nFound; ++i)
        L += bsdf->f(wo, photons[i].photon->wi) *
            (scale * photons[i].photon->alpha);
else {
    Spectrum Li(0.);
    for (int i = 0; i < nFound; ++i)
        Li += photons[i].photon->alpha;
    L += (scale * Li) * bsdf->rho(wo) * INV_PI;
}
```

16.7 ***ADV***: Volume Integration

Just as `SurfaceIntegrator`s are the meeting-point of the scene geometry, materials, and lights, applying sophisticated algorithms to solve the light transport equation and determine the distribution of radiance in the scene, `VolumeIntegrator`s are responsible for incorporating the effect of participating media (as described by `VolumeRegion`s) into this process and determining how it affects the distribution of radiance. This section briefly introduces the equation of transfer, which describes how participating media changes radiance along rays, and then describes the `VolumeIntegrator` interface as well as a few simple `VolumeIntegrator`s.

`BSDF::f()`	373
`BSDF_GLOSSY`	334
`BSDF_REFLECTION`	334
`BSDF_TRANSMISSION`	334
`BxDFType`	334
`ClosePhoton::photon`	623
`INV_PI`	678
`Photon::alpha`	617
`Photon::wi`	617
`Spectrum`	181
`SurfaceIntegrator`	563
`VolumeIntegrator`	630
`VolumeRegion`	465

16.7.1 ***ADV***: The Equation of Transfer

The equation of transfer is the fundamental equation that governs the behavior of light in a medium that absorbs, emits, and scatters radiation (Chandrasekar 1960). It accounts for all of the volume scattering processes described in Chapter 12—absorption, emission, and in- and out-scattering—to give an equation that describes the distribution of radiance in an environment. The light transport equation is a special case of the equation of transfer, simplified by the lack of participating media and specialized for scattering from surfaces (Arvo 1993).

In its most basic form, the equation of transfer is an integro-differential equation that describes how the radiance along a beam changes at a point. After being transformed it into a purely integral equation, it describes the effect of participating media from the infinite number of points along a line. It can be derived in a straightforward manner by subtracting the effects of the scattering processes that reduce energy along a beam (absorption and out-scattering) from the processes that increase energy along it (emission and in-scattering). Recall the source term from Section 12.1.4; it gives the change in radiance along a ray at a point in a particular direction due to emission and in-scattered light from other points in the medium:

$$S(\mathrm{p},\omega) = L_{\mathrm{ve}}(\mathrm{p},\omega) + \sigma_s(\mathrm{p},\omega)\int_{\mathcal{S}^2} p(\mathrm{p},-\omega' \to \omega)\,L_{\mathrm{i}}(\mathrm{p},\omega')\,\mathrm{d}\omega'.$$

The source term accounts for all of the processes that add radiance to a ray.

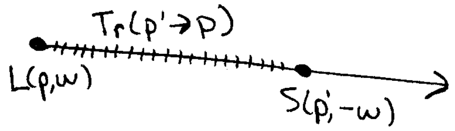


Figure 16.15: The equation of transfer gives the radiance along a ray (p,ω) passing through participating media. At each point along the ray, the source term $S(\mathrm{p}',w)$ gives the differential added radiance added at the point due to scattering and emission. This radiance is then attenuated by the beam transmittance $T_r(\mathrm{p}' \to p)$ from the point p' to the ray's origin.

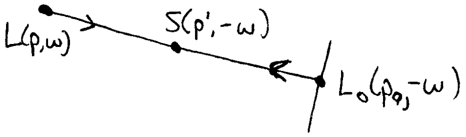


Figure 16.16: For a finite ray that intersects a surface, the radiance arriving at (p,ω) is equal to the outgoing radiance from the surface, $L_{\mathrm{o}}(\mathrm{p}_0,\omega)$ times the beam transmittance to the surface plus the added radiance from all points along the ray from p to p_0.

The attenuation coefficient, $\sigma_t(\mathrm{p},\omega)$ accounts for all processes that reduce radiance at a point: absorption and out-scattering.

$$\mathrm{d}L(\mathrm{p},\omega) = -\sigma_t(\mathrm{p},\omega)L(\mathrm{p},\omega)\mathrm{d}t$$

The overall differential change in radiance at a point p' along a ray is found by adding these two effects together to get the integro-differential form of the equation of transfer.[1]

$$\frac{\partial}{\partial t}L(\mathrm{p}',\omega) = -\sigma_t(\mathrm{p},\omega)L(\mathrm{p}',\omega) + S(\mathrm{p}',\omega)$$

With suitable boundary conditions, this equation can be transformed to a purely integral equation. For example, if we assume that there are no surfaces in the scene so that the rays are never blocked and have an infinite length, the integral equation of transfer is

$$L(\mathrm{p},\omega) = \int_0^\infty T_r(\mathrm{p}' \to \mathrm{p})S(\mathrm{p}',-\omega)\mathrm{d}t',$$

where $\mathrm{p}' = \mathrm{p} + t\omega$. (See Figure 16.15.) The meaning of this equation is reasonably intuitive–it just says that the radiance arriving at a point from a given direction is contributed to by the added radiance along all points along the ray from the point. The amount of the added radiance at each point along the ray that reaches the ray's origin is reduced by the total beam transmittance from the ray's origin to the point.

More generally, if there are reflecting and/or emitting surfaces in the scene, rays don't necessarily have infinite length and the surface a ray hits affects its radiance, adding outgoing radiance from the surface at the point and preventing radiance

[1] It is an integro-differential equation due to the integral over the sphere in the source term.

from points along the ray beyond the intersection point from contributing to radiance at the ray's origin. If a ray (p, ω) intersects a surface at some point p_0 a distance t along the ray, then the integral equation of transfer is

$$L(\mathrm{p},\omega) = T_r(\mathrm{p}_0 \to \mathrm{p}) L_\mathrm{o}(\mathrm{p}_0, -\omega) + \int_0^t T_r(\mathrm{p}' \to \mathrm{p}) S(\mathrm{p}', -\omega) \mathrm{d}t' \qquad (16.7.6)$$

Where $\mathrm{p}_0 = \mathrm{p} + t\omega$ is the point on the surface and $\mathrm{p}' = \mathrm{p} + t'\omega$ are points along the ray. (See Figure 16.16).

This equation describes the two effects that contribute to radiance along the ray: first, reflected radiance back along the ray from the surface is given by the L_o term, which gives the emitted and reflected radiance from the surface. This radiance may be attenuated by the participating media; the beam transmittance from the ray origin to the point accounts for this. The second term accounts for the added radiance along the ray due to volume scattering and emission, but only up to the point where the ray intersects the surface–points beyond that one don't affect the radiance along the ray.

In the interests of brevity, we will refrain from further generalization of the equation of transfer here. However, just as the light transport equation could be written in a more general form as a sum over paths of various numbers of vertices and by adding and importance function to turn it into the measurement equation, the equation of transfer can be generalized in a similar manner. Likewise, we will only present a few simple `VolumeIntegrator`s in the remainder of this section, though the general types of algorithms used for the surface integrators in this chapter, such as path tracing, bidirectional path tracing, photon mapping, etc., can be applied to volume integration as well. The further reading section has pointers to more information about these topics.

`Integrator`	562
`Ray`	36
`Scene`	8
`Spectrum`	181

16.7.2 ***ADV***: Volume Integrator Interface

The `VolumeIntegrator` interface inherits from `Integrator`, picking up the `Preprocess()`, `RequestSamples()` and `L()` methods declared there. The first two of these methods are used by volume integrators in the same way as by surface integrators. The `L()` method is similar to the surface integrator versions in that it returns the radiance along the given ray, though volume integrators should assume that the ray has already been intersected with the scene geometry and that if the ray does intersect a surface, its `Ray::maxt` value will have been set to be at the intersection point. As such, the volume integrator must only compute the effect of volume scattering from the range `[mint, maxt]` along the ray.

The `VolumeIntegrator` interface adds an additional method that implementations must provide, `Transmittance()`, which is responsible for computing the beam transmittance along the given ray from `Ray::mint` to `Ray::maxt`.

⟨*Volume Scattering Declarations*⟩+≡

```
class VolumeIntegrator : public Integrator {
public:
    virtual Spectrum Transmittance(const Scene *scene,
        const Ray &ray, const Sample *sample, Float *alpha) const = 0;
};
```

With this background, the `Scene::L()` method can be fully understood. It is a direct implementation of Equation 16.7.6. The surface integrator computes outgoing radiance at the ray's intersection point, ignoring attenuation back to the ray origin. The volume integrator's `Transmittance()` method is called to compute the beam transmission to the point on the surface, and its `L()` method gives the radiance along the ray due to participating media. The sum of these two terms gives the total radiance arriving at the ray origin.

16.7.3 ***ADV***: Emission-Only Integrator

The simplest-possible volume integrator (other than one that ignored participating media completely) simplifies the source term by ignoring in-scattering completely and only accounting for emission and attenuation. Because in-scattering is ignored, the integral over the sphere in the source term at each point along the ray disappears and the simplified equation of transfer is:

$$L(\mathrm{p},\omega) = T_r(\mathrm{p}_0 \to \mathrm{p}) L_{\mathrm{o}}(\mathrm{p}_0, -\omega) + \int_0^t T_r(\mathrm{p}' \to \mathrm{p}) L_{\mathrm{ve}}(\mathrm{p}', -\omega) \mathrm{d}t. \qquad (16.7.7)$$

The `EmissionIntegrator`, defined in `integrators/emission.cpp`, takes this approach.

630 VolumeIntegrator

⟨EmissionIntegrator Declarations⟩≡

```
class EmissionIntegrator : public VolumeIntegrator {
public:
    ⟨EmissionIntegrator Public Methods⟩
private:
    ⟨EmissionIntegrator Private Data⟩
};
```

The `EmissionIntegrator`'s `Transmittance()` and `L()` methods both have to evaluate one-dimensional integrals along points t' along a ray. Rather than using a fixed number of samples for each estimate the implementation here the number based on the distance the ray travels in the volume—the longer the distance, the more samples are taken. This approach is worthwhile for the naturally intuitive reasons—the longer the ray's extent in the medium, the more acuracy is desirable, and the more samples are likely to be needed to capture greater variation in optical properties along the ray. The number of samples taken is determined indirectly by a user-supplied parameter giving a step-size between samples; the ray is divided into segments of the given length and a single sample is taken in each one.

⟨EmissionIntegrator Public Methods⟩≡

```
EmissionIntegrator(Float ss) { stepSize = ss; }
```

⟨EmissionIntegrator Private Data⟩≡

```
Float stepSize;
```

Only a single 1D sample is needed for of the integrals along rays in the two methods below.

⟨*EmissionIntegrator Method Definitions*⟩≡

```
void EmissionIntegrator::RequestSamples(Sample *sample,
        const Scene *scene) {
    tauSampleOffset = sample->Add1D(1);
    scatterSampleOffset = sample->Add1D(1);
}
```

⟨*EmissionIntegrator Private Data*⟩+≡

```
int tauSampleOffset, scatterSampleOffset;
```

The `Transmittance()` method is reasonably straightforward. The `VolumeRegion`'s `Tau()` method takes care of computing the optical thickness τ from the ray's starting point to its ending point. The only work for the integrator here is to choose a step size (in case `Tau()` does Monte Carlo integration, as the implementation in Section 15.5 does), pass a single sample value along to that method, and return $e^{-\tau}$. (If the `Tau()` method called can compute τ analytically, these additional values are ignored.)

This method also takes advantage of the fact that the `Sample` value is only non-`NULL` for camera rays to increase the step size, thus reducing computational demands (and accuracy) for shadow and indirect rays. The reduction in accuracy for those rays generally isn't noticeable, however.

⟨*EmissionIntegrator Method Definitions*⟩+≡

```
Spectrum EmissionIntegrator::Transmittance(const Scene *scene,
        const Ray &ray, const Sample *sample, Float *alpha) const {
    if (!scene->volumeRegion) return Spectrum(1.f);
    Float step = sample ? stepSize : 4.f * stepSize;
    Float offset = sample ? sample->oneD[tauSampleOffset][0] :
        RandomFloat();
    Spectrum tau = scene->volumeRegion->Tau(ray, step, offset);
    return Exp(-tau);
}
```

EmissionIntegrator	631
Exp()	183
Ray	36
Sample::Add1D()	240
Sample::oneD	241
Scene	8
Scene::volumeRegion	9
Spectrum	181
VolumeRegion	465
VolumeRegion::Tau()	466

The `L()` method is responsible for evaluating the second term of the sum in Equation 16.7.7. If the ray enters the volume at $t = t_0$ along the ray, then no attenuation happens from start of ray up to t_0, and the `L()` method can instead consider integral from t_0 to t_1, the minimum of the point where the ray exits the volume and the point where it intersects a surface—see Figure 16.17. Given this integral,

$$\int_{t_0}^{t_1} L_{\mathrm{ve}}(\mathrm{p}', -\omega)\, T_r(\mathrm{p}' \to \mathrm{p})\, \mathrm{d}t',$$

a straightforward approach is to uniformly select random points along the ray between t_0 and t_1 and evaluate the estimator

$$\frac{1}{N}\sum_i \frac{L_{\mathrm{ve}}(\mathrm{p_i}, -\omega) T_r(\mathrm{p_i} \to \mathrm{p})}{p(\mathrm{p_i})} = \frac{t_1 - t_0}{N}\sum_i L_{\mathrm{ve}}(\mathrm{p_i}, -\omega) T_r(\mathrm{p_i} \to \mathrm{p})$$

since $p(\mathrm{p_i}) = 1/(t_1 - t_0)$.

The L_{ve} term in the estimator can be evaluated directly with the corresponding `VolumeRegion` method, and the optical thickness τ to evaluate T_r can either be

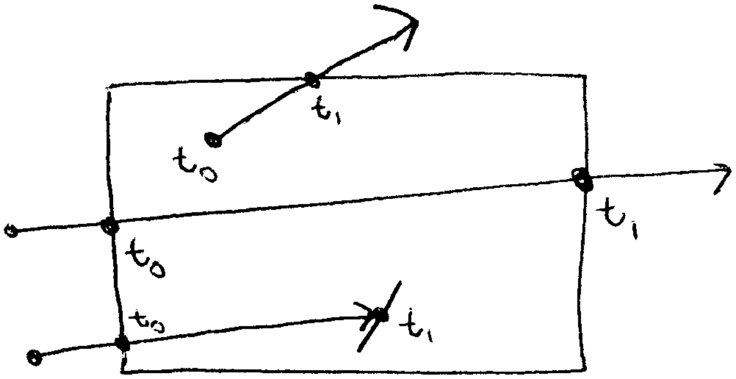


Figure 16.17: The starting point of the ray marching process, t_0, is the maximum ray origin or the ray's intersection point with the participating media's bound. Similarly, the end, t_1, is the minimum of the point where the ray exits the medium and the point where it intersects a surface, if any.

evaluated directly (for a homogeneous or exponential atmosphere), or via Monte Carlo integration as described in Section 15.5.

In order to do this computation, the implementation of the `L()` method thus starts out by finding the t range for the integral and initializing `t0` and `t1` accordingly.

631	`EmissionIntegrator`
37	`RayDifferential`
8	`Scene`
181	`Spectrum`
465	`VolumeRegion`
466	`VolumeRegion::IntersectP()`

⟨*EmissionIntegrator Method Definitions*⟩+≡

```
Spectrum EmissionIntegrator::L(const Scene *scene,
        const RayDifferential &ray, const Sample *sample,
        Float *alpha) const {
    VolumeRegion *vr = scene->volumeRegion;
    Float t0, t1;
    if (!vr || !vr->IntersectP(ray, &t0, &t1)) return 0.f;
    ⟨Do emission-only volume integration in vr⟩
}
```

Two additional details round out the implementation here. First, just as the `VolumeRegion::Tau()` method in Section 15.5 used a uniform step size between sample points, this implementation also steps uniformly for similar reasons. Second, the beam transmittance values T_r can be evaluated efficiently if the points $\mathrm{p_i}$ are sorted from the one closest to p to the one farthest away. qThen, the multiplicative property of T_r can be used to incrementally compute T_r from its value from the previous point:

$$T_r(\mathrm{p_i} \to \mathrm{p}) = T_r(\mathrm{p_{i-1}} \to \mathrm{p})\, T_r(\mathrm{p_i} \to \mathrm{p_{i-1}}).$$

Because $T_r(\mathrm{p_i} \to \mathrm{p_{i-1}})$ covers a shorter distance than $T_r(\mathrm{p_i} \to \mathrm{p})$, fewer samples can be used to estimate its value if it is evaluated with Monte Carlo. Both of these ideas are illustrated in Figure 16.18

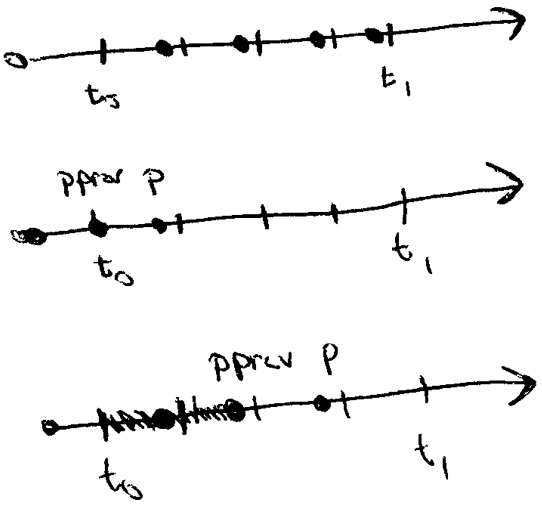


Figure 16.18: volume integration ray sampling: the ray's extent from t_0 to t_1 is subdivided into a number of segments based on the `stepSize` parameter. A single sample is taken in each segment, where the first sample is placed randomly in the first segment and all additional samples are offset by equal-sized steps (top). Ray-marching tracks the previous point to which transmittance was computed, `pPrev` as well as the current point, `p`. Initially, `pPrev` is the point where the ray enters the volume (middle). At each subsequent step, beam transmittance is computed as the product of transmittance to `pPrev` and the additional transmittance from `pPrev` to `p`.

⟨*Do emission-only volume integration in* `vr`⟩≡

```
  Spectrum Lv(0.);
  ⟨Prepare for volume integration stepping⟩
  for (int i = 0; i < N; ++i, t0 += step) {
      ⟨Advance to sample at t0 and update T⟩
      ⟨Compute emission-only source term at p⟩
  }
  return Lv * step;
```

⟨*Prepare for volume integration stepping*⟩≡

```
  int N = Ceil2Int((t1-t0) / stepSize);
  Float step = (t1 - t0) / N;
  Spectrum T(1.f);
  Point p = ray(t0), pPrev;
  Vector w = -ray.d;
  if (sample) t0 += sample->oneD[scatterSampleOffset][0] * step;
  else        t0 += RandomFloat() * step;
```

To find the overall transmittance at the current point, it's only necessary to find the transmittance from the previous point to the current point and multiply it by the transmittance from the ray origin to the previous point, as described above.

33	Point
679	RandomFloat()
36	Ray
241	Sample::oneD
181	Spectrum
185	Spectrum::y()
27	Vector
466	VolumeRegion::Lve()
466	VolumeRegion::Tau()

⟨*Advance to sample at* `t0` *and update* `T`⟩≡

```
  pPrev = p;
  p = ray(t0);
  Spectrum stepTau = vr->Tau(Ray(pPrev, p - pPrev, 0, 1),
      .5f * stepSize, RandomFloat());
  T *= Exp(-stepTau);
  ⟨Possibly terminate raymarching if transmittance is small⟩
```

In a thick medium, the transmittance may become very low after the ray has passed a sufficient distance through it. To reduce the time spend computing source term values that are likely to have very little contribution to the radiance at the ray's origin, ray stepping is randomly terminated with Russian roulette when transmittance is sufficiently small. Because $T_r \leq 1$, transmittance never increases as we step from one point to the next in the medium, so once the transmittance is low, termination in this manner is a reasonable technique.

⟨*Possibly terminate raymarching if transmittance is small*⟩≡

```
  if (T.y() < 1e-3) {
      const Float continueProb = .5f;
      if (RandomFloat() > continueProb) break;
      T /= continueProb;
  }
```

Having done all this other work, actually computing the source term at the point is trivial.

⟨*Compute emission-only source term at* `p`⟩≡

```
  Lv += T * vr->Lve(p, w);
```

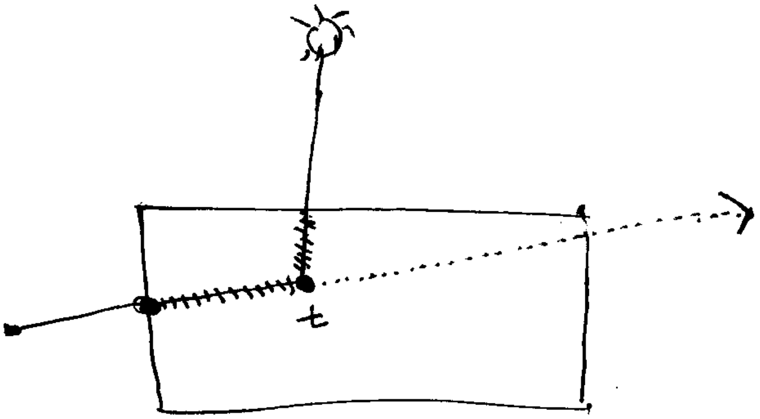


Figure 16.19: When the direct lighting contribution is evaluated at some point *t* along a ray passing through participating media, it's necessary to compute the attenuation of the radiance from the light passing through the volume to the scattering point as well as the attenuation from that point back to the eye.

EmissionIntegrator 631
VolumeIntegrator 630

16.7.4 ***ADV***: Single Scattering Integrator

Almost all of the implementation of the `SingleScattering` integrator parallels the `EmissionIntegrator`, so only the fragments in the parts that differ are included here.

```
⟨SingleScattering Declarations⟩≡
  class SingleScattering : public VolumeIntegrator {
  public:
      ⟨SingleScattering Public Methods⟩
  private:
      ⟨SingleScattering Private Data⟩
  };
```

In addition to worrying about the emission at each point along the ray, this integrator also considers the incident radiance due to direct illumination but ignores incident radiance due to multiple scattering. Thus, its `L()` method evaluates

$$\int_0^t T_r(\mathrm{p}' \to \mathrm{p}) \left(L_{\mathrm{ve}}(\mathrm{p}', \omega) + \sigma_s(\mathrm{p}', \omega) \int_{S^2} p(\mathrm{p}', -\omega' \to \omega) L_{\mathrm{d}}(\mathrm{p}', \omega') \mathrm{d}\omega' \right) \mathrm{d}t'.$$

where L_{d} only includes radiance from direct lighting. This radiance may be blocked by geometry in the scene and may itself be attenuated by participating media between the light and the point p' along the ray—see Figure 16.19.

This integrator uses the same general ray-marching approach to evaluate the equation of transfer as the `EmissionIntegrator`. One difference compared to that integrator is that before it enters the `for` loop over sample positions, it computes sample values for light source sampling. Because it isn't known how many of samples will be necessary until `L()` is called since this depends on the length of the ray segment over which integration is being done, it's not possible to have the `Sampler` generate samples and pass them into `L()` via the `Sample` with the current interfaces in `lrt`. Therefore, a three-dimensional set of Latin hypercube samples are generated here. The first dimension is used to choose which light to sample at each scattering point, and the other two are used by the light to select a point on the light source.

⟨*Compute sample patterns for single scattering samples*⟩≡

```
Float *samp = (Float *)alloca(3 * N * sizeof(Float));
LatinHypercube(samp, N, 3);
int sampOffset = 0;
```

The other difference from the `EmissionIntegrator` is how the source term is evaluated at each term p'. At each sample point along the ray, the fragment below computes the single-scattering approximation of the source term at the point `p`. It serves the same role as the ⟨*Compute emission-only source term at* `p`⟩ fragment above. After including volume emission as the `EmissionIntegrator` does, it finds the value of σ_s at the point, selects a light to sample using the sample table, and computes its contribution to scattering at the point. Because the source term will generally be evaluated at many points along the ray, only a single light is sampled at each one and its contribution will be scaled by the number of lights, similar to the direct lighting integrator's "sample one light" strategy.

251	`LatinHypercube()`
478	`Light`
9	`Scene::lights`
181	`Spectrum`
182	`Spectrum::Black()`
466	`VolumeRegion::Lve()`
466	`VolumeRegion::sigma_s()`

⟨*Compute single-scattering source term at* `p`⟩≡

```
Lv += T * vr->Lve(p, w);
Spectrum ss = vr->sigma_s(p, w);
if (!ss.Black()) {
    int nLights = scene->lights.size();
    int lightNum = min(Floor2Int(samp[sampOffset] * nLights), nLights-1);
    Light *light = scene->lights[lightNum];
    ⟨Add contribution of light due to scattering at p⟩
}
sampOffset += 3;
```

Computing the estimate of the direct lighting contribution involves estimating the integral

$$\int_{S^2} p(\mathrm{p}', -\omega' \to \omega) L_{\mathrm{d}}(\mathrm{p}', \omega') \mathrm{d}\omega'.$$

Here, rather than sampling both the phase function and the light source and applying multiple importance sampling, the implementation here has the light choose a sample position and computes the estimator directly. For media that aren't extremely anisotropic, this approach works well.

⟨*Add contribution of* `light` *due to scattering at* `p`⟩≡

```
Float pdf;
VisibilityTester vis;
Vector wo;
Float u1 = samp[sampOffset+1], u2 = samp[sampOffset+2];
Spectrum L = light->Sample_L(p, u1, u2, &wo, &pdf, &vis);
if (!L.Black() && vis.Unoccluded(scene)) {
    Spectrum Ld = L * vis.Transmittance(scene);
    Lv += T * Ld * ss * vr->p(p, w, -wo) * nLights / pdf;
}
```

Further Reading

Lommel was the apparently first to derive the equation of transfer (Lommel 1889), in a not-widely-known paper. Not only did he derive the equation of transfer, but he solved it in some simplified cases in order to estimate reflection functions from real world surfaces (including marble and paper) and compared his solutions to measured reflectance data from these surfaces.

Apparently unaware of Lommel's work, Schuster was the next worker in radiative transfer to consider the effect of multiple scattering (Schuster 1905). He used the term *self-illumination* to describe the fact that each part of the medium is illuminated by every other part of the medium and derived differential equations that described reflection from a slab along the normal direction assuming the presence of isotropic scattering; the conceptual framework that he developed remains essentially unchanged in the field of radiative transfer

`Light::Sample_L()`	542
`Spectrum`	181
`Spectrum::Black()`	182
`Vector`	27
`VisibilityTester`	479
`yTester::Transmittance()`	480
`lityTester::Unoccluded()`	480
`VolumeRegion::p()`	466

Soon thereafter, Scharzchild introduced the concept of radiative equilibrium (**?**) and Jackson expressed Schuster's equation in integral form, also noting that "the obvious physical mode of solution is Liouville's method of successive substitutions." (i.e. a Neumann series solution) (Jackson 1910). Finally, King completed the re-discovery of the equation of transfer by expressing it in the general integral form (King 1913). (Yanovitskij (Yanovitskij 1997) traces the origin of the integral equation of transfer to Chvolson (Chvolson 1890), but we have been unable to find a copy of this paper.)

Russian roulette introduced to graphics by Kirk and Arvo (Kirk and Arvo 1991). Hall and Greenberg had previously suggested adaptively terminating ray trees by not tracing rays with less than some minimum contribution (Hall and Greenberg 1983). Kirk and Arvo's technique is unbiased, though in some situations, bias and less noise may be the less undesirable artifact.

Lafortune bidir (Lafortune and Willems 1994). Veach and Guibas (Veach and Guibas 1994). Kollig and Keller bidir with quasi-random sample patterns (Kollig and Keller 2000).

Irradiance caching (Ward, Rubinstein, and Clear 1988; Ward and Heckbert 1992; Ward 1994b)

Kajiya (Kajiya 1986), Immel et al (Immel, Cohen, and Greenberg 1986)

Photon maps–resampling step between light and eye makes tricky situations a lot easier... Arvo (Arvo 1986). Heckbert (Heckbert 1990). Collins (Collins 1994). Jensen (Jensen 1996; Jensen and Christensen 1998). Photon mapping improvements (Peter and Pietrek 1998; Suykens and Willens 2000; Keller and Wald 2000). Also can be used to derive better importance sampling methods for final gathering, path tracing, etc, based on approximation of incident illumination using nearby photons (Jensen 1995; Hey and Purgathofer 2002).

Per TVCG paper on adjoints and importance (Christensen 2003)

Shirley thesis (Shirley 1990a; Shirley 1990b), incl sum over paths formulation

Metropolis (Veach and Guibas 1997) (Pauly, Kollig, and Keller 2000)

Radiance (and radiosity stuff) for virtual mirrors for light paths...

The equation of transfer was first introduced to graphics by Kajiya and von Herzen (Kajiya and Herzen 1984); Rushmeier was the first to compute solutions of it in a general setting (Rushmeier 1988). However, Arvo was the first to make the essential connections between previous formalizations of light transport in graphics and the equation of transfer and radiative transfer in general (Arvo 1993).

Bhate and Tokuta spherical harmonic approach (Bhate and Tokuta 1992). Pérez/Pueyo/Sillion volume globillum survey (Pérez, Pueyo, and Sillion 1997).

Blasi et al two pass Monte Carlo algorithm, somewhat in the spirit of Kajiya and von Herzen, where first pass shoots energe from lights and stores it in a grid, second pass does final rendering (Blasi, Saẽc, and Schlick 1993).

Lafortune and Willems bidir stuff (Lafortune and Willems 1996).

Jensen book (Jensen 2001).

Reichert final gather (Reichert 1992).

Blinn first introduced basic volume scattering algorithms to graphics (Blinn 1982b). Other important early work includes Kajiya and von Herzen (Kajiya and Herzen 1984), Max (Max 1986), and Nishita et al (Nishita, Miyawaki, and Nakamae 1987). Glassner's book has a thorough overview of this topic and previous applications of it in graphics (Glassner 1995), and Max's survey article also concisely summarizes the topic (Max 1995).

One key application of volume scattering algorithms in computer graphics has been simulating atmospheric scattering. Work on this topic includes Klassen (Klassen 1987) and Nishita et al (Nishita, Miyawaki, and Nakamae 1987). More recently, Preetham et al's SIGGRAPH paper introduced a physically rigorous and computationally efficient atmospheric and sky-lighting model (Preetham, Shirley, and Smits 1999).

There are a number of important applications of visualizing volumetric datasets for medical and engineering applications–this area is called *volume rendering*. In many of these applications, radiometric accuracy is substantially less important then developing techniques that help make structure in the data apparent (e.g. where the bones are in CT scan data.) Early papers in this area include Levoy's (Levoy 1988; Levoy 1990b; Levoy 1990a) and Drebin et al (Drebin, Carpenter, and Hanrahan 1988).

Rushmeier and Torrance finite element stuff (Rushmeier and Torrance 1987).

Exercises

16.1 To further improve efficiency, Russian roulette can be applied to skip tracing many of the shadow rays that make a low contribution to the final image. Tentatively compute the potential contribution of each shadow ray to the final overall radiance value before tracing the ray; if the contribution is below some threshold, apply Russian roulette to possibly skip tracing the ray.

XXX Russian roulette always increases variance; its value comes from improving efficiency...

16.2 Read Veach's description of efficiency-optimized Russian roulette, which adaptively chooses a threshold for applying Russian roulette (Veach 1997, Section XX). Implement this algorithm in `lrt` and evaluate its effectiveness in comparison to manually set thresholds.

16.3 Path tracing to be able to flag important stuff for indirect lighting, be able to sample it according to dA. Then use MIS to compute weights. Experiments with scene with substantial indirect lighting: how much help, how much does it hurt when mostly direct? What if the wrong objects are flagged as important? Or if MIS isn't used? What about dynamically changing probabilities based on experience...

16.4 Light transport algorithms that trace paths from the lights like bidirectional path tracing and particle tracing implicitly assume that the BSDFs in the scene are symmetric–that $f(\mathrm{p},\omega,\omega') = f(\mathrm{p},\omega',\omega)$. However, both real-world BSDFs like the one describing specular transmission as well as synthetic BSDFs like the ones that are used when shading normals are used are not symmetric; not accounting for this can lead to substantial errors in these light transport algorithms (Veach 1997, Section XX). Fix `lrt` to properly handle non-symmetric BSDFs.

`IrradianceCache 598`

16.5 Photons as paths from light for bidir–use as tiny light sources–unbiased.

16.6 Improve the `BidirectionalIntegrator` so that it uses multiple importance sampling to weight path contributions (Veach and Guibas 1995). How much is variance reduced by this improvement?

16.7 Extent the `IrradianceCache` so optionally sample the glossy components of the BSDF and recursively trace rays to evaluate indirect radiance along them. Set up a scene that illustrates a situation where this gives a substantially better result than using the irradiance estimate for glossy reflection.

16.8 Algorithms for handling many lights... Ward, Wald...

16.9 Improve the `IrradianceCache`'s computation of irradiance values by modifying it to use bidirectional path tracing instead of regular path tracing to compute the individual radiance values. Construct a scene where this approach is much better than path tracing. How much does it improve the results for scenes where the path tracing approach works well already?

16.10 By taking a directionally-varying distribution of incoming radiance, using it to compute irradiance, and then just storing an irradiance value, the irradiance cache makes a well-chosen engineering trade-off that minimizes the amount of information that must be stored with each entry in the irradiance cache. (Substantially more memory would be needed to store all of these radiance values with each cache entry, for example.) For diffuse BS-DFs, irradiance is sufficient to accurately compute reflection, but for glossy

BSDFs, the lack of directionally-varying illumination information may introduce substantial error. Investigate compact representations for functions on the sphere such as spherical harmonics and spherical wavelets and extend the irradiance cache to represent the incident illumination with coefficients from such a set of basis functions. Modify the `IrradianceCache`'s `L()` method to use this representation to compute reflected light from points on surfaces and compare the accuracy of images rendered with this technique to images rendererd by the original irradiance cache, particularly for scenes with glossy surfaces.

16.11 Expected values for many light source handling. Can probabilistically assume a value for part of the integrand. Then x% of the time, compute it for real, weight result by $(\text{guess} - \text{actual})/x$%... Show that this is an unbiased estimator, etc...

16.12 kajiya-von herzen stuff, precompute illumination on a grid, save all those redundant-ray marching computations

16.13 Extent the `SingleScattering` volume integrator to use multiple importance sampling based on sampling points on the light source and sampling the phase function for the direct lighting contribution. Under what circumstances will this method give substantially less variance than the current implementation?

598 `IrradianceCache`
636 `SingleScattering`

16.14 Design and implement a general Monte Carlo path tracing approach to computing images with multiple scattering in participating media.

16.15 bidir for lighting in volumes, cite path integral generalization to volumes by the volume metropolis guys, also lafortune paper.

16.16 If photon mapping is only being used to render caustics in a scene, generation of the caustic map can be accelerated by flagging which objects in the scene potentially have specular components in their BSDFs and then building a directional table around each light source recording which directions could potentially result in specular paths (Jensen 1996). Photons are only shot in these directions, saving work of tracing photons in directions that are certain to not lead to caustics. Extend `lrt` to support this technique. Note that different solutions will be needed for point light sources, directional light sources, and area light sources. How much does this end up improving efficiency for these sorts of scenes?

16.17 Another approach to improving the efficiency of photon shooting is to start out by shooting photons from lights in all directions with equal probability, but to then dynamically update the probability based on which directions are lead to light paths that have high throughput and which directions are less effective. Photons then must be re-weighted based on the probability for shooting a photon in a particular direction. (This is similar in spirit to the `WeightedSampleOneLight()` light source sampling method.) So long as there is always non-zero possibility of sending a photon in any direction, this approach doesn't introduce bias into the shooting algorithm. One advantage of this second approach is that the user of the renderer doesn't need to flag which objects may cast caustics ahead of time.

16.18 Performance of the photon mapping integrator can be substantially improved by using irradiance caching to compute reflection from diffuse objects–for example, final gathering can be avoided for diffuse surfaces and only needs to be done for glossy surfaces. Extend the photon mapping integrator to compute and interpolate irradiance estimates like the `IrradianceCache`. When an irradiance value needs to be computed at a point, use the photon map to compute the reflected radiance at the intersection points of the rays traced to compute the irradiance estimate; this further improves performance since additional bounces of rays don't need to be traced to compute the estimate.

16.19 Even if they aren't used directly for computing reflected radiance, the photons around a point carry useful information about the illumination there that can be used for importance sampling. First, see Keller and Wald's paper (Keller and Wald 2000) and extend the photon mapping integrator to record which light source each direct lighting photon originated from. When computing direct illumination at a point, use the nearby direct lighting photons to choose light source sampling probabilities based on the estimated contribution of each of the light sources.

`VolumeIntegrator 630`

Next, see Jensen's first paper on photon mapping and Hey and Purgathofer's more recent paper (Jensen 1995; Hey and Purgathofer 2002). Both of these describe methods for using importance sampling to choose ray directions for path tracing and final gathering based on a pdf built using the photons around a point. Since the distribution of incident directions of these photons gives information about the distribution of indirect illumination there, they can be used to construct a continuous distribution of directions over the unit sphere. Implement one of these methods and apply it for sampling some of the rays used for the final gathering computation (continue to sample the BSDF for the rest of the rays). Apply multiple importance sampling to compute weights for these samples.

16.20 use depth-mapped shadowmap stuff for fast light beams through atmosphere

16.21 pass radiance into `attenuation()/L()` functions of `VolumeIntegrator`, use their magnitudes to guide how many MC samples to take, etc...

16.22 With inhomogeneous volume regions, where the optical depth between two points must be computed with raymarching, the `SingleScattering` volume integrator may spend a lot of time finding the attenuation between lights and points on rays where single scattering is being computed. One approach to reducing this computation is to take advantage of the fact that the amount of attenuation for nearby rays is generally smoothly varying and to use a precomputed approximation to the attenuation. For example, Kajiya and Von Herzen computed the attenuation to a light source at a grid of points in 3D space and then found attenuation at any particular point could by interpolating among nearby grid samples (Kajiya and Herzen 1984). A more sophisticated approach was developed by Lokovic and Veach in the form of deep shadow maps, which is based on a clever compression technique that takes advantage of the smoothness of the reattenuation (Lokovic and Veach 2000). Implement one of these approaches in `lrt` and measure how much it speeds up the `SingleScattering` integrator. Are there situations where this approach results in image errors?

16.23 spotlight volume region, let light sample t along path as well... interesing/generally illustrative a la specular stuff, in that if you don't deal with it, it gets harder the more specular it gets, but if you do, then it's really easy to nail...

16.24 bidir by sampling a bunch of lights paths, then randomly importance sampling among them based on amount of energy that they carry? what other things can we do to get local path space exploration a la metropolis?

16.25 way to partition lights to handle directly or not

16.26 phomap: accelerate final gathering by precomputing outgoing irradiance at photons; then for final gather, just fine one nearest photon at gather ray hit and return its value–way faster than doing lookups and BSDF evaluations...

16.27 Fluence caching for volume rendering

630 VolumeIntegrator

16.28 Varying step-size for ray-marching: one shortcoming of the `VolumeIntegrators` is that they take a fixed step-size through the participating medium. If the medium is very thick in some parts but very sparse in others, this may be inefficient, as a short step-size is needed to accurately resolve detail in the thick parts but is wasteful in the rest of the volume. Modify the implementations in this chapter to vary the step-size based on the local scattering properties. How much does this speed up rendering the **smoke.lrt** scene?

16.29 Ray differentials for global illumination rays: Suykens and Willems (Suykens and Willems 2001), Christensen et al (Christensen, Laur, Fong, Wooten, , and Batali 2003).

17. Summary and Conclusion

lrt represents a single point in the space of rendering system designs. The basic decisions we made early on—that ray tracing would be the basic geometric visibility algorithm used, that physical correctness would be a cornerstone of the system, and that Monte Carlo would be the main approach used for numerical integration—all had pervasive implications in the system's design. An entirely different set of trade-offs would have been made if this was a renderer designed instead for maximal performance for real-time rendering or for maximum flexibility for artistic expression, for example. This chapter looks back at some of the details of the complete system, discusses some design alternatives, and also sketches out how it could be extended in more complex ways than have been described in the exercises at the ends of the chapters.

17.1 Design Retrospective

One of the basic assumptions in lrt's design was that the most interesting types of images to render are images with complex geometry and lighting and that rendering these images well—with good sampling patterns, ray differentials, and anti-aliased texture—is worth the additional overhead that it introduces. A result of this assumption is that lrt is relatively inefficient at rendering simple images, however.

For example, if all of the lights and geometry are removed from the **foo.lrt** scene, the system still takes roughly **half as much** time to generate a completely black image as it does to generate the full image. All of the effort used to compute samples and ray differentials for camera rays and to add the contribution of the rays to the image is clearly a substantial fraction of the time spent rendering simple scenes. For example, with the standard parameter settings, each image sample contributes to sixteen pixels in the image on average; if very little time is spent computing the ray's radiance, then the time to update the image using a non-trivial reconstruction filter will generally be much higher than if a one-pixel-wide box filter was used for image reconstruction and each sample only contributed to a single pixel.

Of course, for a more complex scene, the time spent finding ray intersections, evaluating textures, and applying Monte Carlo integration algorithms comes to dominate running time, and the relative time spent on sample generation and image sample filtering becomes less important. Because we believe that these are the most important types of scenes to render, the fact that some parts of the system are a bottleneck for simple scenes isn't a high priority to improve.

Another example of performance implications from the basic design decisions is that because `lrt` makes some effort to filter texture maps well and compute ray differentials, finding the BSDF at a ray intersection is more computationally intensive than it is in other renderers that don't expend as much effort in this area. `lrt` generally performs better as more samples are taken to compute outgoing radiance at a given intersection point–for example, increasing the number of shadow rays traced to an area light source amortizes the anti-aliasing work done in computing the BSDF's textures over more samples, while tracing more camera rays to reduce noise in shadows is relatively less efficient, since a BSDF needs to be computed at each of their intersection tests.

`BSDF`	370
`Camera`	202
`DifferentialGeometry`	58
`Sampler`	237
`Shape`	63

Furthermore, some parts of the system may sometimes do unnecessary work due to the way the system is currently designed. For example, `Sampler`s always compute lens and time samples, even if they aren't needed by the `Camera`; there's no way for the `Camera` to communicate its sample needs. Similarly, if an intergrator doesn't use all of the samples requested in its `GetSamples()` method for some ray, the `Sampler`'s work for generating those samples is wasted. This can happen if the ray doesn't intersect any geometry, for example.

Another example of potentially wasted computation is that `Shape`s always compute the partial derivatives of their normal $\partial \mathbf{n}/\partial u$ and $\partial \mathbf{n}/\partial v$, even though these may not be needed. (Currently, they are only needed if the `BSDF` has specular components and ray differentials are being computed for the reflected or refracted rays.) There's no way for the intersection routine to know if they will be needed at intersection-time, however, so they are always computed. Indeed, there's currently no way to know if the BSDF will have specular components at ray–shape intersection time; the BSDF must be created by the material for this to be known, and the material needs the differential geometry at the intersection point to compute the BSDF!

One way to address this shortcoming would be to allow the material to conservatively describe all of the fields in the `DifferentialGeometry` that it needs, for example by setting bits in an `int`. This mask could be passed to the `Shape` intersection routines, which could then skip setting the member variables that aren't needed. This approach could further save execution time by allowing shapes to skip computing (u,v) parametric coordinates if they weren't needed, etc.

what else sucks about the design?

17.1.1 Abstraction Versus Efficiency

One of the basic tensions when designing interfaces between parts of a software system is making a reasonable trade-off between abstraction and efficiency. For example, many programmers who have just learned object-oriented programming concepts, religiously make all data in all classes private and provide accessor functions to get and set the values of the data items. For simple classes (e.g. Vector), we believe that approach is misguided, since it needlessly hides an implementation detail–that the class holds three floating-point coordinates–that can reasonably expected to never be changed (e.g. by switching to a spherical coordinate representation).

Of course, the other extreme of using no information-hiding and exposing all details of all classes internals, is also a recipe for a code maintenance nightmare. (Indeed, lrt's plug-in design has helped enforce compliance to the system's basic interfaces.) Yet we believe that there is nothing wrong with judiciously exposing basic design decisions and allowing code throughout the system to base their implementations on these decisions. For example, the fact that a Ray is represented with a point, a vector, and two-floating point values that give its extent is a decision that doesn't need to be hidden behind a layer of abstraction.

An important thing to keep in mind when writing a software system is its expected final size. The core of lrt (excluding its plug-in modules), where all of the basic interfaces, abstractions, and policy decisions are defined, is almost exactly ten thousand lines of code. Adding additional functionality to the system will generally only increase the amount of code in the plug-ins. The system is never going to grow to be a million lines of code; this fact can be reflected in the amount of information hiding used in the system. It would be a waste of programmer time (and likely a source of runtime inefficiency) to design the interfaces to be prepared for a system of that level of complexity.

36 Ray
27 Vector

17.1.2 Design Alternatives: Triangles Only

While the ability of ray tracing algorithm to handle a wide variety of shape representations directly without requiring tessellation is an elegant property, it's not as useful in practice as one might expect. Most scenes are either modeled directly with polygons ore with smooth surfaces like spline patches and subdivision surfaces that either have difficult-to-implement or relatively inefficient ray–shape intersection algorithms, if any. As such, they are usually tessellated into triangles for ray intersection tests in practice. In short, not many shapes that are commonly encountered in real-world scenes can be described well with quadrics!

Given this state of affairs, there are some advantages to designing the system around a common low-level shape representation and to only operate on triangles throughout much of the the pipeline. Such a renderer would still support a variety of primitives in the scene description but always would must tessellate them into a single representation like triangles. Advantages of this design include:

1. The renderer can depend on the fact that the triangle vertices can be completely transformed into world space, so that no transformations of rays into object space are ever necessary.

2. The acceleration structures can be specialized so that their nodes directly store the triangles that overlap them, improving the locality of the geometry in memory and making it possible to do ray–primitive intersection tests directly in the traversal routine, without needing to pass through two levels of virtual function calls, as to do intersection tests is currently necessary.

3. The presence of per-vertex normals in triangle meshes could be supported in a more direct manner, rather than through the `VertexTexture` class. The current abstraction for them in `lrt` leads to inefficiency in the computation of $\partial\mathbf{n}/\partial u$ and $\partial\mathbf{n}/\partial v$ for shading geometry in Section 10.2, for example, since these values could be computed directly from the vertex normals like $\partial \mathrm{p}/\partial u$ and $\partial \mathrm{p}/\partial v$ is in the `Triangle::Intersect()` routine currently.

4. Displacement mapping, where geometry is subdivided into small triangles which can then have their vertices perturbed procedurally or with texture maps, can be more easily implemented if all primitives are able to dice themselves into triangles.

These advantages are substantial, both in their performance implications and the complexity that they remove from the design of many parts of the system; for a professional rendering system, rather than one that with pedagogical goals like `lrt`, this alternative is probably worth pursuing.

17.1.3 Design Alternatives: Streaming Computation

One of the most exciting recent developments in computer architecture has been the development of *streaming* models of processor design (Kapasi, Rixner, Dally, Khailany, Ahn, Mattson, and Owens 2003). Streaming architectures differ from conventional CPUs in that they are highly-parallel, with many computational units on a single chip, and they are optimized for processing that has with a high ratio of computation to memory bandwidth used. They offer substantial performance advantages compared to general-purpose CPUs, which focus more on making a small number of computational units run quickly than exploiting parallelism over many units.

As modern graphics hardware has become programmable, it has used the streaming model. This has allowed commodity graphics hardware to see performance growth at a substantially faster rate than CPUs have been able to achieve—a sustained rate of doubling performance every six months, as opposed to the eighteen months that Moore's law would predict (Hanrahan 2002; Kirk 2002). Thus, almost all modern computers have specialized streaming processors inside them and are likely to continue to for the foreseeable future.

Streaming architectures like these have shown substantial promise in addressing the problem of what to do with the enormous growth in the number number of transistors that are available on a single chip—they can allocate them directly to more computational elements—while conventional CPU architectures don't support a programming model that makes it possible to keep more computational units busy, so they tend to be limited to using these transistors for secondary tasks like increasing the amount of on-chip cache memory.

Many graphics and media-related applications fit the streaming programming model well (Owens, Dally, Kapasi, Rixner, Mattson, and Mowery 2000; Owens, Khailany, Towles, and Dally 2002; Owens 2002). Purcell and collaborators have demonstrated the implementation of a general-purpose ray tracer on streaming graphics hardware with no native support for ray tracing, an important milestone in the process of applying the capabilities of these architectures to general-purpose computational problems (Purcell, Buck, Mark, and Hanrahan 2002; Purcell, Donner, Cammarano, Jensen, and Hanrahan 2003).

Writing efficient code for streaming architectures well requires different approaches than writing efficient code for CPUs. For example, the mechanisms for writing data to main memory are more constrained than on CPUs. However, the performance rewards for writing software that matches this model well will come to be substantial. Indeed, given the performance potentially available from architectures based on this model in the future, we believe that should a second edition of this book be written in a few years, the system may well target streaming processor architectures rather than general-purpose CPUs.

17.2 Major Projects

Most of the exercises at the end of the chapters have been reasonably self-contained, involving modifying or extending plug-in modules, rather than making substantial changes to the system's overall architecture. This section outlines a number of more ambitious changes that could be made to the system, involving more wide-reaching changes to its main abstractions and interfaces.

17.2.1 Parallel Rendering

Given the computational cost of ray tracing, there has been interest in parallel algorithms for ray tracing since shortly after the algorithm was first introduced (Cleary, Wyvill, Vatti, and Birtwistle 1983; Green and Paddon 1989; Badouel and Priol 1989). With the computational capabilities available in modern CPUs, researchers have started to demonstrate interactive ray tracing using tens of processors. For example, Parker et al developed an interactive ray-tracer on a shared-memory computer with 64 processors (Parker, Martin, Sloan, Shirley, Smits, and Hansen 1999) and Wald and collaborators have interactively ray traced complex scenes on a cluster of PCs (Wald, Slusallek, Benthin, and Wagner 2001; Wald, Slusallek, and Benthin 2001; Wald, Kollig, Benthin, Keller, and Slusallek 2002; Wald, Benthin, and Slusallek 2003).[1] Chalmers et al's book has good coverage of the state of the art in parallel ray tracing (Chalmers, Davis, and Reinhard 2002).

Parallelizing `lrt` would require changes to many parts of the system, depending on the particular approach used and the hardware target. One alternative is target the system to be run on a cluster of networked machines with each processes running in their own address spaces and communicating with the others via explicit message passing. The other main alternative is to design the system for a shared-memory environment, where a number of threads share the same memory

[1] Indeed, even if one isn't writing a parallel ray tracer, there is a wealth of information in papers on this topic about extremely efficient implementation of ray tracing algorithms, quickly building high-quality acceleration structures, cache-friendly programming for ray-tracing, and so forth.

space. This section will focus on issues related to the second approach, as more lower-level details of the system as currently written are relevant to discussion of this approach. See for example Wald et al's work and Chalmers et al's book for information about techniques related to the first approach.

The key problem to solve when multiple independent threads of execution have access to a shared region of memory is access of shared data. It is critical to ensure that one thread isn't part way through modifying a data structure that another is simultaneously reading, thus giving it inconsistent or invalid results, that two threads don't simultaneously try to update the same memory location, etc. The mechanism that solves this problem is *mutual exclusion*: threads must coordinate among themselves in some manner so that these problems don't occur. This can be done by convention–if only one thread needs to read or write some data structure and the other threads never access it, this problem can't come up.

More generally, mechanisms like *locks* can be used, where threads can request ownership of a lock object that they later give up, and the operating system ensures that no more than one thread holds the lock at any point in time. If shared data structures are protected by locks and all threads follow the convention that they acquire the appropriate lock before accessing or updating shared data structures, then the program can execute correctly. This idea is straightforward; implementing it consistently and correctly in a complex system is quite difficult, however, particularly since mutual exclusion bugs are typically non-deterministic across multiple runs of the program, as some show no errors and others may crash, purely due to a lucky or unlucky scheduling of threads by the operating system.

`Sampler`	237
`Scene`	8

If we were parallelizing `lrt`, we would probably start by parsing the scene description file and building the scene representation, including the acceleration structures, using just a single thread. This phase of the program's execution is difficult to parallelize, since all of the work being done is creation of data structures, and generally isn't the main bottleneck in image synthesis. We would then design the system to create multiple threads to share the work rendering the image, but then write out the final image and clean up with just a single thread.

Once the `Scene` has been created and the rendering threads are ready to start, a mechanism for determining what rendering work each thread should do is needed. An effective and straightforward approach is to partition the image plane into regions and to have different threads work on different regions, taking responsibility for all of the work needed to compute the contributions of a set of image samples.

The image can be partitioned statically or dynamically—a static partition divides the image into N regions for N threads; each one can work on its region without needing to communicate with the others. Because some parts of the image may be complex and require more computation than others, a better approach is to divide the image into more regions than there are threads and have threads access a shared data structure to determine what region of the image to work on next. The granularity of the image decomposition must be chosen carefully as well; if it is too fine, threads will finish work on their image region quickly and spend too much time waiting for access to the shared work queue. On the other hand, if the regions are too large, a thread working on a complex region may be the bottle neck that holds up the completion of rendering, as other threads sit idle. In either case, the `Sampler`s would need to be updated to support the partitioning method used, e.g. so that they would generate samples for multiple regions of the image as needed.

In simple ray tracing systems, the main rendering phase is easily parallelizable, since all of the data structures used except for the output image are read-only. Many threads can thus easily share the same scene data and only have to coordinate among each other for image output. (Indeed, for this reason ray tracing has somewhat unfairly been called “embarrassingly parallelizable”.) In more complex ray tracing systems, there are more issues to address. Fortunately, many of the classes in lrt are safe for multi-threading as they are currently written. For example, the Shapes, Cameras, Filters, VolumeRegions, and Materials operate on the data passed into their methods and don’t modify their member data after they have been created. As such, there is no potential for trouble if multiple threads are calling methods of one of them simultaneously.

Other parts of the system, for example the Film, would require locking, however. In lrt, having threads acquire and release a lock to the Film for each image sample they computed radiance for would probably lead to poor performance, as the time spent acquiring and releasing locks would dwarf the time spent on updating the image. The easiest way to work around this would be for threads to accumulate multiple sample contributions in thread-local memory and then pass them to the Film in batches of a few tens or hundreds of samples. An approach like this would amortize the time spent on locking over more work. Alternatively, each thread could hold a separate Film object and update it without locking, though the memory cost of this would be higher. When rendering completed, all of the individual Films would be are merged into a single Film object for final processing and output.

202	Camera
294	Film
281	Filter
598	IrradianceCache
141	MailboxPrim
375	Material
611	PhotonIntegrator
130	Primitive
63	Shape
465	VolumeRegion

There are also issues related to the accelerators. If an accelerator that refines all of the Primitives as the acceleration structure is being built, then the acceleration structure is read-only, except for the MailboxPrims, and locking may be avoided. If the accelerator refines primitives for intersection lazily, threads should use an *reader–writer* lock to protect the accelerator. A reader–writer lock allows multiple threads to hold a read lock for read-only access to a data structure, but only allows one to hold a writer lock, and only hands out a writer lock if no threads hold a reader lock. Primitive refinement and updates to the acceleration data structure should only be done by a thread holding a writer lock.

Mailboxing introduces a few additional complications to the acclerators for multi-threading. The variable used to assign rays ides is a potential source of contention, and having multiple threads try to update the shared mailboxes in the MailboxPrims is a likely source of trouble. The best solution to this is probably for each thread to assign rays mailbox ids independently of the others and for each thread to use local private memory to store the id of the last ray tested against each primitive.

Of the remainder of the system, most of the rest is thread-safe. The statistics system would need to be rewritten to accurately collect statistics in the presence of multiple threads, and the PhotonIntegrator and IrradianceCache would both need additional locking for their data structures, though.

17.2.2 Increased Scene Complexity

Given well-built accelerator structures, one of ray tracing’s strengths is that the time spent on ray–primitive intersections grows slowly with added scene complexity. As such, the possible complexity that a ray tracer can handle may be limited more by memory requirements than by computation time. Because the rays traced by a ray tracer may pass through many different regions of the scene during a short period of time, virtual memory often doesn't perform well with complex scenes, due to the incoherent memory access patterns.

One way to increase the potential complexity that a renderer is capable of handling is to reduce the memory used to store the scene. For the **ecosys.lrt** scene, for example, `lrt` currently uses approximately 300MB of memory for the one million triangles tin the scene—300 bytes per triangle, if all memory use (acceleration structures, geometry, materials, etc.) is amortized over the geometric complexity. We have previously written ray tracers that managed an average of 40 bytes per triangle, including all memory overhead. To do this successfully requires careful attention to memory use throughout the system; for example, we had the equivalent of three different `Triangle` implementations, one using 8 bit `u_char`s to store vertex indices, one using 16 bit `u_short`s, and one using 32 bit `u_int`s; the smallest one that was needed for the range of vertex indices in the mesh was chosen at runtime. Deering's paper on geometry compression and Ward's packed color format are both good inspirations for thinking along these lines (Deering 1995; Ward 1991).

`Triangle 90`

A more complex approach is geometry caching (Pharr and Hanrahan 1996), where the renderer holds a fixed amount of geometry in memory and discards geometry that hasn't been accessed recently. This approach is useful for scenes with a lot of tessellated geometry, where a compact higher-level shape representation like a subdivision surface can explode into a large number of triangles. When available memory is low, it can be worthwhile to discard some of this geometry and regenerate it later if needed.

The performance of such a cache can be substantially improved by reordering the rays that are traced in order to improve their spatial and thus memory coherence (Pharr, Kolb, Gershbein, and Hanrahan 1997). An easier-to-implement and more effective approach to improving the cache's behavior is described by Christensen et al, who wrote a ray tracer that uses simplified representations of the scene geometry in a geometry cache to substantially increase its effectiveness (Christensen, Laur, Fong, Wooten, , and Batali 2003).

17.2.3 Subsurface Scattering

There is an important assumption implicit in the BSDF and the scattering equation: that the only incident light that has an effect on the outgoing radiance at a point p is also incident on the surface at p–light that hits the surface at other points p' is assumed to not affect outgoing radiance at p. (Equivalently, the BSDF assumes that the distribution of incident radiance on the surface is uniform over a relatively large area of the surface with respect to the distance light travels beneath the surface.)

For many types of surfaces–human skin, marble, etc.–there is a significant amount of *subsurface light transport*, however. Indeed, the snow scene in Figure **snow figure in intro** simulates subsurface scattering in the snow on the trees and on the ground, giving it the soft diffused look of real snow. Light that enters such a surface at one location may travel for some distance underneath the surface, undergoing scattering there, before exiting at another position–see Figure 17.1.

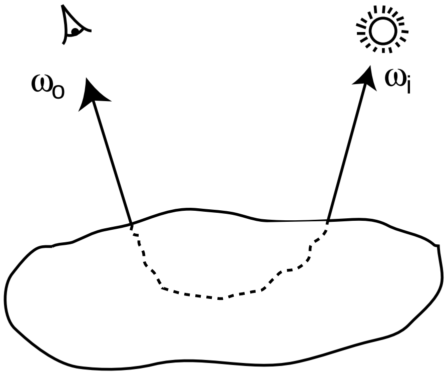


Figure 17.1: The bidirectional scattering-surface reflectance distribution function generalizes the BSDF to account for light that exits the surface at a point other than where it enters. It is more difficult to evaluate than the BSDF, though subsurface light transport can make a substantial contribution to the appearance of many real-world objects.

The *bidirectional scattering-surface reflectance distribution function* (BSSRDF) is the formalism that describes this scattering process. It is a distribution function $S(\mathrm{p}', \omega_i, \mathrm{p}, \omega_o)$ that describes the proportion of outgoing differential radiance at point p in direction ω_o due to differential irradiance at p' from direction ω_i. The scattering equation for the BSSRDF requires integration over surface area *and* incoming direction; with two more dimensions to integrate over, it is substantially more complex than Equation 5.4.9.

58 DifferentialGeometry
63 Shape

$$L_\mathrm{o}(\mathrm{p}, \omega_o) = \int_A \int_{S^2} S(\mathrm{p}', \omega_i, \mathrm{p}, \omega_o) \cos\theta_i \, \mathrm{d}\omega_i \, \mathrm{d}A$$

Fortunately, points p' that are far away from p generally contribute little to $L_\mathrm{o}(\mathrm{p}, \omega_o)$. This fact can be a substantial help in implementations of subsurface scattering algorithms.

Light transport inside a surface is volume light transport in participating media and is described by the equation of transfer—subsurface scattering is based on the same effects as light scattering in clouds, just at a smaller scale. Indeed, one of the key characteristics of subsurface scattering is that it makes possible a number of simplifications to the general volume light transport problem due to the fact that in the end, the quantity of interest is the distribution of light leaving a surface at a point, rather than the actual distribution of light inside the participating medium.

`lrt` currently has deep-seated assumptions that the BSDF is the abstraction that will be used to model reflection from surfaces. In order to support subsurface scattering, it would need to be extended to support methods for describing the volume scattering properties of translucent materials. Furthermore, integrators would be needed that applied subsurface light transport algorithms to compute reflection. Because some of these algorithms require the ability to determine more information about local surface geometry than is available in `DifferentialGeometry`, including the ability to move across points on the surface around the intersection point, the `Shape` interface will likely require extension to implement these algorithms as well.

Subsurface scattering was first introduced to graphics by Hanrahan and Krueger (Hanrahan and Krueger 1993), though their approach did not accurately simulate light that entered the object at points other than at the point being shaded. Dorsey et al applied photon maps to simulating true subsurface scattering (Dorsey, Edelman, Legakis, Jensen, and Pedersen 1999). Other recent work in this area includes papers by Pharr and Hanrahan (Pharr and Hanrahan 2000) and Jensen et al (Jensen, Marschner, Levoy, and Hanrahan 2001; Jensen and Buhler 2002).

The two most easily implemented approaches to this problem are volume photon maps for subsurface scattering, as described by Dorsey et al (Dorsey, Edelman, Legakis, Jensen, and Pedersen 1999) and Jensen et al's dipole approach (Jensen, Marschner, Levoy, and Hanrahan 2001; Jensen and Buhler 2002). The latter approach has been the basis of a number of fast implementations for scanline and interactive rendering (Hery 2003; Hao, Baby, and Varshney 2003; Dachsbacher and Stamminger 2003).

17.2.4 Precomputation for Interactive Rendering

Monte Carlo ray tracing algorithms have application beyond synthesis of final images for display. Recently there has been interest in algorithms for precomputing information about geometric models that encodes a description of how they respond to illumination, rather than computing how they reflect a particular distribution of illumination. This information can then be used in scanline or interactive z-buffer rendering to compute realistic shading based on arbitrary illumination conditions. For example, precomputed radiance transfer (PRT) algorithms account for inter-reflection of light in geometric models, representing it in a way that can be efficiently evaluated in interactive applications (Sloan, Kautz, and Snyder 2002; Sloan, Liu, Shum, and Snyder 2003; Sloan, Hall, Hart, and Snyder 2003).

Because `lrt`'s overall design is geared toward image synthesis rather than this type of pre-processing, extending the system to compute this information for a given model isn't just a matter of writing a new `SurfaceIntegrator`; the basic `Integrator::L()` interface isn't flexible enough to naturally match the needs of these algorithms. For example, the PRT algorithms in the papers cited above need to compute coefficients that encode response to incident illumination at the vertices of a triangle mesh; the task of the integrator is no longer to compute radiance along a set of independent rays. Furthermore, the `Sample` structure isn't rich enough to naturally encode where the samples are to be taken.

Another shortcoming in `lrt`'s interfaces is that they don't provide access to the scene geometry beyond its bounding boxes and ray–object intersection queries; there's no way to iterate over all of the vertices of a triangle mesh, for example. This was an intentional design decision, since by minimizing the number and variety of methods that `Shape`s must provide, it's easier to add new and unusual shapes to the system. For many types of shapes (spheres even), the very idea of iterating over its vertices has no meaning.

One approach to all of these problems is to write an integrator that overloads the `Integrator::Preprocess()` method and does this precomputation there. The scene description file could then be set up to render an image that was one pixel wide and one pixel tall, which could be ignored. In this case, the integrator would also be responsible for determining the points on the model at which values needed to be computed (e.g. by reading a file from disk, or via parameters to the integrator), doing the appropriate computation in `Preprocess()`, and also writing the results to disk.

Alternatively, given an appropriate importance function and leaving the emitted radiance function as an arbitrary function as opposed to a specific function specified by the lights in the scene, it's possible to express some precomputed radiance transfer algorithms in terms of a series of measurements that can be computed with the measurement equation. An alternative would be to derive an appropriate importance function for a particular PRT approach, show that the measurement equation gives the value of the appropriate quantity, and to design a replacement interface for the `Integrator::L()` method that takes a representation of an importance function and computes an estimate of the measurement equation for that importance function.

A.Utilities

598 IrradianceCache
611 PhotonIntegrator

In additional to all of the graphics-related code, there are a number of general low-level utility routines that are useful throughout the system. These routines, though key to lrt's operation, are relatively less interesting than the rest of the code in the system. It is good to have basic familiarity with them in order to understand other code, but understanding their implementation in detail isn't necessary to understand lrt.

This appendix starts by reviewing the parts of the C++ Standard Library used in lrt and then describes routines used for error reporting, memory management, pseudo-random number generation, and other basic details. These sections will omit much of the associated source code, as it isn't particularly interesting. Most of these routines are declared in core/lrt.h and defined in core/util.cpp. Finally, we wrap up with the implementations of generic octree and kd-tree data structures. These are currently only used by the IrradianceCache and PhotonIntegrator, but were written so that they could be re-used for other applications as well.

A.1 The C++ Standard Library

Finally, simple functions that compute the minimum or maximum of two values and a function that swaps the values of two variables. We just use the appropriate functions provided by the standard C++ library.

⟨*Global Include Files*⟩≡

```
#include <algorithm>
using std::min;
using std::max;
using std::swap;
using std::sort;
```

XXX start discussing container classes in general, then specialize down to vectors, sets, and maps XXX

For the benefit of readers unfamiliar with C++'s standard library, we will briefly review some of its facilities that we will be using. The `vector` class from the C++ standard library is a parameterized container class. It is similar to an array, though it can automatically grow as items are added to it. As it is a are template classe, a vector of `ints` (for example) is declared as `vector<int> v;`.

To add a new item to the end of a vector, a `push_back` method is available:

```
vector<int> vec;
for (int i = 0; i < 10; ++i)
    vec.push_back(i);
```

We can't say `vec[i] = i` in the above loop, since the vector needs to be informed that the user needs it to grow bigger, so that space may need to be allocated if needed.

A useful operation supported by `vectors` is the `reserve()` call. This lets us inform the vector the number of items that we will be adding to it; this lets it allocate sufficient space once, rather than needing to grow repeatedly as we insert items into it (e.g. `vec.reserve(100)` reserves 100 spaces in the vector.)

`vector` 658

The vector class provides a `size()` method, which returns the total number of items inside of it. This method can be be used in conjunction with the `[]` operator to access items in the vector directly:

```
for (int i = 0; i < vec.size(); ++i)
    printf ("%d\n", vec[i]);
```

After a `vector` has been filled (e.g. with `push_back()`), its members an be modified with the `[]` operator as well.

Vectors also provide an erase method; this takes two iterators to the sequence and removes all of the items from the first to the one before the last. Thus,

```
v.erase(v.begin(), v.end());
```

empties a vector completely.

Finally, the `pair` template class will be occasionally used; it provides a convenient way to construct a new object that holds two other objects. For example, if we're filling a hash table and are storing an array of pointers to hashed objects `Foo` with their integer hash values, we might declare an array of `pair<Foo *, int>`. Given a variable `p` that is a pair of objects, the constituent objects can be accessed as `p.first` and `p.second`. We can create a pair object with the `make_pair()` function:

```
int i = 0, Foo *foop = NULL;
pair<Foo *, int> p = make_pair(foop, i);
p.first = new Foo;
```

XXX sets and maps XXX

XXX string XXX

A.2 Communicating with the User

The functions and classes in this section are all involved with communicating information to the user of the system–reporting warnings and errors at rendering-time, gathering statistics about the runtime performance of the system, and other related tasks. The intent behind these routines is twofold: first, they are a convenience for code in other parts of the system, providing useful functionality in one place so that it doesn't need to be duplicated. Second, by centralizing communication with the user in a well-defined set of routines, they ensure that policy about how this communication is done can be easily modified. For example, if `lrt` was embedded in another application that had a graphical user interface, errors might be reported via a dialog box or a routine provided by the other application. If `printf()` calls were strewn throughout the system, this would be a more difficult modification to make.

A.2.1 Error Reporting

`lrt` provides four functions for reporting error conditions. In increasing severity, they are `Info()`, `Warning()`, `Error()`, and `Severe()`. All of them take a formatting string as their first argument and then a variable number of arguments providing values for the format. The syntax is identical to that used by the `printf` family of functions. For example, if `rayNum` is an `int`, then

```
Info("Now tracing ray number %d\n", rayNum);
```

will print the expected result.

`lrt` also has its own version of the standard `assert()` macro, here named `Assert()`. It checks that the given expression's value evaluates to true; if not, `Severe()` is called with information about where the assertion failed. It is used for basic sanity checks where if the test fails, there is little possibility of recovery and continuing execution. A failed assertion is a particularly user-unfriendly way to for system to exit, as the message printed will likely be cryptic to anyone other than the developer.

⟨*Global Inline Functions*⟩+≡

```
  #ifdef NDEBUG
  #define Assert(expr) ((void)0)
  #else
  #define Assert(expr) \
      ((expr) ? (void)0 : Severe("Assertion " #expr " failed in %s, line %d", \
                                  __FILE__, __LINE__))
  #endif // NDEBUG
```

A.2.2 Reporting Progress

The `ProgressReporter` class lets the system give the user some indication of how much of a lengthy task has been completed and how much more time it is expected to take. For example, the `Scene::Render()` method uses it to show how many of the camera rays have been traced. The current implementation prints a row of plus signs, the elapsed time, and the estimated remaining time to complete the task to the screen.

```
⟨Global Classes⟩≡
  struct ProgressReporter {
      ⟨ProgressReporter Public Methods⟩
      ⟨ProgressReporter Data⟩
  };
```

The constructor takes the total number of units of work to be done (e.g. the total number of camera rays to that will be traced), and a short string describing what task is bring performed.

```
⟨ProgressReporter Public Methods⟩≡
  ProgressReporter(int t, const string &title);
```

Once the `ProgressReporter` has been created, each call to its `Update()` method reports that one more unit of work has been completed. (An optional integer value can be used to indicate that multiple units are done.) As enough units are completed to warrant updating the display, an updated report is printed to the screen.

```
⟨ProgressReporter Public Methods⟩+≡
  void Update(int num = 1) const;
```

When all of the work is done, the `Done()` method should be called to ensure that the fact that the task has been finished is reported to the user.

```
⟨ProgressReporter Public Methods⟩+≡
  void Done() const;
```

A.2.3 Statistics

`lrt` also has a unified interface for gathering statistics which provides a single unified format for statistics reporting at the conclusion of rendering. At program termination, a single function call causes all of the statistics to be printed out.

Three types of statistics can be gathered:

- *Counters*: These provide a way to count the frequency of something—e.g. the total number of rays that are traced while making an image. They can also be track the minimum or maximum of some quantity (such as the number of primitives overlapping a kd-tree leaf node, for example.)
- *Percentages*: This records the percentage of times an event happens out of the possible times it could have happened—e.g. the number of successful ray–triangle intersection tests versus the total number of ray–triangle intersection tests.
- *Ratios*: These are essentially equivalent to percentages, but are just reported as a ratio. For example, a ratio is used to track the average number of grid voxels that the primitives in the scene overlap.

When a statistic type is reported to the statistics system, the caller must provide a category and a name for the particular statistic. The category gives a way to gather related types of statistics in output (e.g. all of the statistics gathered by the camera module can be reported together.) The name specifically describes the particular statistic.

For example, a counter is created by creating a `StatsCounter` object passing it a general category name (e.g. “Camera”) the name of the specific statistic (e.g. “Rays Generated”). The `StatsCounter` should be declared so that it is persistent throughout the program’s execution, for example as a dynamically-allocated or a `static` object. If it is only used in one function, it’s convenient to just declare it `static` inside that function.

⟨*Global Classes*⟩+≡

```
class StatsCounter {
public:
    ⟨StatsCounter Public Methods⟩
private:
    ⟨StatsCounter Private Data⟩
};
```

⟨*StatsCounter Public Methods*⟩≡

```
StatsCounter(const string &category, const string &name);
```

⟨*StatsCounter Private Data*⟩≡

```
StatsCounterType num;
```

The counter is incremented by calling the `++` operator of the variable that represents it.

⟨*StatsCounter Public Methods*⟩+≡

```
void operator++() { ++num; }
void operator++(int) { ++num; }
```

Alternatively, if the counter is being used to track the minimum or maximum of some range of values, the `Min()` and `Max()` methods can be used to report a new value to it.

⟨*StatsCounter Public Methods*⟩+≡

```
void Max(StatsCounterType val) { num = max(val, num); }
void Min(StatsCounterType val) { num = min(val, num); }
operator int() const { return (int)num; }
```

`StatsRatio` and `StatsPercentage` both have identical interfaces; the format of how they report their results is all that is different. Like the `StatsCOunter`, both also take the category name and statistic name as the only parameters to their constructors.

⟨*StatsRatio Public Methods*⟩+≡

```
void Add(int a, int b) { na += a; nb += b; }
```

⟨*StatsPercentage Public Methods*⟩≡

```
void Add(int a, int b) { na += a; nb += b; }
```

The `StatsPrint()` function prints all of the statistics that have been registered with the statistics system. It does some rudimentary work to sort them by category, make the columns of numbers line up, and report large numbers in units of thousands, millions, or billions, as appropriate.

⟨*Global Function Declarations*⟩+≡
```
  extern void StatsPrint(FILE *dest);
```

When the program is freeing up memory when it's about to exit, it calls the `StatsCleanup()` function, which frees up memory allocated by the statistics system.

⟨*Global Function Declarations*⟩+≡
```
  extern void StatsCleanup();
```

A.3 Memory Management

Memory management is often a complex issue in a system written in a language without garbage collection. It is a big more straightforward in `lrt`, thanks to the fact that most dynamic memory allocation is done as the scene representation is assembled as the scene description file is parsed, and most of this memory is in use until rendering is finished. Nevertheless, there are a few issues related to memory management that warrant classes and routines to address them. Most of these issues are performance-related, though an automatic reference-counting class is also useful for cleanly tracking the lifetimes of objects where multiple pointers to them are held in different parts of the system.

A.3.1 Variable stack allocation

Sometimes a routine needs to allocate a variable amount of temporary space for some computation it performs. If a relatively small amount of memory is needed, the overhead of dynamic allocation via `new` and `delete` (or `malloc()` and `free()`) may be high relative to the work actually being done. Instead, it can be substantially more efficient to use the `alloca()` call, which allocates memory on the routine's execution stack with just a few machine instructions. This memory is automatically deallocated when the routine exits, a feature than can also save bookkeeping work in the routine that uses it.

`alloca()` is an extremely useful tool, though there are two pitfalls to be aware of when using it. First, because the memory is deallocated when the routine returns,m the pointer must be not returned by the routine that allocated it or stored in a data-structure with a longer lifetime than the function that allocated it. (This pointer may be passed into other functions called by the allocating function, however.) Second, stack space is limited on some systems, so `alloca()` shouldn't be used for more than a few kilobytes. For larger allocations, the overhead of `new` and `delete` isn't too bad, anyway.

A.3.2 Reference-Counted Objects

In languages like C++, where the language doesn't provide automatic memory management and the user must deallocate dynamically allocated memory when through with it, it can be tricky to deal with the case when multiple objects hold a pointer to some other object. We want to free the second object as soon as no other object holds a pointer to it, but no sooner, so that we avoid both memory leaks as well as subtle errors due to memory corruption.

As long as there aren't circular references (e.g. object A holds a reference to object B, which holds a reference to object A.), a good solution to this problem is to use *reference counting*. An integer count is associated with objects that may be held by multiple objects; it is incremented when another object stores a reference to it and decremented when a reference goes away (e.g. due to the holding object being destroyed.) When the reference count goes to zero, the object can be safely freed.

We will define two classes to make it easy to use reference counted objects in `lrt`. First is a template, `ReferenceCounted`. An object of type `Foo` should inherit from `ReferenceCounted` if it is to be managed via reference counting. This adds an `nReferences` field to it. The actual count will be managed by the `Reference` class, defined below.

⟨*Global Classes*⟩+≡

```
class COREDLL ReferenceCounted {
public:
    ReferenceCounted() { nReferences = 0; }
    int nReferences;
private:
    ReferenceCounted(const ReferenceCounted &);
    ReferenceCounted &operator=(const ReferenceCounted &);
};
```

664 Reference

Rather than holding a pointer to a reference counted object `Foo`, other objects should declare a `Reference<Foo>` to hold the reference. The `Reference` template class handles updating the reference count as appropriate. For example, consider the function below:

```
void func() {
    Reference<Foo> r1 = new Foo;
    Reference<Foo> r2 = r1;
    r1 = new Foo;
    r2 = r1;
}
```

In the first line, a `Foo` object is allocated; `r1` holds a reference to it, and the object's `nReferences` count should be one. A second reference to the object is made in the second line; `r1` and `r2` refer to the same `Foo` object, with a reference count of two. Next, a new `Foo` object is allocated. When a reference to it is assigned to `r1`, the reference count of the original object is decremented to one. Now `r1` and `r2` point to separate objects. Finally, in the last line, `r2` is assigned to refer to the newly-allocated `Foo` object. The original `Foo` object now has zero references, and is automatically deleted. At the end of the function, when both `r1` and `r2` go out of scope, the reference count for the second `Foo` object goes to zero, causing it to be freed as well.

The only trick to all this is the low-level C++ syntax that makes all this happen automatically, so that other code can treat `References` as much like pointers as possible. (For example, if the `Foo` class has a `bar()` method, we'd like to be able to write code like `r1->bar()` in the function above, etc.)

⟨*Global Classes*⟩+≡

```
template <class T> class COREDLL Reference {
public:
    ⟨Reference Public Methods⟩
private:
    T *ptr;
};
```

The constructors are straightforward; after dealing with the possibility of `NULL` pointers, they just need to increment the reference count.

⟨*Reference Public Methods*⟩≡

```
Reference(T *p = NULL) {
    ptr = p;
    if (ptr) ++ptr->nReferences;
}
```

⟨*Reference Public Methods*⟩+≡

```
Reference(const Reference<T> &r) {
    ptr = r.ptr;
    if (ptr) ++ptr->nReferences;
}
```

Given a reference that is being assigned to hold a different reference, it is just necessary to decrement our old reference count and increment the count of the new object. The increments and decrements are ordered carefully below, so that code like `r1 = r1` doesn't inadvertently delete the object `r1` is refering to if it only has one reference.

⟨*Reference Public Methods*⟩+≡

```
Reference &operator=(const Reference<T> &r) {
    if (r.ptr) r.ptr->nReferences++;
    if (ptr && --ptr->nReferences == 0) delete ptr;
    ptr = r.ptr;
    return *this;
}
```

⟨*Reference Public Methods*⟩+≡

```
Reference &operator=(T *p) {
    if (p) p->nReferences++;
    if (ptr && --ptr->nReferences == 0) delete ptr;
    ptr = p;
    return *this;
}
```

⟨*Reference Public Methods*⟩+≡

```
~Reference() {
    if (ptr && --ptr->nReferences == 0)
        delete ptr;
}
```

Finally, a few methods take care of the details so that references can be used much like pointers–e.g. so that `->` can be used to call methods, etc. The `operator` `bool` method makes it possible to check to see if a reference points to a `NULL` object with code like `if (!r) ...`.

⟨*Reference Public Methods*⟩+≡

```
T *operator->() { return ptr; }
const T *operator->() const { return ptr; }
operator bool() const { return ptr != NULL; }
bool operator<(const Reference<T> &t2) const {
    return ptr < t2.ptr;
}
```

A.3.3 Cache-Friendly Memory Behavior

While the speed of modern CPUs has continued to increase at roughly the rate predicted by Moore's law (doubling every eighteen months), modern memory technologies haven't been able to keep up; memory has been getting faster at a rate of roughly 10% per year. The result of many years of this state of affairs is that a CPU may have to wait 100 or more cycles to receive the result of a read from amin memory. The CPU is usually idle for much of this time and a substantial amount of its computational potential may be lost.

One of the most effective architectural innovations to address this problem has been the addition of small fast cache memory either to the CPU itself or closer to it than main memory. The cache hold recently-accessed data and is able to service memory traffic from the CPU in many fewer cycles than if it had to go all of the way to main memory, thus helping the CPU be able to do more computation and less waiting for memory.

154	`KdTreeAccel`
417	`MIPMap`
664	`Reference`

Due to the high penalty for going to main memory (or alternatively, the large reward for being able to service a memory read from the cache), using algorithms and data structures in the system that are cache friendly can substantially improve its performance. This section will give an overview of current cache technology and discuss general programming techniques for improving cache performance. These techniques are used in many parts of `lrt`, particularly the `KdTreeAccel`, `MIPMap`, and `ImageFilm`.

When the CPU makes a memory reference, the cache (or caches) are checked to see if they have that memory location's value copied in the cache. If so, a cache *hit* is said to have occurred; otherwise there has been a cache *miss* and the value is loaded from memory and stored in the cache. The basic goal of cache-efficient programming is to minimize cache misses, for example by reducing the total number of memory accesses or by reordering them to improve the probability of a cache hit.

L1, L2, ...

Currently, 1-5 cycles to L1 cache, 5-20 to L2, 40-100 to main memory.

3 types of misses: compulsory, capacity, conflict.

cache line is...

When a miss occurs, date currently in the cache generally needs to be discarded to make room for the new data. How a cache entry is chosen to be discarded depends on how addresses in memory are mapped to cache lines. For a *direct-mapped* cache, each main memory address can only map to a single cache address,

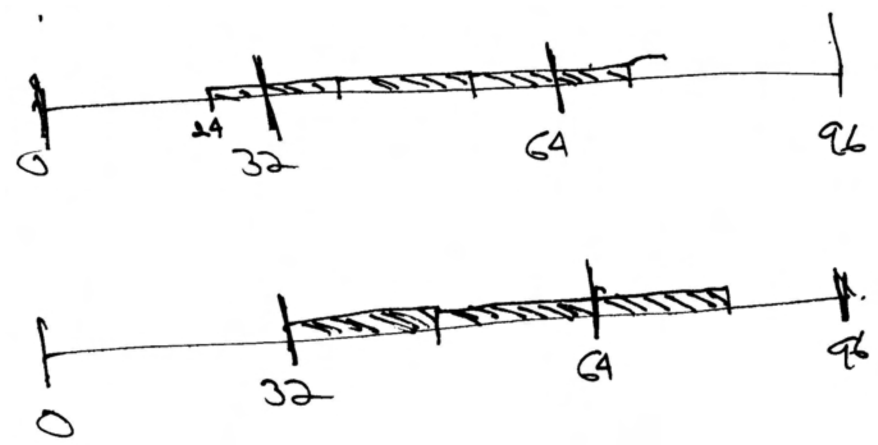


Figure A.1: Cache-aligned memory allocation ensures that the address returned is aligned with the start of a cache line. This figure shows the layout of three 16 byte objects in memory on a system with 32 byte cache lines. On the top, the starting address is not cache aligned–the first and last of the three objects span two cache lines, such that we may incur two cache misses when accessing their elements. On the bottom, the memory is cache aligned, guaranteeing that a maximum of one cache miss will be incurred per object.

so there is no question about which cache entry must be discarded for a new one. For a direct-mapped cache, if the cache has total size s, and block size b then memory ranges $[0, b-1]$, $[s, s+b-1]$, $[2s, 2s+b-1]$, $\ldots$ all map to the same cache line.

A more general organization is n-way set associative (e.g. $n = 2$, $n = 4$). Then any memory location maps to any one of n different cache lines. Then, the cache line to replace is chosen with a heuristic that tries to choose the one least likely to be accessed again in the future, e.g. by choosing the least recently used entry. Can improve performance by reducing thrashing.

One relatively-easy way to reduce the number of cache misses incurred by `lrt` and slightly improve its overall performance is to make sure that some memory allocations are well aligned with the blocks of memory that the cache manages. Figure A.1 shows the basic setting. There, we are allocating three 16 byte objects on a system with 32 byte cache lines. By making sure that the first object starts at the start of a cache entry (bottom), we ensure that we will incur no more than one cache miss when accessing any one of the items. If we expect to be accessing only some of the items (as opposed to looping over all of them in order), then performance will generally be improved with cache-aligned allocation. (`lrt`'s overall performance speed up by approximately 3% when allocation for the kd-tree accelerator in Section 4.4 was switched to use aligned allocation.)

The `AllocAligned()` and `FreeAligned()` functions provide a wrapper around system memory allocation and freeing routines to do cache-aligned allocation. If the pre-processor constant `L1_CACHE_LINE_SIZE` hasn't been set previously, we guess a cache line size of 64 bytes, which is typical of many architectures today.

$\langle$*Memory Allocation Functions*$\rangle\equiv$

```
void *AllocAligned(size_t size) {
#ifndef L1_CACHE_LINE_SIZE
#define L1_CACHE_LINE_SIZE 64
#endif
    return memalign(L1_CACHE_LINE_SIZE, size);
}
```

$\langle$*Memory Allocation Functions*$\rangle+\equiv$

```
void FreeAligned(void *ptr) {
    free(ptr);
}
```

Another family of techniques for improving cache performance is based on reorganizing data structures themselves. For example, simply reducing the size of an frequently-used data structure by packing multiple integer values into a single word with bit fields can be helpful. Doing so improves the *spatial locality* of memory access at runtime, since code that accesses more than one of the packed values won't incur more than one cache miss to get them all. Furthermore, by reducing the overall size of the structure, this technique can reduce capacity misses, if fewer cache lines are needed to store the structure.

If not all of the elements of a structure are usually accessed, there are a few possible strategies. For example, if the structure is 128 bytes large and the computer has 64 byte cache lines, two cache misses may be needed to access it. If the commonly-used fields are moved to the first 64 bytes rather than being spread throughout, then no more than one cache miss will be incurred when only those fields are needed (Truong, Bodin, and Seznec 1998).

A related technique is "splitting", where data structures are split into "hot" and "cold" parts, each stored in separate regions of memory. For example, given an array of some structure type, splitting it into two arrays where the less frequently accessed parts of the original structure are in a new structure in a second array reduces misses when accessing the "hot" parts since each cache line is able to hold more of them–once it is filled, multiple structures can be accessed without additional misses. And in a similar manner to other cache-friendly techniques, "cold" data doesn't displace useful information in the cache except when it is actually needed.

Trees: eliminate pointers, e.g. for a fully-populated binary tree, allocate as array, left child of node n is node $2n$, right child is $2n+1$.

Prefetching

compulsory, capacity, conflict misses

A.3.4 Arena-Based Allocation

The conventional wisdom about memory allocation is that allocation based on the system's `malloc()` and `new()` routines is slow and that it is often worth-while to write custom allocation routines for objects that will be frequently allocated and freed. However, this conventional wisdom seems to be wrong. Wilson et al (Wilson, Johnstone, Neely, and Boles 1995), Johnstone and Wilson (Johnstone and Wilson 1999), and Berger et al (Berger, Zorn, and McKinley 2001; Berger, Zorn, and McKinley 2002) have all investigated the performance of memory allocation routines with real applications and have found that user-written allocators almost always result in *worse* performance in both execution time and memory use compared to a well-written generic system memory allocator.

One type of custom allocation technique that was found to be useful was *arena-based allocation*, which allows the user to quickly allocate objects from a large contiguous region of memory. In this scheme, individual objects can't be freed; only when the lifetime of all of the allocated objects is over is the entire region of memory freed. This is a natural fit for many of the allocation patterns in `lrt`. `lrt` has two classes that provide this type of allocation. The first, `ObjectArena`, is a template parameterized by the type of object to be allocated. The other, `MemoryArena`, supports variable-sized allocation. In some sense, it subsumes the functionality of the `ObjectArena`, though the `ObjectArena` provides slightly cleaner syntax in code that only needs an arena for a single type.

There are two main advantages to arena-based allocation: first allocation is extremely fast, usually just requiring a pointer increment. Second, it can improve locality of reference and lead to fewer cache misses, since the allocated objects are contiguous in memory, without any extra space between them taken up for bookkeeping by the dynamic memory allocator.

```
MemoryArena 670
     vector 658
```

⟨*Global Classes*⟩+≡

```
template <class T> class ObjectArena {
public:
    ⟨ObjectArena Public Methods⟩
private:
    ⟨ObjectArena Private Data⟩
};
```

⟨*ObjectArena Public Methods*⟩≡

```
ObjectArena() {
    nAvailable = 0;
}
```

⟨*ObjectArena Private Data*⟩≡

```
T *mem;
int nAvailable;
vector<T *> toDelete;
```

The `Alloc()` method returns a pointer to enough memory to hold a single instance of the type `T` that it handles allocation for. If there isn't enough space for another `T` object in the last allocated block of memory, a new block is allocated.

⟨*ObjectArena Public Methods*⟩+≡

```
T *Alloc() {
    if (nAvailable == 0) {
        int nAlloc = max((unsigned long)16,
            (unsigned long)(65536/sizeof(T)));
        mem = (T *)AllocAligned(nAlloc * sizeof(T));
        nAvailable = nAlloc;
        toDelete.push_back(mem);
    }
    --nAvailable;
    return mem++;
}
```

A more convenient alternative is provided by its `operator T *` method, which just calls `Alloc()` itself. This allows one to write code that uses C++'s placement new operator to simultaneously allocate memory and run the constructor to initialize it:

```
ObjectArena<Foo> arena;
Foo *f = new (arena) Foo;
```

⟨*ObjectArena Public Methods*⟩+≡

```
operator T *() {
    return Alloc();
}
```

667 `AllocAligned()`
667 `FreeAligned()`
670 `MemoryArena`

The `ObjectArena`'s destructor frees all of the blocks of memory that have been allocated by the arena. To free this memory sooner, the `FreeAll()` method can be called. Note that the `ObjectArena` doesn't run the objects' destructors when the memory is freed; it is the caller's responsibility to do this manually if appropriate:

```
f-> Foo();
```

Note that the object must not be `delete`d.

⟨*ObjectArena Public Methods*⟩+≡

```
void FreeAll() {
    for (u_int i = 0; i < toDelete.size(); ++i)
        FreeAligned(toDelete[i]);
    toDelete.erase(toDelete.begin(), toDelete.end());
    nAvailable = 0;
}
```

The `MemoryArena` quickly allocates memory for objects of varaible size by handing out adjacent blocks of memory for a pre-allocated block. Like the `ObjectArena`, it also does not support freeing of individual blocks of memory, only freeing of all of the memory in the zone all at once. Thus, it is useful for cases where a number of allocations need to be done with low overhead, and where all of the allocated objects have similar lifetimes and where it's easy to determine when all of them have been deallocated.

⟨Global Classes⟩+≡

```
class MemoryArena {
public:
    ⟨MemoryArena Public Methods⟩
private:
    ⟨MemoryArena Private Data⟩
};
```

The implementation of the `MemoryArena` is pretty straightforward. It allocates memory in chunks of size `blockSize`, the value of which is set by a parameter passed to the constructor. It maintains a pointer to the current block of memory and the offset of the first free location in the block.

⟨MemoryArena Public Methods⟩≡

```
MemoryArena(u_int bs = 32768) {
    blockSize = bs;
    curBlockPos = 0;
    currentBlock = (char *)AllocAligned(blockSize);
}
```

The `MemoryArena` also uses two `vector`s to hold pointers to blocks of memory that have been fully used as well as available blocks that were previously allocated but aren't currently in use.

`vector 658`

⟨MemoryArena Private Data⟩≡

```
u_int curBlockPos, blockSize;
char *currentBlock;
vector<char *> usedBlocks, availableBlocks;
```

To service an allocation request, we round up the requested amount of memory so that it is in tune with the computer's word alignment requirements. We then see if the current block has enough space to handle the request, allocating a new block if necessary. We then return the appropriate pointer and update our record of our current offset into the current block.

⟨MemoryArena Public Methods⟩+≡

```
void *Alloc(u_int sz) {
    ⟨Round up sz to minimum machine alignment⟩
    if (curBlockPos + sz > blockSize) {
        ⟨Get new block of memory for MemoryArena⟩
        curBlockPos = 0;
    }
    void *ret = currentBlock + curBlockPos;
    curBlockPos += sz;
    return ret;
}
```

Most modern computer architectures impose alignment requirements on the positioning of objects in memory. For example, it is typically a requirement that 4 byte wide `float` values be stored at memory locations that are themselves multiples of 4. To be safe, we will always hand out memory at 8 byte boundaries, which is a conservative requirement for modern architectures.

⟨*Round up `sz` to minimum machine alignment*⟩≡

```
sz = ((sz + 7) & (~7));
```

We first need to store the pointer to the current block of memory on the `usedBlocks` list so that it's not lost; later, when `MemoryArena::FreeAll()` is called, we'll be able to reuse the block for the next series of allocations. We then check to see if there are any already-allocated free blocks in the `availableBlocks` list before falling back to calling the system allocation routine to allocate a brand new block.

⟨*Get new block of memory for `MemoryArena`*⟩≡

```
usedBlocks.push_back(currentBlock);
if (availableBlocks.size()) {
    currentBlock = availableBlocks.back();
    availableBlocks.pop_back();
}
else
    currentBlock = (char *)AllocAligned(max(sz, blockSize));
```

When the user is done with all of the memory, we just reset our offset in the current block and move all of the memory from the `usedBlocks` list onto the `availableBlocks` list.

⟨*MemoryArena Public Methods*⟩+≡

672 BlockedArray

```
void FreeAll() {
    curBlockPos = 0;
    while (usedBlocks.size()) {
        availableBlocks.push_back(usedBlocks.back());
        usedBlocks.pop_back();
    }
}
```

A.3.5 Blocked 2D Arrays

In C++, the natural way that 2D arrays are arranged in memory is to have entire rows of values contiguous in memory, as shown on the left side of Figure A.2. For an array indexed by (u,v), the problem with this layout is that nearby (u,v) array positions will often map to substantially different memory locations; while adjacent values in the u direction are adjacent in memory, for an array of objects of some type `T`, adjacent values in v are `width * sizeof(T)` locations apart. Thus, spatially-coherent access in terms of 2D array positions does not necessarily lead to the spatially-coherent memory access patterns that modern memory caches depend on.

For all but the smallest arrays, the adjacent values in the y direction will be on different cache lines, and thus, if the cost of a cache miss is incurred to reference a value at a particular location (u,v), there is no chance that handling that miss will also load into memory the data for values $(u,v+1)$, $(u,v-1)$, etc.

To address these problems, the `BlockedArray` template implements a generic 2D array of values, with the items ordered in memory using a *blocked* memory layout, as shown on the right side of Figure A.2. The array is subdivided into square blocks of a small fixed size that is a power of two, `BLOCK_SIZE`. Within each block, the items are laid out row-by-row, as in the usual layout. This scheme

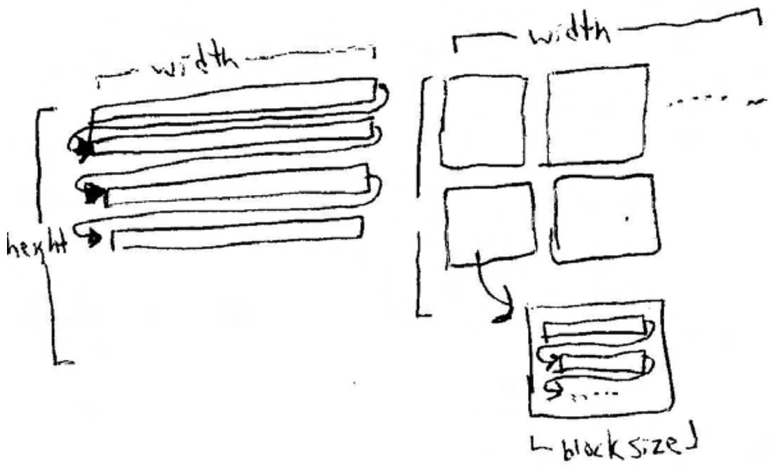


Figure A.2: In C++, the natural layout for a 2D array of size `width*height` is a block of `width*height` entries, where the (u, v) array element is at the `u+v*width` offset. This approach is shown on the left. On the right is a blocked array; it has been split into smaller square blocks, each of which is laid out linearly. Though it is slightly more complex to find the memory location associated with a given (u, v) array position, the improvement in cache performance due to more coherent memory access patterns often more than makes up for this for overall faster performance.

substantially improves the memory coherence of memory references in practice, with very small added computation to compute the memory address for a particular position (Lam, Rothberg, and Wolf 1991).

The size of the blocks is set via a template parameter, `logBlockSize`.

⟨*Global Classes*⟩+≡

```
template<class T, int logBlockSize> class BlockedArray {
public:
    ⟨BlockedArray Public Methods⟩
private:
    ⟨BlockedArray Private Data⟩
};
```

The constructor allocates space for the array and optionally initializes it from a standard array passed in. Because the array size may not be an exact multiple of the block size, it may be necessary to round up the size in one or both directions to find the total amount of memory needed for the blocked array. The `RoundUp()` method, defined below, rounds up the value passed to it to be a multiple of the block size, if it isn't already; this gives us the size that must be allocated in each direction.

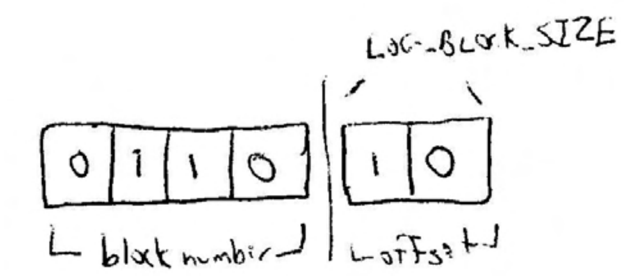


Figure A.3: Given an array coordinate, the (u, v) block number that it is in can be found by shifting off the `logBlockSize` low order bits for both u and v. (For example, with a `logBlockSize` of 2 and thus a block size of 4, we can see see that this correctly maps 1D array positions from 0 to 3 to block 0, 4 to 7 to block 1, etc. To find the offset within the particular block, we just mask off the high order bits, leaving the `logBlockSize` low order bits. Because the block size is a power of two, these computations can all be done with a few efficient bit operations.

⟨*BlockedArray Public Methods*⟩≡

```
BlockedArray(int nu, int nv, const T *d = NULL) {
    uRes = nu;
    vRes = nv;
    uBlocks = RoundUp(uRes) >> logBlockSize;
    int nAlloc = RoundUp(uRes) * RoundUp(vRes);
    data = (T *)AllocAligned(nAlloc * sizeof(T));
    for (int i = 0; i < nAlloc; ++i)
        new (&data[i]) T();
    if (d)
        for (int v = 0; v < nv; ++v)
            for (int u = 0; u < nu; ++u)
                (*this)(u, v) = d[v * uRes + u];
}
```

667 `AllocAligned()`
672 `BlockedArray`

⟨*BlockedArray Private Data*⟩≡

```
T *data;
int uRes, vRes, uBlocks;
```

⟨*BlockedArray Public Methods*⟩+≡

```
int BlockSize() const { return 1 << logBlockSize; }
int RoundUp(int x) const {
    return (x + BlockSize() - 1) & ~(BlockSize() - 1);
}
```

For convenience, the `BlockedArray` can also reports its size in each dimension.

⟨*BlockedArray Public Methods*⟩+≡

```
int uSize() const { return uRes; }
int vSize() const { return vRes; }
```

Looking up a value from a particular (u, v) position in the array requires some slightly tricky indexing work to find the memory offset for that value. There are two steps to this process: first, finding which block the value is in, and second, finding its offset within the block. Because the block sizes are always powers of two, the `logBlockSize` low order bits in each of the *u* and *v* array positions give the offset within the block and the high order bits give the block number–Figure A.3 shows how the address bits are computed.

$\langle$*BlockedArray Public Methods*$\rangle+\equiv$

```
int Block(int a) const { return a >> logBlockSize; }
int Offset(int a) const { return (a & (BlockSize() - 1)); }
```

Then, given the block number (b_u, b_v) and the offset within the block (o_u, o_v), we need to compute what memory location this maps to in the blocked array layout. First consider the task of finding the starting address of the block; since the blocks are laid out row-by-row, this corresponds to the block number `bu + bv * uBlocks`, where `uBlocks` is the number of blocks in the *u* direction, which was computed in the constructor. Because each block has `BlockSize()*BlockSize()` values in it, the product of the block number and this value gives us the offset to start of the block. We then just need to account for the additional offset from the start of the block, which is `ou + ov * BlockSize()`.

$\langle$*BlockedArray Public Methods*$\rangle+\equiv$

```
T &operator()(int u, int v) {
    int bu = Block(u), bv = Block(v);
    int ou = Offset(u), ov = Offset(v);
    int offset = BlockSize() * BlockSize() * (uBlocks * bv + bu);
    offset += BlockSize() * ov + ou;
    return data[offset];
}
```

We will also provide a convenience method to convert the blocked array back to a standard C++ array; the caller is responsible for allocating enough memory to hold the `uRes * vRes` values.

$\langle$*BlockedArray Public Methods*$\rangle+\equiv$

```
void GetLinearArray(T *a) const {
    for (int v = 0; v < vRes; ++v)
        for (int u = 0; u < uRes; ++u)
            *a++ = (*this)(u, v);
}
```

A.4 Mathematical Routines

A.4.1 2x2 Linear Systems

There are a number of places throughout `lrt` where we need to solve a 2x2 linear system $Ax = B$ of the form

$$\begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \end{bmatrix}$$

for the values x_0 and x_1. The SolveLinearSystem2x2() routine implements the closed form solution to such a system, returning false if the determinant of A suggests that the system is degenerate and there are no solutions.

⟨*Matrix Method Definitions*⟩≡

```
bool SolveLinearSystem2x2(const Float A[2][2],
        const Float B[2], Float x[2]) {
    Float det = A[0][0]*A[1][1] - A[0][1]*A[1][0];
    if (fabsf(det) < 1e-5)
        return false;
    Float invDet = 1.0f/det;
    x[0] = (A[1][1]*B[0] - A[0][1]*B[1]) * invDet;
    x[1] = (A[0][0]*B[1] - A[1][0]*B[0]) * invDet;
    return true;
}
```

A.4.2 4x4 Matrices

The Matrix4x4 structure provides a low-level representation of 4 by 4 matrices. It is an integral part of the Transform class, which holds two matrices, one representing a transform and the other representing its inverse. Because many Shapes often have identical transformations, Matrix4x4s are reference counted so that the Transform class only needs to hold shared Matrix4x4 references, rather than holding the much larger complete matrices.

663	ReferenceCounted
63	Shape
43	Transform

⟨*Global Classes*⟩+≡

```
struct Matrix4x4 : public ReferenceCounted {
    ⟨Matrix4x4 Public Methods⟩
    Float m[4][4];
};
```

The default constructor sets the matrix to the identity matrix.

⟨*Matrix4x4 Public Methods*⟩≡

```
Matrix4x4() {
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            if (i == j) m[i][j] = 1.;
            else m[i][j] = 0.;
}
```

There are also provide constructors that allow the user to pass an array of floats, or sixteen individual floats to initialize the Matrix4x4 with.

⟨*Matrix4x4 Public Methods*⟩+≡

```
Matrix4x4(Float mat[4][4]);
Matrix4x4(Float t00, Float t01, Float t02, Float t03,
          Float t10, Float t11, Float t12, Float t13,
          Float t20, Float t21, Float t22, Float t23,
          Float t30, Float t31, Float t32, Float t33);
```

The `Matrix4x4` supports a few low-level matrix operations, each of which returns a reference to a newly allocated matrix that holds the result of the operation. For example, `Transpose()` transposes the matrix's elements.

⟨*Matrix Method Definitions*⟩+≡

```
Reference<Matrix4x4> Matrix4x4::Transpose() const {
    return new Matrix4x4(m[0][0], m[1][0], m[2][0], m[3][0],
                         m[0][1], m[1][1], m[2][1], m[3][1],
                         m[0][2], m[1][2], m[2][2], m[3][2],
                         m[0][3], m[1][3], m[2][3], m[3][3]);
}
```

The product of two matrices M_1 and M_2 is computed by setting the (i,j)th element of the result to the sum of the products of the elements of the ith row of M_1 with the jth column of M_2.

⟨*Matrix4x4 Public Methods*⟩+≡

```
static Reference<Matrix4x4> Mul(const Reference<Matrix4x4> &m1,
        const Reference<Matrix4x4> &m2) {
    Float r[4][4];
    for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j)
            r[i][j] = m1->m[i][0] * m2->m[0][j] +
                      m1->m[i][1] * m2->m[1][j] +
                      m1->m[i][2] * m2->m[2][j] +
                      m1->m[i][3] * m2->m[3][j];
    return new Matrix4x4(r);
}
```

`Matrix4x4 675`
`Reference 664`

Finally, `Inverse()` returns the inverse of the matrix. The implementation uses a numerically stable Gauss–Jordan elimination routine to compute the inverse.

⟨*Matrix4x4 Public Methods*⟩+≡

```
Reference<Matrix4x4> Inverse() const;
```

A.4.3 Utility Functions

Now we'll define a few very short functions that will be useful throughout the program. First is `Lerp()`, which performs linear interpolation between two values, `v1` and `v2`, with position given by the `t` parameter. When `t` is zero, the result is `v1`; when `t` is one, the result is `v2`.

`Lerp()` is implemented with the computation

$$(1-t)v_1 + tv_2$$

in the function below, rather than in the more terse and potentially more efficient form of

$$v_1 + t(v_2 - v_1)$$

in the interests of reducing floating-point error. In the event that the magnitudes of `v1` and `v2` are substantially different, it may not be possible to accurately represent the difference $v_2 - v_1$ with a floating-point value in the latter form. When this inaccurate value is scaled by t and added to v_1, it may be substantially different than the correct value. With the first formulation, not only is this problem avoided but `Lerp()` returns *exactly* the values `v1` and `v2` when `pos` has values 0 and 1, respectively, and always returns a value in the range $[v_1, v_2]$ if t is in $[0, 1]$; this property also isn't guaranteed by the second approach.

⟨*Global Inline Functions*⟩+≡

```
inline Float Lerp(Float t, Float v1, Float v2) {
    return (1.f - t) * v1 + t * v2;
}
```

`Clamp()` clamps a value `val` to be between the values `low` and `high`. If `val` is out of that range, `low` or `high` is returned as appropriate.

⟨*Global Inline Functions*⟩+≡

```
inline Float Clamp(Float val, Float low, Float high) {
    if (val < low) return low;
    else if (val > high) return high;
    else return val;
}
```

`Mod()` computes the remainder of a/b. This function is handy since it behaves predictably and reasonably with negative numbers—the C and C++ standards leave the behavior of the `%` operator undefined in that case.

678 M_PI

⟨*Global Inline Functions*⟩+≡

```
inline int Mod(int a, int b) {
    int n = int(a/b);
    a -= n*b;
    if (a < 0)
        a += b;
    return a;
}
```

Two simple functions convert from angles expressed in degrees to radians, and vice versa.

⟨*Global Inline Functions*⟩+≡

```
inline Float Radians(Float deg) { return ((Float)M_PI/180.f) * deg; }
inline Float Degrees(Float rad) { return (180.f/(Float)M_PI) * rad; }
```

Because the math library doesn't provide a base-2 logarithm function, we provide one here.

⟨*Global Inline Functions*⟩+≡

```
inline Float Log2(Float x) {
    static Float invLog2 = 1.f / logf(2.f);
    return logf(x) * invLog2;
}
```

Sometimes we just need an integer valued base-2 logarithm. (For example in the EWA texture filtering implementation.) This can be computed quickly by shifting and masking the mantissa bits from an IEEE float and returning them directly.

⟨*Global Inline Functions*⟩+≡

```
inline int Log2Int(Float v) {
    return ((*(int *) &v) >> 23) - 127;
}
```

Finally, some fancy bit-twiddling can quickly determine if a given integer is an exact power of two and can round an integer up to the next power of two greater than or equal to it.

⟨*Global Inline Functions*⟩+≡

```
inline bool IsPowerOf2(int v) {
    return (v & (v - 1)) == 0;
}
```

⟨*Global Inline Functions*⟩+≡

```
inline u_int RoundUpPow2(u_int v) {
    v--;
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    return v+1;
}
```

Unfortunately, not all system `math.h` files store the value of π in `M_PI`. If it is not defined, we do it ourself.

⟨*Global Constants*⟩+≡

```
#ifndef M_PI
#define M_PI           3.14159265358979323846f
#endif
```

Other useful constants include $\frac{1}{255}$, $\frac{1}{\pi}$, and $\frac{1}{2\pi}$:

⟨*Global Constants*⟩+≡

```
#define INV_255 0.00392156862745098039f
#define INV_PI  0.31830988618379067154f
#define INV_TWOPI  0.15915494309189533577f
```

We define a generally-useful `INFINITY` value using `FLT_MAX` from the standard math library, which is the largest representable floating point number.

⟨*Global Constants*⟩+≡

```
#ifndef INFINITY
#define INFINITY FLT_MAX
#endif
```

A.4.4 Floating-point to integer conversion

On the x86 architecture, it can take as many as 80 processor cycles to convert a floating-point value to an integer value; the conversion to integer in a simple sequence of code like:

```
Float a = ..., b = ...;
int i = (int)(a * b);
```

may take 80 times longer than the multiplication `a*b`! The root problem is that the floating-point unit's rounding mode needs to be changed from the default before the built-in conversion instruction is used, and this requires an expensive flush of the entire floating-point pipeline.

`lrt` needs to convert `Float`s to integers in a number of performance-sensitive areas. These include the sample filtering code, where for every camera sample it is necessary to compute the extent of pixel samples that are affected by the sample based on the filter extent. Similarly, in the Perlin noise evaluation routines, the integer lattice cell that a floating-point position is in must be found.

Sree Kotay and Mike Herf have developed some techniques to these conversions much more quickly without needing to change the rounding mode by taking advantage of low-level knowledge of the layout of IEEE floating-point values in memory. Using these routines in `lrt` sped it up by up to 5% for some scenes. We will not include the details of their implementation here as they are arcane, to say the least. However, there are four key functions, all of them taking a `Float` value and returning an integer:

1. `Float2Int(f)`: This is the same as the basic cast `(int)f`.
2. `Round2Int(f)`: This rounds the floating point value `f` to the nearest integer, returning the result as an `int`.
3. `Floor2Int(f)`: The first integer value less than or equal to `f` is returned.
4. `Ceil2Int(f)`: And similarly, the first integer value greater than or equal to `f` is returned.

A.4.5 Pseudo-Random Numbers

`lrt` uses a custom pseudo-random number generator rather than calling the one provided by the system. Doing so is worth the trouble, both because it allows us to ensure that the system produces the same results regardless of machine architecture and C library implementation but also because many systems provide random number generation routines with poor statistical distributions.

The random number generator used in `lrt` is the "Mersenne Twister" by Makoto Matsumoto and Takuji Nishimura. The code to the random number generator is both complex and subtle, and we will not attempt to explain it here. Nevertheless, it is one of the best random number generators known, can be implemented very efficiently, and has a period of $2^{19937}-1$ before it repeats the series again. Pointers to the paper describing its algorithm can be found at the end of this section.

The `RandomFloat()` and `RandomUInt()` routines provide uniformly-distributed floating point values from the range $[0,1]$ and $[0,2^{32}-1]$, respectively.

⟨*Global Inline Functions*⟩+≡

```
inline Float RandomFloat();
inline unsigned long RandomUInt();
```

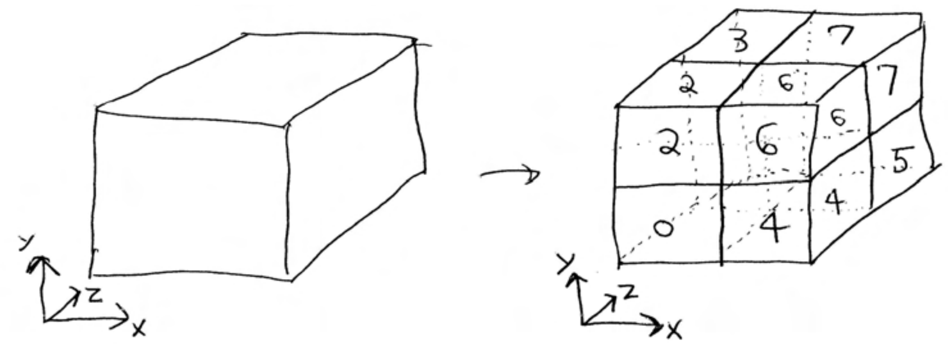


Figure A.4: Basic octree refinement: starting with an axis-aligned bounding box, the octree is defined by progressively splitting each node into eight equal-sized child nodes. The order in which the child nodes are assigned numbers 0...7 is significant–details of this will be explained later in this section. Different sub-trees may be refined to different depths, giving an adaptive discretization of 3D space.

A.5 Octrees

IrradianceCache 598
OctNode 681

The octree is a three-dimensional data structure that recursively splits a region of space into axis-aligned boxes. Starting with a single box at the top level, each level of refinement splits the previous level's boxes into eight child boxes, each covering one-eigth of the volume of the previous ones–Figure A.4 shows the basic idea. The octree implementation defined in this section is in the file `core/octree.h`.

While octrees have many applications, among them acceleration structures for ray tracing, the implementation here helps accelerate the query "given a collection of objects and their axis-aligned bounding boxes, which of their bounds overlap a given point"? For large numbers of objects, using an octree to answer this question can be substantially faster than looping over all of the objects directly. `lrt` currently only uses octrees to store the irradiance estimates computed by the `IrradianceCache` integrator–each estimate has a bounding box associated with it that gives the maximum region of space where the estimate may be used for shading computations. However, the octree implementation is independent of the irradiance cache in order to simplify the description of the `IrradianceCache` as well as to make it easier to re-use the octree for other applications.

First, we will define the `OctNode` structure, which represents a node of the tree. It holds pointers to the eight possible children of the node (some or all of which may be `NULL`) and a `vector` of `NodeData` objects. `NodeData` is the object type that the user of the octree wants to store in the tree; for the `IrradianceCache`, it's the `IrradSample` structure, which records the results from a single irradiance estimate. The constructor and destructor of the `OctNode` just initialize the `children` to `NULL` and delete them, respectively; their implementations won't be shown here.

⟨*Octree Declarations*⟩≡

```
template <class NodeData> struct OctNode {
    OctNode() {
        for (int i = 0; i < 8; ++i)
            children[i] = NULL;
    }
    ~OctNode() {
        for (int i = 0; i < 8; ++i)
            delete children[i];
    }
    OctNode *children[8];
    vector<NodeData> data;
};
```

The `Octree` class is parameterized by the `NodeData` type as well as by a "lookup procedure", `LookupProc`, which is essentially a callback function that lets the `Octree` communicate back to the caller which elements of `NodeData` overlap a given lookup position.

⟨*Octree Declarations*⟩+≡

```
template <class NodeData, class LookupProc> class Octree {
public:
    ⟨Octree Public Methods⟩
private:
    ⟨Octree Private Data⟩
};
```

```
 38 BBox
154 KdTreeAccel
658 vector
```

The constructor takes overall bound of the tree and a maximum depth beyond which it will never refine nodes to have children.

⟨*Octree Public Methods*⟩≡

```
Octree(const BBox &b, int md = 16)
    : bound(b) {
    maxDepth = md;
}
```

⟨*Octree Private Data*⟩≡

```
int maxDepth;
BBox bound;
OctNode<NodeData> root;
```

To add an item to the tree, the octree walks down levels the tree, creating new nodes as needed until termination criteria are met and the item is added to the node it overlaps. Similarly to the `KdTreeAccel` of Section 4.4, performance is substantially affected depending on what the specific termination criteria are. For example, we could trivially decide to never refine the tree and add all items to the root node. This would be a valid octree, though it would perform poorly for large numbers of objects. However, if the tree is refined too much, items may span many nodes, leading to excessive memory use.

The entrypoint for adding an item directly calls an internal "add item" method with a few additional parameters, including the current node being considered, the bounding box of the node, and the squared length of the diagonal of the data item's bounding box. This method calls itself recursively as it works down the octree to the nodes where the item is stored.

⟨*Octree Public Methods*⟩+≡

```
void Add(const NodeData &dataItem, const BBox &dataBound) {
    add(&root, bound, dataItem, dataBound,
        DistanceSquared(dataBound.pMin, dataBound.pMax));
}
```

The internal `add()` method either adds the item to the current node and returns or determines which child nodes the item overlaps, allocates them if necessary, and recursively calls `add()` to allow the children to decide whether to stop the recursion and add the item or to continue down the tree.

⟨*Octree Method Definitions*⟩≡

```
template <class NodeData, class LookupProc>
void Octree<NodeData, LookupProc>::add(OctNode<NodeData> *node,
        const BBox &nodeBound, const NodeData &dataItem,
        const BBox &dataBound, Float diag2, int depth) {
    ⟨Possibly add data item to current octree node⟩
    ⟨Otherwise add data item to octree children⟩
}
```

BBox	38
BBox::pMax	39
BBox::pMin	39
DistanceSquared()	34
OctNode	681
Octree	681
Octree::bound	681
Octree::maxDepth	681

The item is added to the current node once the maximum tree depth is reached or when the length of the diagonal of the node is less than the length of the diagonal of the item's bounds. This ensures that the item overlaps a relatively small number of tree nodes, while not being too small relative to the extent of the nodes that it's added to. Figure A.5 shows the basic operation of the algorithm in two dimensions (where the corresponding data structure is known as an *quadtree*).

⟨*Possibly add data item to current octree node*⟩≡

```
if (depth == maxDepth ||
    DistanceSquared(nodeBound.pMin, nodeBound.pMax) < diag2) {
    node->data.push_back(dataItem);
    return;
}
```

If the algorithm continues down the tree, it needs to determine which of the child nodes the item's bounding box overlaps. The fragment ⟨*Determine which children the item overlaps*⟩ efficiently sets an array of boolean values, `over[]`, such that the *i*th element is true only if the bounds of the data item being added overlap the *i*th child of the current node. It can then loop over the eight children and recursively call `add()` for the ones that the object overlaps.

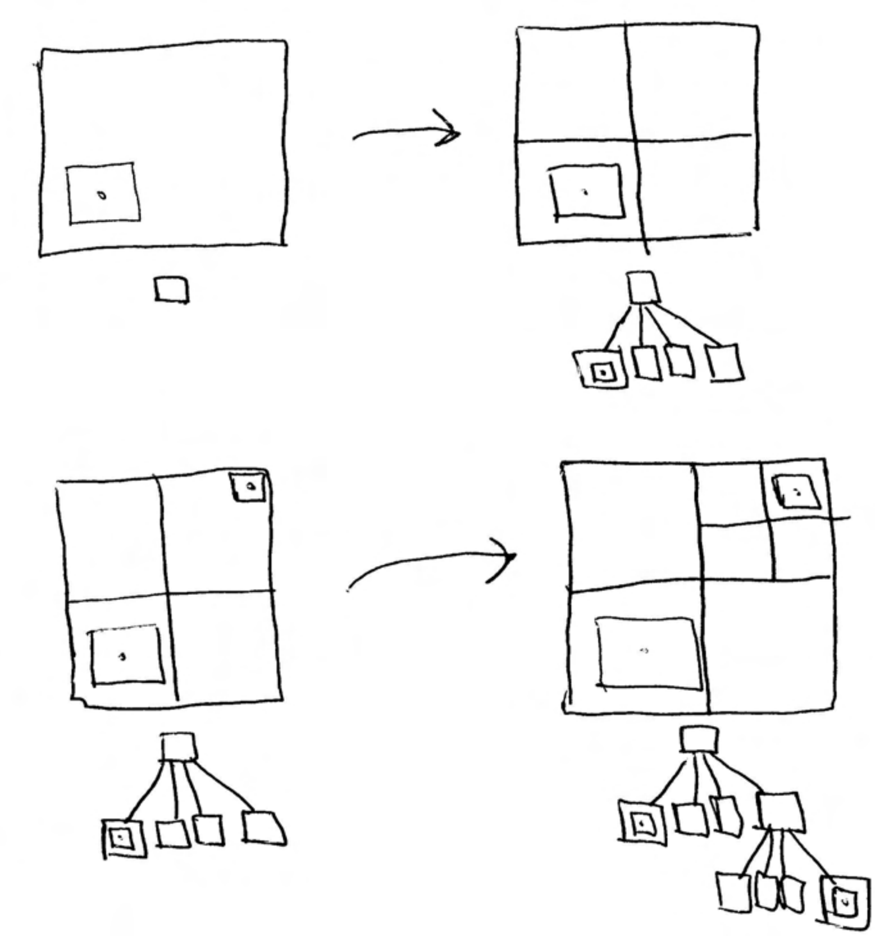


Figure A.5: Creation of a quadtree (the 2D analog of an octree): in the top row, the tree is comprised of just the root node and an object with bounds around a given point is being added. The tree's topology is illustrated with a small box beneath it, corresponding to the root node with no children. Next, the tree is refined one level and the object is added to the single child node that it overlaps (again shown schematically underneath the tree.) In the bottom row, another new object with a smaller bounding box than the first is being added. The `add()` method ends up going down two levels of the tree before adding the item, again to the single node that it overlaps. In general, items may be stored in multiple nodes of the tree.

```
⟨Otherwise add data item to octree children⟩≡
  Point pMid = .5 * nodeBound.pMin + .5 * nodeBound.pMax;
  ⟨Determine which children the item overlaps⟩
  for (int child = 0; child < 8; ++child) {
      if (!over[child]) continue;
      if (!node->children[child])
          node->children[child] = new OctNode<NodeData>;
      ⟨Compute childBound for octree child child⟩
      add(node->children[child], childBound, dataItem, dataBound, diag2,
          depth+1);
  }
```

Rather than computing the bounds of each child and doing a bounding box overlap test, it is possible save work by taking advantage of symmetries, such as the fact that if the *x* range of the object's bounding box is entirely on the left side of the plane that splits the tree node in the *x* direction, there is no way that it overlaps any of the four child nodes on the right side. Careful selection of the child node numbering scheme in Figure A.4 is key to the success of this approach. Here the child node numbering scheme becomes important. The child nodes are numbered such that the low bit of a child's number is zero if its *z* component is on the low side of the *z* splitting plane and one if it is on the high side. Similarly, the second bit is set based on which side the child is of the *y* plane, and the third bit is set based on its position with respect to the *x* plane. Given boolean variables that classify a child node with respect to the splitting planes (`true` if it is above the plane, the child number of a given node is equal to:

`BBox::pMax`	39
`BBox::pMin`	39
`OctNode`	681
`OctNode::children`	681
`Point`	33

```
4*(xHigh ?  1 :  0) + 2*(yHigh ?  1 :  0) + 1*(zHigh ?  1
:  0)
```

It is possible to quickly determine which child nodes a given bounding box overlaps by classifying its extent with respect to the center point of the node. For example, if the bounding box's starting *x* value is less than the midpoint, then the node potentially overlaps children numbers 0, 1, 2, and 3. If its ending *x* value is greater than the midpoint, it potentially overlaps 4, 5, 6, and 7. We check the *y* and *z* dimensions in turn, computing the logical and of the results: the item only overlaps a child node if it overlaps its extent in all three dimensions.

⟨*Determine which children the item overlaps*⟩≡

```
bool over[8];
over[0] = over[1] = over[2] = over[3] = (dataBound.pMin.x <= pMid.x);
over[4] = over[5] = over[6] = over[7] = (dataBound.pMax.x  > pMid.x);
over[0] &= (dataBound.pMin.y <= pMid.y);
over[1] &= (dataBound.pMin.y <= pMid.y);
over[4] &= (dataBound.pMin.y <= pMid.y);
over[5] &= (dataBound.pMin.y <= pMid.y);
over[2] &= (dataBound.pMax.y  > pMid.y);
over[3] &= (dataBound.pMax.y  > pMid.y);
over[6] &= (dataBound.pMax.y  > pMid.y);
over[7] &= (dataBound.pMax.y  > pMid.y);
over[0] &= (dataBound.pMin.z <= pMid.z);
over[2] &= (dataBound.pMin.z <= pMid.z);
over[4] &= (dataBound.pMin.z <= pMid.z);
over[6] &= (dataBound.pMin.z <= pMid.z);
over[1] &= (dataBound.pMax.z  > pMid.z);
over[3] &= (dataBound.pMax.z  > pMid.z);
over[5] &= (dataBound.pMax.z  > pMid.z);
over[7] &= (dataBound.pMax.z  > pMid.z);
```

Here again taking advantage of the child node numbering scheme, the bounding box of a particular child is easily found based on the child number and the parent node's bound.

38	BBox
40	BBox::Inside()
681	Octree::bound
686	Octree::lookup()
33	Point

⟨*Compute* `childBound` *for octree child* `child`⟩≡

```
BBox childBound;
childBound.pMin.x = (child & 4) ? pMid.x : nodeBound.pMin.x;
childBound.pMax.x = (child & 4) ? nodeBound.pMax.x : pMid.x;
childBound.pMin.y = (child & 2) ? pMid.y : nodeBound.pMin.y;
childBound.pMax.y = (child & 2) ? nodeBound.pMax.y : pMid.y;
childBound.pMin.z = (child & 1) ? pMid.z : nodeBound.pMin.z;
childBound.pMax.z = (child & 1) ? nodeBound.pMax.z : pMid.z;
```

After items have been added to the tree, the user can use the tree find up the items that have bounds that overlap a given point. The `Lookup()` method walks down the tree, processing the nodes that the given point overlaps. The user-supplied callback, `process` is called for each `NodeData` item that overlaps the given point.

As with the `Add()` method, the main lookup function directly calls to an internal version that takes a pointer to the current node and the current node's bounds.

⟨*Octree Public Methods*⟩+≡

```
void Lookup(const Point &p, const LookupProc &process) {
    if (!bound.Inside(p)) return;
    lookup(&root, bound, p, process);
}
```

If the internal lookup function has been called with a given node, the point `p` must be inside the node. The user-supplied callback is called for each `NodeData` item that is stored in the octree node, allowing the user to do whatever processing is appropriate.[1] The callback must either be a pointer to a function that takes a position and a NodeData object, or a class that has an operator() method that takes those arguments.

After the items are processed, this method continues down the tree into the single child node that p is inside the bottom is reached.

```
⟨Octree Method Definitions⟩+≡
  template <class NodeData, class LookupProc>
  void Octree<NodeData, LookupProc>::lookup(OctNode<NodeData> *node,
          const BBox &nodeBound, const Point &p,
          const LookupProc &process) {
      for (u_int i = 0; i < node->data.size(); ++i)
          process(p, node->data[i]);
      ⟨Determine which octree child node p is inside⟩
      if (node->children[child]) {
          ⟨Compute childBound for octree child child⟩
          lookup(node->children[child], childBound, p, process);
      }
  }
```

Again taking advantage of the child numbering scheme, it is possible to quickly determine which child a point overlaps by classifying it with respect to the center of the parent node in each direcion.

```
⟨Determine which octree child node p is inside⟩≡
  Point pMid = .5f * nodeBound.pMin + .5f * nodeBound.pMax;
  int child = (p.x > pMid.x ? 4 : 0) +
      (p.y > pMid.y ? 2 : 0) + (p.z > pMid.z ? 1 : 0);
```

BBox	38
BBox::pMax	39
BBox::pMin	39
KdTree	688
KdTreeAccel	154
OctNode	681
OctNode::children	681
OctNode::data	681
Octree	681
PhotonIntegrator	611
Point	33

A.6 Kd-Trees

Like the octree, the kd-tree is another data structure that accelerates the processing of spatial data. (And of course it was the basis of the KdTreeAccel in Chapter 4. In contrast to the octree, where the data items had a known bounding box and the caller wanted to find all items that overlap a given point, the generic kd-tree that is the topic of this section is useful for handling data items that are just single points in space, with no associated bound, but where the caller wants to find all such points within a user-supplied distance of a given point. It is a key component of the PhotonIntegrator.

The KdTree that will be described here is generally similar to the KdTreeAccel of Section 4.4 in that 3D space is progressively split in half by planes. There are two main differences, however:

- Here, each tree node stores a single data item. As such, there is exactly one kd-tree node for each data item stored in the tree.

[1] Note that the Octree actually passes all of the data items in the node to the call back, not just the subset of them that p is inside the bounds of. This isn't too much of a problem in practice, since the node using the Octree can always store a BBox in the NodeData and do the check itself. For uses where it doesn't matter if a few extra NodeData items are passed back, writing the implementation in this way saves the BBox storage space.

- Because each item being stored is just a single point, items never straddle the splitting plane and need to be stored on both sides of a split.

One result of these differences is that it is possible to build a perfectly balanced tree, which can improve the efficiency of data lookups. Furthermore, the total number of nodes in the tree will always be exactly the same as the number of data items.

Like the `KdTreeAccel`, the implementation here stores all of the nodes of the tree in a single contiguous array. The left child of any node with a left child will be one after the node in the array, and the `rightChild` member of `KdNode` gives the offset to the right child of the node, if any. `rightChild` will be set to a very large number if there is no right child.

To further improve the cache efficiency of the kd-tree, we will apply the cache optimization described previously of separating “hot” and “cold” data. “Hot” data is data that is frequently accessed while we are traversing the tree, while cold data is less frequently accessed. By splitting the kd-tree node data structure into two pieces in this way, we are able to pack hot data close together in contiguous memory, improving the cache efficiency of accessing it, since we can pack more tree nodes into a single cache line.

The representation of the basic structure of the tree is stored in `KdNode` structures; they record information about splitting and the children of the node, if any. The additional data that the user wants to store at each node is stored in a separate array, indexed identically to the `KdNode` array (i.e. the `i`th `KdNode`’s data is `i` into the data array.)

154 `KdTreeAccel`

⟨*KdTree Declarations*⟩≡

```
struct KdNode {
    void init(Float p, u_int a) {
        splitPos = p;
        splitAxis = a;
        rightChild = ~0;
        hasLeftChild = 0;
    }
    void initLeaf() {
        splitAxis = 3;
        rightChild = ~0;
        hasLeftChild = 0;
    }
    ⟨KdNode Data⟩
};
```

⟨*KdNode Data*⟩≡

```
Float splitPos;
u_int splitAxis:2;
u_int hasLeftChild:1;
u_int rightChild:29;
```

The `KdTree` is parameterized by the type of object stored in the nodes, `NodeData` and the type of callback object that is used for reporting which nodes are within a given search radius of the lookup position. The `KdTree` depends on `NodeData` having a `Point p` member variable that gives its position; if it is used with a type without such a variable, a compile-time error will result.

⟨*KdTree Declarations*⟩+≡

```
template <class NodeData, class LookupProc> class KdTree {
public:
    ⟨KdTree Public Methods⟩
private:
    ⟨KdTree Private Data⟩
};
```

⟨*KdTree Private Data*⟩≡

```
KdNode *nodes;
NodeData *nodeData;
u_int nNodes, nextFreeNode;
```

All of the data items must be supplied to the `KdTree` constructor. Incremental addition or removal of `NodeData` items isn't supported since this functionality isn't needed in `lrt` and doing so keeps the implementation here straightforward. The constructor allocates all of the memory needed for the tree and the data and calls the recursive tree construction function, which takes an array of `NodeData` pointers, chooses one to store in the current node, and recursively builds the children of that node with the remaining data items. Because the tree is built such that the indexing of the `node` and `nodeData` arrays is identical, the `nodeData` array will be initialized from the items in the vector passed into the constructor as the tree is being built—only when a `KdNode` at a particular offset in `nodes` is initialized do we know which position to store the corresponding `NodeData` item.

`AllocAligned()`	667
`KdNode`	687
`KdTree::recursiveBuild()`	689
`vector`	658

⟨*KdTree Method Definitions*⟩≡

```
template <class NodeData, class LookupProc>
KdTree<NodeData, LookupProc>::KdTree(const vector<NodeData> &d) {
    nNodes = d.size();
    nextFreeNode = 1;
    nodes = (KdNode *)AllocAligned(nNodes * sizeof(KdNode));
    nodeData = new NodeData[nNodes];
    vector<const NodeData *> buildNodes;
    for (u_int i = 0; i < nNodes; ++i)
        buildNodes.push_back(&d[i]);
    recursiveBuild(0, 0, nNodes, buildNodes);
}
```

Tree construction is handled by the `recursiveBuild()` method. It takes the node number of the current node to be initialized and offsets into the array data indicating the subset of data items $[\mathrm{start}, \mathrm{end})$ from the `buildNodes` array to be stored beneath this node.

The tree building process selects the "middle" element of the user-supplied data (to be explained precisely below) and partitions the data, so that all items below the middle are in the first half of the array and all items above the middle are in the second half. It constructs a node with the middle element as its data item and then recursively initializes the two children of the node by processing the first and

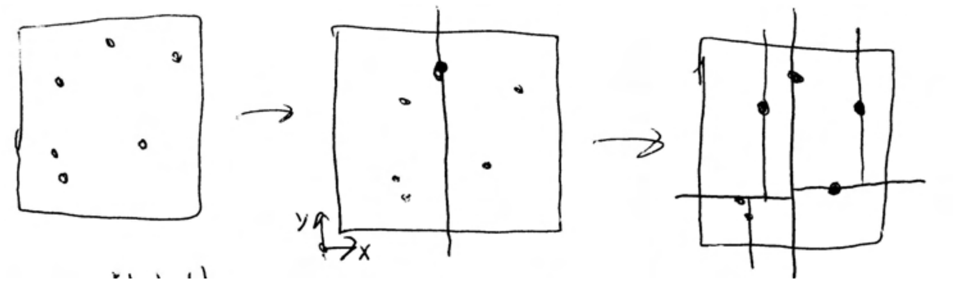


Figure A.6: Creation of a kd-tree to store a set of points: given a collection of points (left), a split direction is first chosen. Here, we have decided to split in the *x* direction. We find the point in the middle along *x* and split along the plane that goes through the point. Roughly half of points are to the left of the splitting plane and half are to the right. We then continue recursively in each half, allocating new tree nodes, splitting and partitioning, until all data points have been processed.

second halves of the array (minus the middle element.) Figure A.6 shows the basic process of bulding the kd-tree.

⟨*KdTree Method Definitions*⟩+≡

```
template <class NodeData, class LookupProc> void
KdTree<NodeData, LookupProc>::recursiveBuild(u_int nodeNum,
        int start, int end,
        vector<const NodeData *> &buildNodes) {
    ⟨Create leaf node of kd-tree if we've reached the bottom⟩
    ⟨Choose split direction and partition data⟩
    ⟨Allocate kd-tree node and continue recursively⟩
}
```

687 KdNode::initLeaf()
688 KdTree
658 vector

When there just a single item to be processed, the bottom of the tree has been reached, the node is flagged as a leaf, and the `nodeData` array item can be initialized to be at the appropriate offset.

⟨*Create leaf node of kd-tree if we've reached the bottom*⟩≡

```
if (start + 1 == end) {
    nodes[nodeNum].initLeaf();
    nodeData[nodeNum] = *buildNodes[start];
    return;
}
```

Otherwise, the data is partitioned into two halves and a non-leaf node is initialized. Splitting is along whichever coordinate axis the remaining data items span the largest extent. The standard library `nth_element()` function finds the middle node along that axis; it takes takes three pointers `start`, `mid`, and `end` into a sequence and partitions it such that the `mid`th element is in the position it would be in if the range was sorted and where all elements from `start` to `mid-1` are less than mid, and elements from `mid+1` to `end` are greater than mid. This can all be done more quickly than sorting the entire range–in $O(n)$ time rather than $O(n \log n)$.

⟨*Choose split direction and partition data*⟩≡

```
⟨Compute bounds of data from start to end⟩
int splitAxis = bound.MaximumExtent();
int splitPos = (start+end)/2;
std::nth_element(&buildNodes[start], &buildNodes[splitPos],
    &buildNodes[end], CompareNode<NodeData>(splitAxis));
```

⟨*Compute bounds of data from* `start` *to* `end`⟩≡

```
BBox bound;
for (int i = start; i < end; ++i)
    bound = Union(bound, buildNodes[i]->p);
```

The `nth_element()` function needs a “comparison object” that determines the ordering between two data elements. `CompareNode` compares positions along the chosen axis.

⟨*KdTree Declarations*⟩+≡

```
template<class NodeData> struct CompareNode {
    CompareNode(int a) { axis = a; }
    int axis;
    bool operator()(const NodeData *d1,
            const NodeData *d2) const {
        return d1->p[axis] < d2->p[axis];
    }
};
```

BBox 38
Octree 681

Once the data has been partitioned, the current node is initialized to store the middle item and its two children are recursively initialized with the two sets of remaining items.

⟨*Allocate kd-tree node and continue recursively*⟩≡

```
nodes[nodeNum].init(buildNodes[splitPos]->p[splitAxis], splitAxis);
nodeData[nodeNum] = *buildNodes[splitPos];
if (start < splitPos) {
    nodes[nodeNum].hasLeftChild = 1;
    u_int childNum = nextFreeNode++;
    recursiveBuild(childNum, start, splitPos, buildNodes);
}
if (splitPos+1 < end) {
    nodes[nodeNum].rightChild = nextFreeNode++;
    recursiveBuild(nodes[nodeNum].rightChild, splitPos+1, end,
        buildNodes);
}
```

When code elsewhere wants to look up items from the tree, they provide a point `p`, a callback procedure (similar to the one used in the `Octree` above), and a maximum squared search radius. (Taking the squared radius rather than the radius directly leads to some optimizations in the traversal code below.) All data items within that radius will be passed back to the caller.

Rather than being passed by value, the squared search radius is passed into the lookup function by reference. This will allow the lookup routine to pass it to the

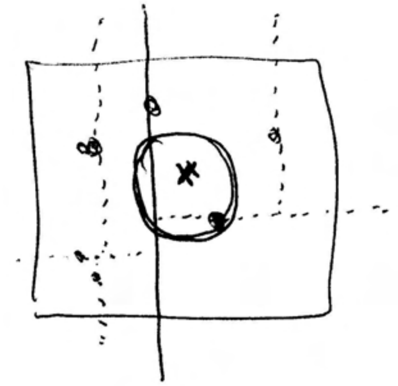


Figure A.7: Basic process of kd-tree lookups. The point marked with an “x” is the lookup position, and the region of interest is denoted by the circular region around it. At the root node of the tree (indicated by a bold splitting line), the data item is outside of the region of interest, so it is not handed to the callback function. However, the region overlaps both children of the node, so we have to recursively consider each of them. We will consider the right child (child number one) first, however, in order to examine the nearby data items before examining the ones farther away.

688 KdTree
692 KdTree::recursiveLookup()
623 PhotonProcess
33 Point

callback procedure by reference, so that it can potentially reduce the search radius as the search goes on. This can speed up lookups when the callback routine can determine partway along that a smaller search radius was appropriate (as the `PhotonProcess` object does.)

As usual, the lookup method immediately calls an internal lookup procedure, passing in a pointer to the current node to be processed.

⟨*KdTree Method Definitions*⟩+≡

```
template <class NodeData, class LookupProc> void
KdTree<NodeData, LookupProc>::Lookup(const Point &p,
        const LookupProc &proc, Float &maxDistSquared) const {
    recursiveLookup(0, p, proc, maxDistSquared);
}
```

The lookup function has two responsibilities: it needs to recursively process the children of the current node based on which of them the search region overlaps (potentially both of them), and it needs to call the callback routine, passing it the data item in the current node if it is inside the search radius. Figure A.7 shows the basic process.

⟨*KdTree Method Definitions*⟩+≡

```
template <class NodeData, class LookupProc> void
KdTree<NodeData, LookupProc>::recursiveLookup(u_int nodeNum,
        const Point &p, const LookupProc &process,
        Float &maxDistSquared) const {
    KdNode *node = &nodes[nodeNum];
    ⟨Process kd-tree node’s children⟩
    ⟨Hand kd-tree node to processing function⟩
}
```

The tree is traversed in a depth-first manner, first heading toward the leaf nodes that are close to the lookup point p before the callback method is called to process data items. This approach ensures that data points are passed to the callback function in a generally near-to-far order. If the caller is only interested in finding a fixed number of points around the lookup point, after which it will end the search, this is a more efficient order.

Therefore, this method first walks down the side of the tree that the current point lies on. Only after that lookup has returned does it go down the other side, if the search radius indicates that the lookup region covers both sides of the tree. Leaf nodes are denoted by a value of 3 in the node’s splitAxis field, in which case thesesteps are skipped.

KdNode	687
KdTree	688
KdTree::nodes	688
Point	33

⟨*Process kd-tree node’s children*⟩≡

```
int axis = node->splitAxis;
if (axis != 3) {
    Float dist2 = (p[axis] - node->splitPos) * (p[axis] - node->splitPos);
    if (p[axis] <= node->splitPos) {
        if (node->hasLeftChild)
            recursiveLookup(nodeNum+1, p, process, maxDistSquared);
        if (dist2 < maxDistSquared && node->rightChild < nNodes)
            recursiveLookup(node->rightChild, p, process, maxDistSquared);
    }
    else {
        if (node->rightChild < nNodes)
            recursiveLookup(node->rightChild, p, process, maxDistSquared);
        if (dist2 < maxDistSquared && node->hasLeftChild)
            recursiveLookup(nodeNum+1, p, process, maxDistSquared);
    }
}
```

Finally, at the end of the lookup function, the check is made to see if the point stored in the node is inside the search radius. An expensive square root computation is saved by comparing squared distances, and the data item is passed back to the callback function if appropriate. In addition to doing whatever processing it needs to do based on the item, the callback function may decrease maxDistSquared in order to reduce the region of space searched for the remainder of the processing.

⟨*Hand kd-tree node to processing function*⟩≡

```
Float dist2 = DistanceSquared(nodeData[nodeNum].p, p);
if (dist2 < maxDistSquared)
    process(nodeData[nodeNum], dist2, maxDistSquared);
```

A.7 Image Input Output

An important part of a rendering system is its routines for reading and writing images. Because many image file formats have been developed, all of them special in their own way, it is helpful for a renderer to support a variety of them. Unfortunately, the details of dealing with a particular format, let alone a variety of formats, are tedious. Therefore, lrt has a plug-in architecture for image input and output. The rest of the system uses three functions for image and output that hide the details of loading and using these plug-ins.

A.8 Main Include File

The lrt.h file is included by all other that all source files in the system. It has global function declarations and inline functions, a few small widely-used classes (e.g. the statistics-related classes) and other widely-used data.

All files that include lrt.h get a number of other include files in the process; this makes it possible for them to just include lrt.h and not repeatedly include the others. in the interests of compile-time efficiency, it's worth trying to keep number of such automatically included files to a minimum; the ones here are necessary for almost all other modules, however.

658 vector

⟨*Global Include Files*⟩+≡

```
#include <math.h>
#include <stdlib.h>
#define _GNU_SOURCE 1
#include <stdio.h>
#include <string.h>
```

Also, we include files from the standard library to get the vector, and pair template classes. The using directive brings these container classes into our namespace.

⟨*Global Include Files*⟩+≡

```
#include <string>
using std::string;
#include <vector>
using std::vector;
using std::pair;
using std::make_pair;
#include <iostream>
using std::ostream;
```

We will also define a number of types with typedef here. First is Float; rather than using the built-in float and double types for floating point variables, we abstract away this choice with Float. This makes it convenient to globally change from one representation to the other. In general, as long as numerical algorithms with egregious stability are avoided, the precision provided by float is sufficient in a ray tracer.

For convenience, we also define shorthand names for unsigned cardinal types: u_char, u_short, u_int, and u_long.

⟨*Global Type Declarations*⟩+≡

```
typedef float Float;
typedef unsigned char u_char;
typedef unsigned short u_short;
typedef unsigned int u_int;
typedef unsigned long u_long;
```

We will also define a macro that holds `lrt`'s current version number. This is a floating-point value that will be increased as future versions of `lrt` are developed.

⟨*Global Constants*⟩+≡

```
#define LRT_VERSION 1.0
```

Further Reading

Grunwald et al were one of the first groups of researchers to investigate the inter-play between memory allocation algorithms and the cache behavior of applications (Grunwald, Zorn, and Henderson 1993).

Or can reorder the computation, so that the program accesses pharr 97, cache paper on ray tracing...

Lam et al investigated blocking (tiling) for improving cache performance and developed techniques for selecting appropriate block sizes, given the size of the arrays and the cache size (Lam, Rothberg, and Wolf 1991).

In `lrt`, we only worry about cache layout issues for dynamically-allocated stuff. However, Calder et al show a profile-driven system that optimizes memory layout of global variables, constant values, data on the stack, and dynamically-allocated data from the heap in order to reduce cache conflicts among them all (Calder, Chandra, John, and Austin 1998), giving an average 30% reduction in data cache misses for the applications they studied.

Blocking for tree data structures–keep node and a few levels of children contiguous (Chilimbi, Hill, and Larus 1999). Among other applications, they applied their tool to the layout of the acceleration octree in the *radiance* renderer and reported a 42% speedup in runtime.

More on structures: possibly split into "hot" and "cold" parts, allocated separately, to improve hits on hot parts. Also more on reordering fields inside structure to improve locality (Chilimbi, Davidson, and Larus 1999).

Christer cache chapter

Detailed information about the random number generator we are using, including the original paper from ACM Transactions on Modelling and Computer Simulation (Matsumoto and Nishimura 1998) are available at `http://www.math.keio.ac.jp/˜matumoto/e`

Sean Anderson bit twiddling hacks.

Float to int stuff at `http://www.stereopsis.com/FPU.html`.

Gaussian elimination, pivot stuff(Atkinson 1993).

Numerical Recipes, Press (Press, Teukolsky, Vetterling, and Flannery 1992).

Samet's book on octrees (Samet 1990)

de Berg et al computational geoemtry (de Berg, van Kreveld, Overmars, and Schwarzkopf 2000)

The TIFF specification (cite XXX)...

Exercises

1.1 Modify the ObjectArena and MemoryArena so that they just call new for each memory allocation. Render images of a few scenes and measure how much slower lrt runs (be sure to choose scenes that end up running code that uses these custom allocators.) Can you quantify how much of this is due to different cache behavior and how much is due to overhead in the dynamic memory management routines?

1.2 Change the BlockedArray class so that it doesn't do any blocking and just uses a linear addressing scheme for the array. Measure the change in lrt's performance as a result. (Scenes with many image map textures are most likely to show differences, since the MIPMap class is a key user of BlockedArray.

1.3 The KdNode for the KdTree can be brought down to use just four bytes of storage; making this change may further improve its memory performance. Modify the KdNode to just store the split position and split axis in four bytes, using the same technique as was used to overlap the flags and the split position in the KdAccelNode. Then, modify the tree construction routine to build a *left-balanced* kd-tree, where the tree's topology is organized such that for the node at position i in the array of nodes, the left child is at 2*i and the right child is at 2*i+1, and the tree is balanced such that if only the first nNodes elements of the array are used, XXXX. **Explain left-balanced binary trees better.**

672	BlockedArray
155	KdAccelNode
688	KdTree
670	MemoryArena
417	MIPMap
668	ObjectArena

B.Scene Description Interface

This appendix describes the application programming interface (API) that is used to describe the scene to be rendered to `lrt`. Users of the renderer typically don't call the functions in this interface directly, but instead use the text file format described in Appendix C to describe their scenes to the renderer; the statements in these text files have a direct correspondence to the API functions described here.

The need for such an interface to the renderer is clear: there must be a convenient way in which all of the properties of scene to be rendered can be communicated to the renderer. In choosing an API style for a renderer, there are two main decisions to make: should the API be focused on configuring a graphics pipeline or should it be focused on describing the physical scene; and should the API be based on an *immediate mode* or a *retained mode* style?

As for the first question, there have historically been two main approaches to scene description in graphics: the interface may specify *how* to rendering the scene, configuring a rendering pipeline at a low-level, or it may specify *what* the scene's objects, lights, and material properties are, and leave it to the renderer to decide how to transform that description into the best-possible image.

The first approach has been successfully used for interactive graphics, as seen in the OpenGL and Direct3D APIs. There, it's not possible to just dmonte an object as being mirror reflective and have reflections appear automatically; rather, the user must choose an algorithm for rendering reflections, render the scene multiple times (e.g. to generate a cube-map environment map), and then configure the graphics pipeline to use the cube-map when rendering the reflective object. The advantage of this approach is that the full flexibility of the rendering hardware is exposed to the user, making it possible to carefully control the actual computation being done and to use the hardware in ways not originally imagined by its designers.

The second, declarative approach to scene description, has been most successful for applications like high-end offline rendering, e.g. as embodied by the RenderMan interface. There, users are willing to cede control of more the low-level rendering details to the renderer in exchange for the ability to specify the scene's properties at a higher level of abstraction.

For `lrt`, we will use an interface based on the declarative approach. Because `lrt` is fundamentally physically-based, the API is less flexible than many non-physically-based rendering APIs. For example, instead of making it possible to flag some lights as only illuminating some objects in the scene, a capability that can be useful for artistic effect, in `lrt` all lights in the scene must illuminate all objects.

The other decision to make in the API design is whether to use an immediate mode or a retained mode style. In an immediate mode API, the user specifices the scene via a stream of commands that the renderer processes as they arrive. In general, it's not possible for the user to make changes to any of the scene description data already specified (e.g. "change the material of that sphere I described previously from plastic to glass"); once it has been given to the renderer, the information is no longer accessible to the user.

Retained mode APIs give the user some degree of access to the scene data structure that the renderer has built up from the scene description given to it so far. The user can modify the scene description in a variety of ways before finally instructing the renderer to render the scene.

Immediate mode has been very successful for hardware graphics APIs since it allows the graphics hardware to immediately start to draw the objects in the scene as they are given by the user. By not needing to build up data structures to represent the scene (and thus being able to discard geometry immediately after drawing it) and by being able to immediately cull objects that are outside of the viewing frustum without worrying that the user will change the camera position before rendering, these APIs have been key to high-performance interactive graphics.

For ray tracing renderers like `lrt`, where the entire scene must be described and stored in memory before rendering can begin, some of these advantages of immediate mode aren't applicable. Nonetheless, we will still use immediate mode semantics in our API, since it leads to a clean and straightforward scene description language. This choice makes it more difficult to use `lrt` for applications like quickly re-rendering a scene after making a small change to it (e.g. by moving a light source), and may make rendering animations less straightforward, since the entire scene needs to be re-described for each frame of an animation.

The rendering API consists of roughly thirty-five carefully chosen functions, all of which are declared in the `core/api.h` header file.

```
⟨api.h*⟩≡
  #include "lrt.h"
  ⟨API Function Declarations⟩
```

The implementation of these functions is in `core/api.cpp`.

```
⟨api.cpp*⟩≡
  #include "api.h"
  #include "paramset.h"
  #include "color.h"
  #include "scene.h"
  #include "film.h"
  #include "dynload.h"
  #include "volume.h"
  #include <map>
  using std::map;
  ⟨API Local Classes⟩
  ⟨API Static Data⟩
  ⟨API Macros⟩
  ⟨API Static Methods⟩
  ⟨API Function Definitions⟩
```

B.1 Parameter Sets

A key problem that the API must address is extensibility: if a developer has added new modules to `lrt` to implement new shapes, cameras, etc., minimal or no changes to the API should be necessary. In particular, the API should be as unaware as possible of what particular parameters these objects take and what their meaning is.

700 ParamSet

One key piece of our solution to this is the `ParamSet` class. It handles collections of named parameters and their values in a generic way. Most of the API routines take a `ParamSet` as one of their parameters–for example, the shape creation routine, `lrtShape()` takes a string giving the name of the shape to make and a `ParamSet` with parameters for it. The appropriate shape implementation is loaded from disk, and a creation routine is called, passing along the `ParamSet`. This style makes the API's implementation very straightforward.

The `ParamSet` is a key part of how objects are created at run-time, bundling up the values of the parameters to the constructors in a single object. For example, it might record that there is a single floating-point value named “radius” with a value of 2.5, and an array of four color values named “specular” with various color values. The `ParamSet` provides methods for both setting and retrieving values from this kind of set of parameters.

```
⟨paramset.h*⟩≡
  #include "lrt.h"
  #include "geometry.h"
  #include "color.h"
  ⟨ParamSet Macros⟩
  ⟨ParamSet Types⟩
  ⟨ParamSet Declarations⟩
  ⟨ParamSetItem Methods⟩
```

```
⟨paramset.cpp*⟩≡
  #include "paramset.h"
  ⟨ParamSet Methods⟩
```

⟨*ParamSet Declarations*⟩≡

```
class ParamSet {
public:
    ⟨ParamSet Public Methods⟩
private:
    ⟨ParamSet Data⟩
};
```

The ParamSet holds seven types of parameters: integers, scalar floating point vaules, points, vectors, normals, spectra, and strings. Internally, it stores a vector of named values for each of the different types that it stores–each bound parameter is represented by a ParmSetItem of the appropriate type.

⟨*ParamSet Data*⟩≡

```
vector<ParamSetItem<int> *> ints;
vector<ParamSetItem<Float> *> floats;
vector<ParamSetItem<Point> *> points;
vector<ParamSetItem<Vector> *> vectors;
vector<ParamSetItem<Normal> *> normals;
vector<ParamSetItem<Spectrum> *> spectra;
vector<ParamSetItem<string> *> strings;
```

Normal	34
ParamSetItem	701
Point	33
Spectrum	181
vector	658
Vector	27

The constructor for the ParamSet does no work; it starts out with unfilled vectors of parameters.

⟨*ParamSet Public Methods*⟩≡

```
ParamSet() { }
```

However, because it holds pointers to ParamSetItems in the vectors, we need to do some work when one ParamSet is assigned to another one; we can't just let the default assignment operator copy the contents of all of the vectors, since we would then free the ParamSetItem memory twice in the two ParamSet destructors. The implementation of this is straightforward, allocating duplicates of all of the ParamSetItems, so we will not include it here.

⟨*ParamSet Public Methods*⟩+≡

```
ParamSet &operator=(const ParamSet &p2);
```

B.1.1 ParamSetItem

The ParamSetItem structure stores all of the relevant information about a single parameter–name, value, etc. Though many parameters just hold a single value of their type, we may also hold multiple values, either because the parameter is an array type and/or because multiple values were given for the parameter. For example (using the syntax that is used to describe parameters in lrt's input files), the foo parameter:

```
"float foo[2]" [ 0 1 2 3 4 5 ]
```

has a basic type of float[2]. Here, three items of this type have been given, [0 1], [2 3], and [4 5].

XXX make more clear array size versus num items relationship, distinction... XXX

⟨*ParamSet Declarations*⟩+≡

```
template <class T> struct ParamSetItem {
    ⟨ParamSetItem Public Methods⟩
    ⟨ParamSetItem Data⟩
};
```

The `ParamSetItem` directly initializes its members from the values passed in and makes a copy of the values in the array of parameter values in locally-allocated memory.

⟨*ParamSetItem Methods*⟩≡

```
template <class T>
ParamSetItem<T>::ParamSetItem(const string &n, const T *v, int t,
        int c, int ni) {
    name = n;
    type = t;
    arraySize = c;
    nItems = ni;
    int nAlloc = arraySize * nItems;
    data = new T[nAlloc];
    for (int i = 0; i < nAlloc; ++i)
        data[i] = v[i];
    lookedUp = false;
}
```

We also store a boolean value `lookedUp` that is initially set to `false` but it is changed to be `true` after the value has been retrieved from the `ParamSet`. Later, this will allow us to print warning messages if any parameters were added to the parameter set but never used (probably indicating a misspelling or other user error.)

⟨*ParamSetItem Data*⟩≡

```
string name;
int type, arraySize, nItems;
T *data;
bool lookedUp;
```

We represent the base type of a parameter with an integer. This type includes both the underlying datatype–float, point, etc.–as well as the parameter's storage class. These two are stored together in the `type` member. We'll define some constants to represent each of the possible types.

⟨*ParamSet Types*⟩≡

```
#define PARAM_TYPE_INT     (1<<0)
#define PARAM_TYPE_FLOAT   (1<<1)
#define PARAM_TYPE_POINT   (1<<2)
#define PARAM_TYPE_VECTOR  (1<<3)
#define PARAM_TYPE_NORMAL  (1<<4)
#define PARAM_TYPE_STRING  (1<<5)
#define PARAM_TYPE_COLOR   (1<<6)
```

The storage class accounts for the idea that we may want to have multiple values of a parameter defined in a way that it can be interpolated over a surface, taking on a different value at each point being shaded. For example, a triangle mesh might be defined with a single diffuse color for all of the triangles, but with specular colors defined at each vertex and interpolated inside each face.

There are three different storage classes to handle these sorts of situations:

- Uniform parameters take on a single value over the entire object
- Varying parameters are specified with four values which are bilinearly interpolated according to the (u,v) parameter value for a point on the surface.
- Vertex parameters are only available for mesh shapes, and represent values specefied at each vertex of the mesh.

⟨*ParamSet Types*⟩+≡
```
#define PARAM_TYPE_UNIFORM (1<<7)
#define PARAM_TYPE_VARYING (1<<8)
#define PARAM_TYPE_VERTEX  (1<<9)
```

Naturally, the `ParamSetItem` needs to free allocated memory in its destructor.

⟨*ParamSetItem Public Methods*⟩+≡
```
~ParamSetItem() {
    delete[] data;
}
```

PARAM_TYPE_FLOAT	701
ParamSet	700
ParamSetItem	701

B.1.2 Adding to the parameter set

To add an entry to the parameter set, the user just calls an appropriate method of the `ParamSet`, passing the name of the parameter, a pointer to its value, and storage class information.

⟨*ParamSet Methods*⟩+≡
```
void ParamSet::AddFloat(const string &name, const Float *data,
        int type, int narray, int nitems) {
    type |= PARAM_TYPE_FLOAT;
    EraseFloat(name);
    floats.push_back(new ParamSetItem<Float>(name, data, type,
        narray, nitems));
}
```

We won’t include the rest of the methods to add other data types to the `ParamSet`, but will include their prototypes here for reference.

⟨*ParamSet Public Methods*⟩+≡

```
void AddInt(const string &, const int *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1,
    int nItems = 1);
void AddPoint(const string &, const Point *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1,
    int nItems = 1);
void AddVector(const string &, const Vector *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1,
    int nItems = 1);
void AddNormal(const string &, const Normal *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1,
    int nItems = 1);
void AddSpectrum(const string &, const Spectrum *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1,
    int nItems = 1);
void AddString(const string &, const string *,
    int type = PARAM_TYPE_UNIFORM, int nArray = 1,
    int nItems = 1);
```

34	Normal
702	PARAM_TYPE_UNIFORM
33	Point
181	Spectrum
27	Vector

B.1.3 Looking up values in the parameter set

Looking up parameter values is similarly straightforward; we just loop through the values we have of the requested type and return the value, if any. There are two versions of the lookup method, a simple one for uniform parameters with array size of one (or non-array types) with a single data value that returns the data value directly, and a more complex one that returns a pointer to the values of more complex types.

The methods that look up a single item (e.g. `FindOneFloat()`) take the name of the parameter and a default value. If the parameter is not found, the default value is silently returned. This makes it easy to write initialization code like:

```
Float radius = params.FindOneFloat("radius", 1.f);
```

Here, it’s not an error if there isn’t a “radius” parameter–we just want to use the default value. For the case where it is an error for a parameter to be unavailable, the second set of lookup methods can be used instead.

⟨*ParamSet Methods*⟩+≡

```
Float ParamSet::FindOneFloat(const string &name, Float d) const {
    for (u_int i = 0; i < floats.size(); ++i)
        if (floats[i]->name == name &&
            (floats[i]->type & PARAM_TYPE_UNIFORM) &&
            floats[i]->nItems == 1 &&
            floats[i]->arraySize == 1) {
            floats[i]->lookedUp = true;
            return *(floats[i]->data);
        }
    return d;
}
```

As above, here are the declarations for the rest of the analogous methods.

⟨*ParamSet Public Methods*⟩+≡

```
int FindOneInt(const string &, int d) const;
Point FindOnePoint(const string &, const Point &d) const;
Vector FindOneVector(const string &, const Vector &d) const;
Normal FindOneNormal(const string &, const Normal &d) const;
Spectrum FindOneSpectrum(const string &,
    const Spectrum &d) const;
string FindOneString(const string &, const string &d) const;
```

The second lookup method returns a pointer to the data if it's present. It returns the storage class information in the given `type` pointer, the number of array elements in the `nArray` value, and then umber of data items in `nItems`. (Thus, the total number of values in the memory pointed to by the returned pointer `nArray * nItems`.) It's up to the caller to interpret these appropriately when accessing the returned pointer.

⟨*ParamSet Methods*⟩+≡

```
const Float *ParamSet::FindFloat(const string &name, int *type,
        int *nArray, int *nItems) const {
    for (u_int i = 0; i < floats.size(); ++i)
        if (floats[i]->name == name) {
            *nArray = floats[i]->arraySize;
            *type = floats[i]->type;
            *nItems = floats[i]->nItems;
            floats[i]->lookedUp = true;
            return floats[i]->data;
        }
    return NULL;
}
```

Normal	34
PARAM_TYPE_UNIFORM	702
ParamSet	700
Point	33
Spectrum	181
Vector	27

These are the rest of the analogous lookup functions.

⟨*ParamSet Public Methods*⟩+≡

```
const int *FindInt(const string &, int *type,
    int *nArray, int *nItems) const;
const Point *FindPoint(const string &, int *type,
    int *nArray, int *nItems) const;
const Vector *FindVector(const string &, int *type,
    int *nArray, int *nItems) const;
const Normal *FindNormal(const string &, int *type,
    int *nArray, int *nItems) const;
const Spectrum *FindSpectrum(const string &, int *type,
    int *nArray, int *nItems) const;
const string *FindString(const string &, int *type,
    int *nArray, int *nItems) const;
```

Because the user may misspell parameter names in the scene description file, we'll also provide a function that goes through the parameter set and reports if any of the parameters present were never looked up. If this happens, odds are good the user has given an incorrect parameter.

⟨*ParamSet Methods*⟩+≡

```
void ParamSet::ReportUnused() const {
#define CHECK_UNUSED(v) \
    for (i = 0; i < (v).size(); ++i) \
        if (v[i]->name[0] != ’_’ && !(v)[i]->lookedUp) \
            Warning("Parameter \"%s\" not used", \
                (v)[i]->name.c_str())
    u_int i;
    CHECK_UNUSED(ints);
    CHECK_UNUSED(floats);
    CHECK_UNUSED(points);
    CHECK_UNUSED(vectors);
    CHECK_UNUSED(normals);
    CHECK_UNUSED(spectra);
    CHECK_UNUSED(strings);
}
```

We break the functionality of the destructor out into a separate `Clear()` method; this allows us to clear out an existing `ParamSet` without fully destroying it. (This is useful in the copy constructor, for example.)

⟨*ParamSet Public Methods*⟩+≡

700 ParamSet

```
ParamSet::~ParamSet() {
    Clear();
}
```

The `Clear()` method is straightforward, just looping over all of the vectors and deleting their individual `ParamSetItem`s before resetting the `vector`s to have zero length.

⟨*ParamSet Methods*⟩+≡

```
void ParamSet::Clear() {
    u_int i;
#define DEL_PARAMS(name) \
    for (i = 0; i < (name).size(); ++i) \
        delete (name)[i]; \
    (name).erase((name).begin(), (name).end())

    DEL_PARAMS(ints);
    DEL_PARAMS(floats);
    DEL_PARAMS(points);
    DEL_PARAMS(vectors);
    DEL_PARAMS(normals);
    DEL_PARAMS(spectra);
    DEL_PARAMS(strings);
#undef DEL_PARAMS
}
```

Finally, we’ll provide a method that returns a `string` representation of the `ParamSet`, using the same syntax as would be used to initialize the `ParamSet` in a `lrt` input file. The implementation of this is tedius and straightforward, so has been elided here.

```
⟨ParamSet Public Methods⟩+≡
  string ToString() const;
```

B.2 Global Options

Now that we have a general mechanism for passing collections of parameters and their values into the renderer, we can move forward to the actual rendering API. Before any other API function calls are made, the rendering system must be initialized by a call to `lrtInit()`. After this has been done, general rendering options like the camera and sampler properties, the tone mapping algorithm to be used, etc., can be set, but the user can not yet start to describe the lights, shapes, and materials in the scene.

After the basic rendering options have been set, the `lrtWorldBegin()` function is called and the options are fixed; it's no longer allowed to change them. The user then describes the geometric primitives and lights that are in the scene along with their various attributes. The reason we disllow further changes to graphics options after `lrtWorldBegin()` is that doing so can simplify the overall implementation of the renderer. For example, consider a spline surface shape that tessellated itself into triangles at creation time at a resolution based on the area of the screen that it covered: if the camera's position and image resolution have been locked down by the time the shape is created, than we don't have to defer the tessellation work until later.

`StatsCleanup() 662`

When all of the primitives have been supplied, `lrtWorldEnd()` is called. At this point the renderer knows that the scene description is complete and that rendering can begin. The image is rendered and written to disk before `lrtWorldEnd()` returns. The user may then specify new graphics options for another frame of an animation and then another `lrtWorldBegin()`/`lrtWorldEnd()` block to describe the geometry for the next frame, repeating as many times as desired. When there is no more rendering to be done, `lrtCleanup()` should be called; this handles final cleanup of the system.

The fragments that define these two functions will be filled in throughout the rest of this appendix as we come to need to do various pieces of initialization and cleanup.

```
⟨API Function Definitions⟩≡
  void lrtInit() {
      ⟨System-wide initialization⟩
      ⟨API Initialization⟩
  }
```

```
⟨API Function Definitions⟩+≡
  void lrtCleanup() {
      StatsCleanup();
      ⟨API Cleanup⟩
  }
```

B.2.1 State tracking

Because all of the API calls besides `lrtInit()` are illegal before `lrtInit()` is called and because most of the others are only legal before or after `lrtWorldBegin()`, we will provide some facilities for tracking what state the API is in. We use a module static variable `currentApiState`. It starts out with value `STATE_UNINITIALIZED`, indicating that the API system hasn't yet been initialized. Its value is is updated appropriately by `lrtInit()`, `lrtWorldBegin()`, and `lrtCleanup()`.

⟨*API Static Data*⟩≡

```
#define STATE_UNINITIALIZED  0
#define STATE_OPTIONS_BLOCK  1
#define STATE_WORLD_BLOCK    2
static int currentApiState = STATE_UNINITIALIZED;
```

Now we can start to define the fragment in the implementation of `lrtInit()`. It makes sure that `lrtInit()` hasn't already been called and sets the api state to reflect that any of the graphics options API calls are now legal.

⟨*API Initialization*⟩≡

```
if (currentApiState != STATE_UNINITIALIZED)
    Error("lrtInit() has already been called.");
currentApiState = STATE_OPTIONS_BLOCK;
```

Similarly, `lrtCleanup()` makes sure that `lrtInit()` has been called and that we're not in the middle of a `lrtWorldBegin()`/`lrtWorldEnd()` block before resetting the state to the uninitialized state.

⟨*API Cleanup*⟩≡

```
if (currentApiState == STATE_UNINITIALIZED)
    Error("lrtCleanup() called without lrtInit().");
else if (currentApiState == STATE_WORLD_BLOCK)
    Error("lrtCleanup() called while inside world block.");
currentApiState = STATE_UNINITIALIZED;
```

All API procedures that are only valid in particular states call one of the state verification macros, `VERIFY_UNINITIALIZED()`, `VERIFY_OPTIONS()`, or `VERIFY_WORLD()`, to ensure that `currentApiState` holds the appropriate value. If the states don't match, we print an error message and return immediately from the function.

⟨*API Macros*⟩≡

```
#define VERIFY_INITIALIZED(func) \
if (currentApiState == STATE_UNINITIALIZED) { \
    Error("lrtInit() must be before calling \"%s()\". " \
        "Ignoring.", func); \
    return; \
} else /* swallow trailing semicolon */
```

⟨*API Macros*⟩+≡

```
#define VERIFY_OPTIONS(func) \
VERIFY_INITIALIZED(func); \
if (currentApiState == STATE_WORLD_BLOCK) { \
    Error("Options cannot be set inside world block; " \
        "\"%s\" not allowed.  Ignoring.", func); \
    return; \
} else /* swallow trailing semicolon */
```

⟨*API Macros*⟩+≡

```
#define VERIFY_WORLD(func) \
VERIFY_INITIALIZED(func); \
if (currentApiState == STATE_OPTIONS_BLOCK) { \
    Error("Scene description must be inside world block; " \
        "\"%s\" not allowed. Ignoring.", func); \
    return; \
} else /* swallow trailing semicolon */
```

⟨*API Macros*⟩+≡

```
#define LRT_UNIMP(func) { \
static bool first = true; \
if (first) { \
    first = false; \
    Warning("Call to unimplemented API function \"%s\"!", func); \
} }
```

currentApiState	707
Scene	8
STATE_OPTIONS_BLOCK	707
STATE_WORLD_BLOCK	707
Transform	43
VERIFY_INITIALIZED	707

B.2.2 Transformations

While the scene is being described, `lrt` maintains a *current transformation matrix* (CTM). When shapes are created, for example, the value of the CTM is used as the object to world transformation for that shape. Similarly, the CTM sets the camera to world transformation when the camera for the scene is created.

The scene description API provides a number of routines that update the CTM. Most of them post-multiply the CTM with a given new transformation matrix. These routines are slightly complicated by the need to be able to specify multiple transformations for moving objects that are at different positions at different points in time. We store up to two current transformations, updating only one of them when a transformation call is made, depending on which transform of a moving object is being specified. If the object is not moving, we just update the first of the two of them.

⟨*API Local Classes*⟩≡

```
struct TransformSet {
    ⟨TransformSet Methods⟩
    Transform shutterOpen, shutterClose;
};
```

⟨*API Static Data*⟩+≡

```
static TransformSet curTransform;
```

⟨*TransformSet Methods*⟩≡

```
void operator*=(const Transform &x) {
    shutterOpen = shutterOpen * x;
}
```

⟨*TransformSet Methods*⟩+≡

```
void SetEnd() { shutterClose = shutterOpen; }
```

⟨*TransformSet Methods*⟩+≡

```
const Transform &GetOpen() const {
    return shutterOpen;
}
```

⟨*TransformSet Methods*⟩+≡

```
const Transform &GetClose() const {
    return shutterClose;
}
```

⟨*TransformSet Methods*⟩+≡

```
void Reset() {
    shutterOpen = Transform();
    shutterClose = Transform();
}
```

708	curTransform
43	Transform
708	TransformSet
707	VERIFY_INITIALIZED

The transformations of moving objects are given within motion blocks, like:

```
AttributeBegin
Translate 1 0 0
Translate 0 1 0
SetEndTransform
AttributeEnd
```

This specifies that at time 10, the first translation should be appended to the current transformation and at time 11, the second translation should be. The `lrtMotionBegin` function takes an array of time values that specifies how many transformations will be given.

If there is no current motion block, then we just update the current transform. Otherwise, we update the appropriate one depending on how many transforms have been given in this block so far.

⟨*API Function Definitions*⟩+≡

```
void lrtSetEndTransform() {
    VERIFY_INITIALIZED("SetEndTransform");
    curTransform.SetEnd();
}
```

The actual transformation functions are quite straightforward; most just use the multiplication operator from the `TransformSet` to update the current transformation.

⟨*API Function Definitions*⟩+≡

```
void lrtIdentity() {
    VERIFY_INITIALIZED("Identity");
    curTransform.Reset();
}
```

```
⟨API Function Declarations⟩+≡
  extern void lrtTransform(Float transform[16]);
  extern void lrtConcatTransform(Float transform[16]);
  extern void lrtRotate(Float angle, Float dx, Float dy, Float dz);
  extern void lrtScale(Float sx, Float sy, Float sz);
  extern void lrtLookAt(Float ex, Float ey, Float ez, Float lx, Float ly,
      Float lz, Float ux, Float uy, Float uz);
```

```
⟨API Function Definitions⟩+≡
  void lrtTranslate(Float dx, Float dy, Float dz) {
      VERIFY_INITIALIZED("Translate");
      curTransform *= Translate(Vector(dx, dy, dz));
  }
```

```
⟨API Function Definitions⟩+≡
  void lrtCoordinateSystem(const string &name) {
      VERIFY_INITIALIZED("CoordinateSystem");
      namedCoordinateSystems[name] = curTransform;
  }
```

curTransform	708
TransformSet	708
Vector	27
VERIFY_INITIALIZED	707

```
⟨API Static Data⟩+≡
  static map<string, TransformSet> namedCoordinateSystems;
```

```
⟨API Function Definitions⟩+≡
  void lrtCoordSysTransform(const string &name) {
      VERIFY_INITIALIZED("CoordSysTransform");
      if (namedCoordinateSystems.find(name) != namedCoordinateSystems.end())
          curTransform = namedCoordinateSystems[name];
  }
```

B.2.3 Options

The user can set a variety of options before the scene to be rendered is specified. These include things such as the camera position and type, image sampling and reconstruction options, the type of image file to write out, etc. We store all of this information in a GraphicsOptions structure. It has a number of public data members that subsequent API calls will set and a number of methods to help create objects used by the rest of the system for rendering.

```
⟨API Local Classes⟩+≡
  struct GraphicsOptions {
      ⟨GraphicsOptions Public Methods⟩
      ⟨Graphics Options⟩
  };
```

```
⟨API Local Classes⟩+≡
  GraphicsOptions::GraphicsOptions() {
      ⟨GraphicsOptions Constructor Implementation⟩
  }
```

We have a single instance of the `GraphicsOptions` that is available to the rest of the functions in this file.

⟨*API Static Data*⟩+≡

```
static GraphicsOptions *curGraphicsOptions = NULL;
```

When `lrtInit()` is called, we need to ensure that the `GraphicsOptions` is re-initialized to hold default values.

⟨*API Initialization*⟩+≡

```
curGraphicsOptions = new GraphicsOptions;
```

And similarly, we need to free it at `lrtCleanup()`-time.

⟨*API Cleanup*⟩+≡

```
delete curGraphicsOptions;
curGraphicsOptions = NULL;
```

The various API functions for setting options are quite similar in both their interface and their implementation. For example, `lrtPixelFilter()` specifies the `Filter` to be used for filtering image samples. It takes two parameters, a string giving the name of the filter to use, and a `ParamSet`, giving the parameters to the filter. The dynamic loading code in Appendix D will later use the string name to load the appropriate filter implementation from disk, passing it the `ParamSet` we were given in `lrtPixelFitler()`. For now, however, all that we need to do is to verify that the API is in an appropriate state for `lrtPixelFilter()` to be called and store away the name of the filter and the parameters in the graphics options structure.

```
710 GraphicsOptions
700 ParamSet
708 VERIFY_OPTIONS
```

⟨*API Function Definitions*⟩+≡

```
void lrtPixelFilter(const string &name, const ParamSet &params) {
    VERIFY_OPTIONS("PixelFilter");
    curGraphicsOptions->FilterName = name;
    curGraphicsOptions->FilterParams = params;
}
```

⟨*Graphics Options*⟩≡

```
string FilterName;
ParamSet FilterParams;
```

The default filter function, in case the user doesn't specify one, is set here.

⟨*GraphicsOptions Constructor Implementation*⟩≡

```
FilterName = "mitchell";
```

Most of the rest of the graphics options do exactly the same thing in their implementations. Therefore, we will only include the declarations of these functions here. What each of thse functions controls should be readily apparent; Appendix C gives more information about each of them.

⟨*API Function Declarations*⟩+≡

```
extern void lrtFilm(const string &type, const ParamSet &params);
extern void lrtSampler(const string &name, const ParamSet &params);
extern void lrtAccelerator(const string &name,
    const ParamSet &params);
extern void lrtToneMap(const string &name, const ParamSet &params);
extern void lrtSurfaceIntegrator(const string &name,
    const ParamSet &params);
extern void lrtVolumeIntegrator(const string &name,
    const ParamSet &params);
```

`lrtCamera()` is slightly different than the other options, since the camera-to-world transformation needs to be set. The current transformation when `lrtCamera()` is called is used to initialize this value.

⟨*API Function Definitions*⟩+≡

```
void lrtCamera(const string &name, const ParamSet &params) {
    VERIFY_OPTIONS("Camera");
    curGraphicsOptions->CameraName = name;
    curGraphicsOptions->CameraParams = params;
    curGraphicsOptions->WorldToCamera = curTransform;
    TransformSet camInv = curTransform;
    camInv.shutterOpen = camInv.shutterOpen.GetInverse();
    camInv.shutterClose = camInv.shutterClose.GetInverse();
    namedCoordinateSystems["camera"] = camInv;
}
```

Camera	202
curGraphicsOptions	711
curTransform	708
namedCoordinateSystems	710
ParamSet	700
TransformSet	708
VERIFY_OPTIONS	708

⟨*Graphics Options*⟩+≡

```
string CameraName;
mutable ParamSet CameraParams;
TransformSet WorldToCamera;
```

⟨*GraphicsOptions Constructor Implementation*⟩+≡

```
CameraName = "orthographic";
```

The only other slightly unusual graphics options API call is `lrtSearchPath()`, which is used to set the directory path that `lrt` uses to search for plug-in implementations. It takes a single string, which holds a colon-separated list of directories.

⟨*API Function Declarations*⟩+≡

```
extern void lrtSearchPath(const string &path);
```

B.3 Scene Definition

After the user has set up the overall graphics options, the `lrtWorldBegin()` call indicates the start of the description of the shapes, materials, and lights in the scene. It updates the record of the current state and resets the current transformation matrix (previously used to specify the world-to-camera transformation) to the identity. For the rest of the world block, it will hold the current object-to-world transformation.

⟨*API Function Definitions*⟩+≡

```
void lrtWorldBegin() {
    VERIFY_OPTIONS("WorldBegin");
    currentApiState = STATE_WORLD_BLOCK;
    curTransform.Reset();
    namedCoordinateSystems["world"] = curTransform;
}
```

B.3.1 Hierarchical Graphics State

As the stream of commands comes in that specifies the scene geometry, a variety of attributes can be updated as well. These include information about the current material, the current transformation matrix, etc. When a geometric primitive or light source is then added to the scene, various parts of the current set of attributes are used to initialize their specific parameters.

The current set of active attributes is managed with an *attribute stack*. This allows the user to *push* the current set of attributes, make changes to their values and then later *pop* back to the previously pushed attribute values. For example, a scene description file might have lines such as:

```
Material "matte"
AttributeBegin # push current attributes
Material "plastic"
Translate 5 0 0
Shape "sphere" "float radius" [1] # plastic and translated
AttributeEnd # pop attributes
Shape "sphere" "float radius" [1] # matte and not translated
```

714	curGraphicsState
707	currentApiState
708	curTransform
710	GraphicsOptions
710	namedCoordinateSystems
707	STATE_WORLD_BLOCK
708	VERIFY_OPTIONS

Changes to attributes made inside an `lrtAttributeBegin()`/`lrtAttributeEnd()` block are forgotten at the end of the block. Being able to save and restore attributes in this manner is a classic idiom for scene description in computer graphics. XXX.

We store the rest of set of current attributes in the `GraphicsState` structure. As with `GraphicsOptions`, we'll be adding members to it throughout this section.

⟨*API Local Classes*⟩+≡

```
struct GraphicsState {
    ⟨Graphics State Methods⟩
    ⟨Graphics State⟩
};
```

⟨*API Local Classes*⟩+≡

```
GraphicsState::GraphicsState() {
    ⟨GraphicsState Constructor Implementation⟩
}
```

When `lrtInit()` is called, we initialize the current graphics state to hold default values.

⟨*API Initialization*⟩+≡

```
curGraphicsState = GraphicsState();
```

We also keep a list of `GraphicsStates`; when `lrtAttributeBegin()` is called, we copy the current `GraphicsState` and push it on to the `vector`. Attribute end pops the state to restore to back off of the `vector`.

⟨*API Static Data*⟩+≡

```
static GraphicsState curGraphicsState;
static vector<GraphicsState> pushedGraphicsStates;
static vector<TransformSet> pushedTransforms;
```

⟨*API Function Definitions*⟩+≡

```
void lrtAttributeBegin() {
    VERIFY_WORLD("AttributeBegin");
    pushedGraphicsStates.push_back(curGraphicsState);
    pushedTransforms.push_back(curTransform);
}
```

⟨*API Function Definitions*⟩+≡

```
void lrtAttributeEnd() {
    VERIFY_WORLD("AttributeEnd");
    if (!pushedGraphicsStates.size()) {
        Error("Unmatched lrtAttributeEnd() encountered.  "
            "Ignoring it.");
        return;
    }
    curGraphicsState = pushedGraphicsStates.back();
    curTransform = pushedTransforms.back();
    pushedGraphicsStates.pop_back();
    pushedTransforms.pop_back();
}
```

currentApiState	707
curTransform	708
GraphicsState	713
namedCoordinateSystems	710
STATE_OPTIONS_BLOCK	707
TransformSet	708
vector	658
VERIFY_WORLD	708

B.3.2 WorldEnd and Rendering

When `lrtWorldEnd()` is called, the scene has been fully specified and rendering can begin. We make sure that there aren't excess graphics state structures on the state stack, issuing a warning if so, before creating the `Scene` object and calling its `Render()` method.

⟨*API Function Definitions*⟩+≡

```
void lrtWorldEnd() {
    VERIFY_WORLD("WorldEnd");
    ⟨Ensure there are no pushed graphics states⟩
    ⟨Create scene and render⟩
    currentApiState = STATE_OPTIONS_BLOCK;
    StatsPrint(stdout);
    curTransform.Reset();
    namedCoordinateSystems.erase(namedCoordinateSystems.begin(),
        namedCoordinateSystems.end());
}
```

⟨*Ensure there are no pushed graphics states*⟩≡
```
while (pushedGraphicsStates.size()) {
    Error("Missing end to lrtAttributeBegin");
    pushedGraphicsStates.pop_back();
    pushedTransforms.pop_back();
}
```

The `GraphicsOptions::MakeScene()` method handles all of the detail work involved in creating all of the objects corresponding to the settings provideed by the user. It is described in Section B.4. After the `Scene` has been created, the `Render()` method takes over and executes the main rendering loop.

⟨*Create scene and render*⟩≡
```
Scene *scene = curGraphicsOptions->MakeScene();
if (scene) scene->Render();
delete scene;
```

B.3.3 Surface and Material Description

The current material is specified by `lrtMaterial()`. We gather up all of the additional parameters and their values passed along with the name of the material and store them away in the graphics state. When we later go create the material, we'll use these to set up its textures.

⟨*API Function Declarations*⟩+≡
```
extern void lrtMaterial(const string &name, const ParamSet &params);
extern void lrtBumpMap(const string &name, const ParamSet &params);
```

711	curGraphicsOptions
714	curGraphicsState
720	GraphicsOptions::MakeScene()
700	ParamSet
714	pushedGraphicsStates
714	pushedTransforms
8	Scene
11	Scene::Render()
708	VERIFY_WORLD

B.3.4 Geometric Primitives

⟨*API Function Definitions*⟩+≡
```
void lrtReverseOrientation() {
    VERIFY_WORLD("ReverseOrientation");
    curGraphicsState.reverseOrientation =
        !curGraphicsState.reverseOrientation;
}
```

⟨*Graphics State*⟩+≡
```
bool reverseOrientation;
```

⟨*GraphicsState Constructor Implementation*⟩+≡
```
reverseOrientation = false;
```

⟨*API Function Definitions*⟩+≡
```
void lrtShape(const string &name, const ParamSet &params) {
    VERIFY_WORLD("Shape");
    Reference<Shape> s = CreateShape(name,
        curGraphicsOptions->SearchPath,
        curTransform.GetOpen(),
        curGraphicsState.reverseOrientation, params);
    curGraphicsState.AddShape(s, params);
}
```

After the shape creation function has created a new `Shape`, it passes it along to `AddShape()` for further processing.

```
⟨API Static Methods⟩+≡
  void GraphicsState::AddShape(const Reference<Shape> &shape,
          const ParamSet &geomParams) {
      if (!shape) return;
      AreaLight *area = NULL;
      ⟨Initialize area light for shape⟩
      ⟨Initialize material for shape⟩
      Reference<Primitive> prim = new GeometricPrimitive(shape,
          mtl, area);
      if (curGraphicsOptions->currentInstance) {
          if (area)
              Warning("Area lights not supported with object instancing");
          curGraphicsOptions->currentInstance->push_back(prim);
      }
      else {
          curGraphicsOptions->primitives.push_back(prim);
          if (area != NULL) {
              ⟨Create area lights given number of light samples⟩
          }
      }
      geomParams.ReportUnused();
  }
```

`AreaLight`	491
`areaLightParams`	719
`curGraphicsOptions`	711
`curGraphicsState`	714
`curTransform`	708
`GeometricPrimitive`	132
`GraphicsState`	713
`Light`	478
`Material`	375
`ParamSet`	700
`Primitive`	130
`Reference`	664
`Scene`	8
`SearchPath`	743
`Shape`	63
`vector`	658
`VERIFY_WORLD`	708
`VolumeRegion`	465

```
⟨Create area lights given number of light samples⟩≡
  int nSamples = areaLightParams.FindOneInt("nsamples", 1);
  if (nSamples == 1)
      curGraphicsOptions->lights.push_back(area);
  else
      for (int i = 0; i < nSamples; ++i)
          curGraphicsOptions->lights.push_back(
              new MultiAreaLight(area, nSamples));
```

All of the `Primitive`s and `Light`s that are defined are stored in a big vector as we process the input file; they are later passed off to the `Scene` when it is created.

```
⟨Graphics Options⟩+≡
  mutable vector<Reference<Primitive> > primitives;
  mutable vector<Light *> lights;
  mutable vector<VolumeRegion *> volumeRegions;
```

We need to create the `Material` that is bound to the shape. We first determine which one to create based on the string stored in `GraphicsState::material`, which was set by `lrtMaterial()`.

⟨*Initialize material for shape*⟩≡

```
Texture<Float> *displace = CreateBump(displacement, curGraphicsOptions->SearchPath,
    curTransform.GetOpen(), geomParams, displaceParams);
Reference<Material> mtl = CreateMaterial(material, curGraphicsOptions->SearchPath,
    curTransform.GetOpen(), geomParams, materialParams, displace);
if (!mtl)
    mtl = CreateMaterial("plastic", curGraphicsOptions->SearchPath,
        curTransform.GetOpen(), geomParams, materialParams,
        displace);
if (!mtl)
    Severe("Unable to create \"plastic\" material?!");
```

Each of the various materials takes a number of parameters to set its properties. The binding of these parameters is a bit tricky; consider the “matte” material, which takes a color texture named “Kd”. Matte defines a default value for Kd that can be overridden when the `lrtMaterial()` call is made.

```
Material "matte" "color Kd" [ .5 1 .5 ]
```

However, this value can then be overridden again when the primitive is created:

```
Material "matte" "color Kd" [ 1 0 0 ]
Shape "sphere" "float radius" [1] # red
Shape "sphere" "float radius" [1] "color Kd" [ 0 1 0 ] #
green
```

Therefore, we create a `ParamSet` from the parameters given when the material is defined in an `lrtMaterial` call. When creating the `Material`, however, we first look for parameter values in `geomParams`, which was set from the parameters to the primitive-creation API call. If this doesn't have a value, we fall back to the value in `GraphicsState::materialParams`, and from there to a default value.

711	curGraphicsOptions
708	curTransform
375	Material
700	ParamSet
130	Primitive
664	Reference
743	SearchPath
394	Texture
658	vector
708	VERIFY_WORLD

B.3.5 Object Instancing

Note that these implicitly save and restore attributes

XXX not currently... XXX

⟨*API Function Definitions*⟩+≡

```
void lrtObjectBegin(const string &name) {
    VERIFY_WORLD("ObjectBegin");
    if (curGraphicsOptions->currentInstance)
        Error("ObjectBegin called inside of instance definition");
    curGraphicsOptions->instances[name] = vector<Reference<Primitive> >();
    curGraphicsOptions->currentInstance = &curGraphicsOptions->instances[name];
}
```

⟨*Graphics Options*⟩+≡

```
map<string, vector<Reference<Primitive> > > instances;
vector<Reference<Primitive> > *currentInstance;
```

⟨*GraphicsOptions Constructor Implementation*⟩+≡

```
currentInstance = NULL;
```

⟨*API Function Definitions*⟩+≡

```
void lrtObjectEnd() {
    VERIFY_WORLD("ObjectEnd");
    if (!curGraphicsOptions->currentInstance)
        Error("ObjectEnd called outside of instance definition");
    curGraphicsOptions->currentInstance = NULL;
}
```

⟨*API Function Definitions*⟩+≡

```
void lrtObjectInstance(const string &name) {
    VERIFY_WORLD("ObjectInstance");
    if (curGraphicsOptions->currentInstance) {
        Error("ObjectInstance can’t be called inside instance definition");
        return;
    }
    if (curGraphicsOptions->instances.find(name) == curGraphicsOptions->instanc
        Error("Unable to find instance named \"%s\"", name.c_str());
        return;
    }
    vector<Reference<Primitive> > &in = curGraphicsOptions->instances[name];
    if (in.size() == 0) return;
    if (in.size() > 1 || !in[0]->CanIntersect()) {
        Reference<Primitive> accel = CreateAccelerator(curGraphicsOptions->Acce
            curGraphicsOptions->SearchPath, in, curGraphicsOptions->Accelerator
        if (!accel)
            accel = CreateAccelerator("kdtree",
                curGraphicsOptions->SearchPath, in, ParamSet());
        if (!accel)
            Severe("Unable to find \"kdtree\" accelerator");
        in.erase(in.begin(), in.end());
        in.push_back(accel);
    }
    Reference<Primitive> prim = new InstancePrimitive(in[0], curTransform.GetOp
    curGraphicsOptions->primitives.push_back(prim);
}
```

curGraphicsOptions	711
curTransform	708
InstancePrimitive	134
ParamSet	700
Primitive	130
Reference	664
SearchPath	743
vector	658
VERIFY_WORLD	708

B.3.6 Light Sources

Finally, we’ll define the routines that allow the user to specify light sources for the scene. The API provides two ways of doing this: the first, `lrtLightSource()` defines a light source that doesn’t have geometry associated with it (e.g. a point light or a directional light). The second, `lrtAreaLightSource()` specifies an active are light source; the primitives that follow it up to the end of the current attribute block are treated as emitting geometry as given by the area light description.

⟨*API Function Definitions*⟩+≡

```
void lrtLightSource(const string &name, const ParamSet &params) {
    VERIFY_WORLD("LightSource");
    Light *lt = CreateLight(name, curGraphicsOptions->SearchPath,
        curTransform.GetOpen(), params);
    ⟨Add new light to graphics state⟩
}
```

⟨*Add new light to graphics state*⟩≡

```
if (lt == NULL)
    Error("lrtLightSource: light type \"%s\" unknown.", name.c_str());
else
    curGraphicsOptions->lights.push_back(lt);
```

When an area light is specified, we can't create it immediately–we need to wait for the upcoming primitives which will define the light source's geometry. Therefore, as in `lrtMaterial()`, we just save away the name of the area light source type and the parameters given here.

⟨*API Function Definitions*⟩+≡

```
void lrtAreaLightSource(const string &name, const ParamSet &params) {
    VERIFY_WORLD("AreaLightSource");
    curGraphicsState.areaLight = name;
    curGraphicsState.areaLightParams = params;
}
```

⟨*Graphics State*⟩+≡

```
ParamSet areaLightParams;
string areaLight;
```

491	AreaLight
711	curGraphicsOptions
714	curGraphicsState
708	curTransform
478	Light
700	ParamSet
743	SearchPath
63	Shape
708	VERIFY_WORLD
465	VolumeRegion

We can now define the fragment ⟨*Initialize area light for shape*⟩ from the `GraphicsState::AddShape()` method. This just takes the area light information from `lrtAreaLightSource()` and the `Shape` passed in to `GraphicsState::AddShape()` to create an `AreaLight` object.

⟨*Initialize area light for shape*⟩≡

```
if (areaLight != "")
    area = CreateAreaLight(areaLight, curGraphicsOptions->SearchPath,
        curTransform.GetOpen(), areaLightParams, shape);
```

B.3.7 Volumes

⟨*API Function Definitions*⟩+≡

```
void lrtVolume(const string &name, const ParamSet &params) {
    VERIFY_WORLD("Volume");
    VolumeRegion *vr = CreateVolumeRegion(name,
        curGraphicsOptions->SearchPath,
        curTransform.GetOpen(), params);
    if (vr) curGraphicsOptions->volumeRegions.push_back(vr);
}
```

B.4 Scene Object Creation

Scene destructor frees up all this memory we allocated...

⟨*API Function Definitions*⟩+≡

```
Scene *GraphicsOptions::MakeScene() const {
    ⟨Initialize filter with pixel filter⟩
    ⟨Initialize film and camera from API settings⟩
    ⟨Initialize sampler from API settings⟩
    ⟨Initialize surfaceIntegrator from API settings⟩
    ⟨Initialize volumeIntegrator from API settings⟩
    ⟨Initialize accelerator from API settings⟩
    if (!camera || !sampler || !film || !accelerator ||
        !filter || !surfaceIntegrator || !volumeIntegrator) {
        Severe("Unable to create scene due to missing DSOs");
        return NULL;
    }
    ⟨Initialize volumeRegion from volume region(s)⟩
    Scene *ret = new Scene(camera, surfaceIntegrator, volumeIntegrator,
        sampler, accelerator, lights, volumeRegion);
    primitives.erase(primitives.begin(), primitives.end());
    lights.erase(lights.begin(), lights.end());
    volumeRegions.erase(volumeRegions.begin(), volumeRegions.end());
    return ret;
}
```

Filter	281
GraphicsOptions	710
phicsOptions::FilterName	711
icsOptions::FilterParams	711
Scene	8

⟨*Initialize* `filter` *with pixel filter*⟩≡

```
Filter *filter = CreateFilter(FilterName, SearchPath, FilterParams);
```

The fragments for creating the rest of the `Scene`-related objects are similarly straightforward calls to dynamic object creation routines, so won't all be included here.

Further Reading

RenderMan companion(Upstill 1989)
RI Spec(Pixar Animation Studios 1989)
Advanced RMan book
OpenGL stuff
What about scene graph, hierarchical graphics state ideas, etc?

C.Input File Format

The scene description files used by lrt are text files with a reasonably direct mapping between statements in these input files to calls to the API functions in Appendix B. For example, when the WorldBegin statement is parsed in an input file, the lrtWorldBegin() function is called. The file format was designed so that it would be both easy to parse by lrt and easy for other applications to generate from their own internal representations of a scene. While a binary file format could lead to smaller files and faster parsing of them, a text format is easier to edit by hand if necessary and makes it easier to convert to from other formats. The input file parser (which we won't include here) is very simple; it has no logic about when which statements are valid or not but just just passes them on to the corresponding API call, which worries about all such details.

The file examples/simple.lrt, reproduced in Figure C.1, is a short example of a lrt input file. Note that at the start of the file, up to the WorldBegin statement, overall options for rendering the scene like the camera type and position, the sampler and the image file to be written out are all configured. After WorldBegin, the lights and geometry in the scene are defined, up until the WorldEnd statement, which causes the image to be rendered. The hash character # denotes that the rest of the line is a comment and should be ignored by the parser.

Some of the statements in the input file have no additional arguments–examples include WorldBegin, AttributeBegin, etc.–and those related to specifying transformations, such as Rotate and LookAt take a fixed set of arguments. Most of them take a variable number of arguments and are of the form:

identifier "type" parameter-list

For example, Shape describes a shape to be added to the scene, where the name of the type of shape is given by a string, and then a set of parameters are specified.

```
# example lrt scene:  renders a disk with a checkerboard
LookAt 0 5 50 0 -1 0 0 1 0
Camera "perspective" "float fov" [30]
PixelFilter "mitchell" "float xwidth" [2] "float ywidth" [2]
Sampler "bestcandidate"
Film "image" "string filename" ["checkerboard.tiff"]
"integer xresolution" [300] "integer yresolution" [300]
WorldBegin
LightSource "distant" "point from" [0 10 0] "point to" [0 0 0]
"color L" [1 1 1]
AttributeBegin
Rotate 15 0 1 0
Rotate 100 1 0 0
Material "checkerboard" "float frequency" [30] "float Ks" [0]
Shape "disk" "float radius" [100] "float height" [-1]
AttributeEnd
WorldEnd
```

Figure C.1: Basic example of a `lrt` scene description file.

`ParamSet 700`

Statements of this type all correspond to one of the plug-in types that `lrt` supports; the *type* string gives the name of the particular plug-in implementation to use, and the *parameter list* gives the parameters to pass to the plug-in.

With this design, the parser doesn't need to know anything about the semantics of the parameters in the parameter list–it just needs to know how to parse general parameter lists and initialize a `ParamSet` from them. As such, an advantage of this approach is that the parser doesn't need to be modified as new plug-in implementations are addeed to the system.

Comments are delineated by hash character #.

Other files can be included via `Include`

```
Include "geometry.lrt"
```

C.1 Parameter Lists

The lists of named parameters and their values that appear in statements that support parameter lists are the key meeting-ground between the parsing system and the dynamically-loaded plug-ins. Each of these lists holds an arbitrary number of name/value pairs, with the name in quotation marks and the value or values inside brackets:

”*type name*” [*value*]

For example,

```
"float fov" [30]
```

specifies a parameter named “fov” that is a single floating-point value, here with value 30. Or,

```
"float cropwindow[4]" [0 .5 0 .25]
```

specifies that “cropwindow” is an array of four floats, with the given values. It is necessary that the complete type of each parameter be given along with its name; `lrt` has no built-in knowledge that the “from” parameter to a `PointLight` is of type `point`, for example. This is a small extra burden for the person or program creating the input file, it simplifies the parsing system in `lrt`.

`lrt` supports seven basic parameter types: `integer`, `float`, `point`, `vector`, `normal`, `color`, and `string`. The point, vector, and normal types all take three floating point values, while `color` takes `COLOR_SAMPES` floating point values. `string` parameters must be inside quotation marks:

```
"string filename" [ "output.tiff" ]
```

As shown by the “cropwindow” example above, arrays of these types may be given as well; the number of array elements is given by an integer value in brackets immediately after the name of the parameter.

In addition to the basic types described above, parameters can optionally have a *variability* associated with them. For example, in order to specify per-vertex normals for a triangle mesh, we’d like to be able to specify a parameter that is of type `normal` but has one value for each vertex of the mesh. Such a parameter is specified as type `vertex normal`. In a similar manner, we might want to specify a parameter that has values that are interpolated across the (u,v) parameterization of the surface, with four values specified for the points $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$. This type of parameter is of type `varying`. Most of the time, the variablity isn’t appropriate (e.g. for the a camera’s field of view) and doesn’t need to be specified. Alternatively, `uniform` variability can be redundantly provided.

C.2 Statement Types

With those basics of the input file format down, we will now enumerate the basic types of statement that are supported, what their parameters are, and which API calls they correspond to.

C.2.1 Transformations

The following set of routines update the current transformation matrix; see Section B.2.2 for further information about the API routines related to transformations. Each of them takes a fixed set of parameters of floating-point or string type, as corresponds to their API call.

Name	API Call
`Identity`	`lrtIdentity()`
`Translate` *x y z*	`lrtTranslate()`
`Scale` *x y z*	`lrtScale()`
`Rotate` *angle x y z*	`lrtRotate()`
`LookAt` *ex ey ez lx ly lz ux uy uz*	`lrtLookAt()`
`CoordinateSystem` "*name*"	`lrtCoordinateSystem()` ”
`CoordSysTransform` "*name*"	`lrtCoordSysTransform()`
`Transform` *m00* ...*m33*	`lrtTransform()`
`ConcatTransform` *m00* ...*m33*	`lrtConcatTransform()`

C.2.2 Options

Rendering options all must be specified before the `WorldBegin` statement that begins the description of the contents of the scene.

`SearchPath` can be used to specify the set of directories `lrt` looks in to find plug-ins at runtime. By default, it looks in the current directory and the directories that the supplied plug-ins were stored in when they were originally compiled.

Name	API Call
`SearchPath` *"path"*	`lrtSearchPath()`

The string *path* given can provide multiple directory names separated by colons; the previous value of the search path can be specified with &. For example,

```
SearchPath "/ext/lrtplugins:&:./plugins"
```

The rest of the options all specify a plug-in of a various type to use. All take the name of the specific plug-in to use and its parameter list.

Name	API Call
`Camera` *"name" parameter-list*	`lrtCamera()`
`Sampler` *"name" parameter-list*	`lrtSampler()`
`PixelFilter` *"name" parameter-list*	`lrtPixelFilter()`
`Film` *"name" parameter-list*	`lrtFilm()`
`Sampler` *"name" parameter-list*	`lrtSampler()`
`Accelerator` *"name" parameter-list*	`lrtAccelerator()`
`ToneMap` *"name" parameter-list*	`lrtToneMap()`
`SurfaceIntegrator` *"name" parameter-list*	`lrtSurfaceIntegrator()`
`VolumeIntegrator` *"name" parameter-list*	`lrtVolumeIntegrator()`

Of these only `Camera` uses the value of the the current transformation matrix when it is encountered in an input file: it is used to initialize the world to camera transformation.

C.2.3 Attributes

The `WorldBegin` statement marks the end of options specification and the start of the description of the lights and geometry in the scene. `WorldEnd` denotes the end of the scene description; rendering starts when it is encountered.

Name	API Call
`WorldBegin`	`lrtWorldBegin()`
`WorldEnd`	`lrtWorldEnd()`

The current graphics state is saved and restored with `AttributeBegin` and `AttributeEnd`.

Name	API Call
`AttributeBegin`	`lrtAttributeBegin()`
`AttributeEnd`	`lrtAttributeEnd()`

In addition to the current transformation matrix, `lrt` stores only a small amount of additional information in the graphics state. The four statements below all set values in the graphics state that will be used when geometry is created for objects in the scene.

Name	API Call
Material *"name" parameter-list*	lrtMaterial()
BumpMap *"name" parameter-list*	lrtBumpMap()
ReverseOrientation	lrtReverseOrientation()

C.2.4 Lights, Shapes, and Volumes

Most of the statements inside the WorldBegin block will specify shapes and light sources for the scene. The corresponding input file statements are listed in this table:

Name	API Call
Shape *"name" parameter-list*	lrtShape()
LightSource *"name" parameter-list*	lrtLightSource()
AreaLightSource *"name" parameter-list*	lrtAreaLightSource()

All of these make use of the current transformation matrix: for shapes, it is used for the object to world transformation; for lights, the light to world transformation. **and for volumes, volume to world.**

C.2.5 Object Instancing

ObjectBegin ”name”
ObjectEnd
ObjectInstance ”name”

217 EnvironmentCamera
210 PerspectiveCamera

C.3 Standard Plug-ins

Reference about the names and parameters used for setting up all of the plug-ins defined throughout this book....

C.3.1 Cameras

There are three standard Camera models in lrt:

Name	Class
"orthographic"	OrthographicCamera
"perspective"	PerspectiveCamera
"environment"	EnvironmentCamera

Parameter	Default Value	Definition
"float hither"	1e-3	
"float yon"	1e30	
"float shutteropen"	0	
"float shutterclose"	0	
"float lensradius"	0	
"float focaldistance"	1e30	
"float pixelaspectratio"	1	
"float frameaspectratio"	xres/yres	
"float screenwindow[4]"	....	

Implementation	Parameter	Default Value	Definition
"perspective"	"float fov"	90	

C.3.2 ToneMaps

Name	Class
"contrast"	ContrastOp
"highconstrast"	HighContrastOp
"maxwhite"	MaxWhiteOp
"nonlinear"	NonLinearOp

nonlinear takes displayadaptationy (default 50)

C.3.3 Samplers

all:

Parameter	Default Value	Definition
"integer xsamples"	2	
"integer ysamples"	2	
"integer xresolution"	640	
"integer yresolution"	480	
"float cropwindow[4]"	0 1 0 1	

"stratified"	StratifiedSampler
"lowdiscrepancy"	LDSampler
"bestcandidate"	BestCandidateSampler

"stratified"	"integer jitter"	1	

"lowdiscrepancy"	"integer zaremba"	0	

BestCandidateSampler	275
ContrastOp	313
HighContrastOp	315
LDSampler	258
MaxWhiteOp	312
NonLinearOp	322
StratifiedSampler	244

C.3.4 Film

image

Parameter	Default Value	Definition
"integer xresolution"	640	
"integer yresolution"	480	
"integer writefrequency"	-1	
"float cropwindow[4]"	(0, 1, 0, 1)	
"string filename"	”lrt.tiff”	
"integer writecoefficients"	0	
"integer premultiplyalpha"	1	
"float XYZToR[3]"	(3.240479, -1.537150, -0.498535)	
"float XYZToG[3]"	(-0.969256, 1.875991f, 0.041556)	
"float XYZToB[3]"	(0.055648, -0.204043f, 1.057311)	
"float gain"	1	
"float gamma"	1	
"integer integerformat"	1	
"integer max"	255	
"float dither"	0.5	
"float bloomradius"	0.	
"float bloomfraction"	0.2	

C.3.5 Filters

all:

Parameter	Default Value	Definition
`"float xwidth"`	2	
`"float ywidth"`	2	

Name	Class
`"box"`	`BoxFilter`
`"triangle"`	`TriangleFilter`
`"gaussian"`	`GaussianFilter`
`"mitchell"`	`MitchellFilter`
`"sinc"`	`SincFilter`

Sinc default width is 4...

`"gaussian"`	`"float alpha"`	2	
`"sinc"`	`"float tau"`	3	

C.3.6 Shapes

Name	Class
`"sphere"`	`Sphere`
`"cone"`	`Cone`
`"cylinder"`	`Cylinder`
`"disk"`	`Disk`
`"hyperboloid"`	`Hyperboloid`
`"paraboloid"`	`Paraboloid`
`"trianglemesh"`	`TriangleMesh`
`"subdiv"`	`LoopSubdiv`
`"nurbs"`	`NURBS`

sphere

Parameter	Default Value	Definition
`"float radius"`	1	
`"float zmin"`	-radius	
`"float zmax"`	radius	
`"float thetamax"`	360	

cone

Parameter	Default Value	Definition
`"float radius"`	1	
`"float height"`	1	
`"float thetamax"`	360	

cylinder

Parameter	Default Value	Definition
`"float radius"`	1	
`"float zmin"`	-1	
`"float zmax"`	1	
`"float thetamax"`	360	

disk

Parameter	Default Value	Definition
`"float height"`	0	
`"float radius"`	1	
`"float thetamax"`	360	

282	`BoxFilter`
78	`Cylinder`
82	`Disk`
285	`GaussianFilter`
100	`LoopSubdiv`
286	`MitchellFilter`
289	`SincFilter`
68	`Sphere`
284	`TriangleFilter`
87	`TriangleMesh`

heightfield

Parameter	Default Value	Definition
"int nu"	u	
"int nv"	v	
"vertex float Pz"	....	

hyperboloid

Parameter	Default Value	Definition
"point p1"	0 0 0	
"point p2"	1 1 1	
"float thetamax"	360	

nurbs

Parameter	Default Value	Definition
"integer nu"		
"integer uorder"		
"float uknots"	...	
"float u0"		
"float u1"		
"integer nv"		
"integer vorder"		
"float vknots"	...	
"float v0"		
"float v1"		
"vertex point P"	...	
"vertex float Pw[4]"	...	

GridAccel 139
KdTreeAccel 154

paraboloid

Parameter	Default Value	Definition
"float radius"	1	
"float zmin"	0	
"float zmax"	1	
"float thetamax"	360	

subdiv

Parameter	Default Value	Definition
"integer nlevels"	3	
"integer vertices"	...	
"integer nvertices"	...	
"vertex point P"	...	
"string scheme"	loop	

trianglemesh

Parameter	Default Value	Definition
"integer indices"	...	
"vertex point P"	...	
"vertex point N"	...	
"vertex point st"	...	

C.3.7 Accelerators

Name	Class
"grid"	GridAccel
"kdtree"	KdTreeAccel

grid

Parameter	Default Value	Definition
"integer refineimmediately"	0	

	Parameter	Default Value	Definition
kd-tree	`"integer intersectcost"`	80	
	`"integer traversalcost"`	1	
	`"integer emptybonus"`	0.2	
	`"integer maxprims"`	1	
	`"integer maxdepth"`	-1	

C.3.8 Materials

Name	Class
`"matte"`	`Matte`
`"plastic"`	`Plastic`
`"translucent"`	`Translucent`
`"mirror"`	`Mirror`
`"glass"`	`Glass`
`"shinymetal"`	`ShinyMetal`
`"substrate"`	`Substrate`
`"clay"`	`Clay`
`"felt"`	`Felt`
`"primer"`	`Primer`
`"skin"`	`Skin`
`"bluepaint"`	`BluePaint`
`"brushedmetal"`	`BrushedMetal`
`"uber"`	`UberMaterial`

Name	Class
`"checkerboard"`	`Plastic XXX`
`"dots"`	`Plastic XXX`
`"marble"`	`Plastic XXX`

	Parameter	Default Value	Definition
matte	`"color basecolor"`	1	
	`"color Kd"`	1	
	`"float sigma"`	0	

	Parameter	Default Value	Definition
plastic	`"color basecolor"`	1	
	`"color specularcolor"`	1	
	`"float Kd"`	1	
	`"float Ks"`	1	
	`"float roughness"`	0.1	
	`"string texturename"`	””	

389	`BluePaint`
389	`BrushedMetal`
389	`Clay`
389	`Felt`
384	`Glass`
381	`Matte`
385	`Mirror`
382	`Plastic`
389	`Primer`
386	`ShinyMetal`
389	`Skin`
387	`Substrate`
383	`Translucent`

	Parameter	Default Value	Definition
translucent	`"color basecolor"`	1	
	`"color Kd"`	1	
	`"color Ks"`	1	
	`"float roughness"`	0.1	
	`"color specularcolor"`	1	
	`"string texturename"`	””	
	`"color reflect"`	.5	
	`"color transmit"`	.5	

	Parameter	Default Value	Definition
mirror	`"color basecolor"`	0.5	
	`"color Kr"`	1	

	Parameter	Default Value	Definition
glass	`"color basecolor"`	1	
	`"color Kr"`	1	
	`"color Kt"`	1	
	`"float index"`	1.5	

	Parameter	Default Value	Definition
shinymetal	`"color basecolor"`	1	
	`"float roughness"`	0.1	
	`"color Ks"`	1	
	`"color Kr"`	1	

	Parameter	Default Value	Definition
substrate	`"color basecolor"`	1	
	`"color glossycolor"`	1	
	`"color Kr"`	1	
	`"color Ks"`	1	
	`"float uroughness"`	0.1	
	`"float vroughness"`	0.1	

	Parameter	Default Value	Definition
checkerboard	`"color Kd"`	1	
	`"color Ks"`	1	
	`"float roughness"`	0.1	
	`"color specularcolor"`	1	
	`"color color1"`	0.1	
	`"color color2"`	1.0	
	`"float frequency"`	1	
	`"integer dimension"`	2	
	`"float noisescale"`	0.	
	`"string aamode"`	{”closedform,supersample,none”}	

dots

Parameter	Default Value	Definition
`"color Kd"`	1	
`"color Ks"`	1	
`"float roughness"`	0.1	
`"color specularcolor"`	1	
`"color dotcolor"`	0.1	
`"color basecolor"`	1.0	
`"float frequency"`	1	

marble

Parameter	Default Value	Definition
`"color Kd"`	1	
`"color Ks"`	1	
`"float roughness"`	0.1	
`"float frequency"`	1	
`"float variation"`	.5	
`"int maxoctaves"`	8	

uber

Parameter	Default Value	Definition
`"color diffusecolor"`	1	
`"color specularcolor"`	1	
`"float roughness"`	0.1	
`"color opacity"`	1.	
`"string basetex"`	””	
`"string opacitytex"`	””	

472 ExponentialDensity
467 HomogeneousVolume
470 VolumeGrid

C.3.9 Bump Functions

imagemap

Parameter	Default Value	Definition
`"float scale"`	1	
`"string texturename"`	””	

windy

Parameter	Default Value	Definition
`"float stretch"`	1	
`"float scale"`	1	

bumpy

Parameter	Default Value	Definition
`"float octaves"`	6	
`"float omega"`	.5	

C.3.10 Volumes

Name	Class
`"exponential"`	`ExponentialDensity`
`"homogeneous"`	`HomogeneousVolume`
`"volumegrid"`	`VolumeGrid`

common:

Parameter	Default Value	Definition
"color sigma_a"	0 0 0	
"color sigma_s"	0 0 0	
"float g"	0	
"color Le"	0 0 0	
"point p0"	0 0 0	
"point p1"	1 1 1	

exponential

Parameter	Default Value	Definition
"float a"	1	
"float b"	1	
"vector updir"	(0,1,0)	

volumegrid

Parameter	Default Value	Definition
"int nx"	1	
"int ny"	1	
"int nz"	1	
"float density[nx*ny*nz]"	””	

C.3.11 Lights

AreaLight	491
DistantLight	489
GoniometricLight	488
InfiniteAreaLight	493
PointLight	480
ProjectionLight	484
SpotLight	482

Name	Class
"area"	AreaLight
"distant"	DistantLight
"goniometric"	GoniometricLight
"infinite"	InfiniteAreaLight
"point"	PointLight
"projection"	ProjectionLight
"spot"	SpotLight

area

Parameter	Default Value	Definition
"color L"	1 1 1	

distant

Parameter	Default Value	Definition
"color L"	1 1 1	
"point from"	0 0 0	
"point to"	0 0 1	

goniometric

Parameter	Default Value	Definition
"color I"	1 1 1	
"string mapname"	””	

infinite

Parameter	Default Value	Definition
"color L"	1 1 1	
"string mapname"	””	

point

Parameter	Default Value	Definition
"color I"	1 1 1	
"point from"	0 0 0	

projection

Parameter	Default Value	Definition
"color I"	1 1 1	
"float fov"	45	
"string mapname"	””	

spot

Parameter	Default Value	Definition
"color I"	1 1 1	
"float coneangle"	30	
"float conedeltaangle"	5	

C.3.12 SurfaceIntegrators

Name	Class
"whitted"	WhittedIntegrator
"directlighting"	DirectLightingIntegrator
"path"	PathIntegrator
"bidirectional"	BidirIntegrator
"photonmap"	PhotonIntegrator
"irradiancecache"	IrradianceCache

whitted

Parameter	Default Value	Definition
"integer maxdepth"	5	

directlighting

Parameter	Default Value	Definition
"integer maxdepth"	5	
"string strategy"	”all”/”one”/”weighted”	

photonmap

Parameter	Default Value	Definition
"integer causticphotons"	20000	
"integer indirectphotons"	100000	
"integer directphotons"	100000	
"integer nused"	50	
"integer maxdepth"	5	
"float maxdist"	.1	
"integer finalgather"	1	
"integer finalgathersamples"	32	
"integer directwithphotons"	0	
"integer nfilter"	8	

591	BidirIntegrator
631	EmissionIntegrator
598	IrradianceCache
586	PathIntegrator
611	PhotonIntegrator
636	SingleScattering
16	WhittedIntegrator

irradiancecache

Parameter	Default Value	Definition
"float maxerror"	.2	
"integer maxspeculardepth"	5	
"integer maxindirectdepth"	3	
"integer nsamples"	4096	
"integer nfilter"	8	

C.3.13 VolumeIntegrators

"single"	SingleScattering
"emission"	EmissionIntegrator

emission

Parameter	Default Value	Definition
"float stepsize"	1	

single

Parameter	Default Value	Definition
"float stepsize"	1	

Exercises

3.1 `lrt`'s scene file parser is written using the standard `lex` and `yacc` tools. While these are an easy way to develop such a parser, carefully implemented hand-written parsers can be substantially more efficient. Replace `lrt`'s parser with a hand-written parser that parses the same file format. Measure the change in performance with your parser. Profile `lrt` when rendering complex scenes like **ecosysXXXX.lrt**; what fraction of time rendering is spent in parsing?

3.2 `lrt`'s scene description format makes it easy for other programs to export `lrt` scenes and makes it easy for users to edit the scene files to make small adjustments to them. However, for complex scenes **such as ecosysXXXX.lrt**, the large text files that are necessary to describe them can take a long time to parse and may occupy a lot of disk space.

Investigate extensions to `lrt` to support compact binary file formats for scene files. Three possible ways to approach this problem are:

`LoopSubdiv`	100
`TriangleMesh`	87

- Full binary file format: each of the scene file directives (`Shape`, `Filter`, etc), could be encoded compactly, for example with a single bte. The name of the particular instance type being created can then be encoded as a NULL-terminated string or as an integer index into a string table. The parameter list that follows could also be encoded compactly, for example using a single byte to denote the type of each parameter, an integer to encode the number of parameter values, the parameter name as a string or an index into a table, and then the parameter values in raw binary form, rather than in text. (See Appendix XXX of the RenderMan specification (Pixar Animation Studios 1989) for an overview of RenderMan's binary encoding, which is along thesel ines.)
- Binary encoding for large meshes: since most of the complexity in detailed scenes comes from large polygon and subdivision surface meshes, providing specialized encodings for just those shapes may be almost as effective as a binary encoding for the entire scene description format and has the advantage of not requiring that the scene file parser be rewritten. Extend the `TriangleMesh` or `LoopSubdiv` to take an optional string parameter that gives the filename for abinary file that holds some or all of the mesh vertex positions and normals and the array of integers that describes which triangles uses which vertices.
- Binary representation of internal data structures: for complex scenes, creating the ray acceleration aggregates may take more time than the initial parsing of the scene file. A third alternative is to modify the system to have the capability of dumping out a representation of the acceleration structure and all of the primitives inside it after it is first created. The resulting file could then be subsequently read back into memory much more quickly than rebuilding the data structure from scratch. However, because C++ doesn't have native support for saving arbitrary objects to disk and then reading them back during a subsequent execution of the program (a capability known as serialization or

pickling in other languages), adding this feature effectively requires extending many of the objects in `lrt` to support this capability on their own.

All three of these approaches have their advantages and disadvantages, though the second is the easiest to implement and will usually solve the basic problem well. The third approach isn't too much harder than the first, and should give he best performance of the three of them.

D.Dynamic Object Creation

One of the key parts of lrt's design was the decision that the lrt executable would only hold the key core logic of the system. All of the shapes, cameras, lights, integrators, and accelerators are stored in separate object files on disk; at run-time, lrt loads in the appropriate object code for the needed objects. This makes it far easier to extend lrt with new implementations of various types and helps ensure a clean design by making it much harder to side-step the basic system interfaces.

⟨*dynload.h**⟩≡

```
  #include "lrt.h"
  ⟨Runtime Loading Declarations⟩
```

⟨*dynload.cpp**⟩≡

```
  #include "dynload.h"
  #include "paramset.h"
  #include "shape.h"
  #include "material.h"
  #ifndef WIN32
  #include <dlfcn.h>
  #endif
  #include <map>
  using std::map;
  ⟨Runtime Loading Forward Declarations⟩
  ⟨Runtime Loading Static Data⟩
  ⟨Runtime Loading Local Classes⟩
  ⟨Runtime Loading Method Definitions⟩
  ⟨DSO Method Definitions⟩
```

⟨*Global Include Files*⟩+≡

```
#ifdef WIN32
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#endif
```

⟨*Platform-specific definitions*⟩+≡

```
#ifdef WINDOWS
#ifdef CORE_SOURCE
#define COREDLL __declspec(dllexport)
#else
#define COREDLL __declspec(dllimport)
#endif
#else
#define COREDLL
#endif
```

D.1 Reading Dynamic Libraries

ParamSet	700
Reference	664
Shape	63
Transform	43

In this section, we will describe the general process that `lrt` uses to link in implementations at runtime. We will focus on the details only for the `Shape` class, since the other times that are loaded at runtime are handled quite similarly.

D.1.1 Creation Functions

All of the object files that hold shape implementations must provide a function with the same signature. When `lrt` needs to create a particular shape, it will call this function from the appropriate object file.

⟨*Shape Creation Declaration*⟩≡

```
Reference<Shape> CreateShape(const Transform &o2w, bool reverseOrientation,
    const ParamSet &params);
```

Because all `Shape`s store an object to world transformation, we pass the appropriate transformation to this function. However, in general we need to be able to pass whichever other parameters the particular shape needs and that the user may have set in the input file. Because we don't want to hard-code knowledge like "spheres need to have a floating-point radius value passed to their constructor" into `lrt`, we use the `ParamSet` to handle marshal parameters and their values for use by the individual shapes.

The dynamic sphere creation routine just pulls the appropriate values out of the `ParamSet` and cals the constructor, returning a newly-allocated sphere.

⟨*Sphere Method Definitions*⟩+≡

```
extern "C" Reference<Shape> CreateShape(const Transform &o2w,
        bool reverseOrientation, const ParamSet &params) {
    Float radius = params.FindOneFloat("radius", 1);
    Float zmin = params.FindOneFloat("zmin", -radius);
    Float zmax = params.FindOneFloat("zmax", radius);
    Float thetamax = params.FindOneFloat("thetamax", 360);
    return new Sphere(o2w, reverseOrientation, radius,
        zmin, zmax, thetamax);
}
```

The creation routines for other shapes are quite similar, so won't be included here.

XXX include the basic signatures for the other object creation functions here, though XXX

D.1.2 Loading object files

Loading an object file with such a function to be called from disk and linking it into a running application can be done relatively easy in modern operating systems. The system calls to use are highly operating-system dependent, however. The `DSO` base-class is one key to this process; it hides the operating-system-dependent parts of it.

700	`ParamSet`
664	`Reference`
63	`Shape`
68	`Sphere`
43	`Transform`

Dynamic shared object DSO

Dynamic link library DLL

XXX what is a DSO, DSO vs DLL. Rename this class? XXX

⟨*Global Classes*⟩+≡

```
class DSO {
public:
    ⟨DSO Public Methods⟩
private:
#if defined(WIN32)
    HMODULE hinstLib;
#else
    void *hinstLib;
#endif
};
```

The `DSO` constructor handles the first step of loading the shared object into `lrt`'s address spade. It takes a pathname to the object file.

⟨*DSO Method Definitions*⟩≡

```
DSO::DSO(const string &fname) {
#ifdef WIN32
    hinstLib = LoadLibrary(fname.c_str());
    if (!hinstLib)
        Error("DSO Loader can’t open DLL \"%s\"", fname.c_str());
#else
    hinstLib = dlopen(fname.c_str(), RTLD_LAZY);
    if (!hinstLib)
        Error("DSO Loader can’t open DLL \"%s\" (%s)", fname.c_str(),
            dlerror());
#endif
}
```

And the destructor makes the system call to remove the library from our address space.

⟨*DSO Method Definitions*⟩+≡

```
DSO::˜DSO() {
#ifdef WIN32
    FreeLibrary(hinstLib);
#else
    dlclose(hinstLib);
#endif
}
```

DSO	739
ShapeDSO	741

Once a library has been loaded into memory, the `GetSymbol()` function lets us ask for a function inside the DSO with a particular name. If that function exists, then this returns a pointer to it which we can use to actually call it.

⟨*DSO Method Definitions*⟩+≡

```
void *DSO::GetSymbol(const string &symname) {
    void *data;
#ifdef WIN32
    data = GetProcAddress(hinstLib, symname.c_str());
#else
#ifdef __APPLE__
    string sym = string("_") + symname;
    data = dlsym(hinstLib, sym.c_str());
#else
    data = dlsym(hinstLib, symname.c_str());
#endif
#endif
    if (!data)
        Error("Couldn’t get symbol \"%s\" in DSO.", symname.c_str());
    return data;
}
```

For each base type for which we are able to load implementations at runtime, we inherit from `DSO`. Here is the implementation of `ShapeDSO`. All of these implementations just call the `DSO` `GetSymbol` function in the constructor, passing in the name of the object creation function (e.g. CreateShape(), which was introduced previously in this section.) All Shape shared object files implement this function and return a new Shape of their particular type when it is called.

⟨*Runtime Loading Local Classes*⟩≡

```
class ShapeDSO : public DSO {
public:
    ⟨ShapeDSO Constructor⟩
    typedef Reference<Shape> (*CreateShapeFunc)(const Transform &o2w,
        bool reverseOrientation, const ParamSet &params);
    CreateShapeFunc CreateShape;
};
```

One possibly dangerous thing that the constructor does is cast the returned symbol to be a pointer to a function with the right signature for creating shapes. If the person who implemented a particular Shape defined it with a CreateShape function that only took a ParamSet and didn't have a Transform parameter, the program would probably crash at run-time if it tried to call that function. In the interests of making it easier to keep lrt portable across architectures, we'll just take that risk and keep the code here simpler.

⟨*ShapeDSO Constructor*⟩≡

```
ShapeDSO(const string &name)
    : DSO(name) {
    CreateShape = (CreateShapeFunc)(GetSymbol("CreateShape"));
}
```

739	DSO
700	ParamSet
664	Reference
63	Shape
43	Transform

XXX call this function something else! XXX

The function that the main section of lrt uses when it actually needs to create a shape is also called CreateShape. It takes the name of the shape to be created, the object to world transformation, and the ParamSet for the new shape. It calls GetShapeDSO, which will be defined shortly–it returns the DSO for the named shape if it exists–and it then calls the creation function pointer that the DSO holds to actually cause the particular shape to be made.

⟨*Runtime Loading Method Definitions*⟩≡

```
Reference<Shape> CreateShape(const string &name, const string &searchpath,
        const Transform &object2world, bool reverseOrientation,
        const ParamSet &paramSet) {
    ShapeDSO *dso = LoadDSO<ShapeDSO>(name, shape_dsos, searchpath);
    if (dso)
        return dso->CreateShape(object2world, reverseOrientation, paramSet);
    return NULL;
}
```

⟨*Runtime Loading Forward Declarations*⟩≡

```
template <class D> D *LoadDSO(const string &name,
        map<string, D *> &loadedDSOs,
        const string &searchPath) {
    if (loadedDSOs.find(name) != loadedDSOs.end())
        return loadedDSOs[name];
    string filename = name;
#ifdef WIN32
    filename += ".dll";
#else
    filename += ".so";
#endif
    string path = SearchPath(searchPath, filename);
    D *dso = NULL;
    if (path != "")
        loadedDSOs[name] = dso = new D(path.c_str());
    else
        Error("Unable to find DSO/DLL for \"%s\"",
            name.c_str());
    return dso;
}
```

DSO	739
SearchPath	743
ShapeDSO	741

⟨*Runtime Loading Forward Declarations*⟩+≡

```
class ShapeDSO;
class FilterDSO;
class MaterialDSO;
class BumpDSO;
class FilmDSO;
class LightDSO;
class AreaLightDSO;
class VolumeRegionDSO;
class SurfaceIntegratorDSO;
class VolumeIntegratorDSO;
class ToneMapDSO;
class AcceleratorDSO;
class CameraDSO;
class SamplerDSO;
```

⟨*Runtime Loading Static Data*⟩≡

```
static map<string, ShapeDSO *> shape_dsos;
static map<string, FilterDSO *> filter_dsos;
static map<string, MaterialDSO *> material_dsos;
static map<string, BumpDSO *> bump_dsos;
static map<string, FilmDSO *> film_dsos;
static map<string, LightDSO *> light_dsos;
static map<string, AreaLightDSO *> arealight_dsos;
static map<string, VolumeRegionDSO *> volume_dsos;
static map<string, SurfaceIntegratorDSO *> surf_integrator_dsos;
static map<string, VolumeIntegratorDSO *> vol_integrator_dsos;
static map<string, ToneMapDSO *> tonemap_dsos;
static map<string, AcceleratorDSO *> accelerator_dsos;
static map<string, CameraDSO *> camera_dsos;
static map<string, SamplerDSO *> sampler_dsos;
```

⟨*Runtime Loading Method Definitions*⟩+≡

```
static string SearchPath(const string &searchpath,
        const string &filename) {
    const char *start = searchpath.c_str();
    const char *end = start;
    while (*start) {
        while (*end && *end != ':')
            ++end;
        string component(start, end);

        string fn = component + "/" + filename;
        FILE *f = fopen(fn.c_str(), "r");
        if (f) {
            fclose(f);
            return fn;
        }
        if (*end == ':') ++end;
        start = end;
    }
    return "";
}
```

375 Material
741 ShapeDSO
181 Spectrum
394 Texture

D.2 Object Creation Functions

⟨*Material Creation Macros*⟩≡

```
#define SURF_TEX_S(var, def) \
    Texture<Spectrum> *(var) = Material::MakeSpecTex(geomParams, surfaceParams, \
        #var, def)
#define SURF_TEX_F(var, def) \
    Texture<Float> *(var) = Material::MakeFloatTex(geomParams, surfaceParams, \
        #var, def)
```

⟨*Material Method Definitions*⟩+≡

```
Texture<Spectrum> *Material::MakeSpecTex(const ParamSet &pGeom, const ParamSet &
        const string &name, const Spectrum &def) {
    int type, narray, nitems;
    const Spectrum *s = pGeom.FindSpectrum(name, &type, &narray, &nitems);
    if (!s) s = pShader.FindSpectrum(name, &type, &narray, &nitems);
    if (!s) return new ConstantTexture<Spectrum>(def);
    Assert(narray == 1); // XXX for now

    if (type & PARAM_TYPE_UNIFORM)
        return new ConstantTexture<Spectrum>(*s);
    else if (type & PARAM_TYPE_VARYING) {
        Assert(nitems == 4);
        return new BilerpTexture<Spectrum>(new IdentityMapping2D,
            s[0], s[1], s[2], s[3]);
    }
    return NULL;
}
```

BilerpTexture	411
BoxFilter	282
ConstantTexture	395
Filter	281
IdentityMapping2D	406
Material	375
PARAM_TYPE_UNIFORM	702
PARAM_TYPE_VARYING	702
ParamSet	700
Spectrum	181
Texture	394

⟨*Material Method Definitions*⟩+≡

```
Texture<Float> *Material::MakeFloatTex(const ParamSet &pGeom, const ParamSet &p
        const string &name, Float def) {
    int type, narray, nitems;
    const Float *s = pGeom.FindFloat(name, &type, &narray, &nitems);
    if (!s)
        s = pShader.FindFloat(name, &type, &narray, &nitems);
    if (!s) return new ConstantTexture<Float>(def);
    Assert(narray == 1); // XXX for now
    if (type & PARAM_TYPE_UNIFORM)
        return new ConstantTexture<Float>(*s);
    else if (type & PARAM_TYPE_VARYING) {
        Assert(nitems == 4);
        return new BilerpTexture<Float>(new IdentityMapping2D,
            s[0], s[1], s[2], s[3]);
    }
    return NULL;
}
```

⟨*Find common filter parameters*⟩≡

```
Float xw = ps.FindOneFloat("xwidth", 2.);
Float yw = ps.FindOneFloat("ywidth", 2.);
```

⟨*Box Filter Method Definitions*⟩+≡

```
extern "C" Filter *CreateFilter(const ParamSet &ps) {
    ⟨Find common filter parameters⟩
    return new BoxFilter(xw, yw);
}
```

⟨*Triangle Filter Method Definitions*⟩+≡

```
extern "C" Filter *CreateFilter(const ParamSet &ps) {
    ⟨Find common filter parameters⟩
    return new TriangleFilter(xw, yw);
}
```

⟨*Gaussian Filter Method Definitions*⟩+≡

```
extern "C" Filter *CreateFilter(const ParamSet &ps) {
    ⟨Find common filter parameters⟩
    Float alpha = ps.FindOneFloat("alpha", 2.f);
    return new GaussianFilter(xw, yw, alpha);
}
```

⟨*Mitchell Filter Method Definitions*⟩+≡

```
extern "C" Filter *CreateFilter(const ParamSet &ps) {
    ⟨Find common filter parameters⟩
    Float B = ps.FindOneFloat("B", 1.f/3.f);
    Float C = ps.FindOneFloat("C", 1.f/3.f);
    return new MitchellFilter(B, C, xw, yw);
}
```

281	Filter
285	GaussianFilter
286	MitchellFilter
700	ParamSet
289	SincFilter
284	TriangleFilter

⟨*Sinc Filter Method Definitions*⟩+≡

```
extern "C" Filter *CreateFilter(const ParamSet &ps) {
    Float xw = ps.FindOneFloat("xwidth", 4.);
    Float yw = ps.FindOneFloat("ywidth", 4.);
    Float tau = ps.FindOneFloat("tau", 3.f);
    return new SincFilter(xw, yw, tau);
}
```

⟨*Initialize common sampler parameters*⟩≡

```
int xsamp = params.FindOneInt("xsamples", 2);
int ysamp = params.FindOneInt("ysamples", 2);
int xstart, xend, ystart, yend;
film->GetSampleExtent(&xstart, &xend, &ystart, &yend);
```

Bibliography

Akenine-Möller, T. and E. Haines (2002). *Real-Time Rendering*. A. K. Peters.

Amanatides, J. (1984, July). Ray tracing with cones. In H. Christiansen (Ed.), *Computer Graphics (SIGGRAPH '84 Proceedings)*, Volume 18, pp. 129–135.

Amanatides, J. (1992, May). Algorithms for the detection and elimination of specular aliasing. In *Graphics Interface '92*, pp. 86–93.

Amanatides, J. and A. Woo (1987, August). A fast voxel traversal algorithm for ray tracing. In *Eurographics '87*, pp. 3–10.

Anton, H. A., I. Bivens, and S. Davis (2001). *Calculus* (7 ed.). John Wiley & Sons.

Apodaca, A. A. and L. Gritz (2000). *Advanced RenderMan: creating CGI for motion pictures*. Morgan Kaufmann.

Appel, A. (1968). Some techniques for shading machine renderings of solids. In *AFIPS 1968 Spring Joint Computer Conf.*, Volume 32, pp. 37–45.

Arnaldi, B., T. Priol, and K. Bouatouch (1987, August). A new space subdivision method for ray tracing CSG modelled scenes. *The Visual Computer 3*(2), 98–108.

Arvo, J. (1986, August). Backward ray tracing.

Arvo, J. (1988, March). Linear-time voxel walking for octrees.

Arvo, J. (1990). Transforming axis-aligned bounding boxes. In A. S. Glassner (Ed.), *Graphics Gems I*, pp. 548–550. Academic Press.

Arvo, J. (1993, August). Transfer equations in global illumination. In *Global Illumination, SIGGRAPH '93 Course Notes*, Volume 42.

Arvo, J. (1995, December). *Analytic Methods for Simulated Light Transport*. Ph. D. thesis, Yale University.

Arvo, J. and D. Kirk (1990, August). Particle transport and image synthesis. *Computer Graphics 24*(4), 63–66.

Arvo, J. and D. B. Kirk (1987, July). Fast ray tracing by ray classification. In M. C. Stone (Ed.), *Computer Graphics (SIGGRAPH '87 Proceedings)*, Volume 21, pp. 55–64.

Ashikhmin, M. (2002, June). A tone mapping algorithm for high contrast images. In *The proceedings of 13th Eurographics Workshop on Rendering*, Pisa, Italy, pp. 145–155.

Ashikhmin, M., S. Premoze, and P. S. Shirley (2000, July). A microfacet-based brdf generator. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pp. 65–74. ACM Press / ACM SIGGRAPH / Addison Wesley Longman. ISBN 1-58113-208-5.

Ashikhmin, M. and P. Shirley (2000, June). An anisotropic Phong light reflection model. Technical report UUCS-00-014.

Ashikhmin, M. and P. Shirley (2002). An anisotropic Phong BRDF model. *Journal of Graphics Tools 5*(2), 25–32.

Atkinson, K. (1993). *Elementary Numerical Analysis*. John Wiley & Sons.

Badouel, D. and T. Priol (1989). An efficient parallel ray tracing scheme for highly parallel architectures. In *Fifth Eurographics Workshop on Graphics Hardware*.

Banks, D. C. (1994, July). Illumination in diverse codimensions. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pp. 327–334.

Barzel, R. (1997). Lighting controls for computer cinematography. *Journal of Graphics Tools 2*(1), 1–20. ISSN 1086-7651.

Berger, E. D., B. G. Zorn, and K. S. McKinley (2001). Composing high-performance memory allocators. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 114–124.

Berger, E. D., B. G. Zorn, and K. S. McKinley (2002). Reconsidering custom memory allocation. In *Proceedings of ACM OOPSLA 2002*.

Betrisey, C., J. F. Blinn, B. Dresevic, B. Hill, G. Hitchcock, B. Keely, D. P. Mitchell, J. C. Platt, and T. Whitted (2000). Displaced filtering for patterned displays. *Society for Information Display International Symposium. Digest of Technical Papers 31*, 296–299.

Bhate, N. and A. Tokuta (1992, May). Photorealistic volume rendering of media with directional scattering. In *Proceedings of the Third Eurographics Rendering Workshop*, pp. 227–245.

Blasi, P., B. L. Saẽc, and C. Schlick (1993). A rendering algorithm for discrete volume density objects. *12*(3), 201–210.

Blinn, J. F. (1978, August). Simulation of wrinkled surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, Volume 12, pp. 286–292.

Blinn, J. F. (1982a, July). A generalization of algebraic surface drawing. *ACM Transactions on Graphics 1*(3), 235–256.

Blinn, J. F. (1982b, July). Light reflection functions for simulation of clouds and dusty surfaces. *Computer Graphics 16*(3), 21–29.

Blinn, J. F. and M. E. Newell (1976). Texture and reflection in computer generated images. *Communications of the ACM 19*, 542–546.

Bolin, M. R. and G. W. Meyer (1998, July). A perceptually based adaptive sampling algorithm. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pp. 299–310.

Bracewell, R. N. (2000). *The Fourier Transform and its Applications*. McGraw-Hill.

Bronsvoort, W. F. and F. Klok (1985, October). Ray tracing generalized cylinders. *ACM Transactions on Graphics 4*(4), 291–303.

Buck, R. C. (1978). *Advanced Calculus*. New York, NY: McGraw–Hill.

Cabral, B., N. Max, and R. Springmeyer (1987, July). Bidirectional reflection functions from surface bump maps. In *Computer Graphics (SIGGRAPH '87 Proceedings),* Volume 21, pp. 273–281.

Calder, B., K. Chandra, S. John, and T. Austin (1998). Cache-conscious data placement. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose.

Cant, R. J. and P. A. Shrubsole (2000, July). Texture potential MIP mapping, a new high-quality texture antialiasing algorithm. *ACM Transactions on Graphics 19*(3), 164–184.

Cazals, F., G. Drettakis, and C. Puech (1995, August). Filtering, clustering and hierarchy construction: a new solution for ray-tracing complex scenes. *Computer Graphics Forum 14*(3), 371–382.

Chalmers, A., T. Davis, and E. Reinhard (Eds.) (2002). *Practical Parallel Rendering*. A. K. Peters.

Chandrasekar, S. (1960). *Radiative Transfer*. New York: Dover Publications. Originally published by Oxford University Press, 1950.

Chilimbi, T. M., B. Davidson, and J. R. Larus (1999). Cache-conscious structure definition. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 13–24.

Chilimbi, T. M., M. D. Hill, and J. R. Larus (1999). Cache-conscious structure layout. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 1–12.

Chiu, K., M. Herf, P. Shirley, S. Swamy, C. Wang, and K. Zimmerman (1993, May). Spatially nonuniform scaling functions for high contrast images. In *Graphics Interface '93*, Toronto, Ontario, Canada, pp. 245–253. Canadian Information Processing Society.

Christensen, P. H. (2003, July). Adjoints and importance in rendering: an overview. *IEEE Transactions on Visualization and Computer Graphics 9*(3), 329–340.

Christensen, P. H., D. M. Laur, J. Fong, W. L. Wooten, , and D. Batali (2003, September). Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. In *Computer Graphics Forum (Eurographics 2003 Conference Proceedings)*, pp. 543–552.

Chvolson, O. D. (1890). Grundzüge einer matematischen teorie der inneren diffusion des lichtes. *Izv. Peterburg. Academii Nauk 33*, 221–265.

Clark, J. H. (1976). Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*(10), 547–554.

Cleary, J. G., B. M. Wyvill, R. Vatti, and G. M. Birtwistle (1983, May). Design and analysis of a parallel ray tracing computer. In *Graphics Interface '83*, pp. 33–38.

Cohen, J., A. Varshney, D. Manocha, G. Turk, H. Weber, P. Agarwal, F. P. Brooks Jr., and W. Wright (1996, August). Simplification envelopes. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pp. 119–128.

Cohen, M. and J. Wallace (1993). *Radiosity and realistic image synthesis*. Academic Press Professional.

Collins, S. (1994, June). Adaptive splatting for specular to diffuse light transport. In *Fifth Eurographics Workshop on Rendering*, Darmstadt, Germany, pp. 119–135.

Cook, R. L. (1984, July). Shade trees. In H. Christiansen (Ed.), *Computer Graphics (SIGGRAPH '84 Proceedings)*, Volume 18, pp. 223–231.

Cook, R. L. (1986, January). Stochastic sampling in computer graphics. *ACM Transactions on Graphics 5*(1), 51–72.

Cook, R. L., L. Carpenter, and E. Catmull (1987, July). The REYES image rendering architecture. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, Anaheim, California, pp. 95–102.

Cook, R. L., T. Porter, and L. Carpenter (1984, July). Distributed ray tracing. In *Computer Graphics (SIGGRAPH '84 Proceedings)*, Volume 18, pp. 137–45.

Cook, R. L. and K. E. Torrance (1981, August). A reflectance model for computer graphics. In *Computer Graphics (SIGGRAPH '81 Proceedings)*, Volume 15, pp. 307–316.

Cook, R. L. and K. E. Torrance (1982, January). A reflectance model for computer graphics. *ACM Transactions on Graphics 1*(1), 7–24.

Crow, F. C. (1977, November). The aliasing problem in computer-generated shaded images. *Communications of the ACM 20*(11), 799–805.

Crow, F. C. (1984, July). Summed-area tables for texture mapping. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, Volume 18, pp. 207–212.

Dachsbacher, C. and M. Stamminger (2003). Translucent shadow maps. In *Proceedings of the 13th Eurographics workshop on Rendering*, pp. 197–201. Eurographics Association.

Dana, K. J., B. van Ginneken, S. K. Nayar, and J. J. Koenderink (1999, January). Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics 18*(1), 1–34.

de Berg, M., M. van Kreveld, M. Overmars, and O. Schwarzkopf (2000). *Computational Geometry: Algorithms and Applications*. Springer-Verlag. ISBN 3-540-65620-0.

de Voogt, E., A. van der Helm, and W. F. Bronsvoort (2000). Ray tracing deformed generalized cylinders. *The Visual Computer 16*(3-4), 197–207.

Debevec, P. (1998, July). Rendering synthetic objects into real scenes: Bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, Orlando, Florida, pp. 189–198. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-999-8.

Deering, M. F. (1995, August). Geometry compression. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pp. 13–20.

DeRose, T. D. (1989). A coordinate-free approach to geometric programming. Also available as Technical Report No. 89-09-16, Department of Computer Science and Engineering, University of Washington, Seattle, WA (September, 1989).

Deussen, O., P. M. Hanrahan, B. Lintermann, R. Mech, M. Pharr, and P. Prusinkiewicz (1998, July). Realistic modeling and rendering of plant ecosystems. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pp. 275–286.

Devlin, K., A. Chalmers, A. Wilkie, and W. Purgathofer (2002, September). Tone reproduction and physically based spectral rendering. In D. Fellner and R. Scopignio (Eds.), *Proceedings of Eurographics 2002*, pp. 101–123. The Eurographics Association.

Dippé, M. A. Z. and E. H. Wold (1985, July). Antialiasing through stochastic sampling. In B. A. Barsky (Ed.), *Computer Graphics (SIGGRAPH '85 Proceedings)*, Volume 19, pp. 69–78.

Dobkin, D. P., D. Eppstein, and D. P. Mitchell (1996, October). Computing the discrepancy with applications to supersampling patterns. *ACM Transactions on Graphics 15*(4), 354–376. ISSN 0730-0301.

Dobkin, D. P. and D. P. Mitchell (1993, May). Random-edge discrepancy of supersampling patterns. In *Graphics Interface '93*, Toronto, Ontario, pp. 62–69. Canadian Information Processing Society.

Dorsey, J., A. Edelman, J. Legakis, H. W. Jensen, and H. K. Pedersen (1999, August). Modeling and rendering of weathered stone. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pp. 225–234.

Dorsey, J., H. K. Pedersen, and P. M. Hanrahan (1996, August). Flow and changes in appearance. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pp. 411–420.

Drebin, R. A., L. Carpenter, and P. Hanrahan (1988, August). Volume rendering. In *Computer Graphics (Proceedings of SIGGRAPH 88)*, Volume 22, pp. 65–74.

Duff, T. (1985, July). Compositing 3-d rendered images. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, Volume 19, pp. 41–44.

Dungan Jr., W., A. Stenger, and G. Sutty (1978, August). Texture tile considerations for raster graphics. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, Volume 12, pp. 130–134.

Durand, F. and J. Dorsey (2000, June). Interactive tone mapping. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pp. 219–230. Eurographics. ISBN 3-211-83535-0.

Durand, F. and J. Dorsey (2002, July). Fast bilateral filtering for the display of high-dynamic-range images. *ACM Transactions on Graphics 21*(3), 257–266. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

Eberly, D. H. (2001). *3D game engine design: a practical approach to real-time computer graphics*. San Francisco, CA: Morgan Kaufmann.

Ebert, D., F. K. Musgrave, D. Peachey, K. Perlin, and S. Worley (2003). *Texturing and Modeling: A Procedural Approach*. San Francisco, CA: Morgan Kaufmann Publishers.

Fante, R. L. (1981, April). Relationship between radiative-transport theory and Maxwell's equations in dielectric media. *Journal of the Optical Society of America 71*(4), 460–468.

Fedkiw, R., J. Stam, and H. W. Jensen (2001, August). Visual simulation of smoke. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pp. 15–22.

Feibush, E. A., M. Levoy, and R. L. Cook (1980, July). Synthetic texturing using digital filters. In *Computer Graphics (Proceedings of SIGGRAPH 80)*, Volume 14, pp. 294–301.

Ferwerda, J. A. (2001, September-October). Elements of early vision for computer graphics. *IEEE Computer Graphics & Applications 21*(5), 22–33. ISSN 0272-1716.

Ferwerda, J. A., S. Pattanaik, P. S. Shirley, and D. P. Greenberg (1996, August). A model of visual adaptation for realistic image synthesis. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, New Orleans, Louisiana, pp. 249–258. ACM SIGGRAPH / Addison Wesley. ISBN 0-201-94800-1.

Ferwerda, J. A., S. N. Pattanaik, P. S. Shirley, and D. P. Greenberg (1997, August). A model of visual masking for computer graphics. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California, pp. 143–152. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-896-7.

Fishman, G. S. (1996). *Monte Carlo: Concepts, Algorithms, and Applications*. New York, NY: Springer Verlag.

Fleischer, K., D. Laidlaw, B. Currin, and A. H. Barr (1995, August). Cellular texture generation. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, pp. 239–248.

Foley, J. D., A. van Dam, S. K. Feiner, and J. F. Hughes (1990). *Computer Graphics: principles and practice*. Reading, Massachusetts: Addison-Wesley.

Fournier, A. (1992, May). Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pp. 45–52.

Fournier, A. and E. Fiume (1988, August). Constant-time filtering with space-variant kernels. In J. Dill (Ed.), *Computer Graphics (SIGGRAPH '88 Proceedings)*, Volume 22, pp. 229–238.

Fournier, A., D. Fussel, and L. Carpenter (1982, June). Computer rendering of stochastic models. *Comm. of the ACM 25*(6), 371–384.

Fraser, C. and D. Hanson (1995). *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley.

Friedel, I. and A. Keller (2000). Fast generation of randomized low discrepancy points sets. In *Monte Carlo and Quasi-Monte Carlo Methods 2000*, Berlin, pp. 257–273. Springer-Verlag.

Fujimoto, A., T. Tanaka, and K. Iwata (1986, April). Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications 6*(4), 16–26.

Gardner, G. Y. (1984, July). Simulation of natural scenes using textured quadric surfaces. In H. Christiansen (Ed.), *Computer Graphics (SIGGRAPH '84 Proceedings)*, Volume 18, pp. 11–20.

Gershbein, R. and P. M. Hanrahan (2000, July). A fast relighting engine for interactive cinematic lighting design. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pp. 353–358.

Gershun, A. (1939). The light field. *Journal of Mathematics and Physics 18*, 51–151.

Glassner, A. (1993, August). Spectrum: an architecture for image synthesis, research, education, and practice. In *Developing Large-Scale Graphics Software Toolkits, SIGGRAPH '93 Course Notes*, Volume 03, pp. 1–14–1–43.

Glassner, A. (1995). *Principles of Digital Image Synthesis*. Morgan Kaufmann Publishers.

Glassner, A. S. (1984, October). Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications 4*(10), 15–22.

Glassner, A. S. (1989, July). How to derive a spectrum from an RGB triplet. *IEEE Computer Graphics & Applications 9*(4), 95–99.

Goldman, D. B. (1997, August). Fake fur rendering. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pp. 127–134.

Goldman, R. (1985). Illicit expressions in vector algebra. *ACM Transactions on Graphics 4*(3), 223–243.

Goldsmith, J. and J. Salmon (1987, May). Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*(5), 14–20.

Goldstein, R. A. and R. Nagel (1971, January). 3-D visual simulation. *Simulation 16*(1), 25–31.

Gondek, J. S., G. W. Meyer, and J. G. Newman (1994, July). Wavelength dependent reflectance functions. In *Proceedings of SIGGRAPH ’94*, pp. 213–220. ACM Press.

Gortler, S. J., R. Grzeszczuk, R. Szeliski, and M. F. Cohen (1996, August). The lumigraph. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pp. 43–54.

Gray, A. (1993). *Modern differential geometry of curves and surfaces*. CRC Press.

Green, S. A. and D. J. Paddon (1989, November). Exploiting coherence for multiprocessor ray tracing. *IEEE Computer Graphics & Applications 9*(6), 12–26.

Greene, N. (1986, November). Environment mapping and other applications of world projections. *IEEE Computer Graphics & Applications 6*(11), 21–29.

Greene, N. and P. S. Heckbert (1986a, June). Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics & Applications 6*(6), 21–27.

Greene, N. and P. S. Heckbert (1986b, June). Creating raster omnimax images from multiple perspective views using the elliptical weighted average filter. *IEEE Computer Graphics and Applications 6*(6), 21–27.

Gritz, L. and J. K. Hahn (1996). BMRT: A global illumination implementation of the RenderMan standard. *Journal of Graphics Tools 1*(3), 29–47.

Grunwald, D., B. G. Zorn, and R. Henderson (1993). Improving the cache locality of memory allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pp. 177–186.

Haines, E. (1994). Point in polygon strategies. In P. Heckbert (Ed.), *Graphics Gems IV*, pp. 24–46. Academic Press.

Haines, E. A. and D. P. Greenberg (1986). The light buffer: a shadow testing accelerator. *IEEE Computer Graphics & Applications 6*(9), 6–16.

Haines, E. A. and J. R. Wallace (1994). Shaft culling for efficient ray-traced radiosity. In *Second Eurographics Workshop on Rendering (Photorealistic Rendering in ComputerGraphics)*.

Hakura, Z. S. and A. Gupta (1997, June). The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, Denver, Colorado, pp. 108–120.

Hall, R. (1989). *Illumination and Color in Computer Generated Imagery*. New York: Springer-Verlag.

Hall, R. A. and D. P. Greenberg (1983, November). A testbed for realistic image synthesis. *IEEE Computer Graphics & Applications 3*, 10–20.

Hanrahan, P. (1983, July). Ray tracing algebraic surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, Volume 17, pp. 83–90.

Hanrahan, P. (2002). Why is graphics hardware so fast?

Hanrahan, P. and W. Krueger (1993, August). Reflection from layered surfaces due to subsurface scattering. In *Computer Graphics (SIGGRAPH Proceedings)*, pp. 165–174.

Hanrahan, P. and J. Lawson (1990, August). A language for shading and lighting calculations. In F. Baskett (Ed.), *Computer Graphics (SIGGRAPH '90 Proceedings)*, Volume 24, pp. 289–298.

Hansen, J. E. and L. D. Travis (1974). Light scattering in planetary atmospheres. *Space Science Reviews 16*, 527–610.

Hao, X., T. Baby, and A. Varshney (2003). Interactive subsurface scattering for translucent meshes. In *ACM Symposium on Interactive 3D Graphics*, pp. 75–82.

Hart, D., P. Dutré, and D. P. Greenberg (1999, August). Direct illumination with lazy visibility evaluation. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pp. 147–154.

Hart, J. C. (1996). Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer 12*(9), 527–545.

Havran, V. and J. Bittner (2002, February). On improving kd-trees for ray shooting. In *Proceedings of WSCG'2002 conference*, pp. 209–217.

He, X. D., K. E. Torrance, F. X. Sillion, and D. P. Greenberg (1991, July). A comprehensive physical model for light reflection. In T. W. Sederberg (Ed.), *Computer Graphics (SIGGRAPH '91 Proceedings)*, Volume 25, pp. 175–186.

Heckbert, P. (1984, July). The mathematics of quadric surface rendering and SOID. 3-D Technical Memo.

Heckbert, P. S. (1986, November). Survey of texture mapping. *IEEE Computer Graphics and Applications 6*(11), 56–67.

Heckbert, P. S. (1989, June). Fundamentals of texture mapping and image warping. M.sc. thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Heckbert, P. S. (1990, August). Adaptive radiosity textures for bidirectional ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, Volume 24, pp. 145–154.

Heckbert, P. S. and P. Hanrahan (1984, July). Beam tracing polygonal objects. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, Volume 18, pp. 119–127.

Heidrich, W. and H.-P. Seidel (1998, June). Ray-tracing procedural displacement shaders. In *Graphics Interface '98*, pp. 8–16.

Heidrich, W., P. Slusallek, and H.-P. Seidel (1998, July). Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics 17*(3), 158–176.

Henyey, L. G. and J. L. Greenstein (1941). Diffuse radiation in the galaxy. *Astrophysical Journal 93*, 70–83.

Hery, C. (2003, July). Implementing a skin BSSRDF.

Hey, H. and W. Purgathofer (2002). Importance sampling with hemispherical particle footprints. In A. Chalmers (Ed.), *Proceedings of the 18th Spring Conference on Computer Graphics*.

Hiller, S., O. Deussen, and A. Keller (2001, November). Tiled blue noise samples. In T. Ertl, B. Girod, G.Greiner, H. Niemann, and H.-P. Seidel (Eds.), *Proceedings of Vision, MOdeling and Visualization*.

Hoffman, G. (2002, December). Windowed sinc interpolation.

Hoppe, H., T. DeRose, T. Duchamp, M. Halstead, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle (1994, July). Piecewise smooth surface reconstruction. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, Orlando, Florida, pp. 295–302. ACM SIGGRAPH / ACM Press. ISBN 0-89791-667-0.

Hurley, J., A. Kapustin, A. Reshetov, and A. Soupikov (2002, September). Fast ray tracing for modern general purpose CPU. In *Proceedgins of GraphiCon 2002*.

Igehy, H. (1999, August). Tracing ray differentials. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pp. 179–186.

Igehy, H., M. Eldridge, and P. Hanrahan (1999, August). Parallel texture caching. In *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 95–106.

Igehy, H., M. Eldridge, and K. Proudfoot (1998, August). Prefetching in a texture cache architecture. In *1998 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 133–142.

Immel, D. S., M. F. Cohen, and D. P. Greenberg (1986, August). A radiosity method for non-diffuse environments. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, Volume 20, pp. 133–142.

Jackson, W. H. (1910). The solution of an integral equation occurring in the theory of radiation. *Bulletin of the American Mathematical Society 16*, 473–475.

Jansen, F. W. (1986). Data structures for ray tracing. In L. R. A. Kessener, F. J. Peters, and M. L. P. Lierop (Eds.), *Data Structures for Raster Graphics, Workshop Proceedings*, pp. 57–73. Springer Verlag.

Jensen, H. W. (1995, June). Importance driven path tracing using the photon map. In *Eurographics Rendering Workshop 1995*, pp. 326–335.

Jensen, H. W. (1996, June). Global illumination using photon maps. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996*, New York City, NY, pp. 21–30. Eurographics: Springer Wien. ISBN 3-211-82883-4.

Jensen, H. W. (2001). *Realistic Image Synthesis Using Photon Mapping*. Natick, MA: A. K. Peters, Ltd.

Jensen, H. W. and J. Buhler (2002, July). A rapid hierarchical rendering technique for translucent materials. *ACM Transactions on Graphics 21*(3), 576–581.

Jensen, H. W. and P. H. Christensen (1998, July). Efficient simulation of light transport in scenes with participating media using photon maps. In M. Cohen (Ed.), *SIGGRAPH 98 Conference Proceedings*, Annual Conference Series, pp. 311–320. Addison Wesley.

Jensen, H. W., S. R. Marschner, M. Levoy, and P. Hanrahan (2001, August). A practical model for subsurface light transport. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pp. 511–518.

Jevans, D. and B. Wyvill (1989, June). Adaptive voxel subdivision for ray tracing. In *Graphics Interface '89*, pp. 164–172.

Johnstone, M. S. and P. R. Wilson (1999). The memory fragmentation problem: solved? *ACM SIGPLAN Notices 34*(3), 26–36.

Kajiya, J. and M. Ullner (1981, August). Filtering high quality text for display on raster scan devices. In *Computer Graphics (Proceedings of SIGGRAPH 81)*, pp. 7–15.

Kajiya, J. T. (1982, July). Ray tracing parametric patches. In *Computer Graphics (SIGGRAPH 1982 Conference Proceedings)*, pp. 245–254.

Kajiya, J. T. (1983, July). New techniques for ray tracing procedurally defined objects. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, Volume 17, pp. 91–102.

Kajiya, J. T. (1985, July). Anisotropic reflection models. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, Volume 19, pp. 15–21.

Kajiya, J. T. (1986, August). The rendering equation. In D. C. Evans and R. J. Athay (Eds.), *Computer Graphics (SIGGRAPH '86 Proceedings)*, Volume 20, pp. 143–150.

Kajiya, J. T. and B. P. V. Herzen (1984, July). Ray tracing volume densities. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, Volume 18, pp. 165–174.

Kajiya, J. T. and T. L. Kay (1989, July). Rendering fur with three dimensional textures. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, Volume 23, pp. 271–280.

Kalos, M. H. and P. A. Whitlock (1986). *Monte Carlo Methods: Volume I: Basics*. New York, NY: Wiley.

Kalra, D. and A. H. Barr (1989, July). Guaranteed ray intersections with implicit surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, Volume 23, pp. 297–306.

Kapasi, U. J., S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens (2003, August). Programmable stream processors. *IEEE Computer*, 54–62.

Kay, D. S. and D. P. Greenberg (1979, August). Transparency for computer synthesized images. In *Computer Graphics (SIGGRAPH '79 Proceedings)*, Volume 13, pp. 158–164.

Kay, T. and J. Kajiya (1986, August). Ray tracing complex scenes. In *Computer Graphics (SIGGRAPH '86 Proceedings)*, Volume 20, pp. 269–278.

Keller, A. (1996, June). Quasi-Monte Carlo radiosity. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996*, New York City, NY, pp. 101–110. Eurographics: Springer Wien.

Keller, A. (1997, August). Instant radiosity. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California, pp. 49–56. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-896-7.

Keller, A. (2001). Strictly deterministic sampling methods in computer graphics. mental images technical report, also in SIGGRAPH 2003 Monte Carlo Course Notes.

Keller, A. and I. Wald (2000). Efficient importance sampling techniques for the photon map. In *Proceedings of Vision, Modeling and Visualization 2000*, pp. 271–279.

King, L. V. (1913). On the scattering and absorption of light in gaseous media, with applications to the intensity of sky radiation. *Philosophical Transactions of the Royal Society of London. Series A. Mathematical and Physical Sciences 212*, 375–433.

Kirk, D. (2002). Gpu talk.

Kirk, D. and J. Arvo (1988, July). The ray tracing kernel. In *Proceedings of Ausgraph '88*, pp. 75–82.

Kirk, D. B. and J. Arvo (1991, July). Unbiased sampling techniques for image synthesis. In T. W. Sederberg (Ed.), *Computer Graphics (SIGGRAPH '91 Proceedings)*, Volume 25, pp. 153–156.

Klassen, R. V. (1987, July). Modeling the effect of the atmosphere on light. *ACM Transactions on Graphics 6*(3), 215–237.

Klimaszewski, K. S. and T. W. Sederberg (1997, January). Faster ray tracing using adaptive grids. *IEEE Computer Graphics and Applications 17*(1), 42–51.

Knuth, D. E. (1984). Literate programming. *The Computer Journal 27*, 97–111. Reprinted in Donald E. Knuth, *Literate Programming*, Stanford Center for the Study of Language and Information, 1992.

Knuth, D. E. (1986). *MetaFont: The Program*. Reading, Massachusetts: Adisson-Wesley.

Knuth, D. E. (1993a). *TEX: The Program*. Reading, Massachusetts: Adisson–Wesley.

Knuth, D. E. (1993b). *The Stanford GraphBase*. New York, NY: ACM Press and Addison–Wesley.

Kolb, C., P. Hanrahan, and D. Mitchell (1995, August). A realistic camera model for computer graphics. In R. Cook (Ed.), *SIGGRAPH 95 Conference Proceedings*, Annual Conference Series, pp. 317–324. Addison Wesley.

Kollig, T. and A. Keller (2000). Efficient bidirectional path tracing by randomized Quasi-Monte Carlo integration. In *Monte Carlo and Quasi-Monte Carlo Methods 2000*, Berlin, pp. 290–305. Springer-Verlag.

Kollig, T. and A. Keller (2002). Efficient multidimensional sampling. In G. Drettakis and H.-P. Seidel (Eds.), *Computer Graphics Forum*, Volume 21, pp. 557–563.

Lafortune, E. and Y. Willems (1994). A theoretical framework for physically based rendering. *Computer Graphics Forum 13*(2), 97–107.

Lafortune, E. P. and Y. D. Willems (1996, June). Rendering participating media with bidirectional path tracing. In *Eurographics Rendering Workshop 1996*, pp. 91–100.

Lafortune, E. P. F., S.-C. Foo, K. E. Torrance, and D. P. Greenberg (1997, August). Non-linear approximation of reflectance functions. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California, pp. 117–126. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-896-7.

Lam, M. S., E. E. Rothberg, and M. E. Wolf (1991). The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, Palo Alto, CA.

Lambert, J. H. (2001). *Photometry, or, on the measure and gradations of light, colors, and shade*. The Illuminating Engineering Society of North America. Transated by David L. DiLaura.

Lang, S. (1986). *An Introduction to Linear Algebra*. New York, NY: Springer Verlag.

Lansdale, R. C. (1991, January). Texture mapping and resampling for computer graphics. M.sc. thesis, Department of Electrical Engineering, University of Toronto.

Larson, G. W. (1998). Logluv encoding for full-gamut, high-dynamic range images. *Journal of Graphics Tools 3*(1), 15–31.

Larson, G. W., H. Rushmeier, and C. Piatko (1997, October - December). A visibility matching tone reproduction operator for high dynamic range scenes. *IEEE Transactions on Visualization and Computer Graphics 3*(4), 291–306. ISSN 1077-2626.

Larson, G. W. and R. A. Shakespeare (1998). *Rendering with Radiance: The Art and Science of Lighting Visualization*. Morgan Kaufmann Publishers.

Lee, M. E., R. A. Redner, and S. P. Uselton (1985, July). Statistically optimized sampling for distributed ray tracing. In *Computer Graphics (Proceedings of SIGGRAPH 85)*, Volume 19, pp. 61–67.

Levoy, M. (1988, May). Display of surfaces from volume data. *IEEE Computer Graphics & Applications 8*(3), 29–37.

Levoy, M. (1990a, July). Efficient ray tracing of volume data. *ACM Transactions on Graphics 9*(3), 245–261.

Levoy, M. (1990b, March). A hybrid ray tracer for rendering polygon and volume data. *IEEE Computer Graphics & Applications 10*(2), 33–40.

Levoy, M. and P. M. Hanrahan (1996, August). Light field rendering. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pp. 31–42.

Levoy, M. and T. Whitted (1995, January). The use of points as a display primitive. Technical Report 85-022, Computer Science Department, University of North Carolina at Chapel Hill.

Li, X., W. Wang, R. R. Martin, and A. Bowyer (2003). Using low-discrepancy sequences and the crofton formula to compute surface areas of geometric models. *Computer Aided Design 35*(9), 771–782.

Liu, J. S. (2001). *Monte Carlo strategies in scientific computing*. New York, NY: Springer-Verlag.

Logie, J. R. and J. W. Patterson (1994, December). Inverse displacement mapping in the general case. *Computer Graphics Forum 14*(5), 261–273.

Lokovic, T. and E. Veach (2000, July). Deep shadow maps. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pp. 385–392.

Lommel, E. (1889). Die Photometrie der diffusen Zurückwerfung. *Annalen der Physik 36*, 473–502.

Loop, C. (1987). *Smooth subdivision surfaces based on triangles*. Ph. D. thesis, University of Utah.

Lu, R., J. J. Koenderink, and A. M. L. Kappers (1999). Specularities on surfaces with tangential hairs or grooves. *Computer Vision and Image Understanding 78*, 320–335.

Lukaszewski, A. (2001). Exploiting coherence of shadow rays. In *AFRIGRAPH 2001*, pp. 147–150. ACM SIGGRAPH.

MacDonald, J. D. and K. S. Booth (1990, June). Heuristics for ray tracing using space subdivision. *The Visual Computer 6*(3), 153–166.

Malacara, D. (2002). *Color vision and colorimetry: theory and applications*. SPIE–The International Society for Optical Engineering.

Mann, S., N. Litke, and T. DeRose (1997, June). A coordinate free geometry ADT. Research Report CS-97-15, Computer Science Department, University of Waterloo. Available at: ftp://cs-archive.uwaterloo.ca/cs-archive/CS-97-15/.

Marschner, S. R., H. W. Jensen, M. Cammarano, S. Worley, and P. Hanrahan (2003, July). Light scattering from human hair fibers. *ACM Transactions on Graphics 22*(3), 780–791.

Marschner, S. R. and R. J. Lobb (1994, October). An evaluation of reconstruction filters for volume rendering. In *Proceedings of Visualization '94*, Washington, DC, pp. 100–107.

Marschner, S. R., S. H. Westin, E. P. F. Lafortune, K. E. Torrance, and D. P. Greenberg (1999, June). Image-based BRDF measurement including human skin. In *Eurographics Rendering Workshop 1999*, Granada, Spain. Springer Wein / Eurographics.

Matsumoto, M. and T. Nishimura (1998, January). Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Transactions on Modeling and Computer Simulation 8*(1), 3–30.

Max, N. (1995, June). Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*(2), 99–108.

Max, N. L. (1986, August). Atmospheric illumination and shadows. In *Computer Graphics (Proceedings of SIGGRAPH 86)*, Volume 20, pp. 117–124.

McCluney, W. R. (1994). *Introduction to radiometry and photometry*. Artech House.

McCool, M. and E. Fiume (1992). Hierarchical Poisson disk sampling distributions. In *Proceedings of Graphics Interface 1992*, pp. 94–105.

McCormack, J., R. Perry, K. I. Farkas, and N. P. Jouppi (1999, August). Feline: Fast elliptical lines for anisotropic texture mapping. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California, pp. 243–250. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.

Meijering, E. (2002, March). A chronology of interpolation: from ancient astronomy to modern signal and image processing. *Proceedings of the IEEE 90*(3), 319–342.

Meijering, E. H. W., W. J. Niessen, J. P. W. Pluim, and M. A. Viergever (1999). Quantitative comparison of sinc-approximating kernels for medial image interpolation. In C. Taylor and A. Colchester (Eds.), *Medical Image Computing and Computer-Assisted Intervention – MICCAI 1999*, Berlin, pp. 210–217. Springer-Verlag. Vol 1679 of Lecture Notes in Computer Science.

Miller, G. S. and C. R. Hoffman (1984). Illumination and reflection maps: Simulated objects in simulated and real environments.

Mitchell, D. P. (1987, July). Generating antialiased images at low sampling densities. In M. C. Stone (Ed.), *Computer Graphics (SIGGRAPH '87 Proceedings)*, Volume 21, pp. 65–72.

Mitchell, D. P. (1991, July). Spectrally optimal sampling for distributed ray tracing. In T. W. Sederberg (Ed.), *Computer Graphics (SIGGRAPH '91 Proceedings)*, Volume 25, pp. 157–164.

Mitchell, D. P. (1992, May). Ray tracing and irregularities of distribution. In *Third Eurographics Workshop on Rendering*, Bristol, UK, pp. 61–69.

Mitchell, D. P. (1996, August). Consequences of stratified sampling in graphics. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, New Orleans, Louisiana, pp. 277–280. ACM SIGGRAPH / Addison Wesley. ISBN 0-201-94800-1.

Mitchell, D. P. and P. Hanrahan (1992, July). Illumination from curved reflectors. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, Volume 26, pp. 283–291.

Mitchell, D. P. and A. N. Netravali (1988, August). Reconstruction filters in computer graphics. In J. Dill (Ed.), *Computer Graphics (SIGGRAPH '88 Proceedings)*, Volume 22, pp. 221–228.

Möller, T. and B. Trumbore (1997). Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools 2*(1), 21–28.

Moon, P. and D. E. Spencer (1936). *The Scientific Basis of Illuminating Engineering*. New York, NY: McGraw-Hill.

Moon, P. and D. E. Spencer (1948). *Lighting Design*. Reading, MA: Addison-Wesley.

Motwani, R. and P. Raghavan (1995). *Randomized Algorithms*. Cambridge University Press.

Musgrave, K. (1992). A panoramic virtual screen for ray tracing. In D. Kirk (Ed.), *Graphics Gems III*, pp. 288–. Academic Press.

Nakamae, E., K. Kaneda, T. Okamoto, and T. Nishita (1990, August). A lighting model aiming at drive simulators. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, Volume 24, pp. 395–404.

Nayar, S. K., K. Ikeuchi, and T. Kanade (1991, July). Surface reflection: Physical and geometrical perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence 17*(7), 611–634.

Naylor, B. (1993, May). Constructing good partition trees. In *Graphics Interface '93*, pp. 181–191.

Niederreiter, H. (1992). *Random number generation and quasi-Monte Carlo methods*. Philadelpha, Pennsylvania: Society for Industrial and Applied Mathematics.

Nishita, T., Y. Miyawaki, and E. Nakamae (1987, July). A shading model for atmospheric scattering considering luminous intensity distribution of light sources. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, Volume 21, pp. 303–310.

Norton, A., A. P. Rockwood, and P. T. Skolmoski (1982, July). Clamping: a method of antialiasing textured surfaces by bandwidth limiting in object space. In *Computer Graphics (Proceedings of SIGGRAPH 82)*, Volume 16, pp. 1–8.

Oren, M. and S. K. Nayar (1994, July). Generalization of Lambert's reflectance model. In A. Glassner (Ed.), *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pp. 239–246. ACM Press.

Owens, J. D. (2002, November). *Computer Graphics on a Stream Architecture*. Ph. D. thesis, Stanford University.

Owens, J. D., W. J. Dally, U. J. Kapasi, S. Rixner, P. Mattson, and B. Mowery (2000, August). Polygon rendering on a stream architecture. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 23–32.

Owens, J. D., B. Khailany, B. Towles, and W. J. Dally (2002, September). Comparing REYES and OpenGL on a stream architecture. In *2002 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pp. 47–56.

Parker, S., W. Martin, P.-P. J. Sloan, P. S. Shirley, B. Smits, and C. Hansen (1999, April). Interactive ray tracing. In *1999 ACM Symposium on Interactive 3D Graphics*, pp. 119–126.

Pattanaik, S. N., J. A. Ferwerda, M. D. Fairchild, and D. P. Greenberg (1998, July). A multiscale model of adaptation and spatial vision for realistic image display. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, Orlando, Florida, pp. 287–298. ACM SIGGRAPH / Addison Wesley. ISBN 0-89791-999-8.

Pattanaik, S. N., J. E. Tumblin, H. Yee, and D. P. Greenberg (2000, July). Time-dependent visual adaptation for realistic image display. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pp. 47–54. ACM Press / ACM SIGGRAPH / Addison Wesley Longman. ISBN 1-58113-208-5.

Patterson, J. W., S. G. Hoggar, and J. R. Logie (1991, June). Inverse displacement mapping. *Computer Graphics Forum 10*(2), 129–139.

Pauly, M., T. Kollig, and A. Keller (2000, June). Metropolis light transport for participating media. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pp. 11–22. Eurographics. ISBN 3-211-83535-0.

Peachey, D. R. (1985, July). Solid texturing of complex surfaces. In B. A. Barsky (Ed.), *Computer Graphics (SIGGRAPH ’85 Proceedings)*, Volume 19, pp. 279–286.

Peachey, D. R. (1990). Texture on demand. unpublished manuscript.

Pearce, A. (1991). A recursive shadow voxel cache for ray tracing. In J. Arvo (Ed.), *Graphics Gems II*, pp. 273–274. Academic Press.

Peercy, M. S. (1993, August). Linear color representations for full spectral rendering. In J. T. Kajiya (Ed.), *Computer Graphics (SIGGRAPH ’93 Proceedings)*, Volume 27, pp. 191–198.

Pérez, F., X. Pueyo, and F. X. Sillion (1997, June). Global illumination techniques for the simulation of participating media. In *Eurographics Rendering Workshop 1997*, pp. 309–320.

Perlin, K. (1985, July). An image synthesizer. In *Computer Graphics (SIGGRAPH ’85 Proceedings)*, Volume 19, pp. 287–296.

Perlin, K. (2002, July). Improving noise. *ACM Transactions on Graphics 21*(3), 681–682. ISSN 0730-0301 (Proceedings of ACM SIGGRAPH 2002).

Perlin, K. and E. M. Hoffert (1989, July). Hypertexture. In *Computer Graphics (Proceedings of SIGGRAPH 89)*, Volume 23, pp. 253–262.

Peter, I. and G. Pietrek (1998, June). Importance driven construction of photon maps. In *Eurographics Rendering Workshop 1998*, pp. 269–280.

Pfister, H., M. Zwicker, J. van Baar, and M. Gross (2000, July). Surfels: Surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pp. 335–342.

Pharr, M. and P. Hanrahan (1996, June). Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996*, pp. 31–40.

Pharr, M. and P. M. Hanrahan (2000, July). Monte Carlo evaluation of non-linear scattering equations for subsurface reflection. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pp. 75–84.

Pharr, M., C. Kolb, R. Gershbein, and P. M. Hanrahan (1997, August). Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of SIGGRAPH 97*, Computer Graphics Proceedings, Annual Conference Series, pp. 101–108.

Phong, B.-T. (1975, June). Illumination for computer generated pictures. *Communications of the ACM 18*(6), 311–317.

Phong, B.-T. and F. C. Crow (1975). Improved rendition of polygonal models of curved surfaces. In *Proceedings of the 2nd USA-Japan Computer Conference*.

Pixar Animation Studios (1989, September). The RenderMan interface. With typographical corrections through May 1995.

Porter, T. and T. Duff (1984, July). Compositing digital images. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, Volume 18, Minneapolis, Minnesota, pp. 253–259.

Potmesil, M. and I. Chakravarty (1981, August). A lens and aperture camera model for synthetic image generation. In *Computer Graphics (Proceedings of SIGGRAPH 81)*, Volume 15, Dallas, Texas, pp. 297–305.

Potmesil, M. and I. Chakravarty (1982, April). Synthetic image generation with a lens and aperture camera model. *ACM Transactions on Graphics 1*(2), 85–108.

Potmesil, M. and I. Chakravarty (1983, July). Modeling motion blur in computer-generated images. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, Volume 17, Detroit, Michigan, pp. 389–399.

Poulin, P. and A. Fournier (1990, August). A model for anisotropic reflection. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, Volume 24, pp. 273–282.

Poynton, C. (2002a). Frequently-asked questions about color. http://www.inforamp.net/˜poynton/ColorFAQ.html.

Poynton, C. (2002b). Frequently-asked questions about gamma. http://www.inforamp.net/˜poynton/GammaFAQ.html.

Preetham, A. J., P. S. Shirley, and B. E. Smits (1999, August). A practical analytic model for daylight. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pp. 91–100.

Preisendorfer, R. W. (1965). *Radiative Transfer on Discrete Spaces*. Oxford: Pergamon Press.

Preisendorfer, R. W. (1976). *Hydrologic Optics*. Honolulu, Hawaii: U.S. Department of Commerce, National Oceanic and Atmospheric Administration. Six volumes.

Press, W. H., S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery (1992). *Numerical Recipes in C: The Art of Scientific Computing (2nd ed.)*. Cambridge: Cambridge University Press.

Prusinkiewicz, P. (1986, May). Graphical applications of L-systems. In *Graphics Interface ’86*, pp. 247–253.

Prusinkiewicz, P., M. James, and R. Mech (1994, July). Synthetic topiary. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pp. 351–358.

Prusinkiewicz, P., L. Mündermann, R. Karwowski, and B. Lane (2001, August). The use of positional information in the modeling of plants. In *Proceedings of ACM SIGGRAPH 2001*, Computer Graphics Proceedings, Annual Conference Series, pp. 289–300.

Purcell, T. J., I. Buck, W. R. Mark, and P. Hanrahan (2002, July). Ray tracing on programmable graphics hardware. *ACM Transactions on Graphics 21*(3), 703–712.

Purcell, T. J., C. Donner, M. Cammarano, H. W. Jensen, and P. Hanrahan (2003, July). Photon mapping on programmable graphics hardware. In *Graphics Hardware 2003*, pp. 41–50.

Ramasubramanian, M., S. N. Pattanaik, and D. P. Greenberg (1999, August). A perceptually based physical error metric for realistic image synthesis. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California, pp. 73–82. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.

Raso, M. and A. Fournier (1991, June). A piecewise polynomial approach to shading using spectral distributions. In *Graphics Interface ’91*, pp. 40–46. Canadian Information Processing Society.

Reeves, W. T., D. H. Salesin, and R. L. Cook (1987, July). Rendering antialiased shadows with depth maps. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, Volume 21, pp. 283–291.

Reichert, M. C. (1992, January). A two-pass radiosity method driven by lights and viewer position. Master’s thesis, Cornell University.

Reinhard, E. (2002). Parameter estimation for photographic tone reproduction. *Journal of Graphics Tools 7*(1), 45–52.

Reinhard, E., M. Stark, P. Shirley, and J. Ferwerda (2002). Photographic tone reproduction for digital images. *ACM Transactions on Graphics 21*(3), 267–276. Proceedings of ACM SIGGRAPH 2002.

Rogers, D. F. and J. A. Adams (1990). *Mathematical elements for computer graphics*. New York, NY: McGraw–Hill.

Ross, S. M. (2002). *Introduction to Probability Models* (8 ed.). Academic Press.

Rougeron, G. and B. Péroche (1998). Color fidelity in computer graphics: A survey. *Computer Graphics Forum 17*(1), 3–16. ISSN 1067-7055.

Rubin, S. M. and T. Whitted (1980, July). A 3-dimensional representation for fast rendering of complex scenes. *Computer Graphics 14*(3), 110–116.

Rushmeier, H. E. (1988). *Realistic Image Synthesis for Scenes with Radiatively Participating Media*. Ph.d. thesis, Cornell University.

Rushmeier, H. E. and K. E. Torrance (1987, July). The zonal method for calculating light intensities in the presence of a participating medium. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, Volume 21, pp. 293–302.

Rushmeier, H. E. and G. J. Ward (1994, July). Energy preserving non-linear filters. In *Proceedings of SIGGRAPH 94*, Computer Graphics Proceedings, Annual Conference Series, pp. 131–138.

Rusinkiewicz, S. and M. Levoy (2000, July). Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, Computer Graphics Proceedings, Annual Conference Series, pp. 343–352.

Saito, T. and T. Takahashi (1990, August). Comprehensible rendering of 3-d shapes. In *Computer Graphics (Proceedings of SIGGRAPH 90)*, Volume 24, pp. 197–206.

Samet, H. (1990). *The Design and Analysis of Spatial Data Structures*. Addison-Wesley. ISBN 0-201-50255-0.

Schaufler, G. and H. W. Jensen (2000, June). Ray tracing point sampled geometry. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pp. 319–328.

Schlick, C. (1993, June). A customizable reflectance model for everyday rendering. In *Fourth Eurographics Workshop on Rendering*, held in Paris, France, 14-16 June 1993, pp. 73–84. Eurographics.

Schneider, P. J. and D. H. Eberly (2003). *Geometric tools for computer graphics*. San Francisco, CA: Morgan Kaufmann Publishers.

Schuster, A. (1905, January). Radiation through a foggy atmosphere. *Astrophysical Journal 21*(1), 1–22.

Shade, J., S. J. Gortler, L. wei He, and R. Szeliski (1998, July). Layered depth images. In *Proceedings of SIGGRAPH 98*, Computer Graphics Proceedings, Annual Conference Series, pp. 231–242.

Shinya, M., T. Takahashi, and S. Naito (1987, July). Principles and applications of pencil tracing. In *Computer Graphics (Proceedings of SIGGRAPH 87)*, Volume 21, pp. 45–54.

Shirley, P. (1990a, November). *Physically Based Lighting Calculations for Computer Graphics*. Ph.D. thesis, Dept. of Computer Science, U. of Illinois, Urbana-Champaign.

Shirley, P. (1990b, May). A ray tracing method for illumination calculation in diffuse-specular scenes. In *Proceedings of Graphics Interface '90*, pp. 205–212.

Shirley, P. (1991, September). Discrepancy as a quality measure for sample distributions. In W. Purgathofer (Ed.), *Eurographics '91*, pp. 183–194. North-Holland.

Shirley, P. (1992). Nonuniform random point sets via warping. In D. Kirk (Ed.), *Graphics Gems III*, pp. 80–83. Academic Press.

Shirley, P. and K. Chiu (1997). A low distortion map between disk and square. *Journal of Graphics Tools 2*(3), 45–52. ISSN 1086-7651.

Shirley, P., C. Y. Wang, and K. Zimmerman (1996, January). Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics 15*(1), 1–36. ISSN 0730-0301.

Sillion, F. and C. Puech (1994). *Radiosity and Global Illumination*. San Francisco: Morgan Kaufmann Publishers. ISBN 1-55860-277-1.

Sims, K. (1991, July). Artificial evolution for computer graphics. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, Volume 25, pp. 319–328.

Sloan, P.-P., J. Hall, J. Hart, and J. Snyder (2003, July). Clustered principal components for precomputed radiance transfer. *ACM Transactions on Graphics 22*(3), 382–391.

Sloan, P.-P., J. Kautz, and J. Snyder (2002, July). Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Transactions on Graphics 21*(3), 527–536.

Sloan, P.-P., X. Liu, H.-Y. Shum, and J. Snyder (2003, July). Bi-scale radiance transfer. *ACM Transactions on Graphics 22*(3), 370–375.

Slusallek, P. (1996, May). *Vision - an architecture for physically-based rendering*. Ph. D. thesis, University of Erlangen.

Slusallek, P. and H.-P. Seidel (1996, June). Towards an open rendering kernel for image synthesis. In *Eurographics Rendering Workshop 1996*, pp. 51–60.

Slusallek, P. and H.-P. Siedel (1995, March). Vision - an architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics 1*(1), 77–96.

Smith, A. R. (1979). Painting tutorial notes.

Smith, A. R. (1984, July). Plants, fractals and formal languages. In *Computer Graphics (Proceedings of SIGGRAPH 84)*, Volume 18, pp. 1–10.

Smith, A. R. (1995, July). A pixel is not a little square, a pixel is not a little square, a pixel is not a little square! (and a voxel is not a little cube). Microsoft Tech Memo 6, http://www.alvyray.com.

Smits, B. (1998). Efficiency issues for ray tracing. *Journal of Graphics Tools 3*(2), 1–14.

Smits, B., P. S. Shirley, and M. M. Stark (2000, June). Direct ray tracing of displacement mapped triangles. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pp. 307–318.

Snyder, J. M. (1992). *Generative Modeling for Computer Graphics and CAD: Symbolic Shape Design Using Interval Analysis*. Academic Press.

Snyder, J. M. and A. H. Barr (1987, July). Ray tracing complex models containing surface tessellations. In M. C. Stone (Ed.), *Computer Graphics (SIGGRAPH '87 Proceedings)*, Volume 21, pp. 119–128.

Snyder, J. M. and J. T. Kajiya (1992, July). Generative modeling: A symbolic system for geometric modeling. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, Volume 26, pp. 369–378.

Spanier, J. and E. M. Gelbard (1969). *Monte Carlo principles and neutron transport problems*. Reading, Massachusetts: Addison–Wesley.

Spencer, G., P. S. Shirley, K. Zimmerman, and D. P. Greenberg (1995, August). Physically-based glare effects for digital images. In *Proceedings of SIGGRAPH 95*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California, pp. 325–334. ACM SIGGRAPH / Addison Wesley. ISBN 0-201-84776-0.

Stam, J. (1999, August). Diffraction shaders. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pp. 101–110.

Stolfi, J. (1991). *Oriented Projective Geometry*. San Diego, CA: Academic Press.

Stroustrup, B. (1997). *The C++ Programming Language*. Addison-Wesley.

Stürzlinger, W. (1998, July). Ray tracing triangular trimmed free-form surfaces. *IEEE Transactions on Visualization and Computer Graphics 4*(3), 202–214.

Sun, Y., F. D. Fracchia, M. S. Drew, and T. W. Calvert (2001). A spectrally based framework for realistic image synthesis. *The Visual Computer 17*(7), 429–444. ISSN 0178-2789.

Sung, K., J. Craighead, C. Wang, S. Bakshi, A. Pearce, and A. Woo (1998, October). Design and implementation of the Maya renderer. In *Pacific Graphics '98*.

Sung, K. and P. Shirley (1992). Ray tracing with the BSP tree. In D. Kirk (Ed.), *Graphics Gems III*, pp. 271–274. Academic Press.

Suykens, F. and Y. Willems (2001, June). Path differentials and applications. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pp. 257–268.

Suykens, F. and Y. Willens (2000, June). Density control for photon maps. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pp. 23–34.

Torrance, K. E. and E. M. Sparrow (1967). Theory for off-specular reflection from roughened surfaces. *Journal of the Optical Society of America 57*(9), 1105–1114.

Trumbore, B., W. Lytle, and D. P. Greenbert (1993, August). A testbed for image synthesis. In *Developing Large-Scale Graphics Software Toolkits, SIGGRAPH '93 Course Notes*, Volume 03, pp. 4–7–4–19.

Truong, D. N., F. Bodin, and A. Seznec (1998). Improving cache behavior of dynamically allocated data structures. In *IEEE PACT*, pp. 322–329.

Tumblin, J., J. K. Hodgins, and B. K. Guenter (1999, January). Two methods for display of high contrast images. *ACM Transactions on Graphics 18*(1), 56–94. ISSN 0730-0301.

Tumblin, J. and H. E. Rushmeier (1993, November). Tone reproduction for realistic images. *IEEE Computer Graphics & Applications 13*(6), 42–48.

Tumblin, J. and G. Turk (1999, August). LCIS: A boundary hierarchy for detail-preserving contrast reduction. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, Los Angeles, California, pp. 83–90. ACM SIGGRAPH / Addison Wesley Longman. ISBN 0-20148-560-5.

Turk, G. (1991, July). Generating textures for arbitrary surfaces using reaction-diffusion. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, Volume 25, pp. 289–298.

Turkowski, K. (1990a). The differential geometry of parametric primitives.

Turkowski, K. (1990b). Filters for common resampling tasks. In A. S. Glassner (Ed.), *Graphics Gems I*, pp. 147–165. Academic Press.

Turkowski, K. (1990c). Properties of surface-normal transformations. In A. S. Glassner (Ed.), *Graphics Gems I*, pp. 539–547. Academic Press.

Turkowski, K. (1993). The differential geometry of texture-mapping and shading.

Upstill, S. (1989). *The RenderMan Companion*. Reading, Massachusetts: Addison–Wesley.

van de Hulst, H. C. (1980). *Multiple Light Scattering*. New York: Academic Press. Two volumes.

van de Hulst, H. C. (1981). *Light Scattering by Small Particles*. New York: Dover Publications. Originally published by John Wiley and Sons, 1957.

Veach, E. (1996, June). Non-symmetric scattering in light transport algorithms. In X. Pueyo and P. Schröder (Eds.), *Eurographics Rendering Workshop 1996*. Springer Wien.

Veach, E. (1997, December). *Robust Monte Carlo Methods for Light Transport Simulation*. Ph. D. thesis, Stanford University.

Veach, E. and L. Guibas (1994, June). Bidirectional estimators for light transport. In *Fifth Eurographics Workshop on Rendering*, Darmstadt, Germany, pp. 147–162.

Veach, E. and L. J. Guibas (1995, August). Optimally combining sampling techniques for Monte Carlo rendering. In *Computer Graphics (SIGGRAPH Proceedings)*, pp. 419–428.

Veach, E. and L. J. Guibas (1997, August). Metropolis light transport. In *Computer Graphics (SIGGRAPH Proceedings)*, pp. 65–76.

Wald, I., C. Benthin, and P. Slusallek (2003, June). Interactive global illumination in complex and highly occluded environments. In *Eurographics Symposium on Rendering: 14th Eurographics Workshop on Rendering*, pp. 74–81.

Wald, I., T. Kollig, C. Benthin, A. Keller, and P. Slusallek (2002, June). Interactive global illumination using fast ray tracing. In *Rendering Techniques 2002: 13th Eurographics Workshop on Rendering*, pp. 15–24.

Wald, I., P. Slusallek, and C. Benthin (2001, June). Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001: 12th Eurographics Workshop on Rendering*, pp. 277–288.

Wald, I., P. Slusallek, C. Benthin, and M. Wagner (2001). Interactive rendering with coherent ray tracing. *Computer Graphics Forum 20*(3), 153–164.

Wallace, B. A. (1981, August). Merging and transformation of raster images for cartoon animation. In *Proceedings of ACM SIGGRAPH '81*, Volume 15, pp. 253–262.

Wallis, B. (1990). Forms, vectors, and transforms. In A. S. Glassner (Ed.), *Graphics Gems I*, pp. 533–538. Academic Press.

Wandell, B. (1995). *Foundations of vision*. Sinauer Associates.

Wang, X. C., J. Maillot, E. L. Fiume, V. Ng-Thow-Hing, A. Woo, and S. Bakshi (2000, June). Feature-based displacement mapping. In *Rendering Techniques 2000: 11th Eurographics Workshop on Rendering*, pp. 257–268.

Ward, G. (1991). Real pixels. In J. Arvo (Ed.), *Graphics Gems II*, pp. 80–83. Academic Press.

Ward, G. (1994a). A contrast-based scalefactor for luminance display. In P. Heckbert (Ed.), *Graphics Gems IV*, pp. 415–421. Boston: Academic Press. ISBN 0-12-336155-9.

Ward, G. J. (1992, July). Measuring and modeling anisotropic reflection. In E. E. Catmull (Ed.), *Computer Graphics (SIGGRAPH '92 Proceedings)*, Volume 26, pp. 265–272.

Ward, G. J. (1994b, July). The Radiance lighting simulation and rendering system. In A. Glassner (Ed.), *Proceedings of SIGGRAPH '94*, pp. 459–472.

Ward, G. J. and P. Heckbert (1992, May). Irradiance gradients. In *Third Eurographics Workshop on Rendering*, Bristol, UK, pp. 85–98.

Ward, G. J., F. M. Rubinstein, and R. D. Clear (1988, August). A ray tracing solution for diffuse interreflection. In J. Dill (Ed.), *Computer Graphics (SIGGRAPH '88 Proceedings)*, Volume 22, pp. 85–92.

Warn, D. R. (1983, July). Lighting controls for synthetic images. In *Computer Graphics (Proceedings of SIGGRAPH 83)*, Volume 17, Detroit, Michigan, pp. 13–21.

Watt, A. and M. Watt (1992). *Advanced Animation and Rendering Techniques*. New York, NY: Addison–Wesley.

Weghorst, H., G. Hooper, and D. P. Greenberg (1984, January). Improved computational methods for ray tracing. *ACM Transactions on Graphics 3*(1), 52–69.

Weisstein, E. (1999). Hypersphere.

Wells, D. (1987). *The Penguin Dictionary of Curious and Interesting Numbers*. Penguin USA.

Westin, S., J. Arvo, and K. Torrance (1992, July). Predicting reflectance functions from complex surfaces. *Computer Graphics 26*(2), 255–264.

Whitted, T. (1980, June). An improved illumination model for shaded display. *Communications of the ACM 23*(6), 343–349.

Williams, L. (1978, August). Casting curved shadows on curved surfaces. In *Computer Graphics (Proceedings of SIGGRAPH 78)*, Volume 12, pp. 270–274.

Williams, L. (1983, July). Pyramidal parametrics. In *Computer Graphics (SIGGRAPH '83 Proceedings)*, Volume 17, pp. 1–11.

Wilson, P. R., M. S. Johnstone, M. Neely, and D. Boles (1995). Dynamic storage allocation: A survey and critical review. In *Proc. Int. Workshop on Memory Management*, Kinross Scotland (UK).

Witkin, A. and M. Kass (1991, July). Reaction-diffusion textures. In *Computer Graphics (Proceedings of SIGGRAPH 91)*, Volume 25, pp. 299–308.

Wong, T.-T., W.-S. Luk, and P.-A. Heng (1997). Sampling with Hammersley and Halton points. *Journal of Graphics Tools 2*(2), 9–24. ISSN 1086-7651.

Worley, S. P. (1996, August). A cellular texture basis function. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, New Orleans, Louisiana, pp. 291–294. ACM SIGGRAPH / Addison Wesley. ISBN 0-201-94800-1.

Wyvill, B. and G. Wyvill (1989, March). Field functions for implicit surfaces. *The Visual Computer 5*(1/2), 75–82.

Yanovitskij, E. G. (1997). *Light Scattering In Inhomogeneous Atmospheres*. Berlin: Springer Verlag.

Yellot, J. I. (1983). Spectral consequences of photoreceptor sampling in the Rhesus retina. *Science 221*, 382–385.

Zorin, D., P. Schröder, T. DeRose, L. Kobbelt, A. Levin, and W. Sweldins (2000, August). Subdivision for modeling and animation. SIGGRAPH 2000 Course Notes.

E. Index of Classes

F.Index of Non-Classes

G.Index of Members 1

H.Index of Members 2

I.Index of Code Chunks