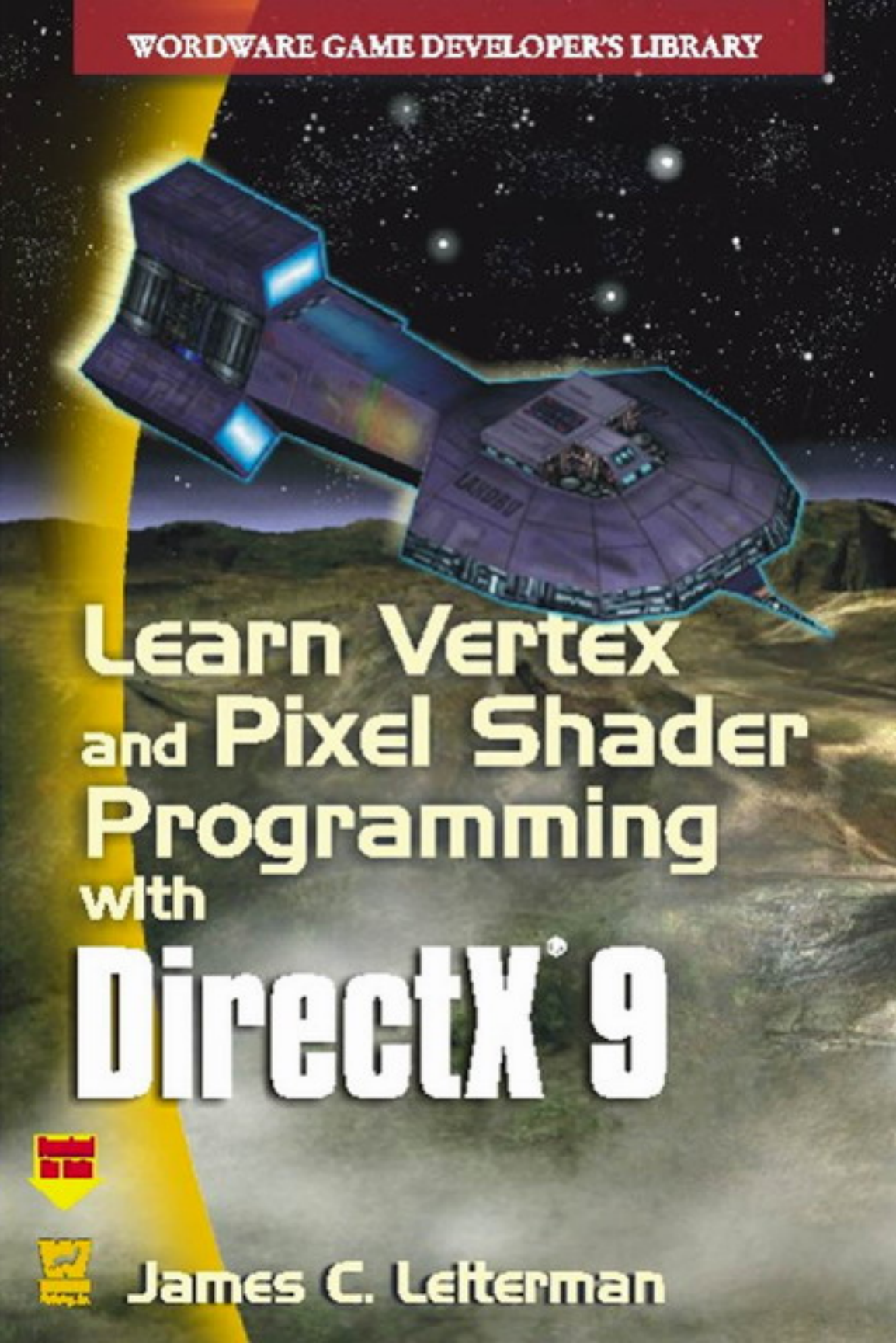
WORDWARE GAME DEVELOPER'S LIBRARY



Learn Vertex and Pixel Shader Programming with DirectX® 9

James C. Leiterman

Learn Vertex and Pixel Shader Programming with DirectX® 9

James C. Leiterman

This page intentionally left blank.

Learn Vertex and Pixel Shader Programming with DirectX® 9

James C. Leiterman

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

Leiterman, James C.
Learn Vertex and Pixel Shader programming with DirectX 9 / by James Leiterman.
p. cm.
ISBN 1-55622-287-4 (pbk.)
1. Vector processing (Computer science) 2. Computer games--Programming. 3. DirectX. I. Title.
QA76.5 .L444 2003
794.8'1666--dc22 2003020944
CIP

2320 Los Rios Boulevard
Plano, Texas 75074

Printed in the United States of America

ISBN 1-55622-287-4

10 9 8 7 6 5 4 3 2 1
0310

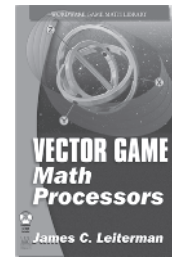
All inquiries for volume purchases of this book should be addressed to Wordware Publishing, Inc., at the above address. Telephone inquiries may be made by calling:

(972) 423-0090

Contents

Preface

(or, Why Did He Write Yet Another Book?)



Another book? Here we go again…

Why, yes! For those of you who have followed my eventful career, you already know that this is actually my third book. Third book? Well, my second book, *Vector Game Math Processors* (the first to be published) was met in the marketplace with enthusiasm.

Everyone who read it and contacted me seemed to like it. It was more of a "niche" book, as it primarily taught how to vectorize code to take advantage of the vector instruction sets of processors only recently introduced into the office and home. These contained functionality that was once confined only to the domain of the "super computer." This book discussed pseudo vector code that actually emulated vector processors using generic optimized C, as well as heavy utilization of the following instruction sets:

- X86 (MMX, SSE(2), 3D-Now! Professional)
- PowerPC and Altivec
- MIPS (MIPS-3D, MIPS V, etc.)

You may ask why I should write about three different processors in one book instead of three separate books. By using the tutorials, you should be able to cross program these and future processors (especially those proprietary ones).

So if you have not already, go buy a copy!

Did I not mention that this was my third book? Well, the first book has a long story, but you'll have to buy the vector book and read the preface in it to get the scoop on that one!

The problem with writing "yet another book" is that the first one tends to have the best jokes or the author's favorite stories.

But I have so many from over the years, especially since I have eight kids (with the same wife), that I have saved some for this book (and possibly the next one).

While I was writing the chapter on programming shaders in my vector book (they are vector processors too, are they not?), I realized the need for a book such as this to learn how to program them. I anxiously called my publisher, Wordware Publishing, to pitch the idea. Unbeknownst to me, they were already in the process of publishing a shader book, *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, edited by Wolfgang F. Engel. But they immediately came back with, "How about a beginner shader book with slightly fewer pages and a lower price tag, which can be purchased by a larger audience?" I thought about it and agreed. I was a glutton for more writing. Here I was, not even done writing the vector book, signing a contract for this shader book.

Bored yet? Here is where things get exciting! I sort of became greedy. Money had nothing to do with it. It was that name-in-print thing. One book is a novelty. Most authors write one, find out how much work it is, and discover that the monetary returns are extremely low, unless they have that one-in-2.8 million best-seller. (Just check out the sales rankings from Amazon or Barnes & Noble to see what I am talking about!) Two or more books indicate a serious author.

As I mention in an actual chapter within this book (just to make sure that those who skip reading this preface will find out), there are multiple manufacturers with their own shader instruction sets doing their own thing. The point is that this is the Intel/AMD processor wars all over again. The technology (as well as the instruction sets) is forked but will probably eventually merge back together again. Maybe it won't take 10+ years this time! That, and vertex shaders are so different from pixel shaders. This almost calls for two books:

- An introduction to vertex shaders
- An introduction to pixel shaders

But alas, both my publisher and technical editor felt that one would suffice.

For several reasons, this book primarily focuses on the DirectX 9 (DX9) functionality and forsakes earlier versions. The problem with this approach is that the video card you may be testing on may not be fully DX9 compatible.

This book is definitely needed because Engel's ShaderX books (two others were released during the editing phase of this book) are essentially a collection of technical white papers and therefore over the heads of beginners or inexperienced game programmers. But I continued on and finished, feeling confident that you would not see this as a "me-too" book!

One last item to note is that this book was actually delayed. I had originally planned to be done by December 2002 and have the book in stores in time for the 2003 Game Developers Conference. There were technical check delays on my vector book. (You would not believe how hard it is to find someone proficient in vector math processing who actually had some time in their schedule.) Finally, a couple of people were found and utilized! The other delay was the economy. I was actually unemployed for three months in 2003 (for a month August/September and then again November/January). It kind of forced us to have an extremely frugal Christmas and holiday season! Some would see that as a lot of free writing time, but regretfully no. Job hunting is a full-time (and then some) occupation.

Okay, I am not going to bore you with my "writing is hard work" speech, and if you *really* do not like my books, go write your own. But please contact me with any comments or recommendations (or bug fixes) by emailing me at books@leiterman.com.

I wish to thank those who have contributed information, hints, testing time, etc., for this book: My friend Paul Stapley for some technical recommendations; my old-time friend Jack Palevich from my Atari days for his review of my book and insight into vertex shaders; Wolfgang Engel for his technical check of this book; Ken Mayfield for some 3D computer art donations and my partner in a new line of game software in conjunction with my company, Wild Goose Games; Matthias Wloka with nVidia for

some technical vertex shader programming help; and others that I have not mentioned here.

Most of all, I'd like to thank my wife, Tina, for not balking too much when I bought that new laptop, G4 Macintosh, top-of-the-line video card, and other computer peripherals during the development of the vector book last year and for her muteness during the development of this book. Although I should note that this time, there was no rhetorical question, "That is the last one, right?" every time she discovered a new piece of equipment, because there were none visible to be discovered. I had purchased *all* the different flavors of graphics cards that supported shaders just to make sure that I had a complete and detailed book for you here!

I should note, however, that the location of my home office was changed by my wife from an entire wall in the master bedroom to a large room with two really big doors. Okay, okay, so my new computer lab is in the garage. Done laughing yet? Building full height wall partitions using 4'x8'x1" insulating foam makes for a pretty neat windowless office space that hides the majority of the garage. But I have plans to resolve the scenery problem! How many of you can say you have a 12'x18' private home office?

I finally wish to thank Jim Hill from Wordware Publishing, Inc. for seeing the possibilities of this and my last book, and Wes Beckwith for the two time extensions and not asking the question I frequently hear from my children: "Is it done yet? Is it done yet?" Finally, I'd like to thank Paula Price of Wordware Publishing for making sure those checks arrived just when I needed them.

So get up from that floor or chair in the bookstore where you are currently reading this book, as you know you will need this book for work. Besides, I filled it with so much stuff you might as well stop copying it into that little notebook. Grab a second copy for use at home, walk over to that check stand, and buy them both. Tell your friends how great the book is so they will buy a copy too! Insist to your employer that the Technical Book Library needs a few copies as well. This book is an instruction manual and a math source library all rolled up into one.

My eight children and outnumbered domestic engineering wife will be thankful that we can afford school clothes as well as

Christmas presents this year! Unlike that old movie's implication that kids are *Cheaper by the Dozen*, they are not! They eat us out of house and home!

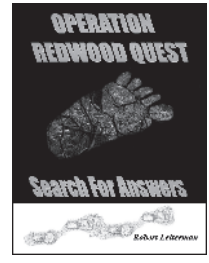
(Déjà vu — sounds like a cut'n'paste from my last book!)

To check for any updates or code supplements to any of my books, check out my web site: http://www.leiterman.com/books.

Send any questions or comments to books@leiterman.com.

■ ■ ■

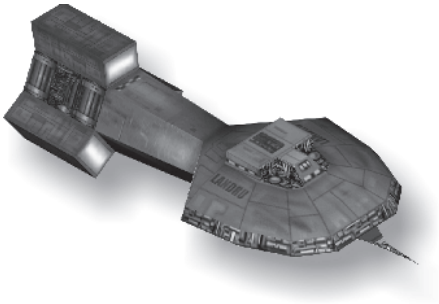
My brother Ranger Robert Leiterman is the writer of mystery-related nature books that cover diverse topics such as natural resources and his BigFoot mystery series. Buy his books also! Especially buy them if you are a game designer and interested in cryptozoology or natural resources or just have kids. If it was not for him having me proofread his manuscripts, I probably would not have started writing my own books as well. (So, blame him!)



Watch out for books from Thomas Leiterman, yet another of my brothers who has joined the Leiterman brothers book-writing club. That leaves three remaining brothers yet to join!

(*Now if we can only get our kids to read our books — something about needing more pictures…*)

This page intentionally left blank.



Chapter 1

Introduction

Hey, dudes and dudettes! You are now about to hang ten on the crest of a new technology wave!

The next logical step in making 3D graphics and animation go faster is to allow a programmer to add his own code directly into the rendering pipeline. The way is now paved where a simple C-style scripting language called Cg is implemented as well as an assembly programming language with miscellaneous levels of syntax for programming registers for the new programmable vertex and pixel shader-based video cards. In my previous book, *Vector Game Math Processors*, I discussed the internals of mathematics using vector-based processors. You can think of it as a foundational support for this book. You did read the preface in this book, didn't you? You did buy my other book, didn't you? Great! This book makes reference to it from time to time, so go buy a copy for your research library if you have not already!

The problem with writing an introductory book (I do not indicate beginner here, as it has certain connotations) is that one really does not know how introductory to make it. Readers can be anywhere from high school students with a weak math foundation trying to break into the game/graphics industry up to professional 3D programmers looking for new insight (or just wanting something to laugh at!). So I wrote this book with the assumption that you have a basic background in linear algebra and trigonometry (that is, sines, cosines, vectors, and matrices for those of you on the weak end of the mathematical skills spectrum).

This book gives a quick refresher, but those of you in doubt, please refer to another mathematics book, such as my vector book mentioned earlier or *3D Math Primer for Graphics and Game*

Development by Fletcher Dunn and Ian Parberry (Wordware Publishing).

This book is laid out as a reference manual in terms of functionality and is organized into three primary sections:

- Vertex shaders
- Pixel shaders
- Reference information

It is not an alphabetical listing like some technical bible documents because I personally prefer types of functionality to be adjacent to each other. Just because I wrote this does not mean that I remember everything. I frequently revisit my other books as reference manuals whenever I'm working with subject matter that they relate to (one of the reasons my home office has very large, overflowing bookcases).

Since I've always felt that the building blocks of code are like the circle of life (insert a lion standing on a rock here!), writing a book to be read and understood should be organized in that same way. Since about (I'm guessing here) 65 percent of you reading this probably do not have a new top-of-the-line video card that supports pixel shaders, then only the vertex shader section will be of use to you, as you probably will not be able to test anything you might learn. Of course, you could always borrow a card from work or get a hand-me-down from a rich friend. I received my first GeForce2 MX board that way and then started spending tons of money on various boards before the price started dropping. But I digress.

Part I, "Vertex Shaders," (Chapters 3 through 9) and Part II, "Pixel Shaders," (Chapters 10 through 11) are organized, for the most part, in a learning order. Instructions are learned early in a section and used in later sections or chapters. They have also been intermixed with some background information to help minimize the need to have to switch back and forth between a math book and this book. Throughout this book I attempt to annoy you to get you to buy my other book (or at the very least buy one or more books from Wordware Publishing).

As this is an introductory book, it is not really going to help you fly (only get you ready to jump out of the nest and glide to the

bookstore to buy, or get into a position to understand, Engel's ShaderX books).

I have painstakingly tried to ensure that the same mistakes present in the actual DirectX 9 documentation and in at least one of the books that appears to have been rushed to market were not repeated here. In fact, this book took a slow development path. There is very little white space on the pages in keeping with my last book, which crammed tons of information into a very little space. If you have purchased and read my vector book, you might find that this book is organized in a similar way. You might also experience some déjà vu. Some of the material in that book was needed in this book, and some cutting and pasting was implemented. Unfortunately, I had to come up with new stories and new jokes.

For each vertex and pixel section, a block diagram of the registers is given and an explanation of each of the registers is implemented, which is followed by information needed to assemble that code type. This is followed by an array of all the instruction-statement types, and by which version they are actually supported. Each of these are individually explained with the version numbers supported repeated next to the instruction to help limit the need for flipping back and forth.

Throughout the chapters you will see sections such as the following:

Pseudocode:

```
dx=ax+bx    dy=ay+by    dz=az+bz    dw=aw+bw
```

Anybody who has read Knuth knows exactly what pseudocode is. For the rest of you, however, pseudocode is not really a programming language but coding data organized in terms of functionality. It contains enough information to allow one to understand the basic functionality of the code and code it in any programming language.

Algebraic law:

Additive Inverse	a – b = a + (–b)

By using source negation:

$$d_x=a_x+(-b_x) \quad d_y=a_y+(-b_y) \quad d_z=a_z+(-b_z) \quad d_w=a_w+(-b_w)$$

Since this is partially a math book and contains linear algebra, it might as well have the algebraic laws handy. I have personally found that having them nearby helps me to understand solutions to problems. That is something I did not do in high school because I found them too wordy, but I heavily push it on my own kids not in an attempt to be an evil father, but to make their life much easier.

Listing 1-1: Vertex shader

```
// Sets c0 with {1.0, 0.0, 2.0, 1.5}
vs.1.1                              // Version 1.1
def  c0, 1.0f, 0.0f, 2.0f, 1.5f   // Set c0 register
```

A vertex shader code listing is just that — code that is written specifically for the vertex shader. There is no CD included with this book, but the samples demonstrated within are available for download from www.wordware.com/files/vshaders. Those samples are labeled by chapter. Additional snippets within the book are actually modifications of existing samples with the DirectX SDK, which can be downloaded directly from the Microsoft web site. Errata can be found at http://www.leiterman.com/books.html.

Listing 1-2: Pixel shader

```
// Sets c0 with {1.0, 0.0, 2.0, 1.5}
ps.1.1                             // Version 1.1
def c0, 1.0f, 0.0f, 2.0f, 1.5f   // Set c0 register
```

A pixel shader code listing is just that — code that is written specifically for the pixel shader. The rest of the vertex shader explanation applies here as well.

Listing 1-3: C++

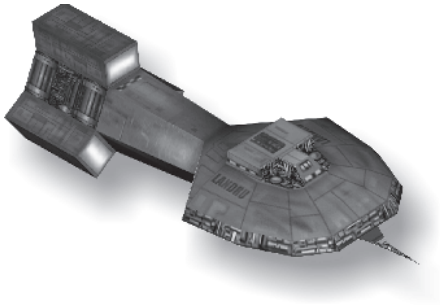
```
D3DMATRIX mtxA, mtxB;

mtxA = mtxB;
```

Sample C code snippets are included (not everywhere, but wherever I felt that it would make understanding a concept easier or if there was some trick that I felt was important enough to be learned). For example, did you know that you could just equate a structure to a structure to copy it within Visual C? No need for the wasteful overhead of a memcpy() function. In this particular case, the matrix is 16x4, thus 64 bytes, and so the compiler actually expands memory move instructions to copy the data.

Finally, whenever macro instructions were utilized, I did my best to expand them to make their functionality easier to understand. It also helps to make evident the reason why source arguments cannot be destination arguments when raw macros are utilized.

Are you still reading? Great, let's get on with it!

This page intentionally left blank.



Chapter 2

Getting Started

Writing shaders can be a simple or complicated process, depending on the visual effect that you are trying to achieve. One thing to remember is that the more complicated your shader, the longer it takes to process a pixel. The nice thing about shaders, as with programming Microsoft Windows, is that they both work with a similar basic shell. In terms of Windows, there is a WinMain as well as a WinProc and the same basic initialization code is used from application to application. For shaders, the same basic shader building blocks are in place. The neat thing about shaders is that you can dig into your library of shaders, pull out the closest one in functionality to the effect that you have in mind, and then modify it to accommodate your needs. The one thing that you should keep in mind as you go through this book is that you need to understand what you can do with a shader and what shader assembly instructions are available for you to do so, depending on the shader.

In this book, the foundations of programming video cards with shader capability are made evident to those of you with access to one of them that supports the vertex and pixel shader chipsets, such as nVidia's GeForce series 3, 4, or higher, ATI's Radeon 8500-9700 series, Matrox's Parhelia-512, and 3Dlabs' Wildcat VP10. It should help those of you who are either new to the industry or have been up until now absorbed by the 2D world and looking for a leg up into the 3D world (especially in this world where games have for the most part gone to 3D rendering). Unless you are a 3D programmer, finding a job becomes more difficult as more and more doors are closed and competition for the few remaining jobs becomes more fierce!

3D programmers working with the Open Graphics Library (OpenGL) and DirectX 3D are spoiled by the luxury of calling application programming interfaces (APIs) to do the core of their 3D work without having to know how to handle rendering, face culling, or hidden line removal. This book jumps you past that into a new realm, helping you get caught up with the industry.

If you really want to jump in feet first, check out *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, edited by Wolfgang F. Engel. It is an excellent collection of white papers for advanced usage of vertex and pixel shaders, but refer back to this book for the fundamental vector functionality.

Do not get bogged down by the Direct3D aspects, as OpenGL uses this technology and Macintosh computers have these graphics chips available to them as well.

With this new technology, you now have a choice. One is to visit one of the following web sites and scroll through their demo aisles with your little shopping cart until you come to the product functionality similar to what you are interested in. Drop the demo into your download cart and check out!

- nVidia: http://www.nvidia.com
- ATI: http://www.ati.com
- Matrox: http://www.matrox.com
- 3Dlabs: http://www.3dlabs.com
- SiS: http://www.xabre.com
- DirectX SDK: http://www.microsoft.com

Then cut'n'paste their code samples into your product. But that would be cheating, and you would not really learn anything. Besides, where is the fun in that? So how about buying and reading this book instead? Use what you have learned. Go ahead and download those samples, but pick them apart and learn how they function and more. *It is all vector processing!*, just from a higher elevational point of view. This is where my vector book comes in handy!

This is a grand, relatively new technology, as rendering calculations are moved from the processor CPU(s) to the video GPU (graphics processor unit). This relieves the CPU of that time-consuming burden, thus freeing its time for more time-worthy game

processing, such as AI (artificial intelligence), game physics, event triggering, terrain following, etc. At the time of publication, DirectX 9 was being released, and there were seven instruction set versions available; those have been grouped in terms of function enhancements.

- Vertex {1.1}, {2.0}, {3.0}
- Pixel {1.1, 1.2, 1.3}, {1.4}, {2.0}, {3.0}

Some of you may be ringing the fault buzzer at the moment. ("Ha! Ha! You missed version 1.0.")

Sorry, I did not. You see, by the nature of progress, Microsoft tends to retire technology, and so version 1.0 is no longer supported. That is also the reason why I personally hang on to old MSDN CDs, because information is dropped due to nonsupport and you never know when you need access to that old "ancient" technology.

Well, actually the hardware for vs.1.0 and ps.1.0 was never released, as nVidia was the first to put out the 1.x for Xbox and 1.1 for the GeForce card, which started the wave of shader programming.

Newer versions will probably become available when this book makes it to the bookshelf of a book dealer near you, as the state of the technology is always advancing. At the time of publication, version 3.0 was only supported by software emulation, as supporting hardware was not out yet.

DirectX Version Checking

So before we get much further, you should obtain, download, or (by some other means) get and install the latest and greatest DirectX (in this book's particular case, DirectX 9). Please note that there are slightly different releases of this version, and your Software Development Kit (SDK) must be an exact match with your installed redistribution version. If not, they will be incompatible and will thus not work together. In the old days of DirectX, as long as you had any version of the product, such as DirectX 3 or DirectX 6, it did not matter which specific minor version you had installed; the code would still work for that major release. But

these days, it has become a wee bit more stringent. In your initialization code, there should be some code similar to the following to verify compatibility:

Listing 2-1: \Chap02\SDKVersion\Bench.cpp

```
IDirect3D9   *m_pD3D;

// Create the Direct3D object
//
// NOTE: - DirectX SDK version 9 MUST be installed!

m_pD3D = Direct3DCreate9(D3D_SDK_VERSION);
if (NULL == m_pD3D)
{
  Log("Couldn't initialize Direct 3D System!");
  return false;
}
```

The keyword here is the definition D3D_SDK_VERSION. It is defined within the SDK header file and returned by the function Direct3DCreate9(). When it is passed to the function during the DirectX initialization, it is compared to the DirectX 9 version, and they *must* match exactly or the function will fail. This guarantees that the SDK headers match the DirectX install of the target so that the appropriate code can be built and the application code can run properly. The defined value has no meaning! It is merely a comparison value. The following definitions might give you an idea of the importance of this statement:

DirectX SDK Version	Header	D3D_SDK_VERSION
8.0 ≤	d3d8.h	220
9.0	d3d9.h	30
9.0	d3d9.h	31

Notice the multiple versions for the same 9.0 SDK that all use the same d3d9.h header file. Your application should prompt the user to upgrade his DirectX to match.

NOTE: Install the entire SDK with all features, especially its sample code. It does not require that much extra space, and this book takes advantage of it!

So install the entire SDK along with the sample code and debug redistribution. It will run slower, but you can software toggle it to release mode for faster throughput when you want to play games. Having the debug option assists you in the development (and especially debugging) of your applications.

The typical install path for the SDK is the directory C:\DXSDK, and so this book is based upon that root path, although you may have installed your SDK in a different location depending on space considerations. You should remember that there are multiple downloads to obtain all the components, so try to make sure that you have the space available.

Once it is installed (or even if it's been installed for some time), check out the following Tools option and make sure that the DirectX include and lib folders are at the top of the list.

Under Visual C++ 6.0, click on the menu item Tools and then Options. Then select the Directories tab on the Options property page. The combo box "Show directories for:" (Include files) should have C:\DXSDK\Include at the top of the list of include folders. If not, select and then move up the list by clicking on the up arrow.

Do the same for the Library files, which should be set to C:\DXSDK\Lib as well. Then click OK.

If you had to change a selection, exit Visual C++ by clicking File and Exit, as this will have Visual Studio save the options so they will be available the next time you execute the tool.

One last item for you to do is unzip the files for this book (available at www.wordware.com/files/vshaders) in the LearnVPShader folder at a location of your choosing. Then copy the Common folder (C:\DXSDK\Samples\C++\Common) and place it into the root folder .\LearnVPShader\ along with the chapter folders, such as Chap02.

So your folder should be similar to the following:

```
C:\LearnVPShader\Common
                \incX86\
                \Chap02\
                etc.
```

STOP: Is the Common folder missing? If so, stop here, and reinstall the latest *complete* DirectX 9 SDK.

Other folders, such as Common, Direct3D, DirectInput, etc., should be visible within the .\DXSDK\Samples\ folder. If the folder is missing, then you did not do a complete install of the SDK. Stop, reinstall the complete SDK, and then try these installation instructions again.

Data Alignment Issues

The one thing that really bothers me about the DirectX manuals is the lack of insistence that a programmer take care to ensure that his or her data is properly aligned! Admittedly, it does make their library more forgiving. I have not detailed it here because I have beaten the concept to death in my vector book. In short, however, make all your code 16-byte aligned to optimize your processor speed. Not enough said, but take it to heart!

HINT: Align as much of your data as possible, regardless if it is being written for DirectX.

Video Cards

Mirror, mirror, on the wall, who has the fairest video card of all?

Look deep into your shader soul and thou shall find that for each, there is no coal or diamond to mine!

Before we get into it, we need a video card that supports vertex and pixel shaders. (Please note that for some of you with laptops or motherboards with an embedded video chip, you do not use a video card, but the term "video card" still applies to you!) Which video card do you get? Well, if you have around $200 to $400 U.S. lounging around burning a hole in your pocket, then by all means, get the top of the line! But if you are on a budget (or worse), you need to decide what functionality you can do without or afford! Get a bottom-end or mid-priced video card and bump up after you have mastered the entry level stuff. But be careful, labels can sometimes be misleading. Examine the next table carefully! Besides, no card supports all features, and therefore you need to

decide what features you wish to utilize and thus choose your card (or cards) accordingly.

Currently, there are five manufacturers putting out video chips with shader support: nVidia, ATI, Matrox, SiS, and 3Dlabs. There are many flavors of video cards out there with GPUs supporting some combination of programmable vertex and pixel shaders, as well as the cards having all sorts of bells and whistles, but the primary task is finding what instructions you want to support, how fast you want it to be, and thus which chip the card needs to contain. The following table shows the latest model chips and their version support, as well as some interesting information such as number of vertex pipes, number of rendering (pixel) pipes, number of textures handled per pixel pipe, and number of constant registers available.

Table 2-1: Manufacturers with basic GPU and relative information.

MFG	Chip	VS Version	PS Version	VS Pipe	PS Pipe	Tex ×N	Const
nVidia	GeForce2	---	---	0	4	×2	0
	"	3.0_{sw}	3.0_{sw}	0	16	?	8192
	GeForce3–Ti	1.1	1.3	1	4	×2	96
	Xbox	1.0	1.0x	2	4	?	96
	GeForce4–MX (NV18)	1.1	---	0	2	×2	96
	GeForce4–Ti (NV28)	1.1	1.3	2	4	×2	96
	GeForceFX (NV30)	2.0+	2.0+	?	?	?	256
ATI	8500/9100 (RV200)	1.1	1.4	2	4	×2	192
	9000 (RV250)	1.1	1.0-1.4	2	4	×1	192
Radeon	9200 (RV280)	1.1	1.0-1.4	2	4	×1	192
	9500/9700 (RV300)	1.1, 2.0	1.4, 2.0	4	8	×1	256
	9600 (RV350)	2.0	2.0	2	4	?	?
	9800 (R350)	2.0	2.0	2	8	×1	256
Matrox	Parhelia-512	1.1, 2.0	1.3	4	4	×4	256
		3.0_{sw}	3.0_{sw}	4	16		8192
SiS	Xabre 200	---	1.3	0	4	?	0
	Xabre 400	---	1.0-1.3	0	4	×2	0
	Xabre 600	1.1_{sw}	1.3	0	4	×2	0
3Dlabs	Wildcat VP	1.1	1.2	2	4	?	16

(#.#$_{sw}$ indicates *only* software emulation.)

I recommend that you follow up on each of the companies' web sites and get additional statistical information, such as fill rates, operations per second, memory bandwidth, etc., in order to make an informed decision. The information in Table 2-1 is only enough to determine what instruction set your prospective card is capable of supporting.

For DirectX 8 (aka vertex shaders {1.0, 1.1}, pixel shaders {1.0 … 1.4}), you must have a high-end video card to support programmable pixel shaders. Vertex shaders can otherwise be hardware-based, thus programmable or nonexistent and emulated in software. With the advent of DirectX 9, the shader support of 2.0 and 3.0 was made available, and support for 1.0 was dropped. At the time of publication, you can see from Table 2-1 that 2.0 was available from only the newer high-end cards, and 3.0, which I labeled as $3.0_{sw,}$ was only emulated! Poor Xbox. By that reasoning, it is now ancient technology!

When I personally buy new computer processors, I typically buy for the instruction set and not so much for speed (but to each his own). Also, do not make the mistake of buying the wrong GeForce4! (Not unless you mean to buy the lower-cost MX.) Zero vertex shaders means that it does not have any hardware support and is thus not applicable to what this chapter is about! However, it does have very fast software emulation due to its onboard hardware assist. If you have little or no money, you can always use software emulation of the vertex shaders. It is much slower, but at least you can still test your vertex algorithms.

As I mentioned in the preface, I only have all of them for the purposes of writing this book in an attempt to be as accurate as possible. As they say in the movie *Jurassic Park*, "I spared no expense!"

Version(s) Determination: Vertex and Pixel Shader

If you are writing vertex or pixel code specifically for your video card, then you can use the version of shader assembly that is compatible with your video card. If you are writing code to work across multiple video cards, then you have a problem. As already explained, each card has its own compatibility and thus capability. This section of the chapter discusses vertex and pixel shader compatibility; version information is evaluated at the same time. Pixel shader version determination, however, is discussed in a bit more detail in Chapter 10, "Pixel Shaders."

To begin with, we must first review the source of the version information. A video adapter contains an ordinal and a set of associative properties with all of the video adapter capabilities. This is referred to as a *Direct3DDevice* object, and is defined by the data structure D3DCAPS9. An application steps through this list of exposed properties and tests each against the features that it requires for its execution. Upon finding a match, it is utilized. For our purposes of examining version control, only a couple of the individual data members need to be inspected.

```
UINT            AdapterOrdinal;         // Video card index
D3DDEVTYPE      DeviceType;             // HAL, REF, SW
DWORD           VertexShaderVersion;    // Vertex shader version
DWORD           PixelShaderVersion;     // Pixel shader version
DWORD           DevCaps;                // Behavioral bits
```

There are other data members, but they are not necessary for version selection. Those data members that are useful are highlighted here:

- **AdapterOrdinal**: The index used to index the video adapter
- **DeviceType**:

D3DDEVTYPE_HAL	= 1	Hardware rasterization
D3DDEVTYPE_REF	= 2	Reference rasterizer
D3DDEVTYPE_SW	= 3	A software device plug-in

- D3DDEVTYPE_HAL: (hardware abstraction layer) Shading is handled with hardware, software, or mixed, with transform and lighting.
- D3DDEVTYPE_REF: Shading is handled only by optimized software and {vector, SIMD, parallel} CPU instructions are utilized whenever possible.
- D3DDEVTYPE_SW: A software device plug-in that has been registered with IDirect3D9:RegisterSoftwareDevice.

- **VertexShaderVersion and PixelShaderVersion**: The vertex and pixel shader version numbers are a 32-bit value consisting of major and minor version numbers. The following macros are used to extract the major and minor version values for both types of shaders:

```
Maj = D3DSHADER_VERSION_MAJOR(pCaps->VertexShaderVersion);
Min = D3DSHADER_VERSION_MINOR(pCaps->VertexShaderVersion);

Maj = D3DSHADER_VERSION_MAJOR(pCaps->PixelShaderVersion);
Min = D3DSHADER_VERSION_MINOR(pCaps->PixelShaderVersion);
```

  To build a 32-bit version number for the vertex shader, the following macro is used to combine the major and minor version numbers:

```
DWORD Version = D3DVS_VERSION(Maj, Min);
```

  …and for a pixel shader:

```
DWORD Version = D3DPS_VERSION(Maj, Min);
```

  So be careful not to mix'n'match, as each has its own masking key!

In the Common folder of your DirectX SDK installation, which is typically located at C:\DXSDK\Samples\C++\Common, examples of the following can be found within the sample applications.

- **DevCaps**: These are the flags (bits) identifying the capabilities of the device object. The following are the bits that we are concerned with:
  - D3DDEVCAPS_PUREDEVICE: If set, this device is the D3DDEVTYPE_HAL. It supports rasterization, transform, shading, and lighting using hardware.

- D3DDEVCAPS_HWTRANSFORMANDLIGHT: If set, this device can support transformation and lighting in hardware.

Okay, here is where things get a little interesting. At the lower level (directly using these bits), it can be determined whether pure hardware, hardware, software, or mixed versions are supported. It helps remap these bits into the following behavioral bit flags:

- D3DCREATE_SOFTWARE_VERTEXPROCESSING: For video cards without hardware shader support, this first definition of *_SOFTWARE_* would apply. This is mentioned earlier, such as in the case of the nVidia GeForce4-MX board, where shader support is emulated in software.
- D3DCREATE_HARDWARE_VERTEXPROCESSING: Video cards that do have shader hardware support of a specific version. For example, a GeForce3-TI can support the vertex shader specification of version 1.1 and thus *_HARDWARE_* would apply.
- D3DCREATE_MIXED_VERTEXPROCESSING: This declaration is a bit trickier, as old cards needing to support new instruction sets cannot fully support a pure vertex shader at the hardware level, and thus a combination of software and hardware can be utilized; so *_MIXED_* would apply!

We should discuss one more item before we continue. This book is an introductory book about how to program shaders and not so much about DirectX programming, but certain information is needed to do so. The pre-existing samples from the SDK are relied upon as well as the graphics framework that Microsoft has prepared to make your learning experience go more smoothly. These can be located in your SDK folder:

.\DXSDK\Samples\C++\Common\Src\

Of all the files found, there are some that will be of immediate usefulness.

D3dapp.cpp contains the generic application interface and D3dutil.cpp has 3D functions related to material, light, and textures.

There are others, but these will help us get running faster. To use these common files used in the SDK samples and your own, include the following at the top of your application code and include the associated CPP files into your builds:

Listing 2-2: \Chap02\VertexVersion\VertexVersion.cpp

```
#include "KariType.h"     // Master definitions

   // Third-party libraries

#define STRICT

#include <windows.h>
#include <commctrl.h>
#include <stdio.h>
#include <math.h>
#include <d3dx9.h>

   // Microsoft DirectX SDK helper functions

#include "DXUtil.h"
#include "D3DEnumeration.h"
#include "D3DSettings.h"
#include "D3DApp.h"
#include "D3DFile.h"
#include "D3DFont.h"
#include "D3DUtil.h"
```

Before building our own code, the first sample project that should be investigated is the vertex shader program found on the SDK at the following location:

.\DXSDK\Samples\C++\Direct3D\VertexShader\

This takes care of the burden of DirectX initialization, Direct 3D initialization, enumerated device selection of our video card, enumerated level of access, etc. As I mentioned earlier, this is not a DirectX programming book, but we need to get up to the level of shader programming as quickly as possible! It also becomes a good test to make sure that your development environment is set up correctly and your video card supports shader technology, at least at a rudimentary level.

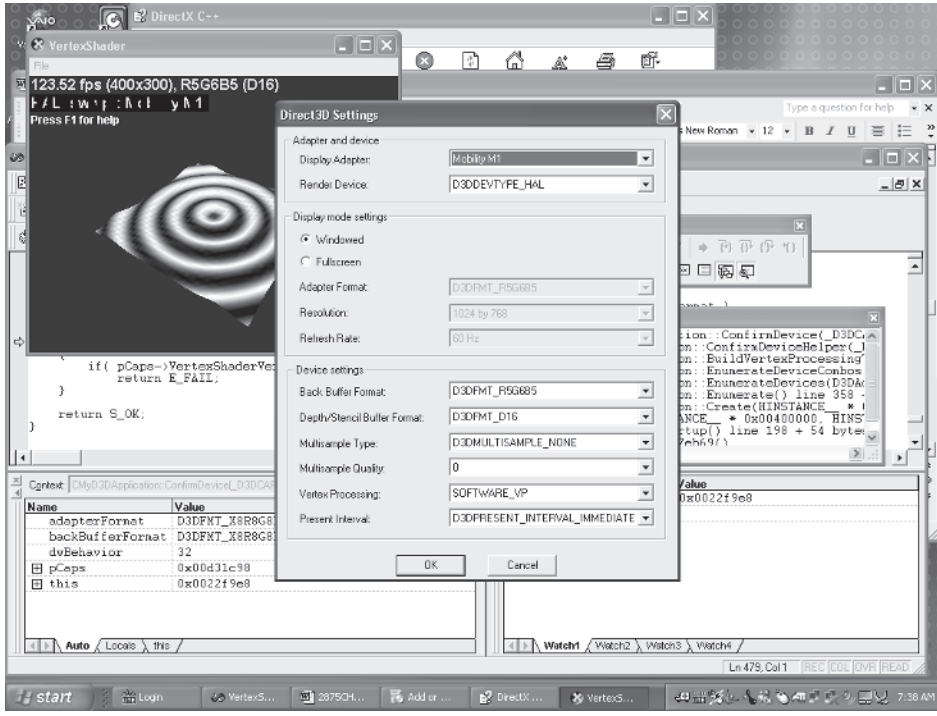


Figure 2-1: Output from the sample VertexShader.exe

A simple compile and magically there exists an instant sample that can be built upon! No spending a day or more trying to build one's own architecture. Besides, you can always do that later!

As I mentioned previously, we may not be running our application on our own computer, and so we are probably not familiar as to what video hardware is inside. Thus, it would be up to our application program in conjunction with appropriate shader code to work together accordingly.

Okay, before we get too far ahead of ourselves, we need to learn how to detect what shader versions can be handled on our development computer. So let's investigate. First, for our version dump experiment, our basic application should look similar to the following:

Listing 2-3: Chap02\VertexVersion\VertexVersion.cpp

```
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, int)
{
    CMyD3DApplication d3dApp;

          // Initialize common code

    if (FAILED(d3dApp.Create(hInst)))
     {
        return 0;
     }

    return 0;
}


HRESULT CMyD3DApplication::ConfirmDevice(
       D3DCAPS9* pCaps,     // Capabilities
       DWORD dwBehavior,    // Behavioral bits
       D3DFORMAT adapterFmt, D3DFORMAT backBufferFmt)
{
   if ((D3DCREATE_HARDWARE_VERTEXPROCESSING & dwBehavior)
        | (D3DCREATE_MIXED_VERTEXPROCESSING & dwBehavior))
   {           // Vertex Version 1.0
      if (pCaps->VertexShaderVersion < D3DVS_VERSION(1,0))
      {
         return E_FAIL; // Reject (doesn't meet criteria!)
      }
   }
   return S_OK;         // Accept!
}
```

The second argument passed into the enumeration callback ConfirmDevice() contains the individual behavioral bits related to device creation.

```
DWORD dwBehavior;

D3DCREATE_SOFTWARE_VERTEXPROCESSING
D3DCREATE_HARDWARE_VERTEXPROCESSING
D3DCREATE_MIXED_VERTEXPROCESSING
D3DCREATE_PUREDEVICE
```

The first argument passed into the enumeration callback ConfirmDevice() contains references to properties related to an individual device feature of a shader capability.

```
D3DCAPS9* pCaps;
```

The following is a snippet of sample output from the VertexVersion sample using a video card that does not support shaders. The _abbreviations_ are behavioral bits represented as follows:

Please note the versions *0.0* and *3.0*, as well as the *HAL* and *REF*.

```
#0 Vertex: 0.0  Pixel: 0.0  HAL  _HARDWARE_
#0 Vertex: 0.0  Pixel: 0.0  HAL  _SOFTWARE_
#0 Vertex: 3.0  Pixel: 3.0  REF  _HARDWARE_  _PURE_
#0 Vertex: 3.0  Pixel: 3.0  REF  _HARDWARE_
```

The hardware abstraction layer (HAL) is defined by D3DDEVTYPE_HAL. This represents actual hardware support but only for version 0.0, which is not usable, as shaders only support from version 1.0 or better. Thus, the version 0.0 is ignored. That leaves version 3.0, but please note the (REF) defined by D3DDEVTYPE_REF. This means that the vertex shaders are emulated in software!

Using our sample VertexShader.exe, a video card that supports hardware vertex and pixel shaders would have an output similar to the following snippet(s) from a Matrox Parhelia and nVidia GeForce4-Ti card with two monitors activated; note the #0 and #1 representing the adapter number.

```
#0 Vertex: 1.1  Pixel: 1.3  HAL  _HARDWARE_  _PURE_
#0 Vertex: 1.1  Pixel: 1.3  HAL  _HARDWARE_
#0 Vertex: 1.1  Pixel: 1.3  HAL  _SOFTWARE_
#0 Vertex: 3.0  Pixel: 3.0  REF  _HARDWARE_
#0 Vertex: 3.0  Pixel: 3.0  REF  _SOFTWARE_
#1 Vertex: 1.1  Pixel: 1.3  HAL  _HARDWARE_  _PURE_
#1 Vertex: 1.1  Pixel: 1.3  HAL  _HARDWARE_
#1 Vertex: 1.1  Pixel: 1.3  HAL  _SOFTWARE_
```

Also notice that vertex shaders 1.1 and pixel shaders 1.3 are supported by hardware by both displays! But software emulation is used for 3.0 for both shaders. Another item of note is that there are no pixel 1.0, 1.1, or 1.2 device object enumeration items. That is because those are subsets of version 1.3, and in DirectX version

8.1 (or 9.0 for that matter), all were supported, thus you as a programmer would specify the latest and greatest.

The dump below for the ATI 8500 has device objects that look similar to the following:

```
#0 Vertex: 1.1  Pixel: 1.4  HAL  _HARDWARE_  _PURE_
#0 Vertex: 1.1  Pixel: 1.4  HAL  _HARDWARE_
#0 Vertex: 1.1  Pixel: 1.4  HAL  _SOFTWARE_
```

But watch carefully! This contains a pixel shader of 1.4, which is a result of the shader wars. This is explained later, but briefly, competing companies have forked the technology (similar to that of the X86 wars of the '90s!). In this case, pixel shaders 1.0 is not supported, but 1.1-1.4 are! Essentially, the versions up to the specified version are supported. If we examine the dump of a newer card, such as the ATI 9700, it has enumerated device objects that look similar to the following:

```
#0 Vertex: 2.0  Pixel: 2.0  HAL  _HARDWARE_  _PURE_
#0 Vertex: 2.0  Pixel: 2.0  HAL  _HARDWARE_
#0 Vertex: 2.0  Pixel: 2.0  HAL  _SOFTWARE_
```

It does not contain a device object line listing of vertex 1.1 or pixel 1.4, but those versions are supported. Remember what I said about shader wars? The version 2.0 specification is inclusive and exclusive of the 1.4 specification. This means that some features are supported by both 1.4 and 2.0, some features are only supported by 1.4, and some features are only supported by 2.0.

Note that the function member ::ConfirmDevice() is in reality written by you to detect and allow various formats of shader cards to work with your code with its supporting shader code.

```
if (pCaps->VertexShaderVersion < D3DVS_VERSION(1,0))
{
   return E_FAIL; // Reject (doesn't meet criteria!)
}
```

In the previous sample of this function, any card that supports the vertex shader at or above the version of 1.0 was not rejected by the return of an S_OK, indicating an acceptable enumerated device object. If your code is more discriminating, then you would need something similar to the following:

```
if ((D3DVS_VERSION(1,4)  == pCaps->PixelShaderVersion)
  && (D3DVS_VERSION(2,0) == pCaps->PixelShaderVersion))
{
   return S_OK;            // All ATI type cards okay!
}
```

And if you needed to reject a specific shader type:

```
if (D3DVS_VERSION(1,0) == pCaps->VertexShaderVersion)
{
  return E_FAIL;          // Do not support this OLD type!
}
```

The idea is that you need to trap for specific shaders that you do handle and then work around those that you do not. You do not want to reject video cards that you do not support, as fewer computers will be able to run your software. You should keep in mind that this rejection does not remove a target computer; it only reduces the number of enumerated items that would be available later as choices of mode selections. If you, however, reject all of these shaders (even the software-emulated ones), you are in fact stating that no shader will run on your code! So be careful here! The sample VertexVersion does just this. Note the no shaders available dialog that occurs.

Figure 2-2: Sample dialog from VertexVersion termination

All of this may seem twisted, but in reality it is not. I explain it in more detail later.

Plumbing: The Rendering Pipeline

Have you ever noticed those tubes (pipes) in some warehouse stores that extend from a checkout register to the ceiling and beyond? Little cylinder-shaped conveyances pushed by air travel through the tubes and are routed through a pipe through several switch boxes in their routes to their eventual destinations. You hand the clerk a claim check that you picked up off a shelf, and you pay for the item. She then sticks the piece of paper in a little cylinder, inserts it into the tube, and presses a button. Then with a whooshing blast of air, the cylinder is gone. Then you stand around (alone) for 30 minutes or more waiting for someone to bring a big oversized box out to you, leaving you to strong-arm it into your car, which is typically too small to…

Okay, okay, my technical editor just pointed out to me that this is ancient U.S. technology! But it is still being used. Visited your drive-through bank lately? Or local home improvement store? It's a good representation of what occurs in the shader pipeline.

Sorry about that long story, but you can think of a 3D rendering system as a similar type of pipeline. Each vertex of a polygon is inserted into a train of these cylindrical conveyances and inserted into the end of the pipe. The vertex shaders are handled early in the pipeline and are used in the manipulation of vertices. The pixel shaders are utilized late in the pipeline and are only designed to process pixels. Together they can be used to create fabulous special effect displays within a scene. Before beginning, however, the rendering pipeline should be examined!

Note the two gray boxes in the path on the right side of Figure 2-3. They indicate the position within the flow of logic of the vertex shader module and the pixel shader module. This rendering pipeline is usually referred to as a pipeline, but in reality the whole architecture is plumbing, which has some parallel rendering pipes. Whether the logic uses an older fixed, semi-static flow, such as on the left that was used by earlier versions of DirectX, or the more robust dynamic flow on the right used by the latest versions such as DirectX 9, it is all fluid! Please note the placements of the vertex and pixel shader boxes, as their utilization is what this book is all about!

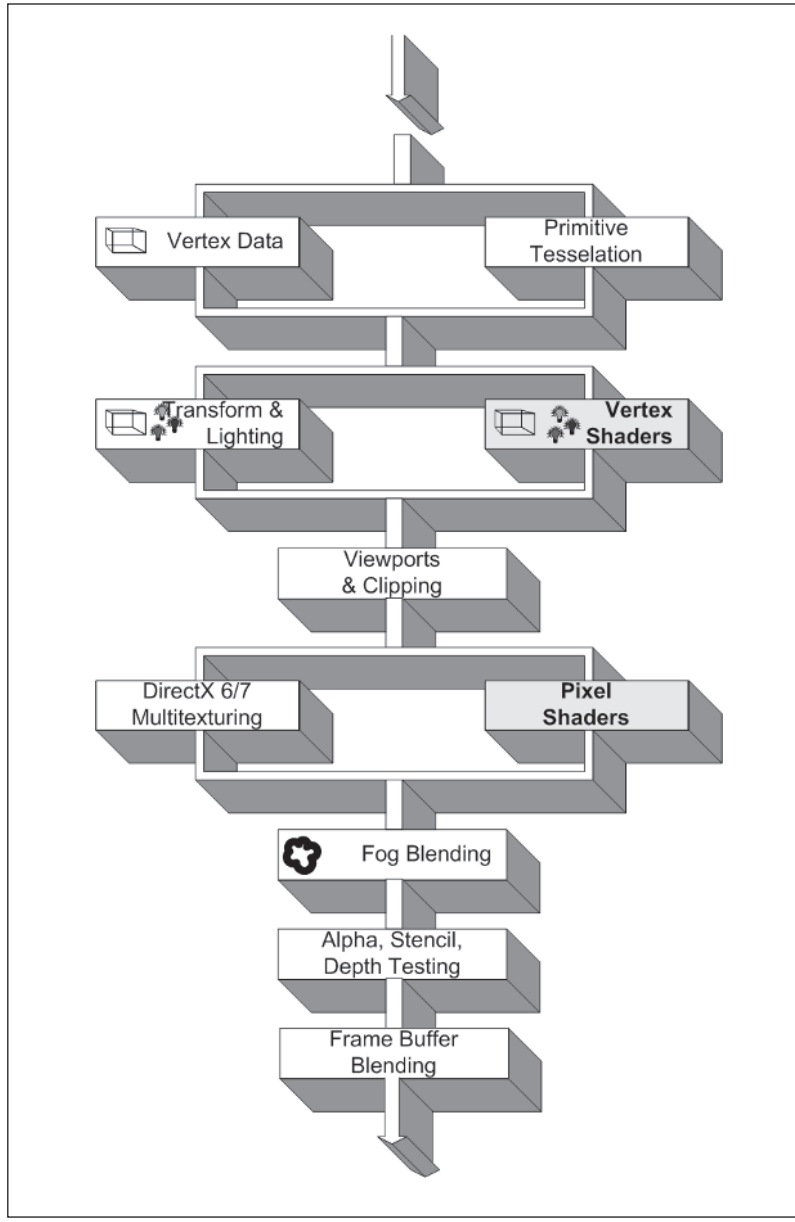


Figure 2-3: Rendering pipeline (plumbing)

The Vector

In a vertex or pixel shader, the concept of computer memory does not exist. Only the method of registers, which is used for accessing data values, exists. The registers in the vertex and pixel shaders are primarily 128-bit vector based. This is known as a quad vector. That is, they have a block of bits organized as four elemental components {XYZW}, each a 32-bit single-precision floating-point value.

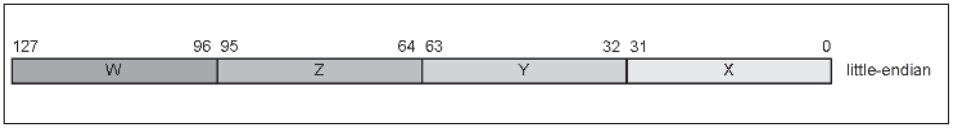


Figure 2-4: A 128-bit vector containing the four elements {XYZW} arranged in a little-endian format

You may have noticed the little-endian labeling. Since this book is about programming shaders on an X86 processor, which is a little-endian processor, emphasis is placed upon the little-endian declaration.

If you are a Macintosh or MIPS programmer using this book as a programming reference for your big-endian platform, then this label should help keep you from getting confused over data ordering for your platform. That's enough for the moment about endian data orientation!

The three elements {XYZ} together is typically considered a vector, and four elements {XYZW} is typically considered a quad vector, where the {W} field is neutral, but for purposes of clarity between this book and other source information related to programmable vertex and pixel shaders, the term "vector" will be used to represent a four-element float represented by a 128-bit register. Okay, I have said this twice. For those of you new to this concept, please revisit it in my vector book!

The following is just a quick background for clarity. In the C programming language, each elemental scalar is known as a float.

```
float val;
```

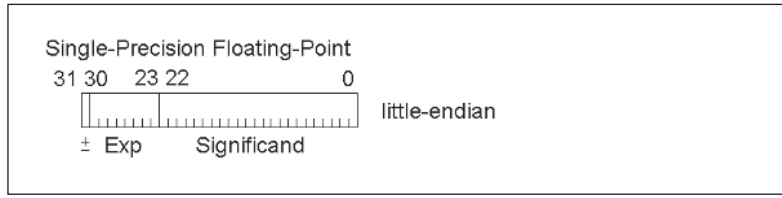


Figure 2-5: The bit encoding of a single-precision floating-point value — {1:8:23} one sign bit, eight bits of exponent, and 23 bits of significand (mantissa)

Wrapped into a quad vector, it would appear like the following using Microsoft C container:

```
typedef struct D3DXVECTOR4
{
  float x;
  float y;
  float z;
  float w;
} D3DXVECTOR4;
```

At this time, please notice the "4" in the term D3DXVECTOR**4**, which is used to indicate four arguments. There are other vector-type definitions, such as:

D3DXVECTOR2
D3DXVECTOR3
D3DXVECTOR4
D3DXQUATERNION

…of which the quaternion declaration is nearly identical; it is discussed later.

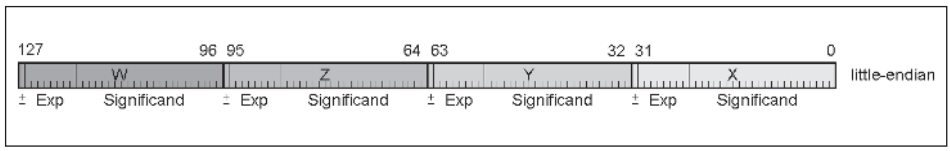


Figure 2-6: Vector of four packed floats in little-endian ordering with their bit encodings exposed

My vector book discusses the concept of vectors in detail but for now, let's just do a superficial overview.

A scalar data value would use a single float, such as in the following addition:

```
 1.0
+2.1
 3.1
```

A single instruction multiple data (SIMD), such as a vertex shader, operates upon a group of data elements, such as the following vector, where a single instruction manipulates one or more data values.

W	Z	Y	X
3.2	1.2	2.5	1.0
+6.3	+5.9	+2.9	+2.1
9.5	7.1	5.4	3.1

This is discussed shortly, but for now just keep in mind that a multiple data item is a group of four floats contained within a 128-bit vector, and each of those items is handled in parallel. Also keep in mind that those four elements (components) can be addressed by an {XYZW} or {RGBA} reference. Mostly, however, for a vertex shader, the {XYZW} reference typically makes more sense since vertices are being accessed in the streaming registers. The {RGBA} makes more sense with the pixel shaders, but keep in mind that there are always exceptions to the rule!

R	G	B	A
3.2	1.2	2.5	1.0
+6.3	+5.9	+2.9	+2.1
9.5	7.1	5.4	3.1

The concept of bits only superficially applies at this time for programmable vertex shaders, as all data is represented by a floating-point value, although it is shown in the figures in a little-endian form. This book only covers the DirectX environment for the X86-based processor, and so it will primarily only discuss little-endian, but we should discuss little-endian versus big-endian orientation of data. The data being imported into the game may be in big-endian form, and it is very important to understand the difference so that manipulation of that data will be correct!

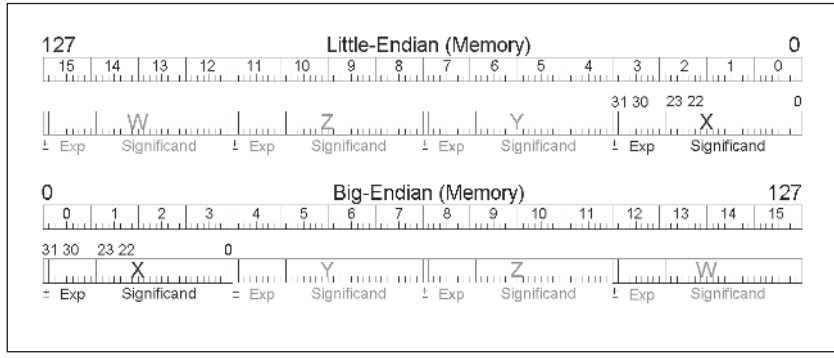


Figure 2-7: Little-endian versus big-endian orientations of single-precision floating-point elements of a 128-bit vector

Another possibility is that you are a non-Microsoft Windows user and are using this book as a shader programming guide, due to the lack of this programming information for your platform, which may be in a big-endian format.

In addition, for those of you who are more advanced programmers, you may find it interesting that the various shader assembly tools export the data in little-endian orientations but in different file formats. From the previous figure, you may have observed that the first four bytes in memory, bytes {0…3}, are related to the X element regardless of the endian orientation! You should also note the byte reversal of the least significant byte and most significant byte of the individual floats between the little-endian and big-endian orientations as well. I hope that should be enough endian for you big-endian-based developers!

Graphics Processor Unit (GPU) Swizzling Data

The programmable vertex and pixel shaders both utilize vector-based registers for their source and destination of values in conjunction with assembly language instructions, and so a method was needed to allow an individual element used as a scalar or up to a full set of four data elements to be written to a destination vector-based register. The use of the following swizzle method is used to shuffle the *source* elements {XYZW} around to make the blending of source data more flexible.

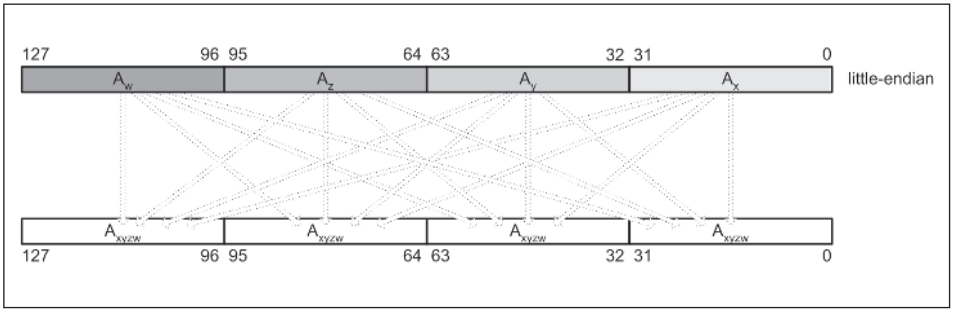


Figure 2-8: Swizzle of floating-point elements within a 128-bit register. Each destination element can be optionally assigned a copy of any of the source elements.

In essence, any set of elements from the source can be selectively used as a source for the computation whose results are stored into the selected element(s) in the destination. It is important to remember that the destination has to be sequenced in an {XYZW} order, but the source elements can be randomly selected! A destination element can be omitted (skipped over). The only exception is when no destination elements are specified and the full resulting {XYZW} vector is copied or the resulting scalar is replicated.

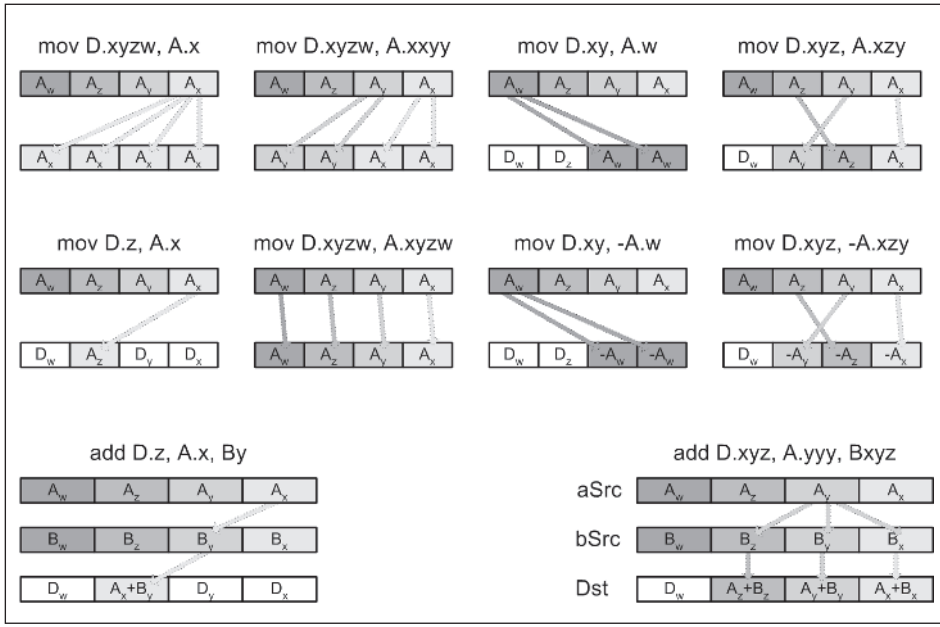


Figure 2-9: Examples of data swizzling

In Figure 2-9, the upper eight diagrams use a *mov* (move) instruction, which takes two arguments. This instruction merely copies the selected second (source) argument *A* to the first (destination) argument *D*. The bottom two diagrams use the summation instruction *add* (addition), which sums the two selected source arguments *A* and *B* and saves the result to argument *D*.

You may notice that in all these cases, when a destination element of a register was *not* specified, the previous result was retained!

You may also have noticed that two of the examples contained a negative sign on the source! This effectively 2's complement (negates) *that* source input passed to the instruction.

2's complement: The process of where all the binary bits are inverted (effectively a not condition) and then the value is incremented.

For those of you who do not understand the operations of a data swizzle thus far or are more advanced and wish to understand all the details of the swizzle operation under the hood, please review Appendix C, "Instruction Dissection." The information supplied there should be sufficient to make the concept of swizzling very simple to understand or, contrarily, help the more advanced developer build his own assembler!

Tools

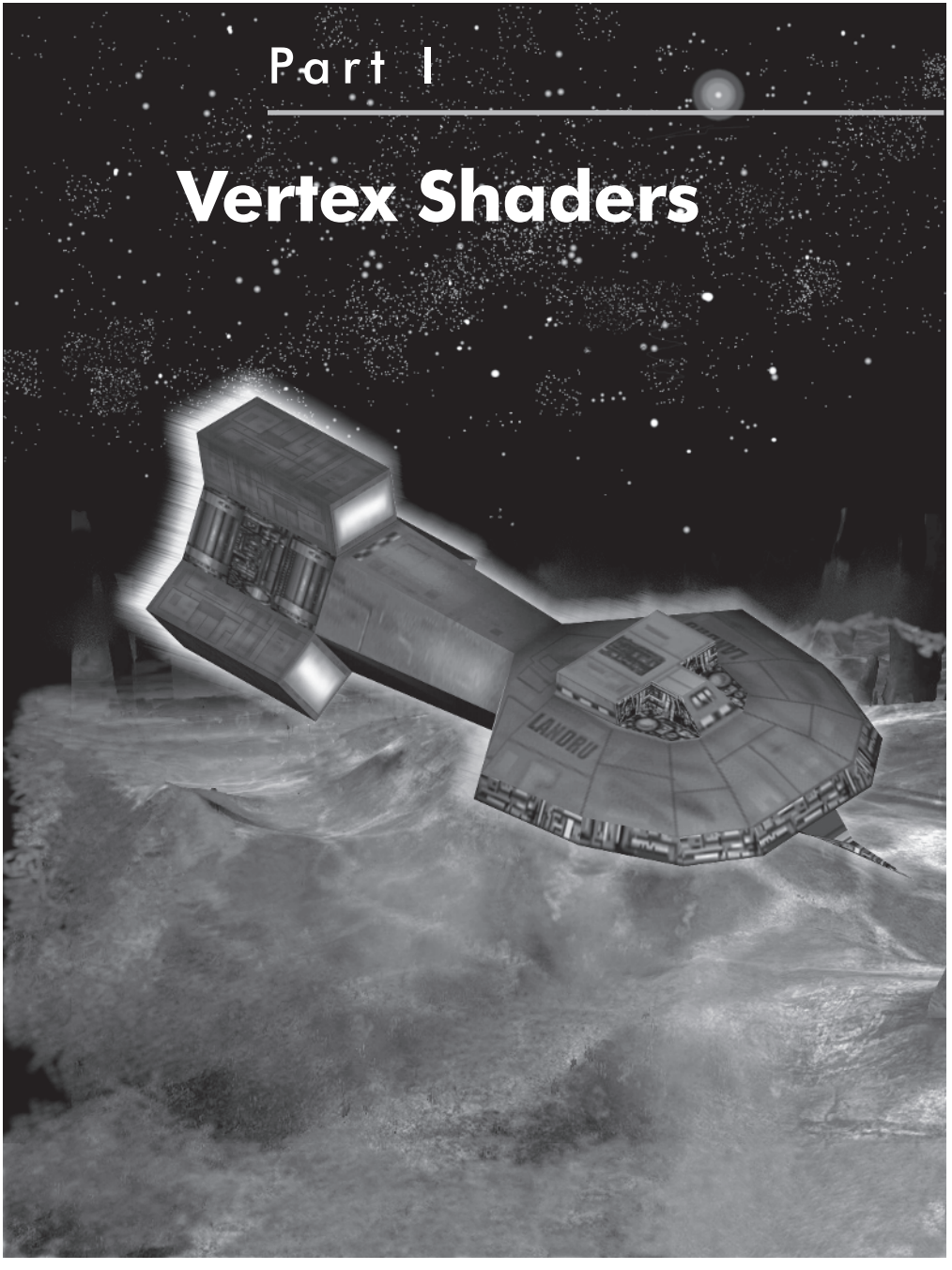
When assembling or compiling your shader code, you will most likely be using one of the following tools:

- nvasm.exe: nVidia –V&P Macro Assembler
- psa.exe: Direct3D 8 Pixel Shader Assembler
- vsa.exe: Direct3D 8 Vertex Shader Assembler
- xsasm.exe: Xbox Shader Assembler
- cg.exe: Cg (C for graphics) compiler is limited in scope to the hardware platform
- fxc.exe: HLSL compiler provided with DX9 SDK, which optimizes code for all hardware platforms
- RenderMonkey.exe: RenderMonkey

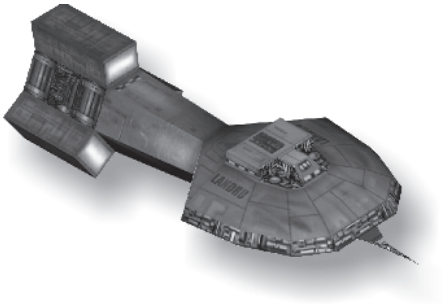
One noteworthy item is that development of nvasm has stalled, so it only supports ps.1.3 and thus has limited use.

Part I

Vertex Shaders



This page intentionally left blank.



Chapter 3

Vertex Shaders

There is so much functionality with the vertex shader that explaining it all takes four chapters, and that is before even getting to quaternions. This chapter covers most of the general functionality as well as scripting methodology. The following chapters cover topics related to other vertex instructions, such as the logic of branching, which is a new feature set introduced with DirectX 9. A topic discussed in the next chapter is what I refer to as branchless code. This is typically logic that one relates to the conditional setting of data.

```
d = (a < b) ? a : b;          // d = min(a, b)
```

Other chapters detail the use of matrices and some trigonometric math in relation to those matrices. So let's begin with the basics!

Vertex Shader Registers

Vertex shaders are used to manipulate vertices, a burden once handled by the CPU processor. Anything that requires vertex movement, such as flapping flags, flowing clothes, bouncing hair, particle fountains, water ripples, etc., can use this programming mechanism instead.

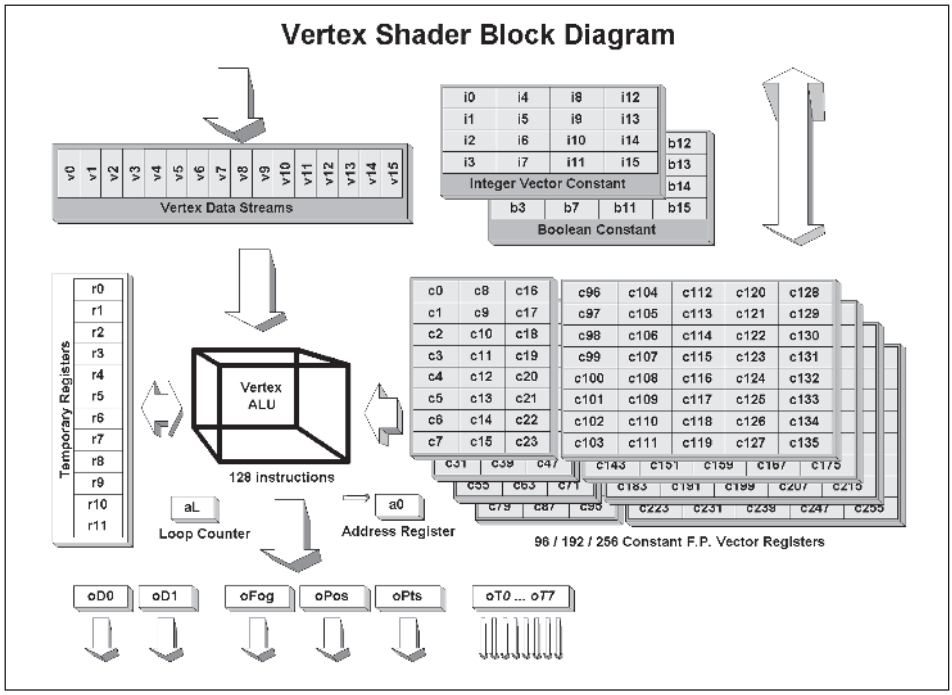


Figure 3-1: Vertex shader block diagram. Note that the grayed out constant registers b_n, c_n, i_n, and vertex stream registers V_n, are all read only from the shader code.

c0 ... c95
c96 ... c255

The standard 96 constant registers (192 on the Radeon 8500 and the additional 160 on the Parhelia for 256 total) are each read-only quad single-precision floating-point vectors. Depending on version, software emulation, and hardware factors, the highest register index can range from {0…8191}. They are set either with the use of the *def* instruction or by calling an external function from an application. Only one vector constant can be used per instruction, but the elements can be negated and/or swizzled. These can only be read by the vertex shader code or from the game application through an API interface. Access is through c[#] or *c*[*a0.x* + #]. For Direct3D, see SetVertexShaderConstantF() under *def* later in this chapter.

i0 ... i15 The standard 16 constant registers are each read-only quad integer vectors. They are set either from the use of the *defi* instruction or from calling an external function from an application. These can only be read by the vertex shader code by a version 2.0 or higher or from the game application through a DX9 API interface. Access is through i[#]. For Direct3D, see SetVertexShaderConstantI() under *defi* later in this chapter.

b0 ... b15 The standard 16 constant registers are each read-only quad Booleans. They are set either from the use of the *defb* instruction or from calling an external function from an application. These can only be read by the vertex shader code by version 2.0 or higher or from the game application through a DX9 API interface. Access is through b[#]. For Direct3D, see SetVertexShaderConstantB() under *defb* later in this chapter.

r0 ... r11 The 12 single-precision floating-point temporary registers are used as scratch registers to temporarily save vertex data in various stages of processing.

v0 ... v15 The 16 read-only vertex data registers each represent a stream of single-precision floating-point and are used as the mechanism to route the data into the Vertex ALU. Only one vertex can be used per instruction, but the elements can be negated and/or swizzled.

aL Loop count register, vs 2.0 and 3.0.

p0 Predicate, vs 2.0 and 3.0.

The output registers are primarily write-only vectors and scalars used to route the processed data to the graphics pipeline for the next pipeline processing stage.

a0	The scalar write-only address register is used as an index offset into the table of registers. It was introduced with version 1.1. Only one use of a0 as a variable index is allowed per instruction. *c*[*a0.x* + #] It can be thought of as a base address plus offset. Moving a register to *a0* used to truncate, but in DX9 it now rounds to the closest integer.
oD0	The *vertex diffuse color register* is a write-only vector that is interpolated and written to the pixel shader color input register *v0*.
oD1	The *vertex specular color register* is a write-only vector that is interpolated and written to the pixel shader color input register *v1*.
oFog	The *vertex fog factor* is a write-only vector of which only the scalar {X} element is interpolated and routed to the fog table.
oPos	The *vertex position register* is a write-only vector that contains the position within homogeneous clipping space.
oPts	The *vertex size register* is a write-only vector of which only the scalar {X} element containing the point size is used.
oT0 ... oT3 *oT4 ... oT7*	*Texture coordinates* {0…7}. These write-only vectors are used as the texture coordinates and routed to the pixel shader. Use {XY} for a 2D texture map. GeForce3 {0…3}, Radeon {0…7}.

Although the programmable vertex shaders only use vector registers for their input and output of values, the source vertex data streams registers (which are talked about later) are preloaded with the contents of the vertex stream data, which actually resides in little-endian-oriented memory. These registers are labeled *v0* through *v15* for the vertex shader.

For register access, each instruction can negate or swizzle the source elements, as discussed in the previous chapter.

r0.xyzw, r1.zxwy	r0.wz, r1.xy	r0.xyzw, –r1.xywz
r0.x = r1.z	r0.w = r1.x	r0.x = –r1.x
r0.y = r1.x	r0.z = r1.y	r0.y = –r1.y
r0.z = r1.w		r0.z = –r1.w
r0.w = r1.y		r0.w = –r1.z

Instruction Modifiers

Now would probably be a good time to discuss an instruction modifier. This is a filter that is applied to the result of the calculation before it is stored to the destination.

	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
_sat = Saturate (0.0 ≤ n ≤ 1.0)								☻	☻

The saturation filter is a value limiter. The results of a calculation are filtered so that if the normalized range of 0.0 through 1.0 is exceeded, the value is inclusively clipped to that limit. This is also known as clamping of a value to the interval n ∈ [0.0, 1.0].

```
if (n < 0.0)
    n = 0.0
else if (n > 1.0)
    n = 1.0
```

This is applied to the individual instructions. So for example, instead of a normal addition such as:

```
add r1, r0, c0          // d = a + b
```

…an instruction modifier is used:

```
add_sat r1, r0, c0      // d = sat(a + b)
```

…so in essence:

$r1_x$=sat($r0_x$+$c0_x$) $r1_y$=sat($r0_y$+$c0_y$) $r1_z$=sat($r0_z$+$c0_z$)
$r1_w$=sat($r0_w$+$c0_w$)

Listing 3-1: Vertex shader

```
add_sat r1.w, r0.y, c0.x    // r1w=sat(r0y+c0x)
```

Vertex Input Streams

There are up to 16 vertex data streams, which works out to 16x4 (16 vectors times four floats each), thus 64 scalar inputs. At a very minimum, a data structure such as {XYZ} would be used.

```
typedef struct _D3DVector
{
  D3DVALUE x;          // v0x
  D3DVALUE y;          // v0y
  D3DVALUE z;          // v0z
} D3DVECTOR;
```

This would be accessible only through the register *v0* and the elements {XYZ}, thus occupying a single stream. A slightly more complex stream mechanism would be as follows, which would actually use multiple streams:

```
struct CUSTOMVERTEX
{
  float x, y, z;       // Vertex position
  float nx, ny, nz;    // Vertex normals
  DWORD color1;        // Diffuse color
  float tu, tv;        // Texture coordinates
};
```

With Direct 3D, each data type is mapped to a vertex stream register, so when remapped as shown below, three streams are utilized: {v0, v1, v2}.

```
DWORD dwDecl[] =
{
  D3DVSD_STREAM(0);
  D3DVSD_REG(0, D3DVSDT_FLOAT3),   // v0xyz Position v0w ignored!
  D3DVSD_REG(4, D3DVSDT_FLOAT3),   // v1xyz Normals
  D3DVSD_REG(7, D3DVSDT_D3DCOLOR), // v1w  Diffuse color
  D3DVSD_REG(8, D3DVSDT_FLOAT2),   // v2xy  Texture coordinates
  D3DVSD_END()                     // v2zw  Ignored!
};
```

This is discussed later, but what is being represented here is that the data structure containing the stream data is mapped to a set of specified vertex data stream registers.

Vertex Shader Instructions

Before beginning, it should be noted that if you are working with Direct3D version 8.0 or 8.1, then only vertex shader instructions for 1.0 through 1.1 are supported. If working with Direct3D version 9.0 or beyond, then instructions up to 2.0 or 3.0 are supported. As version 9.0 is readily available, do not use previous versions of 8.1 or older, as it has restrictions as to what it can do. That, and this book was written specifically for version 9.0, and thus samples may fail.

An item of note is that for each increasing vertex version type, the number of instructions supported is increased!

Vertex shader version	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
Maximum number of instructions	128	256	256	256	512+	512+

Instruction counts for version 2x and above are actually higher, as code looping is supported and the number of executed instructions is effectively increased. For versions 3.0 and above, the minimum limit is set to 512 instructions, but the maximum is specified by the device object's enumeration data member:

```
D3DCAPS9.MaxVertexShader30InstructionSlots
```

Note that there are macros intermixed with the instructions, and they are indicated in italics. You should note the use of the white smiley face ☺ in the following tables, which represents a supported instruction/macro for the specified vertex shader and Direct3D versions. Elsewhere in this book, a shaded smiley face ☻ represents pixel shader functionality, so keep this in mind as you read on.

Table 3-1: Programmable vertex instructions and their relationship with the version of Direct3D and shader versions.

	9.0					
Direct3D	8.1					
Instruction Version	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
Assembly (Scripting) Commands						
dcl_usage	Usage declaration	☺	☺	☺	☺	☺	☺
def	Definition of a FP vector const.	☺	☺	☺	☺	☺	☺
defb	Definition of a Boolean const.		☺	☺	☺	☺	☺
defi	Definition of an Int vector const.		☺	☺	☺	☺	☺
label	A code address location		☺	☺	☺	☺	☺
vs	Version (vertex shader)	☺	☺	☺	☺	☺	☺
Data Conversions							
frc	*Fractional portion of float*	☺	☺	☺	☺	☺	☺
mov	Copy	☺	☺	☺	☺	☺	☺
mova	Copy FP to integer		☺	☺	☺	☺	☺
Add/Sub/Mul/Div							
add	Addition	☺	☺	☺	☺	☺	☺
crs	*Cross product*		☺	☺	☺	☺	☺
dp3	Dot product (Vec)	☺	☺	☺	☺	☺	☺
dp4	Dot product (QVec)	☺	☺	☺	☺	☺	☺
dst	Distance vector	☺	☺	☺	☺	☺	☺
mad	Multiply-Add	☺	☺	☺	☺	☺	☺
mul	Multiply	☺	☺	☺	☺	☺	☺
rcp	Reciprocal	☺	☺	☺	☺	☺	☺
rsq	Reciprocal square root	☺	☺	☺	☺	☺	☺
sub	Subtraction	☺	☺	☺	☺	☺	☺
Special Functions							
exp	*Exponential 2^x full precision*	☺	☺	☺	☺	☺	☺
expp	Exponential 2^x	☺	☺	☺	☺	☺	☺
lit	Lighting	☺	☺	☺	☺	☺	☺
log	*$Log_2(x)$ full precision*	☺	☺	☺	☺	☺	☺
logp	$Log_2(x)$ partial	☺	☺	☺	☺	☺	☺
lrp	Linear interpolation		☺	☺	☺	☺	☺
nop	No operation	☺	☺	☺	☺	☺	☺
pow	2^x		☺	☺	☺	☺	☺

sincos	Sine and cosine		☺	☺	☺	☺	☺
texldl	Texture load with adj. detail					☺	☺

Matrices

m3x2	*Apply 3x2 matrix to vector*	☺	☺	☺	☺	☺	☺
m3x3	*Apply 3x3 matrix to vector*	☺	☺	☺	☺	☺	☺
m3x4	*Apply 3x4 matrix to vector*	☺	☺	☺	☺	☺	☺
m4x3	*Apply 4x3 matrix to vector*	☺	☺	☺	☺	☺	☺
m4x4	*Apply 4x4 matrix to vector*	☺	☺	☺	☺	☺	☺

Flow Control (Branchless)

abs	Absolute		☺	☺	☺	☺	☺
max	Maximum	☺	☺	☺	☺	☺	☺
min	Minimum	☺	☺	☺	☺	☺	☺
nrm	*Normalize*		☺	☺	☺	☺	☺
setp	Set predicate register			☺	☺	☺	☺
sge	Set if (>=)	☺	☺	☺	☺	☺	☺
sgn	*Sign*		☺	☺	☺	☺	☺
slt	Set if (<)	☺	☺	☺	☺	☺	☺

Flow Control (Branching)

if	**If** (Boolean)		☺	☺	☺	☺	☺
if_comp	**If** (comparison)			☺		☺	☺
if_pred	**If** (predicate)			☺		☺	☺
else	If-**else**-endif code block		☺	☺	☺	☺	☺
endif	if-else-**endif** code block		☺	☺	☺	☺	☺
break	Break out of loop			☺		☺	☺
break_comp	Conditional break out of loop			☺		☺	☺
break_pred	Predicate break out of loop			☺		☺	☺
call	Subroutine function call		☺	☺	☺	☺	☺
callnz	Function call if ≠ zero		☺	☺	☺		
callnz_pred	Subroutine call if predicate ≠ 0			☺		☺	☺
ret	Return from subroutine		☺	☺	☺	☺	☺
loop	Start of a **loop**-endloop block		☺	☺	☺	☺	☺
endloop	End of a loop-**endloop** block		☺	☺	☺	☺	☺
rep	Start of a **rep**-endrep block		☺	☺	☺	☺	☺
endrep	End of a rep-**endrep** block		☺	☺	☺	☺	☺

The ☺ indicates that the instruction is supported for that version! Note that the white smiley face represents vertices. The $\#_{sw}$ indicates that only software emulation is supported. The $\#_{x}$ indicates extensions.

Before we dive into the deep end of the pool, I should reiterate that this chapter is not about special effects or any other vertex manipulations but to show the fundamental processing functionality of vectors within the programmable graphics processor unit (GPU).

Assembly (Scripting) Commands

The following are assembly language definitions and not instructions. They are, in essence, scripting commands to the assembler that do not generate code instructions but control the building of that code. They also are used to generate constant data (that is, read-only data).

- **vs:** Definition for the version of the code written for the vertex shader

vs.MajVer.MinVer	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This is an assembly language definition and not an instruction. It is for setting the version for which the code was written and *must* be the first declaration in a code fragment. *MajVer* is the major version number, and *MinVer* is the minor version number of the vertex shader for which the code is targeted. The current range is {1.0, 1.1, 2.0, 2_x, 2_sw, 3.0, 3_sw}. There can only be one version definition per code block. Please note that the version numbers used for this instruction and the version numbers used for the pixel shader discussed in a later chapter have no correlation to each other!

Versions without an appended "sw" are either hardware accelerated or software emulated. Versions with the appended "sw" are only software emulated.

Also note the universal usage of "." and "_".

Listing 3-2: Vertex shader

```
vs.1.1        // Uses 1.1 vertex shader code
vs.2.0
vs.2.x
vs.3.0
vs.3.sw       // Version 3sw software (emulated) vertex shader
vs_3_sw
```

- ***label***: A code address location.

label l#	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This declaration is used to mark a location in code for purposes of branching the program counter of the vertex processor of a particular pipe. A *label* may occur following a *ret* instruction to mark the beginning of a new block of code.

For versions 2.0 and 2_x, the label number (#) must be in a range of {0…15}.

For versions 2_{sw} and 3.0, it must be {0…2047}.

This instruction is discussed in more depth in Chapter 4, "Flow Control."

Listing 3-3: Vertex shader

```
vs.2.0     ; Main shader entry point

nop
call l1
call l5
ret

label l1   ; 1st function entry point
  nop
  ret

label l5   ; 2nd  function entry point
  nop
  ret
```

Style: The label numbers should be ordered but not necessarily enumerated (that is, sequential). This is similar to old BASIC programming, where that language did not support auto-numbering. In this way, space can be maintained for the easy insertion of new code without having labels shuffled in appearance {1, 5, 9, 2, 7, 3, etc.}.

■ ***def***: Definition of a single-precision floating-point vector constant

def *Dst, aSrc, bSrc, cSrc, dSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This declaration is used to set single-precision floating-point vector values within the constant registers (c#) ranging from {0…8191} used by the vertex shader code before it is executed. This instruction must occur *after* the version instruction but *before* any instructions. This is not a programming instruction but a definition, and so it does not use up any of the (128/256/512+) instruction code space. The constant value can only be read by and not written to by the shader code.

Listing 3-4: Vertex shader

```
// Sets c0 with {1.0, 0.0, 2.0, 1.5}
vs.1.1                          // Version 1.1
def  c0, 1.0f, 0.0f, 2.0f, 1.5f   // Set c0 register
```

Those who understand the concept of a constant register can skip this next code snippet. Those who do not, please continue!

To clear up any fuzziness as to the concept of constant data, the following should help. It uses C++ code and a global variable to make its point. Just follow the comments in the code!

Listing 3-5: C++

```
typedef enum {                  // Different kinds of fruit
      iFRUIT_UNDEFINED = 0,
      iFRUIT_APPLE = 1,
      iFRUIT_GRAPE = 2,
         // Insert more fruit here!
      iFRUIT_MAX
} iFRUIT;

iFRUIT Fruit = iFRUIT_GRAPE;   // Preset global variable "Fruit"

// Define our procedure – similar to that of vertex code!
// Note that MyFruit is constant data and thus protected from change!
void PrintBasket(const iFRUIT &MyFruit)
{
cout << "Basket contains fruit: " << MyFruit << endl;

    // The following line will NOT compile! A const cannot
    // be altered!
MyFruit = iFRUIT_UNDEFINED;
}

// Somewhere in our main program call the procedure
PrintBasket(Fruit);
```

As you may have noticed, the global register *Fruit* was preset with an enumerated value, and within the function *PrintBasket()* it was printed but then an attempt was made to clear it. This write attempt generates a compile error as it is not possible to alter a constant. In terms of a vertex shader, writing to a constant is also not possible for similar reasons.

NOTE: Now remember, that was not meant to be condescending to the more advanced programmer reading this book. This book is written as an introduction to shaders, and so it was needed to make certain that the concept was well understood!

An alternative to this method is writing or reading the value directly from a C/C++ application by using the DirectX API. Note that the *def* declaration is considered an immediate definition and will override any previously set value, such as that set by the following API constant access method. This allows the setting of one

or more vectors simultaneously, as opposed to the one at a time that the shader code allows.

IDirect3DDevice9::SetVertexShaderConstantF()

```
HRESULT SetVertexShaderConstantF(
    UINT StartRegister,         // Register c#
    CONST float *pConstantData, // Pointer to array of float vectors
    UINT Vector4fCount          // # of four-float vectors
);
```

If the function succeeds, a return value of D3D_OK will result. If there is an error, then D3DERR_INVALIDCALL will result.

This is similar to that of the following snippet.

Listing 3-6: C++

```
float catPos[] = {1.0f, 0.0f, 1.0f, 0.0f};
UINT nCnt = sizeof(catPos) / (sizeof(float) * 4);  // =1
    // Set the c0 floating-point register
rval = pDev->SetVertexShaderConstantF(0, catPos, nCnt);

    // Alternate methods involve casting a data structure to a
    // float pointer.

    // Copy a vector (four floats) into c7
D3DXVECTOR4  vec;
pDev->SetVertexShaderConstantF(7, (float *)&vec, 1);

    // Copy a matrix (four vectors – 16 floats) into c8...c11
D3DXMATRIXA16  mtx;
pDev->SetVertexShaderConstantF(8, (float *)&mtx, 4);
```

■ ***defb***: Definition of a Boolean constant

defb *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This declaration is used to set a single Boolean value {true : false} within the specified constant register (b#) used by the vertex shader code before it is executed. This must occur *after* the version instruction but *before* any instructions. This is not a programming instruction but a definition and so does not use up any of the (128/256/512+) instruction code space. The constant value can only be read by — not written to — the shader code.

Note that this is only available for shader code versions 2.0 and newer! It is not supported by cards running an older version of shader code. Also of importance is that the definitions of true or false must be lowercase, as this definition is case sensitive.

Note that the Boolean values are used in the conditional branching.

Listing 3-7: Vertex shader

```
vs.2.0                  // Version 2.0
defb  b4, true          // Set b4 register {true : false}
defb  b2, false
```

An alternative to this is writing or reading the value directly from a C/C++ application by using the provided API. Note that the *defb* declaration is considered an immediate definition and will override any previously set value such as that set by the following DirectX constant access method. This allows the setting of one or more vectors simultaneously.

IDirect3DDevice9::SetVertexShaderConstantB()

```
HRESULT SetVertexShaderConstantB(
    UINT StartRegister,        // Register b#
    CONST BOOL *pConstantData, // Pointer to array of Booleans
    UINT BoolCount             // # of Boolean values in an array
);
```

If the function succeeds, a return value of D3D_OK will result. If there is an error, D3DERR_INVALIDCALL will result.

Listing 3-8: C++

```
BOOL bAry[] = {TRUE, FALSE, TRUE};           // b4...b6
UINT nAryCnt = sizeof(bAry) / sizeof(BOOL);  // =3

  // Set Boolean constants  b4...b6
rval = pDev->SetVertexShaderConstantB(4, bAry, nAryCnt);
```

■ ***defi***: Definition of an integer vector constant

defi *Dst, aSrc, bSrc, cSrc, dSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This declaration is used to define vector integer values within the constant registers (i#) by the code of the vertex shader code before it is executed. This must occur *after* the version instruction but *before* any arithmetic instruction. This is not a programming instruction but a definition and so does not use up any of the (128/256/512+) instruction code space. The constant value can only be read by the shader code and not written.

Listing 3-9: Vertex shader

```
vs.2.0                      // Version 2.0
defi  i3, 1, 0, 1, 0        // Set i3 register {1,0,1,0}
```

An alternative to this is writing or reading the value directly from a C/C++ application by using the provided API. Note that the *defi* declaration is considered an immediate definition and will override any previously set value, such as that set by the following DirectX constant access method. This allows the setting of one or more vectors simultaneously.

IDirect3DDevice9::SetVertexShaderConstantI()

```
HRESULT SetVertexShaderConstantI(
    UINT StartRegister,        // Register i#
    CONST int *pConstantData,  // Pointer to array of integer vectors
    UINT Vector4iCount         // # of four-integer vectors
);
```

If the function succeeds, a return value of D3D_OK will result. If there is an error, D3DERR_INVALIDCALL will result.

Listing 3-10: C++

```
integer dog[] = {1, 0, 1, 0,     // i3
                 2, 5, 7, 9};    // i4
  // Set i3 & i4 register
rval = pDev->SetVertexShaderConstantI(3, dog, 2);
```

■ ***dcl_?(usage)?***: Source sampler declarations

dcl *Dst*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺	☺	☺	☺

Up to and including DirectX 8.1, the dcl usage was not implemented, but with the release of DirectX 9, this statement is now required by the vsa assembler for the declaration of input stream data from register v(#). This register has to declare a vertex input usage based upon the D3DDECLUSAGE enumeration type to specify how the vertex input data is being used by the shader code.

```
typedef enum _D3DDECLUSAGE {
    D3DDECLUSAGE_POSITION     = 0,  // dcl_position
    D3DDECLUSAGE_BLENDWEIGHT  = 1,  // dcl_blendweight
    D3DDECLUSAGE_BLENDINDICES = 2,  // dcl_blendindices
    D3DDECLUSAGE_NORMAL       = 3,  // dcl_normal
    D3DDECLUSAGE_PSIZE        = 4,  // dcl_psize
    D3DDECLUSAGE_TEXCOORD     = 5,  // dcl_texcoord
    D3DDECLUSAGE_TANGENT      = 6,  // dcl_tangent
    D3DDECLUSAGE_BINORMAL     = 7,  // dcl_binormal
    D3DDECLUSAGE_TESSFACTOR   = 8,  // dcl_tessfactor
    D3DDECLUSAGE_POSITIONT    = 9,  // dcl_positiont
    D3DDECLUSAGE_COLOR        = 10, // dcl_color
    D3DDECLUSAGE_FOG          = 11, // dcl_fog
    D3DDECLUSAGE_DEPTH        = 12, // dcl_depth
    D3DDECLUSAGE_SAMPLE       = 13  // dcl_sample
} D3DDECLUSAGE;
```

Please note that declarations prior to version 3.0 require the full {xyzw} declaration for each stream type. Versions 3.0 and above allow individual components to be masked. See *dcl_?usage?* (in Chapter 10, "Pixel Shaders") for additional information.

Pseudocode:

```
dcl_position v(m)          – Position {xyz}
dcl_blendweight v(m)
dcl_normal v(m)            – Normals {xyz}
dcl_texcoord0 v(m)         – Texture coordinate {uv}
dcl_texcoord1 v(m)
```

Listing 3-11: Vertex shader

```
dcl_position    v0
dcl_blendweight v1
dcl_normal      v2
dcl_texcoord0   v3
dcl_texcoord1   v4
```

Now that the dcl usage has been assigned to a stream input, it can be used:

```
mov r0.xyz, v0.xyz     // Get position information
```

Vertex Shader Assembly

Now let's peek at the file architecture for this graphics processor assembly language. A vertex shader script (VSH) can exist in a *.vsh file. As such, it would be ordered similar to Listing 3-12.

We have not discussed any instructions as of yet, so ignore them for the most part. For an example of dcl_*usage*:

Listing 3-12: Vertex shader

```
vs.1.1            // The vertex version number

dcl_position v0   // Position register {xyz}
dcl_texcoord v7   // Texture coordinate register {uv}

// Output position   Dx=axbx+cx   Dy=ayby+cy   Dz=azbz+cz   Dw=awbw+cw
mul r1, v0, c3
add oPos, r1, c4

mov oT0.xy, v7   // Copy texture coordinates. {xy} are the {uv}
```

But we are not done yet! The C/C++ needs to set up the foundations for this shader code, so let's take a look at that. The v(#) registers are the stream inputs and thus must define the data as such. So the position {xyz} and texture {uv} data needs to be defined within a custom data structure. Notice that the ordering of the declarations is the same between the shader code and the C/C++ data structure.

```
struct CUSTOMVERTEX_POS_TEX
{
  float  x, y, z;   // v0.xyz - dcl_position {xyz}
  float  tu1, tv1;  // v7.xy  - dcl_texcoord {uv}
};
```

Our custom polygon data would look something like the following:

```
CUSTOMVERTEX_POS_TEX vBoxAry[] =
{   // x      y     z    tu1   tv1
   { -1.0f, -1.0f, 0.0f,  0.0f, 1.0f },  // #0 LL
   {  1.0f, -1.0f, 0.0f,  1.0f, 1.0f },  // #1 LR
   {  1.0f,  1.0f, 0.0f,  1.0f, 0.0f },  // #2 UR
   { -1.0f,  1.0f, 0.0f,  0.0f, 0.0f },  // #3 UL
};
```

Textures are actually discussed later in this book, but basically a texture is a 2D bitmap that gets mapped onto each triangular face. The coordinates of the texture are actually normalized $\{0.0 \leq n \leq 1.0\}$, so the {uv} upper-left corner is addressed as {0.0, 0.0} and the lower-right corner is addressed as {1.0, 1.0}. Regardless of what pixel resolution the texture is, it gets mapped onto the polygon face very nicely.

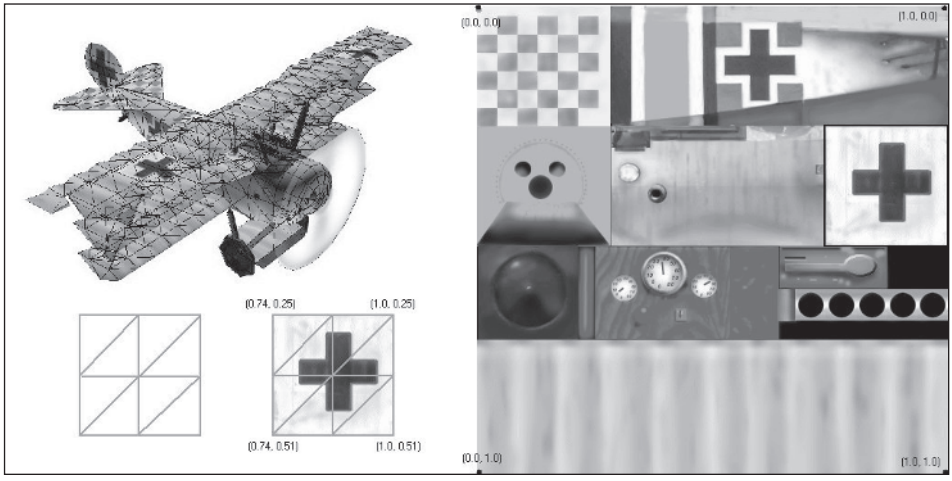


Figure 3-2: Rendered textured (with wireframe overlay) Fokker Dr 1 Triplane (Red Baron) (compliments of Ken Mayfield). The texture is a 512x512. Note the UV location from the source texture as to where the mapped texture resides.

The point here is that all the custom data structure elements are mapped to vertex streams and imported into the vertex shader as a first stage and eventually mapped into the pixel shader as a second stage.

Just for a quick idea of where this is all headed, take a look at the following vertex shader code:

Listing 3-13: DX9SDK\samples\Media\fogshader.vsh

```
// Fog shader code sample from DirectX SDK 9.0
// v0=vector, c8,c9,c10,c11=matrix, c12=limit

// Note the Version (VS) at the top of the file!
  vs.1.1                // Version 1.1

  dcl_position v0

// Here we have the definition (DEF) that was discussed previously.
def c40, 0.0f,0.0f,0.0f,0.0f   // c40={0,0,0,0}

m4x4 r0,v0,c8          // r = v0  [c8,c9,c10,c11]        D=AB
    // r0x = (v0x * c8x)  + (v0y * c8y)  + (v0z * c8z)  + (v0w * c8w)
    // r0y = (v0x * c9x)  + (v0y * c9y)  + (v0z * c9z)  + (v0w * c9w)
    // r0z = (v0x * c10x) + (v0y * c10y) + (v0z * c10z) + (v0w * c10w)
    // r0w = (v0x * c11x) + (v0y * c11y) + (v0z * c11z) + (v0w * c11w)
                       // saturate low of 0.0
                       // r0z = (c40z > r0z) ? c40z : r0z
max r0.z,c40.z,r0.z    // r0z = (0    > r0z) ?   0 : r0z
    // clamp (w) to near clip plane
max r0.w,c12.x,r0.w    // r0w = (c12x > r0w) ? c12x : r0w
mov oPos,r0            // oPosxyzw = r0xyzw

add r0.w,r0.w,-c12.x   // r0w = r0w - c12x
    // Load into diffuse
mul r0.w,r0.w,c12.y    // r0w = r0w * c12y

    // Set diffuse color register
mov oD0.xyzw,r0.w      // oD0x = oD0y = oD0z = oD0w =    r0w
mov oT0.x,r0.w         ;  oT0x = r0w      Set 2D texture X
mov oT0.y,c12.x        // oT0y = c12x     Set 2D texture Y
```

Now, that didn't scare you, did it?

Style: Be consistent! Do not mix and match methods of commenting remarks. Use either the semicolon or the C++ // method. Using both tends to make code look confusing, which is demonstrated in the previous code snippet.

The point here is for you to note that either the assembly language comment designator of ; or the C++ style of comment // can be used for remarks. But for cosmetic reasons, you should really not mix and match but be consistent. Use one or the other but not both! It is just a matter of style.

Let's now move forward!

Vertex Shader Instructions (Data Conversions)

- ***mov***: Copy register data to register d = a

mov *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction copies the referenced source register from *aSrc* to the destination register *Dst*.

Pseudocode:

```
if (a0 <> Src)          // If not the register index offset
    dx=ax  dy=ay  dz=az  dw=aw
else                    // Float to address, so round to integer
   {
    *(int*)&dx = int( floor( a0x ))
    *(int*)&dy = int( floor( a0y ))
    *(int*)&dz = int( floor( a0z ))
    *(int*)&dw = int( floor( a0w ))
}
```

Listing 3-14: Vertex shader

```
mov oD0, c0
mov oD0, r4
mov r1,  c9
mov r0.xz, v0.xx            ; r0x = v0x,    r0z = v0x
mov oPos, r1

def c4, 0.0f, 1.0f, 2.0f, 3.0f
mov r0.yz, c4.yw            ; r0y =  1.0,   r0z =  3.0
```

■ ***mova***: Copy data from floating-point register to address register

mova *Dst*, *aSrc*	1.1	2.0	2x	2sw	3.0	3sw
		☺	☺	☺	☺	☺

This instruction rounds the source floating-point *aSrc* to the nearest integer and stores the result in *Dst*. See *mov* for pseudo conversion.

This instruction is the only method to load the *a0* register with a value for index referencing into the constant array. In versions prior to DX9, the value being copied to the *a0* (index) register is truncated (floor). In version DX9 or later, it is rounded to the nearest integer. A component {XYZW} can be selected instead of the a0.x default.

Listing 3-15: Vertex shader

```
mova a0, c1
mova a0.x, r0.x
```

Note that *Dst* must be register *a0* but can be swizzled and contain one to three elements if vertex version 2.x or newer.

```
mova a0.yz, r1
```

■ ***frc***: Return fractional component of each source input

frc *Dst*, *aSrc*	1.1	2.0	2x	2sw	3.0	3sw
(Macro)	☺	☺	☺	☺	☺	☺

This macro instruction removes the integer component from the source *aSrc*, leaving the fractional component of the elements $d \in [0.0, 1.0)$, which is stored in the destination *Dst*.

Note that for version 1.1, a destination mask of .y and .xy is allowed but not .x, and versions 2.0 and above allow full swizzling capability. The basic functionality is that the real number becomes, in essence, truncated. That is, the whole number portion is removed, leaving only the truncated portion in the range $\{0.0 \leq x < 1.0\}$.

$$\begin{array}{r} 123.\mathbf{456} \\ -\ 123.0 \\ \hline 0.456 \end{array}$$

Pseudocode:

```
float f = 123.456;
int   i = (int) f;

// Version 1.0 - The {ZW} elements are ignored
d_x = a_x – floor(a_x)
d_y = a_y – floor(a_y)

// Version 2.0 and above
d_x = a_x – floor(a_x)
d_y = a_y – floor(a_y)
d_z = a_z – floor(a_z)
d_w = a_w – floor(a_w)
```

Listing 3-16: Vertex shader

```
frc r0.xy, r0.x    ; Fraction of {X} is stored in {XY}
```

Vertex Shader Instructions (Mathematics)

Vector to Vector Summation v + w

The summation of two same-sized vertices is simply the scalar of each element of both vertices that are each summed and then stored in the same element location of the destination vector.

$$v = [v_1 \quad v_2 \quad v_3 \quad v_4] \; w = [w_1 \quad w_2 \quad w_3 \quad w_4]$$

$$v + w = [v_1 + w_1 \quad v_2 + w_2 \quad v_3 + w_3 \quad v_4 + w_4]$$

Equation 3-1: Vector to vector summation: $v+w$

■ ***add***: Addition d = a + b Subtraction d = a + (–b)

add *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction sums each of the specified elements of source *aSrc* and source *bSrc* and stores the result in the destination *Dst*.

Pseudocode:

$d_x=a_x+b_x \quad d_y=a_y+b_y \quad d_z=a_z+b_z \quad d_w=a_w+b_w$

Subtraction is handled by this summation instruction as well as by utilizing the algebraic law of the additive inverse.

Algebraic law:

Additive inverse	a – b = a + (–b)

By using source negation:

$d_x=a_x+(-b_x) \quad d_y=a_y+(-b_y) \quad d_z=a_z+(-b_z) \quad d_w=a_w+(-b_w)$

Listing 3-17: Vertex shader

```
add r0, r0, -c24        // Subtraction
add r0, c23.x, r3       // Addition
add oD0, r0, r1
add r4.x, r4.x, c7.x
```

■ ***sub***: Subtraction d = a – b

sub *Dst*, *aSrc*, *bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction subtracts each of the specified elements of the source *bSrc* from the source *aSrc* and stores the result in the destination *Dst*. This instruction is retired as the *add* instruction, which uses a negation of the source arguments using the algebraic law of the additive inverse and results in the same needed solution! Therefore this instruction is no longer needed! However, it can still be utilized, as the assembler merely remaps it to an *add* instruction with a negated source. A source that is already negated is merely negated again, thus making it a positive source value. See *add*.

3D Cartesian Coordinate System

The 3D Cartesian coordinate system is similar to what you learned in geometry class for describing points and lines (vectors) in three-dimensional space. Each of the three axes {XYZ} are perpendicular to each other. Three planes are constructed with various combinations of the axis (X-Y, Y-Z, X-Z). Three coordinates, as in the following example {10,7,10}, specify a point within 3D space.

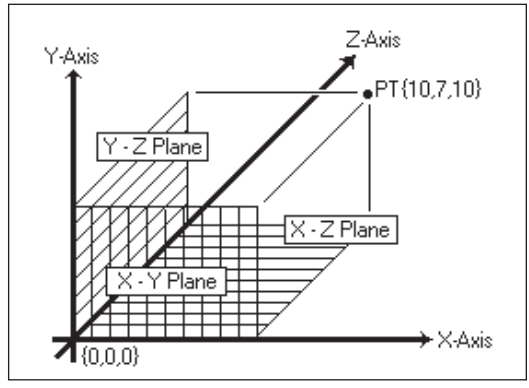


Figure 3-3: 3D Cartesian coordinate system

A line is specified by the difference between two points. In Figure 3-3, the two points are {10,7,10} and {0,0,0}, hence:

{10 - 0, 7 - 0, 10 - 0}. ΔX=10 ΔY=7 ΔZ=10

```
add r0.xyz, v0.xyz, -c0.xyz        // Subtraction
```

■ ***mul***: Multiply d = ab

mul *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in the product of each of the specified elements of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*.

Pseudocode:

```
dx=axbx   dy=ayby   dz=azbz   dw=awbw      ; mul r2,r0,r1
dx=axbx   dy=ayby   dz=azbz                ; mul r0.xyz, r0.xyz, r11.xyz
```

Listing 3-18: Vertex shader

```
mul r2, r0, r1
mul r2, r2, r2
mul r5, r5, c15
mul r0.xyz, r0.xyz, r11.xyz
mul r0,-r7.zxyw,r8.yzxw      ; dx=azby dy=axbz dz=aybx dw=awbw
```

The equation of squares is extremely simple here, as it is the product of itself!

```
mul r0,r0,r0                 ; {r0x2 r0y2 r0z2 r0w2}
```

Q&A: *What is the difference between a cross product and a dot product and what are their equations? Which is also referred to as an outer product and which is an inner product?*

Cross Product v × w

A cross product (*also known as the outer product*) of two vectors is a third vector perpendicular to the plane of the two original vectors. The two vectors define two sides of a polygon face, and their cross product points away from that face. Since multiplication and addition have been discussed, this should be a straightforward implementation of that newly learned knowledge!

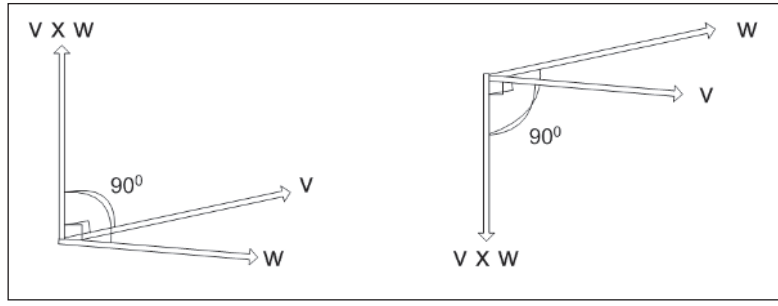


Figure 3-4: Cross product (outer product), the perpendicular to the two vectors v and w

$v = \{v_1, v_2, v_3\}$ and $w = \{w_1, w_2, w_3\}$ are vectors of a plane denoted by matrix R^3. The cross product is represented by the following equation. The standard basic vectors are **i**=(1,0,0) **j**=(0,1,0) **k**=(0,0,1).

$$v \times w = (v_2w_3 - v_3w_2)\mathbf{i} - (v_1w_3 - v_3w_1)\mathbf{j} + (v_1w_2 - v_2w_1)\mathbf{k}$$

$$\det \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ v^1 & v^2 & v^3 \\ w^1 & w^2 & w^3 \end{bmatrix} \text{ thus } \begin{bmatrix} v^2w^3 - v^3w^2 \\ v^3w^1 - v^1w^3 \\ v^1w^2 - v^2w^1 \end{bmatrix}$$

Pseudocode:

Not clear enough? The equation resolves to the following simplified form:

$D_x = A_yB_z - A_zB_y$	`Dx = Ay*Bz - Az*By;`
$D_y = A_zB_x - A_xB_z$	`Dy = Az*Bx - Ax*Bz;`
$D_z = A_xB_y - A_yB_x$	`Dz = Ax*By - Ay*Bx;`

Listing 3-19: Vertex shader

```
mul r1.xyz, v0.zxy, c0.yzx
mul r0.xyz, v0.yzx, c0.zxy
add r0.xyz, r0.xyz, -r1.xyz
```

So with this foundation in mind, let's examine some methods of implementation.

■ *mad*: Multiply Add d = ab + c

mad *Dst, aSrc, bSrc, cSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in the product of each of the specified elements of the source *aSrc* and the source *bSrc,* then sums the elements of the source *cSrc* and stores the result in the destination *Dst*.

Pseudocode:

$d_x=a_xb_x+c_x$ $d_y=a_yb_y+c_y$ $d_z=a_zb_z+c_z$ $d_w=a_wb_w+c_w$

Listing 3-20: Vertex shader

```
mad oT0.xyz, r2, r0, r1
mad oT0,     r1, c8, c3
```

Note that the elements can be negated as a whole and/or crossed:

```
mul r0, r7.zxyw,  r8.yzxw
mad r5, r7.yzxw, -r8.zxyw, r0
```

…so the quad vector multiply: **mul r0, r7.zxyw, r8.yzxw**

$r0_x = r7_z r8_y$
$r0_y = r7_x r8_z$
$r0_z = r7_y r8_x$
$r0_w = r7_w r8_w$

…followed by the multiply-add: **mad r5, r7.yzxw, –r8.zxyw, r0**

$r5_x = –r7_y r8_z + r0_x$
$r5_y = –r7_z r8_x + r0_y$
$r5_z = –r7_x r8_y + r0_z$
$r5_w = –r7_w r8_w + r0_w$

…should look very familiar to you! Well, I hope it does, as it is the cross product that was just discussed!

$D_x = A_y B_z – A_z B_y$	Dx = Ay*Bz – Az*By;
$D_y = A_z B_x – A_x B_z$	Dy = Az*Bx – Ax*Bz;
$D_z = A_x B_y – A_y B_x$	Dz = Ax*By – Ay*Bx;
$D_w = 0.0$	Dw = 0.0;

But alas, we are not done just yet. An instruction was just made available as part of the new version 2 enhancement, and thus it is available to you in DirectX 9.

- ***crs***: Cross product d = a × b

crs *Dst*, *aSrc*, *bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)		☺	☺	☺	☺	☺

This (two-slot) macro instruction results in the cross product (outer product) of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*.

Listing 3-21: Vertex shader

```
mul r0, r7.zxyw,  r8.yzxw
mad r5, r7.yzxw, -r8.zxyw, r0
```

This is done as a single macro. It still requires two instruction slots. As one cannot play without rules, there are some restrictions as to its use:

- Neither *aSrc* nor *bSrc* can also be the destination.
- Neither *aSrc* nor *bSrc* can be swizzled. Only the default *.xyzw* is allowed.
- The destination must be a temporary (r#) register.

- Only one of the following destination masks is allowed: .x .y .z .xy .xz .yz .xyz

Listing 3-22: Vertex shader

```
crs r5.xyz, r7, r8
```

Dot Product

A dot product, also known as an *inner product* of two vectors, is the summation of the results of the product for each of their {XYZ} elements, thus resulting in a scalar. Not to oversimplify it, but this scalar is equal to 0 if the angle made up by the two vectors is perpendicular (=90°), is positive if the angle is acute (<90°), and is negative if the angle is obtuse (>90°).

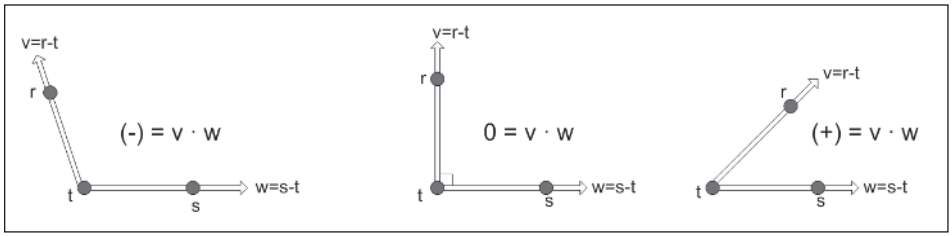


Figure 3-5: Dot product (inner product). A positive number is an acute angle, zero is perpendicular, and negative is an obtuse angle.

$v = \{v_1, v_2, v_3\}$ and $w = \{w_1, w_2, w_3\}$

These are vectors that produce a scalar defined by v • w when their products are combined. The dot product is represented by the following equation:

$$v \bullet w = v_1w_1 + v_2w_2 + v_3w_3$$

The equation resolves to the following simplified form:

$$D = A_xB_x + A_yB_y + A_zB_z \qquad D = Ax*Bx + Ay*By + Az*Bz;$$

This is one of my favorite equations because it does not slice, dice, or chop, but it culls, illuminizes, simplifies; it cosineizes (not a real word, but you know what I mean). It is the Sledge-O-Matic! Well, it's not quite comedian Gallagher's watermelon disintegration kitchen utensil, but it does do many things, and so it is just as useful.

Note in Figure 3-5 that if the resulting scalar value is positive (+), the vectors are pointing in the same general direction. If zero (0), then they are perpendicular to each other. If negative (–), they point in opposite directions.

Before explaining further, it should be pointed out that to keep 3D graphic algorithms as simple as possible, the three vertices for each polygon should all be ordered in the same direction. For example, by using the left-hand rule and keeping all the vertices of a visible face in a clockwise direction, such as in the following diagram, back face culling will result. If all visible face surfaces use this same orientation, then if the vertices occur in a counter-clockwise direction, they are back faced and thus pointing away and need not be drawn, saving render time.

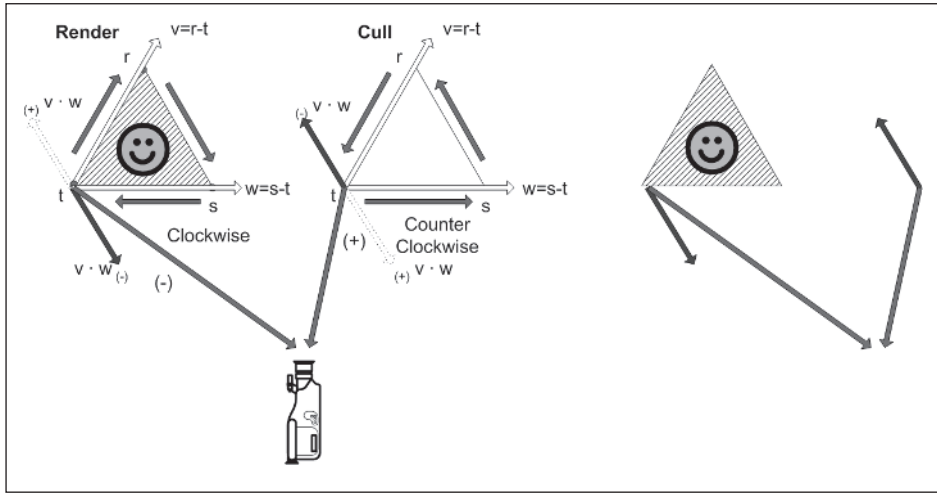


Figure 3-6: Face culling mechanism where if the angle between the camera and the perpendicular to the face plane is obtuse, the face is pointed away from the camera and thus can be culled

Contrarily, if polygons are arranged in a counterclockwise orientation, then the inverse occurs, where a positive value is drawn and a negative value is culled. Keep in mind, however, that most software algorithms keep things in a clockwise orientation.

By calculating the dot product of the normal vector of the polygon with a vector between one of the polygon's vertices and the camera, it can be determined that if the polygon is back facing, it needs to be culled. A resulting positive value indicates that the

face is pointed away, hence back facing, and thus can be culled and not rendered; a negative value is a face oriented toward the camera and thus visible.

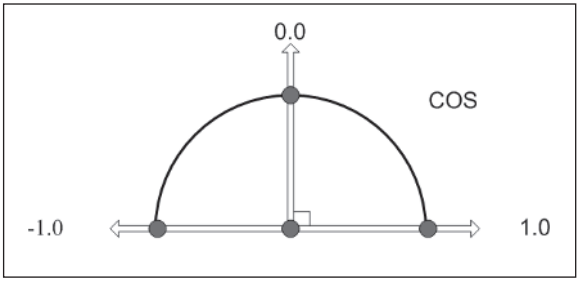


Figure 3-7: Cosine of two intersecting lines

Another use for the dot product equation is that it is also the cosine of the angle. A quick brief here is that the cosine is returned by the reciprocal of the dot product by the product of the magnitudes of the two vectors. Note that *v* and *w* are vectors and |*v*| and |*w*| are their magnitudes.

$$\text{Cos}\,\theta = \frac{A_x B_x + A_y B_y + A_z B_z}{\sqrt{\left(A_x^{\,2} + A_y^{\,2} + A_z^{\,2}\right)}\ \sqrt{\left(B_x^{\,2} + B_y^{\,2} + B_z^{\,2}\right)}} = \frac{v \bullet w}{|v||w|}$$

Equation 3-2: Cosine of the angle

Using standard trigonometric formulas, such as:

$$1 = \text{Cos}^2 + \text{Sin}^2$$

…sine and other trigonometric results can be calculated.

But whoa! There are two problems here. One is that we are beginning to get ahead of ourselves. Two, normally the application will precalculate a transformation matrix with rotations, etc., built in that will then be used by all the vertices in a stream. In that way, we can be efficient. But when we start calculating this within a shader and each shader has to perform the calculation, things do not begin to slow down. They *do* slow down and the idea is to keep the code as light and optimized as possible. So keep in mind that you may want to take advantage of some of this stuff being discussed, but on a limited basis. As you may remember, the constant registers cannot be written to by a shader, and the temporary

registers are garbage between each stream operation upon a vertex, so there is no way to carry over the previous calculations from shader code execution to shader code execution.

■ ***dp3***: Three-element {XYZ} dot product d = a • b

dp3 *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in the dot product of the source *aSrc.xyz* and the source *bSrc.xyz* and stores the replicated scalar result in each element of the destination. The default is *Dst.xyzw*. See *m3x2*, *m3x3*, and *m3x4* in Chapter 5 for use of this instruction in 3xN matrix operations.

Pseudocode:

```
dw=dz=dy=dx=   axbx + ayby + azbz          ; dp3   d,   a, b
dx=            axbx + ayby + azbz          ; dp3   d.x, a, b
```

Listing 3-23: Vertex shader

```
dp3 r2,r0,r1
dp3 r11.x,r0,r0          ; r11x = r0xr0x + r0yr0y + r0zr0z
```

■ ***dp4***: Four-element {XYZW} dot product d = a • b

dp4 *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in the dot product of the source *aSrc.xyzw* and the source *bSrc.xyzw* and stores the replicated scalar result in {w}.

The elements {XYZ} of the destination are left intact. The default is *Dst.w*, but swizzling can override this destination. See *m4x3* and *m4x4* in Chapter 5 for the use of this instruction in 4xN matrix operations.

Pseudocode:

```
dw=   axbx + ayby + azbz + awbw      ; dp4   d,   a, b
dy=    axbx + ayby + azbz + awbw     ; dp4   d.y, a, b
```

Listing 3-24: Vertex shader

```
dp4 r2,   r0,r1
dp4 r5.y, v0, c3      ; r5y=   v0xc3x + v0yc3y + v0zc3z + v0wc3w
```

Q&A: *How does one handle the age-old problem of division by zero?*

Reciprocals

$$\text{Divisor} = \frac{\text{Quotient}}{\text{Dividend}} \quad \text{Remainder}$$

A division is a play on an equation transformation: a multiplication of the dividend by the reciprocal of the divisor.

$$D = A \div B = \frac{A}{1} \div \frac{B}{1} = \frac{A}{1} \bullet \frac{1}{B} = \frac{A}{B}$$

You learned back in grade school that to find the result of a division, one merely takes the reciprocal of the dividend (divisor) and finds the product of that dividend with the quotient. But care needs to be taken when that dividend becomes zero.

$$D = A \div 0 = \frac{A}{1} \div \frac{0}{1} = \frac{A}{1} \bullet \frac{1}{0} = \frac{A}{0}$$

You also learned that you never divide by zero because that would be an undefined answer. Well, that is not quite true. If you were to graph the results of a constant A and slowly reduce the dividend B, the result would become larger and larger. This, in essence, means that as B approaches zero, the result gets closer to ∞ (infinity). Normally, a divide by zero on a processor results in a math exception error, but vector processors and shaders have gone one

better. For purposes of efficiency, if the dividend is exactly 0.0, the largest floating-point value is substituted, FLT_MAX.

Due to limited precision handling with single-precision floating-point, fewer calculations are needed and the longer that accuracy can be kept within reason. The two most precise values within a floating-point value on a computer are the values 0.0 and 1.0. With this in mind, even performing a calculation with 1.0 causes a loss in accuracy. But there is a solution for that as a dividend as well.

Value	Hex	Sign Exp Sig.
–1.0	0xBF800000	1 7F 000000
0.0	0x00000000	0 00 000000
1.0	0x3F800000	0 7F 000000

Note the 23 consecutive bits set to 0. For more in-depth information related to floating-point precision, see Appendix D.

Algebraic law:

Additive identity	n + 0 = 0 + n = n
Multiplicative identity	n1 = 1n = n

If the dividend is 1, then the multiplicative identity shows that.

So:

$$N = 1N = \frac{N}{1} \bullet \frac{1}{1}$$

…therefore:

$$N \div 1 = N$$

Since any number divided by 1 is that same number, the shader code takes this to its advantage. So to maximize accuracy, when the dividend is exactly 1.0, the quotient is returned. For all other quotients, the product of the reciprocal is calculated.

$$\{^{-}\infty < x < 0.0 < x < 1.0 < x < {}^{+}\infty\}$$

■ ***rcp***: Reciprocal of the source scalar d = 1/a

rcp *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in the reciprocal of the source *aSrc* and stores the replicated scalar result in each specified element of the destination. Special case handling is utilized if a source is equal to 1.0 or 0.0. The default is *Dst.xyzw*, Src.x.

Pseudocode:

```
if (0.0 == a_x)                    // 1/0  Divide by zero
    r =   +∞                       // Positive infinity
else if (1.0 == a_x)               // 1/1 = 1
    r =   1.0
else                               // 1/x
    r =   1.0/a_x

d_w=d_z=d_y=d_x=   r
```

Note that unlike most reciprocal instructions, if the denominator is 0, there is no exception (SNaN or QNaN) and/or the data is not marked as invalid! Instead, it is set as a positive infinity, keeping the data valid for additional operations.

Listing 3-25: Vertex shader

```
rcp r2,    r0          ; r2_w=r2_z=r2_y=r2_x=   1/r0_x
rcp r0.z,  r0.z
rcp r1.y,  r1.x
rcp r2.yw, r2.y
rcp r7.w,  r7.w
```

Division

d = a/b = a * 1/b

Listing 3-26: Vertex shader

```
rcp r0.x, r2.x          // d_x = 1/b_x
mul r0.x, r1.x, r0.x    // d_x = a_x/b_x = a_x * 1/b_x
```

Square Roots

The reciprocal and square root are two mathematical operations that have special functionality with vector processors. The division operation is typically performed by multiplying the reciprocal of the denominator by the numerator. A square root is not always just a square root. Sometimes it is a reciprocal square root. So first let's examine some simple forms of these.

$$y \div x = \frac{y}{1} \bullet \frac{1}{x} = \frac{y}{x} = \frac{y^1}{x^1} = y^1 \bullet x^{-1} = y^1 x^{-1}$$

So:

$$\frac{1}{x} = x^{-1}$$

Equation 3-3: Reciprocal

$$\sqrt{x} = x^{1/2} \quad \frac{1}{\sqrt{x}} = x^{-1/2} \quad \text{so} \quad \frac{x}{\sqrt{x}} = x^{1^{-1/2}} = x^{1/2} = \sqrt{x}$$

Another way to remember this is:

$$\frac{x}{\sqrt{x}} = \frac{\sqrt{x} \bullet \sqrt{x}}{\sqrt{x}} = \frac{\sqrt{x} \bullet \sqrt{x}}{\sqrt{x}} = \frac{\sqrt{x}}{1} = \sqrt{x}$$

Equation 3-4: Square root

The simplified form of this scalar instruction individually calculates the square root of the specified floating-point element and returns the result in the destination.

So now I pose a little problem. Hopefully we all know that a negative number should never be passed into a square root because computers go BOOM, as they have no idea how to deal with an identity (**i**).

$$\sqrt{-x} = i\sqrt{x}$$

With that in mind, what is wrong with a reciprocal square root? Remember your calculus and limits?

$$\sum_{x \to 0^+} \frac{1}{\sqrt{x}}$$

HINT: $x \rightarrow 0^+$ (as x approaches zero from the right)

Okay, how about this instead?

$$\sum_{x \rightarrow 0^+} \frac{1}{X}$$

Do you see it now? You normally cannot divide by zero, as it results in infinity and is mathematically problematic. So with a normal CPU, special case handling has to occur. But with a shader, this special case code is handled for you!

If *x* is passed as a negative number, it is passed through an abs() function for you. If *x* is *way* too close to zero, such as specifically being 0.0, then it is conditionally set as close to infinity as possible (in essence, a really high positive value). If it is 1.0, then it is left alone to keep precision loss from affecting it during a reciprocal operation; otherwise, the square root is processed as a reciprocal.

```
ax = abs(ax)
if (0.0 == ax)                  // 1/0   Divide by zero
    r =   +∞                    // Near positive infinity
else if (1.0 == ax)             // 1 = 1/1 = 1/√1
    r =   1.0
else                            // 1/√1
    r =   1.0/√ax
```

It is not perfect, but it is a solution. The number is so close to infinity that the result of its product upon another number is negligible. So, in essence, the result is that other number, thus the multiplicative identity comes to mind (1 * n = n).

So in the case of a reciprocal square root, the square root can be easily achieved by merely multiplying the result by the original *x* value, thus achieving the desired square root. Remember, the square of a square root is the original value!

$$\sqrt{x} = x^{1/2} \quad \frac{1}{x} = x^{-2} \quad \frac{1}{\sqrt{x}} = x^{-1/2}$$

$$\sqrt{x} = \frac{x}{1} \bullet \frac{1}{\sqrt{x}} = x^{1-1/2} = x^{1/2}$$

- ***rsq***: Reciprocal square root of the source scalar $d = 1/\sqrt{|a|}$

rsq *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in the reciprocal square root of the source *aSrc* specified by only one element {.x .y .z .w} and stores the replicated scalar result in each element of the destination. Special case handling is utilized if the source is equal to 1.0 or 0.0. The default is *Dst.xyzw*, aSrc.x.

Pseudocode:

```
dw=dz=dy=dx =   rsqrt(ax)
```

Listing 3-27: Vertex shader

```
rsq r1,   c3          //  r1w = r1z = r1y = r1x =  1/√c3x
rsq r1.y, c3.y        //              r1y =        1/√c3y
```

You should remember that multiplying a reciprocal square root by the original number returns a square root! $\sqrt{x} = x \bullet 1/\sqrt{x}$

```
rsq r1.x, c3.x        //  1/√bx
mul r1.x, r1.x, c3.x  //  ax/√bx
```

Vector Magnitude

This is also known as the 3D Pythagorean theorem. As you know, the shortest distance between two points is typically a straight line. The square of the hypotenuse of a right triangle is equal to the square of each of its two sides, whether in 2D or 3D space. The Pythagorean equation is essentially the distance between two points, in essence the magnitude of their differences.

2D Distance

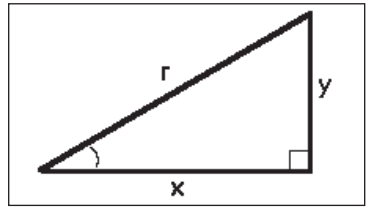


Figure 3-8: 2D right triangle representing a 2D distance

$$x^2+y^2=r^2$$

$$r=\sqrt{(x^2+y^2)}$$

Code:

```
r = sqrt(x*x + y*y);
```

Equation 3-5: 2D distance

3D Distance

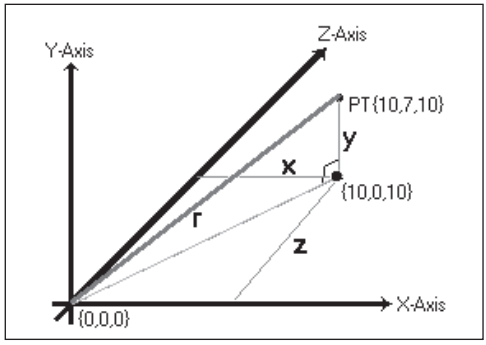


Figure 3-9: Right triangle within a 3D Cartesian coordinate system representing a 3D distance and thus its magnitude

$$x^2+y^2+z^2=r^2$$

$$r=\sqrt{(A_x^2+A_y^2+A_z^2)}$$

Code:

```
r = sqrt(x*x + y*y + z*z);
```

Equation 3-6: 3D distance (magnitude)

Mathematical formula:

Pythagorean	$x^2+y^2=r^2$ $r=\sqrt{(x^2+y^2)}$	$x^2+y^2+z^2=r^2$ $r=\sqrt{(x^2+y^2+z^2)}$

2D Distance	$d(P1,P2)=\sqrt{((x2-x1)^2+(y2-y1)^2)}$
3D Distance	$d(P1,P2,P3)=\sqrt{((x2-x1)^2+(y2-y1)^2+(z2-z1)^2)}$

■ ***dst***: Distance vector

dst *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction calculates the distance between the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*.

The *aSrc* is assumed to be the source vector $\{\#,d^2,d^2,\#\}$ and *bSrc* the vector $\{\#,1/d,\#,1/d\}$, and the result *Dst* is $\{1,d,d^2,1/d\}$. Note that # indicates an *I do not care* condition.

Pseudocode:

```
dx=1.0    dy=ayby    dz=az    dw=bw
```

Listing 3-28: Vertex shader

```
// Find the distance from v1 to the origin {0,0,0}
mov r1.xyz, v1.xyz          // vec = {xyz#} Position
dp3 r1.yz, r1, r1           // d={##yz} = sum of squares
rsq r2.y, r1.y              // {# # 1/√d #}
rcp r2.yw, r2.y             // {√d √d # #} = 1/(1/√d)
dst r0,r1,r2                //  = r1#yz#  r2#yw#
```

Special Functions

We are now coming to the end, with the few remaining non-branch and non-matrix instructions that have special functionality.

■ ***pow***: Power $d = |a|^b$

pow *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This (three-slot) macro instruction occupies three slots and calculates the scalar value using the source *aSrc* as a base value and *bSrc* as an exponent value and stores the replicated result in each component of the destination *Dst*. This is a scalar instruction, thus both the *aSrc* and *bSrc* source arguments require the swizzle of a single replicated component {x, y, z, w}.

The *Dst* register should be a temporary register (r#) and not the same register as *bSrc*.

Pseudocode:

$d_x = d_y = d_z = d_w = |a|^b$

Listing 3-29: Vertex shader

```
pow r3, r4.x, r4.y
pow r2, r0.z, c0.w
```

■ ***expp***: Exponential 2^x — 10-bit precision

expp *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction calculates the exponential number using the source *aSrc* and stores the result in the destination *Dst*.

Pseudocode:

```
uint32 m

w = floor(a_w)
t = pow(2, a_w)
d_x = pow(2, w)
d_y = a_w – w

    // Reduced precision exponent
m = *((uint32*)&t) & 0xffffff00
d_z = *(float*)&m
d_w = 1.0
```

Listing 3-30: Vertex shader

```
expp r1.x, c6.y
expp r5.yw, r5.xxxx
```

■ ***exp***: Exponential 2^x — 21-bit precision

exp *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(vs 1.1 Macro)	☺	☺	☺	☺	☺	☺

This (n-slot — previously a macro) instruction calculates the exponential number using the source *aSrc* and stores the result in the destination *Dst*. See *expp*.

NOTE: Version 1.1 is a ten-slot macro. Version 2.0 and higher is an instruction that only uses one slot. So if you need to use this instruction, by all means try to make sure that your code is targeting at least version 2.0.

Pseudocode:

```
dx = dy = dz = dw = pow(2, aw)
```

Listing 3-31: Vertex shader

```
exp r1.x,   c6.y
```

■ ***lit***: Lighting coefficients — reduced precision

lit *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction calculates the lighting coefficient for the source *aSrc* using two dot products and an exponent and stores the result in the destination *Dst*.

Pseudocode:

```
const float MAXPOWER = 127.9961

ax = Normal · LightVector
ay = Normal · HalfVector
az = 0.0
aw = exponent       // Exponent of ∈ [–128.0, 128.0]

  // The following code fragment shows the operations
```

```
  // performed.
dx = dw = 1.0
dy = dz = 0.0

power = aw

if ((power < –MAXPOWER) || (MAXPOWER < power))
   power = –MAXPOWER                    // 8.8 fixed point format

if (0.0 < ax)                           // positive
{
   dy = a
      // Allowed approx. is EXP(power * LOG(ay))
   if (0.0 < ay)  dz = pow(ay, power)   // positive
}
```

Listing 3-32: Vertex shader

```
lit r1, r0
```

■ ***logp***: $\log_2(x)$ — 10-bit precision

logp *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction calculates a partial log using the source *aSrc* and stores the result in the destination *Dst*.

Pseudocode:

```
v = |aw|

if (0.0 <> v)
{
    int i = (int)(*(DWORD*)&v >> 23) – 127

    dx = (float)I                          // exponent
    i = (*(uint*)&v & 0x7FFFFF) | 0x3f800000
    dy =  *(float*)&i                     // mantissa

    v = log (v) / log (2.0)
```

```
    i = *(uint*)&v & 0xffffff00

    d_z = *(float*)&i;
    d_w = 1.0
}
else                                    // a_w is zero!
{
    d_x = d_z = MINUS_MAX()
    d_y = d_w = 1.0
}
```

Listing 3-33: Vertex shader

```
logp r0, r0.w
```

■ ***log***: $\log_2(x)$ — full precision

log *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(vs 1.1 Macro)	☺	☺	☺	☺	☺	☺

This (n-slot — previously a macro) instruction calculates a full precision log using the source *aSrc* and stores the result in the destination *Dst*. See *logp*.

NOTE: Similar to the exp macro instruction, for shader version 1.1 this was implemented as a (ten-slot) macro instruction. For version 2.0 and later, this is a single-slot instruction. Of course, the same recommendation of targeting at least version 2.0 applies. This will increase rendering speed per vertex.

Pseudocode:

```
v = |aSrc.w|

if (0.0 != v)
  v = (log (v)/log (2))
else
  v = MINUS_MAX()

Dst.x = Dst.y = Dst.z = Dst.w = v
```

Listing 3-34: Vertex shader

```
log r0, r0.w
```

■ ***lrp***: Interpolate between registers

lrp *Dst, aSrc, bSrc, cSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This (two-slot) macro instruction calculates a linear interpolation between the product of the source *aSrc* and the differential of the source arguments *bSrc* less *cSrc*, followed by the summation of the third argument *cSrc*, and the result is stored in the destination *Dst*.

Pseudocode:

$$d_x = a_x (b_x - c_x) + c_x$$
$$d_y = a_y (b_y - c_y) + c_y$$
$$d_z = a_z (b_z - c_z) + c_z$$
$$d_w = a_w (b_w - c_w) + c_w$$

Listing 3-35: Vertex shader

```
lrp r3, r0, r1, r2
```

■ ***texldl***: Texture load

texldl *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
					☺	☺

This instruction combines the texture coordinates referenced by the sampler stage, source *aSrc*, and the source sampler (s#) referenced by *bSrc*.

The mipmap level of detail (LOD) being accessed for this load must be the texture coordinate's fourth element, *aSrc.w.* This is, in essence, the index number of the mipmap with index #0 being the largest mipmap! If the value is negative, then mipmap #0 (the largest mipmap) is referenced.

The *Dst* must be a temporary register *r#.*

bSrc must be register *s#*, be non-negative values, and have been used by the *dcl* declaration! The swizzle functionality can be taken advantage of.

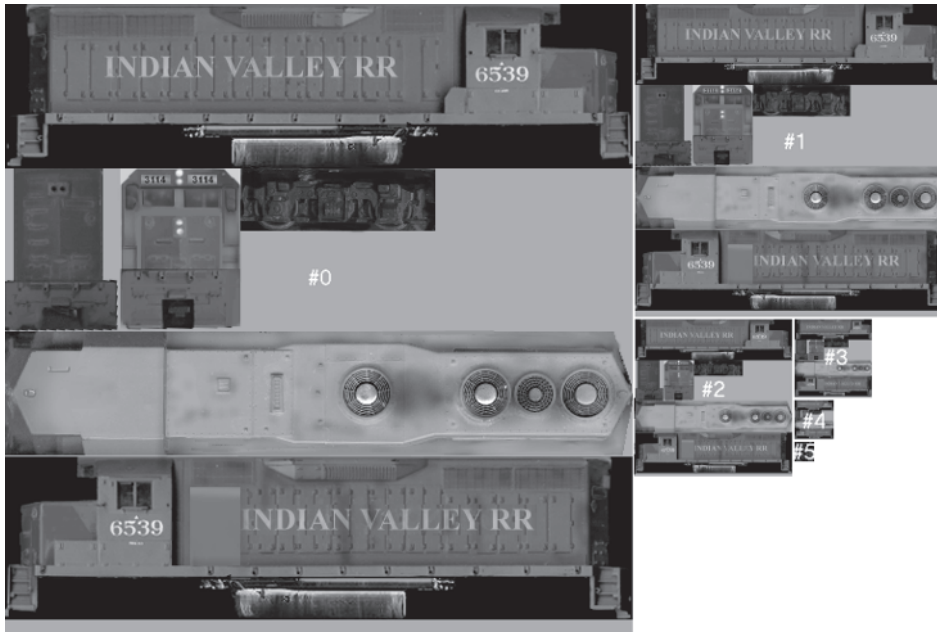
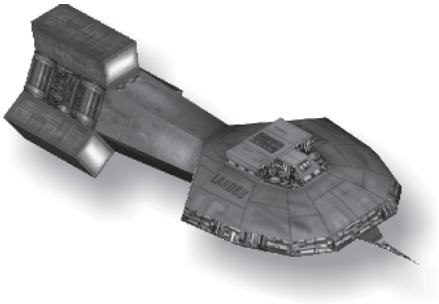


Figure 3-10: Mipmapped GP35 train texture (compliments of Ken Mayfield)

This page intentionally left blank.



Chapter 4

Flow Control

This chapter is related to procedural shaders — the branching and thus controlling the flow of the shader process. It is discussed as two methodologies. The first method is what I like to refer to as *branchless code*. The second is branching code, with which those of you familiar with the C programming language and especially the Microsoft Macro Assembler and X86 assembly language programming would be most familiar.

Branchless Coding

The instructions *abs*, *min, max, slt, sge*, *sgn*, *setp*, and *nrm* can be considered branchless code, as they do a comparison test and substitution without actually branching. These instructions use simple bit masking logic to achieve the desired result. The *slt* and *sge* instructions go one step further, as they return a value of zero or one, which can then be used in conjunction with additional parameters to generate masking logic of numerical expressions that other code may require! Typically, branchless code has to do with optimization (that is, to rework code to make it faster and more efficient by removing branches and substituting alternative logic). In this book's case, the pseudocode functionality of what these instructions do requires branching, although in reality the hardware is doing no branching whatsoever.

■ ***abs***: Absolute d = |a|

abs *Dst*, *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This instruction results in the 2's complement (negation) of a negative value in the source *aSrc* and stores the result in each specified element of the destination. Please note that this instruction is only valid if your vertex version is set to 2.0 or higher! A positive value remains unchanged.

Pseudocode:

```
float f;

if (f < 0.0) f = –f;

dx= |ax|    dy=|ay|    dz=|az|    dw=|aw|
```

Listing 4-1: Vertex shader

```
abs r0,   c0
abs r0.x, c2.z
```

■ ***min***: Minimum d = (a < b) ? a : b

min *Dst*, *aSrc*, *bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in the selection of the *lower* value from each scalar element of the source *aSrc* and *bSrc* and stores the result in the destination *Dst*.

Pseudocode:

```
dx= (ax < bx)  ? ax : bx
dy= (ay < by)  ? ay : by
dz= (az < bz)  ? az : bz
dw= (aw < bw)  ? aw : bw
```

Listing 4-2: Vertex shader

```
min r2, r0, r1
min r0, r0, c4.y
```

■ ***max***: Maximum d = (a > b) ? a : b

max *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in the selection of the *higher* value from each scalar element of the source *aSrc* and *bSrc* and stores the result in the destination *Dst*.

Pseudocode:

$d_x = (a_x > b_x) \ ? \ a_x : b_x$
$d_y = (a_y > b_y) \ ? \ a_y : b_y$
$d_z = (a_z > b_z) \ ? \ a_z : b_z$
$d_w = (a_w > b_w) \ ? \ a_w : b_w$

Listing 4-3: Vertex shader

```
max r2, r0, r1
max r0, r0, c4.y
```

■ ***slt***: Set if less than d = (a < b) ? 1.0 : 0.0

slt *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in a comparison of each selected element of the source *aSrc, bSrc* and stores 1.0 if less than or 0.0 if not in the destination *Dst*. This instruction is similar to the *min* instruction and is very much like the branchless code that I mentioned at the beginning of this chapter. If this were an integer, one would merely take the 0 or 1 resulting from the *set* instruction, subtract a value of one resulting in –1 or 0, which then results in a new value of either all ones or all zeros. This could then be used as a bit mask. But in reality, this instruction is used to process

floating-point values. If you recall your algebraic law of identity, the resulting value calculated from the product of this instruction results in that value or zero.

$$0N = 0 \qquad 1N = N$$

So effectively, we have a branchless mask as well, and that is how we use it.

Pseudocode:

```
dx= (ax < bx)  ? 1.0 : 0.0
dy= (ay < by)  ? 1.0 : 0.0
dz= (az < bz)  ? 1.0 : 0.0
dw= (aw < bw)  ? 1.0 : 0.0
```

Listing 4-4: Vertex shader

```
slt r2,c4,r0
```

- ***sge***: Set if greater than or equal to d = (a >= b) ? 1.0 : 0.0

sge *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
	☺	☺	☺	☺	☺	☺

This instruction results in a comparison of each selected element of the source *aSrc* and *bSrc* and stores 1.0 if greater than or equal to (else 0.0) as the result in the destination *Dst*. This instruction is very similar to the *slt* instruction, except it is the inverse conditional of ≥ instead of <.

Pseudocode:

```
dx= (ax >= bx)  ? 1.0 : 0.0
dy= (ay >= by)  ? 1.0 : 0.0
dz= (az >= bz)  ? 1.0 : 0.0
dw= (aw >= bw)  ? 1.0 : 0.0
```

Listing 4-5: Vertex shader

```
sge r2,c2,r3
```

■ ***sgn***: Sign

sgn *Dst, aSrc, bSrc, cSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)		☺	☺	☺	☺	☺

This (three-slot) macro instruction returns the sign of a value by stripping a value from source *aSrc* of its weight and stores the result in the destination *Dst*. A unit value indicates –1.0 if the value was negative, 0.0 if the value was zero, and 1.0 if the value was positive. This effectively generates a mask.

WARNING: *bSrc* and *cSrc* are temporary scratch registers. Their contents will be destroyed!

Pseudocode:

```
d = –1 if a < 0
d =  0 if a == 0
d =  1 if a > 0
```

Listing 4-6: Vertex shader

```
sgn r1, r0, r10, r11    // Note: r10, r11 scratch Regs
```

■ ***nrm***: 3D vector normalization

nrm *Dst, aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)		☺	☺	☺	☺	☺

This (three-slot) macro instruction calculates the normalization of a 3D vector.

WARNING: As this is a macro instruction, the destination register must be a temporary register, and the source and destination registers cannot be the same register.

$\sqrt{0} = 0 \quad 1/0 = \infty$ (infinity)

The same rules of division apply here, as discussed in Chapter 3.

Pseudocode:

```
r = ax*ax + ay*ay + az*az;
if (r == 0)
   r = FLT_MAX;
else
   r = 1 / sqrt(r);
ay = ay * r;
az = az * r;
```

Please note that the instruction *nrm*, as previously explained, is only available for vertex version 2.0 or higher. If an old shader card is being utilized using hardware rendering, this instruction must be emulated. In the last chapter, a reciprocal was shown as a division:

```
rcp r0.x, r2.x          // dx = 1/bx
mul r0.x, r1.x, r0.x    // dx = ax/bx = ax * 1/bx
```

And a reciprocal square root was shown as a square root:

```
rsq r1.x, c3.x          // 1/bx
mul r1.x, r1.x, c3.x    // ax/bx
```

How to handle the divide by zero in both cases was not discussed. This would normally present a problem because a divide by zero has an invalid solution and quite often it must be trapped and converted to a value of one, as divide by zero is in essence infinity, and a denominator of zero has an infinitesimal effect on a value, so in essence the value remains the same.

But for programmable vertex and pixel shaders, they are trapped and a positive infinity is returned. The interesting thing here is the product of zero and infinity is zero!

HINT: The product of zero and any value (including infinity) is zero!

```
if (0.0 == aw)                          // 1/0   Divide by zero
    r =   +∞                            // Positive infinity
else if (1.0 == aw)                     // 1/1 = 1
    r =   1.0
else                                    // 1/x   or 1/sqrt(x)
    r =   1.0/aw      or      r =   1.0/sqrt(aw)
```

They both require a reciprocal (in essence, a division). But that presents the old problem of a divide by zero, as in the case of an element being too close to zero. When dealing with a normalization, losing precision in the process of manipulating pixels is not a problem.

```
dp3  r0.w, r1.xyz, r1.xyz      // dw=ax2+ay2+az2
rsq  r0.w, r0.w                // dw = 1/√dw
mul  r0.xyz, r1.xyz, r0.w      // {az(1/dw)  ay(1/dw)  ax(1/dw)}
```

If the output of the normalization is being used for additional calculations and precision is required, then use limits (remember your calculus) to trap for the denominator as it approaches zero. To resolve this problem, there is no branching or Boolean masking, which is to the detriment of this assembly code. Hopefully, those instructions will be added to a new version soon. Yet, a normalization is needed, so what can be done?

A comparison in conjunction with a constant can be utilized. Remember that the sum of squares is never a negative number, so the value is d ∈ [0.0, r].

■ ***slt***: Compare less than d = (a < b) ? 1.0 : 0.0

```
def c0, 0.0000001, 0.0000001, 0.0000001, 0.0000001

  // 'r1' too close to zero?
slt r3, r1, c0           // s = (a < 0.0000001) ? 1.0 : 0.0
sge r4, r1, c0           // t = (a >= 0.0000001) ? 1.0 : 0.0

  // Complement masks, d[]=1 if too close to zero, t[]=1 if Not!
  // If too small, 1.0 = (0.00000001*0)+1.0
  // If okay          a   = (a * 1) + 0
mad r0, r1, r4, r3       // d[] = (a[] * t[]) + s[]
```

■ ***setp***: Set predicate

setp_?? *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
setp_gt (a > b)			☺	☺	☺	☺
setp_ge (a ≥ b)			☺	☺	☺	☺
setp_eq (a = b)			☺	☺	☺	☺
setp_ne (a <> b) (a ≠ b)			☺	☺	☺	☺

setp_le (a ≤ b)			☺	☺	☺	☺
setp_lt (a < b)			☺	☺	☺	☺

This instruction results in a per-channel comparison between the source register *aSrc* and the source register *bSrc* and stores the Boolean result in the destination predicate *Dst*. Now remember that when doing an exact equality comparison (a = b) or (a ≠ b), there is the risk of very close numbers not being exactly the same and vice versa due to precision loss, especially when they are processed by a calculation. Check out Appendix D, "Floating-Point 101," for more information.

Pseudocode:

```
p0x = (ax ? bx) ? TRUE : FALSE
p0y = (ay ? by) ? TRUE : FALSE
p0z = (az ? bz) ? TRUE : FALSE
p0w = (aw ? bw) ? TRUE : FALSE
```

Listing 4-7: Vertex shader

```
setp_gt p0, r1, c1
setp_eq p0, r0, c4
```

That predicate can then be used in conjunction with the predicate *if*, *callnz*, and *break* instructions.

HINT: The predicate can be used as a pre-filter on an instruction whereas if an element of the predicate is true, then the result of the correlating element is stored in the destination.

Pseudocode:

For example, a predicate addition would result in the following.

If $p0_?$ is true, then do the calculation for that element, else skip.

```
if (p0x) dx = ax + bx
if (p0y) dy = ay + by
if (p0z) dz = az + bz
if (p0w) dw = aw + bw
```

The NOT symbol ! can be used to invert the logic.

```
if (!p0x) dx = ax + bx
if (!p0y) dy = ay + by
if (!p0z) dz = az + bz
if (!p0w) dw = aw + bw
```

Listing 4-8: Vertex shader

```
(p0) add r3, r2, c6
(!p0) add r3, r2, c6
```

Branching Code

These procedural instructions were introduced with vertex version 2.0, and there are a few simple rules to keep in mind. There are two forms of loops: loop and repeat. Each type of loop has a start marker {*loop*, *rep*} and an end marker {*endloop*, *endrep*}. The if conditional branching also contains an end marker {*endif*}. There are only two methods to escape from a loop. One is not really an escape, as it is a {*call* or *callnz*} instruction, which has an end marker {*ret*}. The *break* instruction is the method of escaping a loop. Except for the *break* instruction, each of these can be thought of as a sub-code block, of which one can be contained within another but not cross connected.

There are limits, however, as the D3DCAPS9.*VS20Caps.Static FlowControlDepth* indicates how deep function calls within loops can nest.

The new integer and Boolean registers are used by these procedural instructions. For purposes of familiarity, this starts with a branch that you should be most familiar with: the if-then conditional.

■ ***if***-endif: (Boolean)

if *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

The branching conditional ***if***-endif is a Boolean conditional. In the C programming language, your code may have looked similar to the following:

Pseudocode:

```
if (16 > a)
{
    // If a is less than 16.
}
```

But how it really works (behind the wrappers) is as follows:

```
Boolean bFlg = (16 > a) ? true : false;

if (true == bFlg)
{
    // If a is less than 16.
}
```

The *if* shader statement is extremely similar, except it does not do a value comparison. Instead it uses one of the Boolean constants defined by *defb* (discussed in Chapter 3) as the Boolean test for the *if*. The other difference here is that instead of braces, an *endif* statement is used to indicate the end of an end block.

■ if-***endif***

endif	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

Listing 4-9: Vertex shader

```
defb b1, TRUE

if b1
   mul r0.xyz, v0, c2.x   // Executed if b1 is true!
else
   mad r2.xyz, v1, c2.y, r0
endif
```

So now let's add another statement that you should be familiar with, the *else* statement.

■ if-*else*-endif

else	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

Pseudocode:

```
if (true == bFlg)
{
    // If a is less than 16.
}
else
{
    // If a is greater or equal to 16.
}
```

Listing 4-10: Vertex shader

```
defb b1, TRUE

if b1
   mul r0.xyz, v0, c2.x   // Executed if b1 is true!
   mad r2.xyz, v1, c2.y, r0
else
   mul r0.xyz, v0, c3.x   // Executed if b1 is false!
   mad r2.xyz, v1, c3.y, r0
endif
```

■ *if_??*-endif (compare)

if_?? *aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
if_gt (a > b)			☺	☺	☺	☺
if_ge (a ≥ b)			☺	☺	☺	☺
if_eq (a = b)			☺	☺	☺	☺
if_ne (a <> b) (a ≠ b)			☺	☺	☺	☺
if_le (a ≤ b)			☺	☺	☺	☺
if_lt (a < b)			☺	☺	☺	☺

This is a bit *more* similar to the C programming language. It actually performs a mathematical comparison between the scalar

source values stored in *aSrc* and *bSrc* and branches based upon the result.

This set of statements can be nested up to 24 levels deep. These can exist outside or inside a loop-based code block but may not cross a loop block boundary.

A component element is required to be selected from *aSrc* and *bSrc* to compare the correct scalar.

When comparing floating-point numbers, care must be taken in discerning which comparisons will behave as expected, especially when trying to see if the two source values are the same (*eq*) or not the same (*ne*).

Having reviewed this new *if* comparison, let's revisit that original comparison: if (16 > a).

Listing 4-11: Vertex shader

```
def c3, 16.0, 0.0, 0.0, 0.0

if_gt c3.x, r0.y
     // Executed if (16 > r0.y)
else
     // Executed if (16 = r0.y)
endif
```

Q&A: *Why can't two floating-point numbers be compared for equality?*

Do not expect the resulting values from different calculations to be identical. For example, 2.0 x 9.0 is about 18.0, and 180.0/10.0 is about 18.0. But the two 18.0 values are not guaranteed to be identical.

For a bumpy but interesting ride on this topic, see Appendix D, "Floating-Point 101."

So the highest accuracy will occur when working with normalized numbers $\{-1.0 \leq x \leq 1.0\}$. Normally, one would not compare two floating-point values except to see if one is greater than the other for purposes of clipping.

So on with the last *if* statement — the *if predicate*!

■ ***if***: endif (predicate)

if_pred *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
			☺	☺	☺	☺

This branch uses the predicate register as the conditional. The ! symbol indicates a NOT condition, and therefore 1's complements the value in the predicate register.

Listing 4-12: Vertex shader

```
if   p0.x
if  !p0.y
```

■ ***rep***-endrep (repeat)

rep *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

The source integer register *aSrc* only uses the constant integers (i#).x, which contains the number of iterations (loops) that occur between the *rep* and *endrep* instructions. The maximum number of loops allowed is 255.

■ rep-***endrep*** (end repeat)

endrep	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

An *endrep* instruction must be used in conjunction with the *rep* instruction and occur at the end of the looped code block.

The repeat loops are not allowed to be nested. When used in conjunction with if statements, the repeat loop must either be a container for the *if* code block or the repeat loop must reside within the *if* block. The *rep* and *endrep* both occupy one instruction slot each.

This can be thought of as a while loop.

Pseudocode:

```
nCount = aSrc.x

while (nCount--)
{
}
```

Listing 4-13: Vertex shader

```
defi i3, 5, 0, 0, 0

rep i3
        // Insert your code here!
endrep
```

■ *loop*-endloop

loop *aSrc*, *bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

The source register *aSrc* is the loop counter *aL* register. The source register *bSrc* is an integer register, where the (i#).*x* component contains the iteration count, the (i#).*y* component contains the initial value of the loop counter, and the (i#).z component contains the incremental value.

■ loop-***endloop***

endloop	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

The *endloop* statement indicates the bottom of a loop. The vertex code between *loop* and *endloop* will cycle up to the value of the loop counter.

Pseudocode:

aL is used as an index into the constant integer array.

```
nCount  = aSrc.x;      // Same as rep, # of loops
aL      = aSrc.y;
iStep   = aSrc.z;

while (nCount--)
{
   aL += iStep;
}
```

Listing 4-14: Vertex shader

```
defi i3, 5, 2, 1, 0   // 5 loops, i2...i6, +1

loop aL, i3
          // Insert your code here using aL index!
endloop
```

■ ***break*** (break out of loop)

break	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
			☺	☺	☺	☺

This instruction is used to break out of repeat and loops and allow the function to execute the instruction just below the *endloop* or *endrep* looping block in which it resides. It has an identical functionality to that of the *break* used in while loops in C.

Listing 4-15: Vertex shader

```
defi i3, 5, 2, 1, 0   // 5 loops, i2...i6, +1

loop aL, i3
                      // Insert your code here using aL index!

  if_gt c3.x, v0.x
    break

endloop
```

■ ***break_??*** (compare break)

break_?? *aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
break_gt if (a > b) break			☺	☺	☺	☺
break_ge if (a ≥ b) break			☺	☺	☺	☺
break_eq if (a = b) break			☺	☺	☺	☺
break_ne if (a <> b) break (a ≠ b)			☺	☺	☺	☺
break_le if (a ≤ b) break			☺	☺	☺	☺
break_lt if (a < b) break			☺	☺	☺	☺

This is a combination of an *if* conditional and a *break* contained within a single instruction. Just like the *if* conditional, an element of both the source *aSrc* and *bSrc* needs to be selected for the individual scalar compare.

Listing 4-16: Vertex shader

```
rep i1
nop
break_gt c3.x, r0.x
endrep
```

■ ***break*** (predicate)

break_pred *aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
			☺	☺	☺	☺

This predicate conditional break uses the predicate register (p0) as the conditional to break out of a loop. The ! symbol indicates a NOT condition, and therefore 1's complements the value in the predicate register.

Listing 4-17: Vertex shader

```
break   p0.x
break   !p0.y
```

■ *call*-ret

call *label*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This instruction is a function call to a code block with an entry point marked by a label and an exit point marked by a *ret* instruction. Labels were discussed in Chapter 3. Similar to how general purpose processors function, the effective address of the next executable instruction is pushed on a stack and execution is branched (jumped) to the address marked by a label. Execution continues from that point forward until a return is encountered.

An indicator is needed to mark the end of called function code, which is the *ret* instruction.

```
vs.2.0
  // base shader code
  call l2
  ret

label l2
  nop
  ret
```

■ call-*ret*

Ret	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This is the exit point of a code block accessed by a *call* instruction. The address pushed onto the stack by the call instruction is popped off the stack, and then the execution is branched (jumped) to.

Listing 4-18: Vertex shader

```
   call l1
   ret

label l1  // insert subroutine code here
   ret
```

In vertex version 2.0, nested subroutine calls are not allowed. In vertex version 3.0, functions can be nested four levels deep!

Pseudocode:

```
call l1
    call l2
        call l3
            call l4
            ret
        ret
    ret
ret
```

■ *callnz*-ret

callnz *label, aSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
		☺	☺	☺	☺	☺

This is similar to a *call* instruction, except it is a conditional call. That is, if the Boolean constant referenced by the source input *aSrc* is NOT zero (thus True), then the function is called.

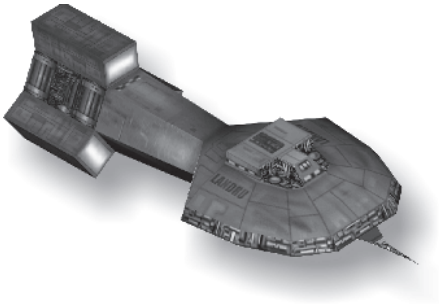
Listing 4-19: Vertex shader

```
callnz l1, b2
```

An alternative argument is to use a scalar element of the predicate (p0).{XYZW} instead of a Boolean. Also similar to the *if* predicate conditional, the NOT of the predicate scalar element can be used as well.

Listing 4-20: Vertex shader

```
callnz - ret
callnz_pred - ret
```



Chapter 5

Matrix Math

Welcome to the Matrix. A place where time is warped by the mind … wait, oh yeah! That was a movie!

This matrix is a wee bit different but can be just as entertaining! Before delving into the vertex shader instructions that support matrix operations, some basics of matrices need to be understood. To keep this chapter from being too big, it is broken into two parts. A mixture of linear algebra (matrices) and 3D rendering technology are brought together in conjunction with vertex shader code in this chapter. Then there is purely trigonometric support (sine-cosine, that sort of thing!) in Chapter 6.

For rendering images, typically two primary orientations of a coordinate system are utilized — the left-handed 3D Cartesian coordinate system on the left or the right-handed on the right, as shown in the following figure.

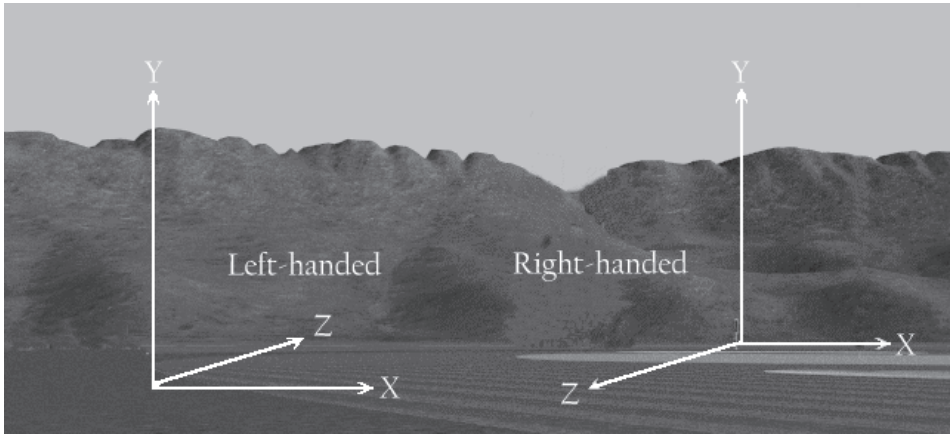


Figure 5-1: Left-handed and right-handed coordinate systems

This is a representation of Euler (pronounced oiler) angles. The following has been done with all sorts of hand manipulations, etc., but I find the following the easiest non-thinking way to remember. In a left-handed system, place your left arm along the x-axis and the hand at the intersection of these axis; the horizontal hitchhiking thumb easily points in the z+ direction. For a right-handed system, do the same with the right hand, and the thumb again easily points in the direction of the z-axis (unless you are double jointed).

The left-handed system has the z-axis increase in a positive direction toward the horizon and is the system that video game system graphic packages, such as Direct3D, typically use. The right-handed system increases the value of the z-axis as it approaches the viewer. This is more along the lines of a high-performance graphics library and standard mathematical conventions. There are variations of these where the z-axis is the height (elevation) information, but this is typically used within some art rendering programs, such as 3D Studio Max.

For purposes of rotation, scaling, and translations of each object within a scene, one of the following two methods is utilized. One method is the use of the vertex, which has been discussed in some detail in previous chapters, and the other method is the use of a quaternion, which is discussed in Chapter 7. Both of these mechanisms use their own implementation of vectors and matrices for the actual multiple-axis transformations of images and their {XYZ} coordinate information.

Vectors

A typical 3D coordinate is contained within an {XYZ} 1x3 column vector, but when extended to a fourth element for an {XYZW} 1x4 column vector, a 1 is typically set for the {W} element. As a note, you should remember that a product identity vector contains an {XYZW} of {0,0,0,1}. When working with translations, the fourth row contains translation (displacement) information, and the {1} in the {W} element allows it to be processed as part of the solution.

A matrix is used to encapsulate simple to complicated mathematical expressions, which can be applied to a scalar, vector, or another matrix. The product, inverse calculations, and summation expressions can all be combined into a single matrix to help minimize the number of overall calculations needed for resolution. These simple mathematical operations resolve to rotations, scaling, and translations of vectors, just to name a few.

Vector to Vector Summation v + w

The summation of two same-sized vertices is simply the scalar of each element of both vertices that are each summed and then stored in the same element location of the destination vector.

$$v = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix} \quad w = \begin{bmatrix} w_1 & w_2 & w_3 & w_4 \end{bmatrix}$$

$$v + w = \begin{bmatrix} v_1 + w_1 & v_2 + w_2 & v_3 + w_3 & v_4 + w_4 \end{bmatrix}$$

Equation 5-1: Vector to vector summation: v+w

The Matrix

Before one can use the shader's matrix instructions one needs to understand their basic functionality. A matrix is an array of scalars that are used in mathematical operations. In the case of this book, only two types of matrices are utilized — a 4x4 matrix denoted by *A* as illustrated below on the left, and a 1x4 matrix *v* on the right, which is used to represent a vector.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad v = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix}$$

Equation 5-2: 4x4 matrix and 1x4 vector

The matrices here are discussed in more depth later in this book, so it is very important that you understand the functionality of a matrix.

Matrices are typically a black box to most game programmers, as they typically cut'n'paste standard matrix algorithms into their

code without really understanding how they work. There may be a similarity between the words "matrix" and "magic," but there are no chants and spells here. It is just mathematics!

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

Equation 5-3: Matrices are arranged into a row column arrangement ($M_{row\ col}$ $A_{row\ col}$). For example, scalar a_{23} is referenced by the second row, third column.

DirectX uses a basic 4x4 matrix of which one is based upon a structure using 1-based indexing: A._23.

Listing 5-1: ...\dx9sdk\include\d3dtypes.h

```
D3DMATRIX mtxA;

 typedef struct _D3DMATRIX {
    float  _11, _12, _13, _14;
    float  _21, _22, _23, _24;
    float  _31, _32, _33, _34;
    float  _41, _42, _43, _44;
 } D3DMATRIX;
```

The other is a two-dimensional float array: A.m[1][2].

```
typedef struct _D3DMATRIX {
   float m[4][4];
} D3DMATRIX;
```

These are actually embedded in the structure as a union but have been individually exposed here for clarity.

Style: Since both matrix implementations are within a single structure, a programmer might be tempted to access different elements using different methods, but this would be a bad programming (style) habit to adopt. Instead keep to one method or another (not both, and especially not in the same function).

```
mtxA.m[3][2] = A._24;
A._24 = 231.0f;
```

A matrix is arranged linearly in memory and sometimes needs to be accessed as a 1x16 array of floats. There are multiple methods of implementation of which the following are only some of them:

```
fpAry = (float*)&mtxA;
fpAry = &mtxA.m[0][0];
fpAry = &mtxA._11;
```

These are similar to the following array. Note that each element of a D3DMATRIX occupies the space of a float.

```
float M[16];
D3DMATRIX A;

  M[0] =A._11  M[1] =A._12  M[2] =A._13 M[3] =A._14
  M[4] =A._21  M[5] =A._22  M[6] =A._23 M[7] =A._24
  M[8] =A._31  M[9] =A._32  M[10]=A._33  M[11]=A._34
  M[12]=A._41  M[13]=A._42  M[14]=A._43  M[15]=A._44
```

This structure is utilized as a base class and has been inherited by another, which is much more versatile, and that is the D3DXMATRIX. (Note the addition of the "X.")

```
typedef struct D3DXMATRIX : public D3DMATRIX
{
public:   // Misc. constructors
    D3DXMATRIX() {};
    D3DXMATRIX(CONST FLOAT *);
    D3DXMATRIX(CONST D3DMATRIX&);
    D3DXMATRIX(FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
               FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
               FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
               FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44);

         // access grants
    FLOAT& operator () (UINT Row, UINT Col);
    FLOAT  operator () (UINT Row, UINT Col) const;

        // casting operators
    operator FLOAT* ();
    operator CONST FLOAT* () const;

         // assignment operators
    D3DXMATRIX& operator *= (CONST D3DXMATRIX&);
    D3DXMATRIX& operator += (CONST D3DXMATRIX&);
    D3DXMATRIX& operator -= (CONST D3DXMATRIX&);
    D3DXMATRIX& operator *= (FLOAT);
```

```
        D3DXMATRIX& operator /= (FLOAT);

              // unary operators
        D3DXMATRIX operator + () const;
        D3DXMATRIX operator - () const;

              // binary operators
        D3DXMATRIX operator * (CONST D3DXMATRIX&) const;
        D3DXMATRIX operator + (CONST D3DXMATRIX&) const;
        D3DXMATRIX operator – (CONST D3DXMATRIX&) const;
        D3DXMATRIX operator * (FLOAT) const;
        D3DXMATRIX operator / (FLOAT) const;

        friend D3DXMATRIX operator *(FLOAT, CONST D3DXMATRIX&);

        BOOL operator == (CONST D3DXMATRIX&) const;
        BOOL operator != (CONST D3DXMATRIX&) const;
    } D3DXMATRIX, *LPD3DXMATRIX;
```

With the DirectX SDK, *D3DXMATRIXA16* can be utilized instead of *D3DXMATRIX* as it uses 16-byte alignment, provided that the Visual C++ compiler is version 7.0 or later or version 6 with an appropriate service pack and a processor pack upgrade.

#define D3DXMATRIXA16 **_ALIGN_16** _D3DXMATRIXA16

The big plus is that the Direct3D functions have been optimized for 3DNow! and SSE. Note that 3DNow! Professional uses the MMX register set. There is a benefit and a drawback. The *benefit* is that the DirectX code was written to handle unaligned memory. The *drawback* is that the code was written to handle unaligned memory. (Huh? Didn't you just read that?) Keep this in mind when you write your applications, as your matrices as well as your vertex stream data *will be more efficient* when aligned properly. All the constant registers, etc., are properly aligned already by the API.

Matrix Copy D = A

A matrix copy $d_{ij}=a_{ij}$ copies each element of a matrix from an ij cell of a source matrix to the equivalent ij cell of a destination matrix. A function such as the following or a memcpy (although a memcpy would be inefficient in this particular implementation) is used to copy that matrix. Also, since data is merely copied, the most efficient memory transfer mechanism available for a processor can be utilized, and it does not necessarily need to be floating-point based, as it is merely tasked with the transfer from memory to memory. Since the matrix is encapsulated as a structure, Visual C++ copies all 16 floats from one matrix to another.

$$[d_{ij}] = [a_{ij}]$$

For shaders, this is as simple as copying each of the four rows, as within each vertex is an element from each of the four columns.

Listing 5-2: Vertex shader

```
mov r0, c0     // row a11...a14
mov r1, c1     // a21...a24
mov r2, c2     // a31...a34
mov r3, c3     // a41...a44
```

Listing 5-3: C++

```
D3DMATRIX mtxA, mtxB;

mtxA = mtxB;
```

Matrix Summation D = A + B

The summation of two same-sized matrices ($c_{ij}=a_{ij}+b_{ij}$) is extremely easy, as the scalar of both matrices is summed and then stored in the same indexed cell location of the same-sized destination matrix.

$$[a_{ij}]+[b_{ij}] = [a_{ij}+b_{ij}]$$

$$A=\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B=\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

$$A+B=\begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & a_{13}+b_{13} & a_{14}+b_{14} \\ a_{21}+b_{21} & a_{22}+b_{22} & a_{23}+b_{23} & a_{24}+b_{24} \\ a_{31}+b_{31} & a_{32}+b_{32} & a_{33}+b_{33} & a_{34}+b_{34} \\ a_{41}+b_{41} & a_{42}+b_{42} & a_{43}+b_{43} & a_{44}+b_{44} \end{bmatrix}$$

Equation 5-4: Matrix to matrix summation: A + B

Algebraic law

Commutative law of addition	a + b = b + a
Commutative law of multiplication	ab = ba

In relationship with the algebraic laws, it is both commutative A+B=B+A and associative A+(B+C)=(A+B)+C, as each element is isolated and no element affects an adjacent element.

The following should be easily recognized as four quad vector summations in parallel.

Listing 5-4: Vertex shader

```
add r4, r0, c0
add r5, r1, c1
add r6, r2, c2
add r7, r3, c3
```

Listing 5-5: C++

```
void Func(D3DMATRIX &mtxD, D3DMATRIX &mtxA, D3DMATRIX &mtxB)
{
mtxD.m[0][0]=mtxA.m[0][0]+mtxB.m[0][0];
mtxD.m[0][1]=mtxA.m[0][1]+mtxB.m[0][1];
     :            :            :
  mtxD.m[3][2]=mtxA.m[3][2]+mtxB.m[3][2];
mtxD.m[3][3]=mtxA.m[3][3]+mtxB.m[3][3];
}
```

DirectX C++ prototype:

```
D3DXMATRIX mtxD, mtxA, mtxB;
mtxD = mtxA + mtxB;
```

Scalar Matrix Product rA

$r[a_{ij}]$

In a scalar multiplication, a scalar is applied to each element of a matrix, and the resulting product is stored in a same-size matrix.

$$rA = \begin{bmatrix} ra_{11} & ra_{12} & ra_{13} & ra_{14} \\ ra_{21} & ra_{22} & ra_{23} & ra_{24} \\ ra_{31} & ra_{32} & ra_{33} & ra_{34} \\ ra_{41} & ra_{42} & ra_{43} & ra_{44} \end{bmatrix}$$

Equation 5-5: Scalar matrix multiplication: *rA*

In the following, the scalar is represented by c0.x, and either c0.x or c0.xxxx can be used, as they both represent the same replication of {X}!

Listing 5-6: Vertex shader

```
mul r4, r0, c0.x
mul r5, r1, c0.x
mul r6, r2, c0.x
mul r7, r3, c0.x
```

Listing 5-7: C++

```
void Func(D3DMATRIX &mtxD, D3DMATRIX &mtxA, float r)
{
   mtxD.m[0][0]=mtxA.m[0][0]*r;
   mtxD.m[0][1]=mtxA.m[0][1]*r;
      :            :
   mtxD.m[3][2]=mtxA.m[3][2]*r;
   mtxD.m[3][3]=mtxA.m[3][3]*r;
}
```

DirectX C++ prototype:

```
D3DXMATRIX mtxD, mtxA;
float r;
mtxD = mtxA * r;
```

Apply Matrix to Vector (Multiplication) vA

A vector can be transformed into another vector by using a 4x4 matrix. When a vector is applied to a matrix, the product of each scalar of the vector and each scalar of a column of the matrix is summed, and the total is stored in the destination vector of the same element of the source vector. The first expression uses the highlighted scalars. Since a vector has four elements, there are four expressions.

$$w_i = v_1 a_{1i} + v_2 a_{2i} + v_3 a_{3i} + v_4 a_{4i}$$

$$w_1 = v_1 a_{11} + v_2 a_{21} + v_3 a_{31} + v_4 a_{41}$$

$$v = \begin{bmatrix} v_1 & v_2 & v_3 & v_4 \end{bmatrix} \quad A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}$$

$$\begin{aligned} w_1 &= v_1 a_{11} + v_2 a_{21} + v_3 a_{31} + v_4 a_{41} \\ w_2 &= v_1 a_{12} + v_2 a_{22} + v_3 a_{32} + v_4 a_{42} \\ w_3 &= v_1 a_{13} + v_2 a_{23} + v_3 a_{33} + v_4 a_{43} \\ w_4 &= v_1 a_{14} + v_2 a_{24} + v_3 a_{34} + v_4 a_{44} \end{aligned}$$

Equation 5-6: Apply matrix to vector (multiplication): $v \times A$

■ ***m4x4***: Apply 4x4 matrix to vector d = aB

m4x4 *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)	☺	☺	☺	☺	☺	☺

This macro applies a 4x4 matrix referenced by the four sequential registers beginning with the source *bSrc* {+0, …, +3} to the {XYZW} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*.

$$w_i = v_1a_{1i} + v_2a_{2i} + v_3a_{3i} + v_4a_{4i}$$

In the following, *a* is the vector, *b*[0] is the first row, *b*[1] is the second row, *b*[2] is the third row, and *b*[3] is the fourth row of the matrix.

$$d_x = (a_x * b[0]_x) + (a_y * b[1]_x) + (a_z * b[2]_x) + (a_w * b[3]_x)$$
$$d_y = (a_x * b[0]_y) + (a_y * b[1]_y) + (a_z * b[2]_y) + (a_w * b[3]_y)$$
$$d_z = (a_x * b[0]_z) + (a_y * b[1]_z) + (a_z * b[2]_z) + (a_w * b[3]_z)$$
$$d_w = (a_x * b[0]_w) + (a_y * b[1]_w) + (a_z * b[2]_w) + (a_w * b[3]_w)$$

The vector is v0, and the matrix is c4…c7. Watch your column and row access when calculating the products. Also note that the source and destination are not one and the same. If they were, then the source would become contaminated by each sequential instruction.

Listing 5-8: Vertex shader

```
m4x4 r5,v0,c4
```

Macro equivalent:

```
dp4 r5.x, v0, c4          ; 1st row
dp4 r5.y, v0, c5          ; 2nd row
dp4 r5.z, v0, c6          ; 3rd row
dp4 r5.w, v0, c7          ; 4th row
```

Note that in the following code, just in case the source and destination vectors are the same, a temporary vector is used and the finished product is copied to the destination!

Listing 5-9: C++

```
void QVecApplyMatrix(D3DXVECTOR4 &vD,
                     D3DXVECTOR4 &vA,
                     D3DMATRIX   &mtxB)
{
  D3DVECTOR4 vT;

  vT.x = (mtxB.m[0][0] * vA.x) + (mtxB.m[1][0] * vA.y)
       + (mtxB.m[2][0] * vA.z) + (mtxB.m[3][0] * vA.w);
  vT.y = (mtxB.m[0][1] * vA.x) + (mtxB.m[1][1] * vA.y)
       + (mtxB.m[2][1] * vA.z) + (mtxB.m[3][1] * vA.w);
  vT.z = (mtxB.m[0][2] * vA.x) + (mtxB.m[1][2] * vA.y)
       + (mtxB.m[2][2] * vA.z) + (mtxB.m[3][2] * vA.w);
```

```
    vD.w = (mtxB.m[0][3] * vA.x) + (mtxB.m[1][3] * vA.y)
         + (mtxB.m[2][3] * vA.z) + (mtxB.m[3][3] * vA.w);

    vD.x = vT.x;   // Now copy final product.
    vD.y = vT.y;
    vD.z = vT.z;
    vD.w = vT.w;
  }
```

Other product terms resulting from different sized matrices will upon occasion need to be calculated. They have been included as shader macro instructions and are discussed in this chapter as well.

■ ***m4x3***: Apply 4x3 matrix to vector d = aB

m4x3 *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)	☺	☺	☺	☺	☺	☺

This macro applies a 4x3 matrix referenced by the three sequential registers beginning with the source *bSrc* {+0, +1, +2} to the {XYZW} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*.

$$w_i = v_1 a_{1i} + v_2 a_{2i} + v_3 a_{3i} + v_4 a_{4i}$$

In the following, *a* is the vector, *b*[0] is the first row, *b*[1] is the second row, and *b*[2] is the third row of the matrix.

$$d_x = (a_x * b[0]_x) + (a_y * b[1]_x) + (a_z * b[2]_x) + (a_w * b[3]_x)$$
$$d_y = (a_x * b[0]_y) + (a_y * b[1]_y) + (a_z * b[2]_y) + (a_w * b[3]_y)$$
$$d_z = (a_x * b[0]_z) + (a_y * b[1]_z) + (a_z * b[2]_z) + (a_w * b[3]_z)$$

Listing 5-10: Vertex shader

```
m4x3 r5,v0,c3          ;c3 1st row, c4 2nd row, c5 3rd row
```

Macro equivalent:

```
dp4 r5.x, v0, c3   ; 1st row
dp4 r5.y, v0, c4   ; 2nd row
dp4 r5.z, v0, c5   ; 3rd row
```

■ ***m3x2***: Apply 3x2 matrix to vector d = aB

m3x2 *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)	☺	☺	☺	☺	☺	☺

This macro applies a 3x2 matrix of the two sequential registers beginning with the source *bSrc* {+0, +1} to the {XYZ} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*.

$$w_i = v_1 a_{1i} + v_2 a_{2i} + v_3 a_{3i}$$

In the following, *a* is the vector, *b*[0] is the first row, and *b*[1] is the second row of the matrix.

$$d_x = (a_x * b[0]_x) + (a_y * b[1]_x) + (a_z * b[2]_x)$$
$$d_y = (a_x * b[0]_y) + (a_y * b[1]_y) + (a_z * b[2]_y)$$

Listing 5-11: Vertex shader

```
m3x2 r5,v0,c3           ;c3 1st row, c4 2nd row
```

Macro equivalent:

```
dp3 r5.x, v0, c3    ; 1st row
dp3 r5.y, v0, c4    ; 2nd row
```

■ ***m3x3***: Apply 3x3 matrix to vector d = aB

m3x3 *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)	☺	☺	☺	☺	☺	☺

This macro applies a 3x3 matrix referenced by the three sequential registers beginning with the source *bSrc* {+0, +1, +2} to the {XYZ} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*.

$$w_i = v_1 a_{1i} + v_2 a_{2i} + v_3 a_{3i}$$

In the following, *a* is the vector, *b*[0] is the first row, *b*[1] is the second row, and *b*[2] is the third row of the matrix.

$d_x = (a_x * b[0]_x) + (a_y * b[1]_x) + (a_z * b[2]_x)$
$d_y = (a_x * b[0]_y) + (a_y * b[1]_y) + (a_z * b[2]_y)$
$d_z = (a_x * b[0]_z) + (a_y * b[1]_z) + (a_z * b[2]_z)$

Listing 5-12: Vertex shader

```
m3x3 r5,v0,c3       ;c3 1st row, c4 2nd row, c5 3rd row
```

Macro equivalent:

```
dp3 r5.x, v0, c3    ; 1st row
dp3 r5.y, v0, c4    ; 2nd row
dp3 r5.z, v0, c5    ; 3rd row
```

■ ***m3x4***: Apply 3x4 matrix to vector d = aB

m3x4 *Dst, aSrc, bSrc*	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)	☺	☺	☺	☺	☺	☺

This macro applies a 3x4 matrix referenced by the four sequential registers beginning with the source *bSrc* {+0, …, +3} to the {XYZ} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*.

$$v_1 a_{1i} + v_2 a_{2i} + v_3 a_{3i}$$

In the following, *a* is the vector, *b*[0] is the first row, *b*[1] is the second row, *b*[2] is the third row, and *b*[3] is the fourth row of the matrix.

$d_x = (a_x * b[0]_x) + (a_y * b[1]_x) + (a_z * b[2]_x)$
$d_y = (a_x * b[0]_y) + (a_y * b[1]_y) + (a_z * b[2]_y)$
$d_z = (a_x * b[0]_z) + (a_y * b[1]_z) + (a_z * b[2]_z)$
$d_w = (a_x * b[0]_w) + (a_y * b[1]_w) + (a_z * b[2]_w)$

Listing 5-13: Vertex shader

```
m3x4 r5,v0,c3       ;c3 1st row, c4 2nd row, c5 3rd row, c6 4th row
```

Macro equivalent:

```
dp3 r5.x, v0, c3    ; 1st row
dp3 r5.y, v0, c4    ; 2nd row
dp3 r5.z, v0, c5    ; 3rd row
dp3 r5.w, v0, c6    ; 4th row
```

Matrix Multiplication D = AB

Hopefully you will not try to implement the following within shader code, as it becomes time consuming and (as explained in a previous chapter) has to be recalculated for each individual vertex. Therefore, it should be precalculated and stored into constants for use by shaders. Because this is an introductory book and this information is necessary to build shader code, it is included here.

The following demonstrates the calculation of a product of two 4x4 matrices ($d_{ik}=a_{ij}b_{jk}$). The *j* indices represent the Einstein summation for all indices of *i* and *k*. The first expression uses the highlighted scalars.

$$d_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + a_{i3}b_{3j} + a_{i4}b_{4j}$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}$$

Equation 5-7: Matrix to matrix multiplication: *AB*

Each of the 16 resulting scalars is the product summation of each scalar in a row of matrix *A* and a scalar from a column of matrix *B*.

Pseudo vec:

$$d_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$
$$d_{12} = a_{11}b_{12} + a_{12}b_{22} + a_{13}b_{32} + a_{14}b_{42}$$
$$d_{13} = a_{11}b_{13} + a_{12}b_{23} + a_{13}b_{33} + a_{14}b_{43}$$
$$d_{14} = a_{11}b_{14} + a_{12}b_{24} + a_{13}b_{34} + a_{14}b_{44}$$

$$d_{21} = a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41}$$
$$d_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42}$$

$$d_{23} = a_{21}b_{13} + a_{22}b_{23} + a_{23}b_{33} + a_{24}b_{43}$$
$$d_{24} = a_{21}b_{14} + a_{22}b_{24} + a_{23}b_{34} + a_{24}b_{44}$$

$$d_{31} = a_{31}b_{11} + a_{32}b_{21} + a_{33}b_{31} + a_{34}b_{41}$$
$$d_{32} = a_{31}b_{12} + a_{32}b_{22} + a_{33}b_{32} + a_{34}b_{42}$$
$$d_{33} = a_{31}b_{13} + a_{32}b_{23} + a_{33}b_{33} + a_{34}b_{43}$$
$$d_{34} = a_{31}b_{14} + a_{32}b_{24} + a_{33}b_{34} + a_{34}b_{44}$$

$$d_{41} = a_{41}b_{11} + a_{42}b_{21} + a_{43}b_{31} + a_{44}b_{41}$$
$$d_{42} = a_{41}b_{12} + a_{42}b_{22} + a_{43}b_{32} + a_{44}b_{42}$$
$$d_{43} = a_{41}b_{13} + a_{42}b_{23} + a_{43}b_{33} + a_{44}b_{43}$$
$$d_{44} = a_{41}b_{14} + a_{42}b_{24} + a_{43}b_{34} + a_{44}b_{44}$$

In relationship with the algebraic laws, it is not commutative but associative. Use the representation most familiar to you (mathematicians on the left and C programmers on the right).

AB <> BA AB != BA

That should make all of you happy!

Thus, the ordering of matrices A versus B needs to be considered when performing this operation.

The following C code is an example of where individual floats are processed.

```
Loop {0...3}
 Dx = AxB[0]x   + AyB[1]x   + AzB[2]x   + AwB[3]x
 Dy = AxB[0]y   + AyB[1]y   + AzB[2]y   + AwB[3]y
 Dz = AxB[0]z   + AyB[1]z   + AzB[2]z   + AwB[3]z
 Dw = AxB[0]w   + AyB[1]w   + AzB[2]w   + AwB[3]w
 D += 1;   A += 1;
```

If the multiplication and sum have to be resolved separately, then the four equations need to be resolved vertically.

```
Loop {0...3}
 Dxyzw =         AxB[0]x     AxB[0]y     AxB[0]z     AxB[0]w
 Dxyzw = Dxyzw + AyB[1]x     AyB[1]y     AyB[1]z     AyB[1]w
 Dxyzw = Dxyzw + AzB[2]x     AzB[2]y     AzB[2]z     AzB[2]w
 Dxyzw = Dxyzw + AwB[3]x     AwB[3]y     AwB[3]z     AwB[3]w
 D++;  A++;
```

The following code shows the code unrolled into a multiply-add supported vector organization. This should look a little familiar to you, as it was recently discussed! Remember applying a matrix to

a vector in shader instruction *m4x4*. Keep in mind r0…r3 is one source matrix, c4…c7 is another source matrix, and r4…r7 is a destination matrix.

Listing 5-14: Vertex shader

```
m4x4  r4,r0,c4
m4x4  r5,r1,c4
m4x4  r6,r2,c4
m4x4  r7,r3,c4
```

That did not look too bad, did it? How about if we unpack those macros!

Macro equivalent:

```
dp4 r4.x, r0, c4          ; 1st row
dp4 r4.y, r0, c5          ; 2nd row
dp4 r4.z, r0, c6          ; 3rd row
dp4 r4.w, r0, c7          ; 4th row

dp4 r5.x, r1, c4          ; 1st row
dp4 r5.y, r1, c5          ; 2nd row
dp4 r5.z, r1, c6          ; 3rd row
dp4 r5.w, r1, c7          ; 4th row

dp4 r6.x, r2, c4          ; 1st row
dp4 r6.y, r2, c5          ; 2nd row
dp4 r6.z, r2, c6          ; 3rd row
dp4 r6.w, r2, c7          ; 4th row

dp4 r7.x, r3, c4          ; 1st row
dp4 r7.y, r3, c5          ; 2nd row
dp4 r7.z, r3, c6          ; 3rd row
dp4 r7.w, r3, c7          ; 4th row
```

Now imagine every vertex having to be run through that bit of shader code. This should sink the point home; for fast rendering, precalculate as much as possible before executing that shader code!

Listing 5-15: C++

```
void Func(D3DMATRIX &mtxD, D3DMATRIX &mtxA, D3DMATRIX &mtxB)
{
  D3DMATRIX mtxT;

  for (uint u = 0; u < 4; u++)
  {
    for (uint v = 0; v < 4; v++)
    {
      mtxT.m[u][v] = mtxA.m[u][0] * mtxB.m[0][v]
                   + mtxA.m[u][1] * mtxB.m[1][v]
                   + mtxA.m[u][2] * mtxB.m[2][v]
                   + mtxA.m[u][3] * mtxB.m[3][v];
    }
  }

  mtxD = mtxT;   // Copy matrix in case source = destination!
}
```

But why do it by hand when there are two perfectly good methods available to you by DirectX.

DirectX C++ prototype:

```
D3DXMATRIX  mtxD, mtxA, mtxB;
mtxD = mtxA * mtxB;
```

…or DirectX C++ prototype:

```
D3DXMATRIX *D3DXMatrixMultiply(D3DXMATRIX *pmD,
    CONST D3DXMATRIX *pmA, CONST D3DXMATRIX *pmB);
```

Matrix Set Identity

An identity matrix (sometimes referred to as I) is typically the base foundation of other matrix types. As shown, all scalars are set to 0, except that the scalars on the diagonal are set to 1's.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ left and right-handed

Equation 5-8: Identity matrix

This is considered an initialized matrix, as when applied to (multiplied with) a vector, the original vector will result. Let's examine this more carefully where matrix ***A*** is applied to vector ***v***:

Pseudo vec:

$$w_1 = v_1a_{11} + v_2a_{21} + v_3a_{31} + v_4a_{41}$$
$$w_2 = v_1a_{12} + v_2a_{22} + v_3a_{32} + v_4a_{42}$$
$$w_3 = v_1a_{13} + v_2a_{23} + v_3a_{33} + v_4a_{43}$$
$$w_4 = v_1a_{14} + v_2a_{24} + v_3a_{34} + v_4a_{44}$$

And when the identity matrix is substituted for the matrix:

$$w_1 = v_1 = \quad v_1(1) + v_2(0) + v_3(0) + v_4(0)$$
$$w_2 = v_2 = \quad v_1(0) + v_2(1) + v_3(0) + v_4(0)$$
$$w_3 = v_3 = \quad v_1(0) + v_2(0) + v_3(1) + v_4(0)$$
$$w_4 = v_4 = \quad v_1(0) + v_2(0) + v_3(0) + v_4(1)$$

Another way of looking at the identity of a matrix is that the results of a product of a matrix and an identity of the same size is a matrix equivalent to the original matrix. Also, do not fear optimizational waste due to the two-dimensional array reference because the const is converted to an offset during compilation.

Listing 5-16: C++

```
void MatrixSetIdentity(D3DMATRIX &mA)
{
        mA._12 = mA._13 = mA._14 =
  mA._21 =       mA._23 = mA._24 =
  mA._31 = mA._32 =       mA._34 =
  mA._41 = mA._42 = mA._43 =       0.0f;

  mA._11 =
           mA._22 =
                    mA._33 =
                             mA._44 = 1.0f;
}
```

DirectX C++ prototype:

```
D3DXMATRIX *D3DXMatrixIdentity(D3DXMATRIX *pM);
```

Merely pass in the address of a matrix, and it will be filled in with the identity matrix. The returned value is the pointer to that same matrix. Another handy little function is:

```
BOOL D3DXMatrixIsIdentity(CONST D3DXMATRIX *pM);
```

This determines if the passed matrix is an identity.

Scaling Triangles

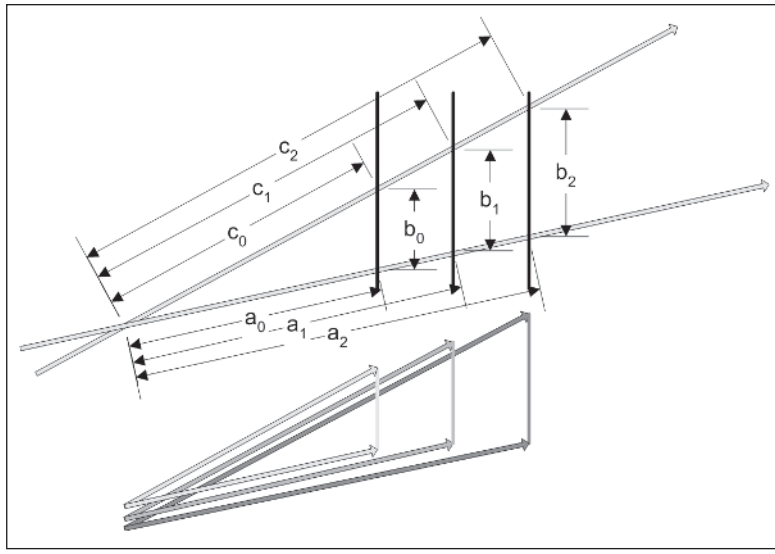


Figure 5-2: If two intersecting lines defining an angle are both intersected by two or more parallel lines, a series of similar overlapping triangles are formed.

The similar-triangle theorem states that given similar triangles, there is a constant k such that:

a' = *k*a b'=*k*b c' = *k*c

In essence, corresponding sides are proportional. Similar triangles are ratios of each other, as the ratio of the differences of each edge is the same ratio of all the edges! For example:

$$\frac{a_0}{b_0}=\frac{a_1}{b_1}=\frac{a_2}{b_2}\,\frac{a_0}{c_0}=\frac{a_1}{c_1}=\frac{a_2}{c_2}\,\frac{b_0}{c_0}=\frac{b_1}{c_1}=\frac{b_2}{c_2}$$

Therefore, using simple ratios:

$$a_2 = \frac{a_1 c_2}{c_1}$$

Equation 5-9: Ratios of similar triangles

Unknown edges can be easily calculated with simple algebraic transformations.

Matrix Set Scale

The scaling factor is set by having all zeros in the application matrix and the scale set for {XYZ} set on the diagonal. In a 3D coordinate system, there are two primary methods of display: a left-handed system and a right-handed system. Some matrices such as for scaling, which are down the diagonal, are identical for both. When a lower or upper diagonal is affected, then the properties of the matrix are dependent upon the hand orientation.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ left and right-handed

Equation 5-10: Scaling matrix

Note that a scale factor of 1 is identical to the identity matrix, and thus no change in any of the elemental values will occur. A factor of 2 doubles the elements in size, and a factor of 0.5 halves (shrinks) them. Each {XYZ} coordinate has a separate scale factor, so an object can be scaled at different rates on each of the axes, such that {0.5, 1, 2} would cause the X element to reduce by half, the Y element to remain the same, and the Z element to double in size. The point is that each vector coordinate is individually scaled.

Listing 5-17: C++

```
void Func(D3DXMATRIX &mA,
                          float fSx, float fSy, float fSz)
{
  mA._12 = mA._13 = mA._14 =
  mA._21 = mA._23 = mA._24 =
  mA._31 = mA._32 = mA._34 =
  mA._41 = mA._42 = mA._43 =  0.0f;

  mA._11 = fSx;
           mA._22 = fSy;
                    mA._33 = fSz;
                             mA._44 = 1.0f;
}
```

Now that you have seen the internal functionality, you might as well use the DirectX equivalent!

DirectX C++ prototype:

```
D3DXMATRIX *D3DXMatrixScaling(D3DXMATRIX *pmD,
    FLOAT fSx, FLOAT fSy, float fSz);
```

Matrix Set Translation

A translation matrix displaces a vector by translating its position by the amount specified by $t_{\{xyz\}}$.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$ left-handed $$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ right-handed

Equation 5-11: Translation matrix left-handed row versus right-handed column matrix

If there is no translation (adjustment of position), then $t_{\{xyz\}}$ are all set to 0, and thus it performs just like an identity matrix. Now if the translation actually has non-zero values, the vector is adjusted, thus displaced, on the {XYZ} axis. Note that the vector really consists of the three coordinates {XYZ} with a fourth tending to be a placeholder. When this matrix is applied to the vector:

Pseudo vec:

$$
\begin{aligned}
w_1 &= v_1 + t_x = & v_1(1) + v_2(0) + v_3(0) + v_4(t_x)\\
w_2 &= v_2 + t_y = & v_1(0) + v_2(1) + v_3(0) + v_4(t_y)\\
w_3 &= v_3 + t_z = & v_1(0) + v_2(0) + v_3(1) + v_4(t_z)\\
w_4 &= v_4 = & v_1(0) + v_2(0) + v_3(0) + v_4(1)
\end{aligned}
$$

…the position is adjusted (displaced) accordingly.

Simplified equation:

```
dx = x + tx;
dy = y + ty;
dz = z + tz;
```

DirectX C++ prototype:

```
D3DXMATRIX *D3DXMatrixTranslation(D3DXMATRIX *pmD,
    FLOAT x, FLOAT y, FLOAT z);
```

Again, this is similar to the matrix identity, so there is no real need to replicate the code here with some minor modification.

Matrix Transpose

A transposed matrix A^T is indicated by the superset ^{T}, and it is effectively the swap of all elements referenced by row-column with column-row indexing, and vice versa. Effectively, as the row and column are equivalent for the diagonal, those elements are retained as indicated by the gray diagonal.

$$
A^T = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}
\qquad
A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}
$$

Equation 5-12: Transpose matrix

Interesting, is it not? Do you recognize it? The starting matrix on the right is similar to that of AoS, or array of structures, {XYZW}[4], and the resulting matrix on the left is that of an SoA, or structure of arrays, {X[4], Y[4], Z[4], W[4]}.

Note that a temporary array needs to be used to save floats just in case the destination matrix to contain the results of the transpose is the source matrix.

Listing 5-18: C++

```
void Func(D3DXMATRIX &mtxD, D3DXMATRIX &mtxA)
{
  float f[6];

  mtxD._11=mtxA._11;    mtxD._22=mtxA._22;
  mtxD._33=mtxA._33;    mtxD._44=mtxA._44;

  f[0]=mtxA._12;  f[1]=mtxA._13;  f[2]=mtxA._14;
  f[3]=mtxA._23;  f[4]=mtxA._24;  f[5]=mtxA._34;

  mtxD._12=mtxA._21;    mtxD._21=f[0];
  mtxD._13=mtxA._31;    mtxD._31=f[1];
  mtxD._14=mtxA._41;    mtxD._41=f[2];
  mtxD._23=mtxA._32;    mtxD._32=f[3];
  mtxD._24=mtxA._42;    mtxD._42=f[4];
  mtxD._34=mtxA._43;    mtxD._43=f[5];
}
```

DirectX C++ prototype:

```
D3DXMATRIX *D3DXMatrixTranspose(D3DXMATRIX *pmD,
    CONST D3DXMATRIX *pmA);
```

Matrix Inverse mD = mA⁻¹

An inverse matrix A^{-1} (also referred to as a reciprocal matrix) is indicated by the superset $^{-1}$. It is effectively the swap of all row-column {XYZ} elements referenced by row-column with column-row indexing, and vice versa. Effectively, as the row and column are equivalent for the diagonal, those elements are retained. The bottom row is set to 0, thus no translation, and the fourth column contains the negative sums indicated by the following expressions.

Pseudo vec:

$$i_1 = -((a_{14}\ a_{11}) + (a_{24}\ a_{21}) + (a_{34}\ a_{31}))$$
$$i_2 = -((a_{14}\ a_{12}) + (a_{24}\ a_{22}) + (a_{34}\ a_{32}))$$
$$i_3 = -((a_{14}\ a_{13}) + (a_{24}\ a_{23}) + (a_{34}\ a_{33}))$$

A sometimes useful expression is that the product of a matrix and its inverse is an identity matrix.

$$AA^{-1} = I$$

By rewriting that equation into the form:

$$A^{-1} = \frac{I}{A}$$

…it visualizes the reasoning why this is sometimes referred to as a reciprocal matrix. It should be kept in mind that as this is a square matrix **A**, then it has an inverse iff (if and only if) the determinant of $|A| \neq 0$; thus it is considered nonsingular (invertible).

An equation to remember is that the inverse transposed matrix is equal to a transposed inverse matrix.

$$(\mathbf{A}^{T})^{-1} = (\mathbf{A}^{-1})^{T}$$

Listing 5-19: C++

```
bool Func(D3DXMATRIX &dMx, D3DXMATRIX &aMx)
{
  D3DXMATRIX s;
  D3DXVECTOR4 t0, t1, t2, *pv;
  float fDet;
  int j;
  bool bRet;

  D3DXMatrixTranspose(&s, &aMx);
                          //     A^T        A
  t0.x = s._33 * s._44;  //  = A2z*A3w   A2z*A3w
  t0.y = s._34 * s._43;  //  = A2w*A3z   A3z*A2w
  t0.z = s._32 * s._44;  //  = A2y*A3w   A1z*A3w
  t0.w = s._33 * s._42;  //  = A2z*A3y   A2z*A1w

  t1.x = s._34 * s._42;  // += A2w*A3y   A3z*A1w
  t1.y = s._31 * s._44;  // += A2x*A3w   A0z*A3w
  t1.z = s._34 * s._41;  // += A2w*A3x   A3z*A0w
  t1.w = s._31 * s._43;  // += A2x*A3z   A0z*Azw

  t2.x = s._32 * s._43;  // += A2y*A3z   A1z*Azw
  t2.y = s._33 * s._41;  // += A2z*A3x   A2z*A0w
  t2.z = s._31 * s._42;  // += A2x*A3y   A0z*A1w
  t2.w = s._32 * s._41;  // += A2y*A3x   A1z*A0w

  dMx[0][0]  = t0.x*s[1][1] + t1.x*s[1][2] + t2.x*s[1][3];
```

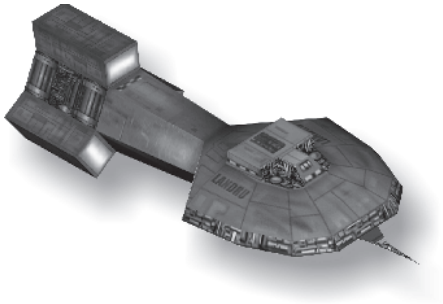
```
dMx[0][0] -= t0.y*s[1][1] + t0.z*s[1][2] + t0.w*s[1][3];

dMx[0][1]  = t0.y*s[1][0] + t1.y*s[1][2] + t2.y*s[1][3];
dMx[0][1] -= t0.x*s[1][0] + t1.z*s[1][2] + t1.w*s[1][3];

dMx[0][2]  = t0.z*s[1][0] + t1.z*s[1][1] + t2.z*s[1][3];
dMx[0][2] -= t1.x*s[1][0] + t1.y*s[1][1] + t2.w*s[1][3];

dMx[0][3]  = t0.w*s[1][0] + t1.w*s[1][1] + t2.w*s[1][2];
dMx[0][3] -= t2.x*s[1][0] + t2.y*s[1][1] + t2.z*s[1][2];

     // calculate {X_ZW} for first two matrix rows

dMx[1][0]  = t0.y*s[0][1] + t0.z*s[0][2] + t0.w*s[0][3];
dMx[1][0] -= t0.x*s[0][1] + t1.x*s[0][2] + t2.x*s[0][3];

dMx[1][1]  = t0.x*s[0][0] + t1.z*s[0][2] + t1.w*s[0][3];
dMx[1][1] -= t0.y*s[0][0] + t1.y*s[0][2] + t2.y*s[0][3];

dMx[1][2]  = t1.x*s[0][0] + t1.y*s[0][1] + t2.w*s[0][3];
dMx[1][2] -= t0.z*s[0][0] + t1.z*s[0][1] + t2.z*s[0][3];

dMx[1][3]  = t2.x*s[0][0] + t2.y*s[0][1] + t2.z*s[0][2];
dMx[1][3] -= t0.w*s[0][0] + t1.w*s[0][1] + t2.w*s[0][2];

     // calculate XY pairs for last two matrix rows

                          //  A^T      A
t0.x = s[0][2]*s[1][3];   // 0=2 7   A2x*A3y
t0.y = s[0][3]*s[1][2];   // 1=3 6   A2x*A2y
t0.z = s[0][1]*s[1][3];   // 2=1 7   A1x*A3y
t1.x = s[0][3]*s[1][1];   // 3=3 5   A3x*A1y
t2.x = s[0][1]*s[1][2];   // 4=1 6   A1x*A2y
t0.w = s[0][2]*s[1][1];   // 5=2 5   A2x*A1y
t1.y = s[0][0]*s[1][3];   // 6=0 7   A0x*A3y
t1.z = s[0][3]*s[1][0];   // 7=3 4   A3x*A0y
t1.w = s[0][0]*s[1][2];   // 8=0 6   A0x*A2y
t2.y = s[0][2]*s[1][0];   // 9=2 4   A2x*A0y
t2.z = s[0][0]*s[1][1];   //10=0 5   A0x*A1y
t2.w = s[0][1]*s[1][0];   //11=1 4   A1x*A0y

     // calculate {XY_W} for last two matrix rows

dMx[2][0]  = t0.x*s[3][1] + t1.x*s[3][2] + t2.x*s[3][3];
dMx[2][0] -= t0.y*s[3][1] + t0.z*s[3][2] + t0.w*s[3][3];

dMx[2][1]  = t0.y*s[3][0] + t1.y*s[3][2] + t2.y*s[3][3];
dMx[2][1] -= t0.x*s[3][0] + t1.z*s[3][2] + t1.w*s[3][3];
```

```
  dMx[2][2]  = t0.z*s[3][0] + t1.z*s[3][1] + t2.z*s[3][3];
  dMx[2][2] -= t1.x*s[3][0] + t1.y*s[3][1] + t2.w*s[3][3];

  dMx[2][3]  = t0.w*s[3][0] + t1.w*s[3][1] + t2.w*s[3][2];
  dMx[2][3] -= t2.x*s[3][0] + t2.y*s[3][1] + t2.z*s[3][2];

    // calculate {XY_W} for first two matrix rows

  dMx[3][0]  = t0.z*s[2][2] + t0.w*s[2][3] + t0.y*s[2][1];
  dMx[3][0] -= t1.x*s[2][2] + t2.x*s[2][3] + t0.x*s[2][1];

  dMx[3][1]  = t1.w*s[2][3] + t0.x*s[2][0] + t1.z*s[2][2];
  dMx[3][1] -= t2.y*s[2][3] + t0.y*s[2][0] + t1.y*s[2][2];

  dMx[3][2]  = t1.y*s[2][1] + t2.w*s[2][3] + t1.x*s[2][0];
  dMx[3][2] -= t1.z*s[2][1] + t2.z*s[2][3] + t0.z*s[2][0];

  dMx[3][3]  = t2.z*s[2][2] + t2.x*s[2][0] + t2.y*s[2][1];
  dMx[3][3] -= t2.w*s[2][2] + t0.w*s[2][0] + t1.w*s[2][1];

       // calculate determinant

  fDet = s[0][0]*dMx[0][0] + s[0][1]*dMx[0][1]
       + s[0][2]*dMx[0][2] + s[0][3]*dMx[0][3];
  if (0.0f == fDet)
  {
    fDet = 1.0f;
    bRet = false;
  }
  else
  {
    fDet = 1.0f / fDet;
    bRet = true;
  }

  pv = (vmp3DQVector *)dMx;
  j = 4;
  do {
    pv->x *= fDet;
    pv->y *= fDet;
    pv->z *= fDet;
    pv->w *= fDet;
    pv++;
  } while (--j);

  return bRet;
}
```

Note that the DirectX version assigns the determinant to the second passed pointer.

DirectX C++ prototype:

```
D3DXMatrixInverse(D3DXMATRIX *pmD,
     float *pDeterminate, D3DXMATRIX *pmA);
```



Chapter 6

Matrix Deux: A Wee Bit o' Trig

As mentioned in the last chapter, the information about matrices was divided into two chapters to help reduce their sizes. From the standpoint of vertex shaders, there is only one vertex instruction that supports trigonometric operations, and that is the macro instruction *sincos*.

■ ***sincos***: Sine – cosine calculation

sincos (Macro)	1.1	2.0	2_x	2_{sw}	3.0	3_{sw}
sincos *Dst*, *aSrc*, *bSrc*, *cSrc*		☺	☺	☺		
sincos *Dst*, *aSrc*					☺	☺

This (eight-slot) macro instruction calculates both the sine and cosine of the source arguments in radians.

aSrc is the radian source scalar value in radians ranging from {–π…0…π}.

Things get a little murky, as the vertex assembler version 3.0 and above only accepts a single source argument; therefore, I must conclude that the two following constants are predefined:

bSrc is a constant defined as D3DSINCOSCONST1.

bSrc = (–1.0f/(7!*128), –1.0f/(6!*64), 1/(4!*16), 1/(5!*32))

```
#define D3DSINCOSCONST1 –1.5500992e–006f, –2.1701389e–005f,
    0.0026041667f, 0.00026041668f
```

cSrc is a constant defined as D3DSINCOSCONST2.

cSrc = (–1.0f/(3!*8), –1.0f/(2!*8), 1, 0.5f)

```
#define D3DSINCOSCONST2 –0.020833334f,
    –0.12500000f, 1.0f, 0.50000000f

D3DXVECTOR4 vSinCos1(D3DSINCOSCONST1);
D3DXVECTOR4 vSinCos2(D3DSINCOSCONST2);

m_pd3dDevice->SetVertexShaderConstantF(12, (float*)&vSinCos1, 1);
m_pd3dDevice->SetVertexShaderConstantF(13, (float*)&vSinCos2, 1);
```

The resulting y=sin(θ) is stored in *Dst.y*, and x=cos(θ) is stored in *Dst.x*.

Normally, an instruction can only read from a single constant register regardless of how many registers it can use as a source. This instruction, however, is the exception to the rule, as it can actually read from two separate constant source registers simultaneously.

The *aSrc* must be a single component {x, y, z, or w}.

The *bSrc* and *cSrc* must be a constant register.

The *Dst* must be a temporary register (r#) and specify one of the following component masks {x, y, xy}. It *cannot* be the same register as *aSrc*. The component *Dst.z* is corrupted.

For channel selection, the source *bSrc* needs to utilize a replication swizzle.

Listing 6-1: Vertex shader

```
sincos r1.xy, r0.x, c1, c2            // version 2.0, 2x, 2sw
sincos r1.xy, r0.x, c1.xyzw, c2.xyzw  // version 2.0, 2x, 2sw
sincos r1.xy, c0.z, c1.xyzw, c2.xyzw  // version 2.0, 2x, 2sw

sincos r1.xy, r0.x                    // version 3.0, 3sw
```

That was simple, was it not? Maybe too simple? For those of you very familiar with the use of trigonometric operations in matrices, you can just skip ahead to the next chapter.

But, if you need a refresher or background on how to do trig operations within matrices, then continue reading!

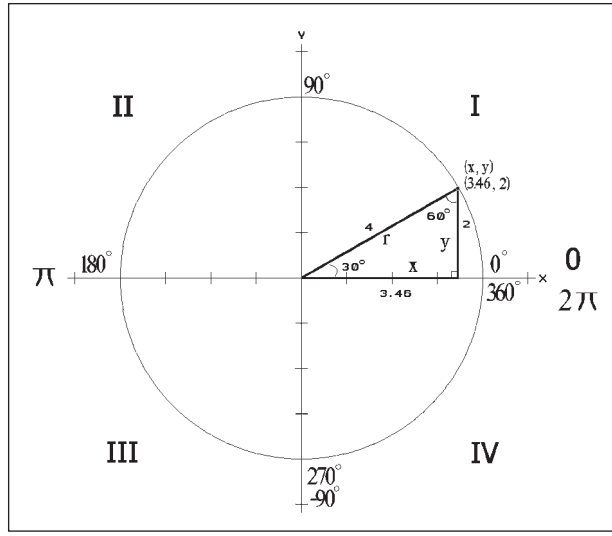


Figure 6-1: 2D geometric circle

Again, I wish to state that if you need more information related to this and other subjects related to math, I recommend my vector book (*Vector Game Math Processors*) or *3D Math Primer for Graphics and Game Development* by Fletcher Dunn and Ian Parberry (also from Wordware Publishing) or a visit to your local university bookstore or one of the multitude of rich, or not-so-rich, web sites.

Sine and Cosine Functions

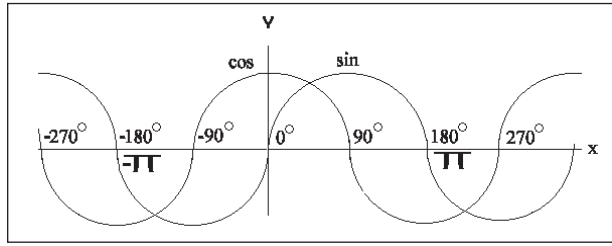


Figure 6-2: Sine-cosine waves

$$\sin\theta = \frac{\text{opposite side}}{\text{hypotenuse}} \qquad \cos\theta = \frac{\text{adjacent side}}{\text{hypotenuse}}$$

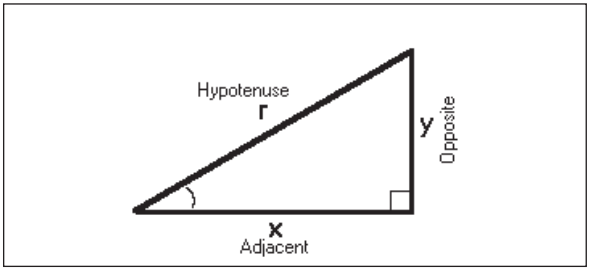


Figure 6-3: Sine and cosine trigonometric relationships

$$\sin\theta = \frac{y}{r} \quad y = r\sin\theta \quad r = \frac{y}{\sin\theta}$$

$$\cos\theta = \frac{x}{r} \quad x = r\cos\theta \quad r = \frac{x}{\cos\theta}$$

Equation 6-1: Sine and cosine

The standard "C" math library contains the following functions:

```
float cos(float r);          double cos(double r);
float sin(float r);          double sin(double r);
```

You should already be familiar with the fact that the angle in degrees is not passed into those functions but instead the equivalent value in radians. If you recall, π (pi) is equivalent to 180°, and 2π is equivalent to 360°. By using the following macro, an angle in degrees can be converted to radians:

```
#define PI          (3.141592f)
#define DEG2RAD(x)  (((r) * PI) / 180.0F)
```

…and used in the calculations. It can then be converted from radians back to degrees:

```
#define RAD2DEG(x)  (((r) * 180.0f) / PI)
```

…if needed for printing or other purposes.

For a simple 2D rotation, the use is merely one of:

```
x = cos(fRadian);
y = sin(fRadian);
```

There is one thing that has always bothered me about these two functions. When a cosine is needed, a sine is needed as well, and

that one is in reality 90 degrees out of phase of the other, which means that they share similar math.

$$\sin(90° - \theta) = \cos\theta$$

Equation 6-2: Angular relationship sine to cosine

As a generic C/C++ *sincos* procedure, define your function similar to the following:

Listing 6-2: C++

```
void FSinCos(float * pfSin, float * pfCos, float fRads)
{
  *pfSin = sinf(fRads);
  *pfCos = cosf(fRads);
}
```

Okay, okay, the following is a little off from the topics of this book (see my vector book for more in-depth information), but for the X86 processor, you can write a faster equivalent function in assembly code!

vmp_SinCos (X86)

The *fwait* instruction is used to wait for the FPU (floating-point unit) operation to complete. The *fsincos* instruction calculates the sine and cosine simultaneously. This is slower than calling just one of them but faster than calling them both consecutively.

Listing 6-3: X86 FPU assembly

```
fld   fRads               ; Load Radians from memory
OFLOW_FIXUP fRads         ; Overflow fixup (optional)

fwait                     ; Wait for FPU to be idle
fsincos                   ; ST(0)=cos ST(1)=sin

mov   eax,pfSin
mov   ecx,pfCos

fwait                     ; Wait for FPU to be idle
                          ; Pop from FPU stack
fstp  (REAL4 PTR [ecx])   ; ST(0)=cos ST(1)=sin
fstp  (REAL4 PTR [eax])   ; sin
```

The vertex shader *sincos* instruction uses this same optimization for speed efficiency.

Matrix Rotations

3D rotations are much more interesting and more difficult to understand than 2D rotations. For example, to rotate the cube in the following diagram, you do not actually rotate on the axis that you wish to rotate. Do not worry. This is not nearly as complicated as trying to solve a Rubik's Cube! It just makes a cool prop and another neat office toy! This is demonstrated later. Now let's put into practice the sine of the trigonometry.

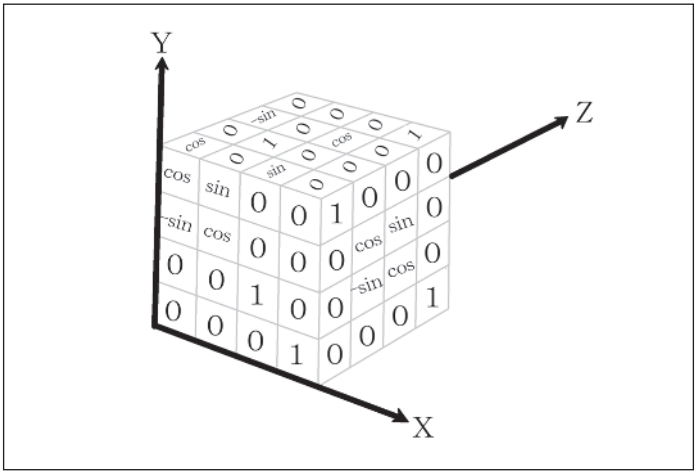


Figure 6-4: Rubik's Cube rotational matrix model

Confusing? It can be! Find yourself a Rubik's Cube. Keep each side as individual colors (do not scramble it)! Attach pencils, straws, or swizzle sticks to represent the x-, y-, and z-axis. (This can be done with Legos or Tinker Toys, but the Rubik's Cube is a lot more visual.) Now try to rotate the x-axis by only rotating the y and z-axis. Fun, huh? Have you ever tried to manually control an object in a 3D test jig just by controlling the three axes individually?

With matrices, the order of rotations is very important, and they should always be in an {XYZ} rotation order. Matrix operations of

Euler angles tend to be the primary use of matrices for x-rot, y-rot, and z-rot, but it comes with a price. This is detailed in Chapter 7.

Matrix Set X Rotation

To rotate on an x-axis, one actually rotates the y and z coordinates, leaving the x coordinate alone. Note the darkened text areas of the following equation; cos and sin are the only difference from an identity matrix. Also note the negative sign on opposite sides of the diagonal depending on if the matrix is for a left-handed or right-handed 3D coordinate system.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \mathbf{\cos\theta} & \mathbf{\sin\theta} & 0 \\ 0 & \mathbf{-\sin\theta} & \mathbf{\cos\theta} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{left-handed} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \mathbf{\cos\theta} & \mathbf{-\sin\theta} & 0 \\ 0 & \mathbf{\sin\theta} & \mathbf{\cos\theta} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{right-handed}$$

Equation 6-3: X-axis (left-handed row and right-handed column) rotation matrix. Note that this is nearly identical to an identity matrix, except for the (non-x axis) y and z rows and columns being set to the trigonometric values.

Pseudo vec:

$$\begin{aligned} w_x = w_1 = v_1 = & \quad v_1(1) + v_2(0) + v_3(0) + v_4(0) \\ w_y = w_2 = v_2 \cos\theta + v_3 \sin\theta = & \quad v_1(0) + v_2(\cos\theta) + v_3(\sin\theta) + v_4(0) \\ w_z = w_3 = -v_2 \sin\theta + v_3 \cos\theta = & \quad v_1(0) + v_2(-\sin\theta) + v_3(\cos\theta) + v_4(0) \\ w_w = w_4 = v_4 = & \quad v_1(0) + v_2(0) + v_3(0) + v_4(1) \end{aligned}$$

Simplified equation:

```
fCos = cos(ang);
fSin = sin(ang);

dx =  x;
dy =  y * fCos + z * fSin;
dz = –y * fSin + z * fCos;
```

Note that to save processing time, the standard C language trigonometric functions sin() and cos() are only used once, and the result of the sine function is merely negated for the inverse result.

Listing 6-4: C++

```
void Func(D3DXMATRIX &mtxD, float fRads)
{
   float fSin, fCos;

   FSinCos(&fSin, &fCos, fRads);

   D3DXMatrixIdentity(&mtxD);

   mtxD._23 =                fSin;
   mtxD._33 =  mtxD._22 =    fCos;
   mtxD._32 = –fSin;
}
```

DirectX C++ prototype:

```
D3DXMATRIX *D3DXMatrixRotationX(D3DXMATRIX *pmD,
    FLOAT fRads);
```

Matrix Set Y Rotation

To rotate on a y-axis, one actually rotates the x and z coordinates, leaving the y alone by not touching the elements of the row and column of y.

$$\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{\text{left-handed}} \qquad \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}_{\text{right-handed}}$$

Equation 6-4: Y-axis (left-handed row and right-handed column) rotation matrix. Note that this is nearly identical to an identity matrix, except for the (non-y axis) x and z rows and columns being set to the trigonometric values.

Pseudo vec:

$$\begin{aligned} w_x = w_1 = v_1 \cos\theta + -v_3 \sin\theta &= v_1(\cos\theta) + v_2(0) + v_3(-\sin\theta) + v_4(0) \\ w_y = w_2 = v_2 = &\quad v_1(0) + v_2(1) + v_3(0) + v_4(0) \\ w_z = w_3 = v_1 \sin\theta + v_3 \cos\theta &= v_1(\sin\theta) + v_2(0) + v_3(\cos\theta) + v_4(0) \\ w_w = w_4 = v_4 = &\quad v_1(0) + v_2(0) + v_3(0) + v_4(1) \end{aligned}$$

Simplified equation:

```
fCos = cos(ang);
fSin = sin(ang);

dx = x * fCos + –z * fSin;
dy = y;
dz = x * fSin +  z * fCos;
```

Listing 6-5: C++

```
void Func(D3DXMATRIX &mtxD, float fRads)
{
  float fSin, fCos;

  FSinCos(&fSin, &fCos, fRads);

  D3DXMatrixIdentity(&mtxD);

  mtxD._31 =               fSin;
  mtxD._33 =  mtxD._11 =   fCos;
  mtxD._13 = –fSin;
}
```

DirectX C++ prototype:

```
D3DXMATRIX *D3DXMatrixRotationY(D3DXMATRIX *pmD,
    FLOAT fRads);
```

Matrix Set Z Rotation

To rotate on a z-axis, one actually rotates the x and y coordinates, leaving the z coordinate alone by not touching the elements of the row and column of z.

$$\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ left-handed

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ right-handed

Equation 6-5: Z-axis (left-handed row and right-handed column) rotation matrix. Note that this is nearly identical to an identity matrix, except for the (non-z axis) x and y rows and columns being set to the trigonometric values.

Pseudo vec:

$$
\begin{aligned}
w_x = w_1 = v_1 \cos\theta + v_2 \sin\theta &= v_1(\cos\theta) + v_2(\sin\theta) + v_3(0) + v_4(0) \\
w_y = w_2 = -v_1 \sin\theta + v_2 \cos\theta &= v_1(-\sin\theta) + v_2(\cos\theta) + v_3(0) + v_4(0) \\
w_z = w_3 = v_3 &= v_1(0) + v_2(0) + v_3(1) + v_4(0) \\
w_w = w_4 = v_4 &= v_1(0) + v_2(0) + v_3(0) + v_4(1)
\end{aligned}
$$

Simplified equation:

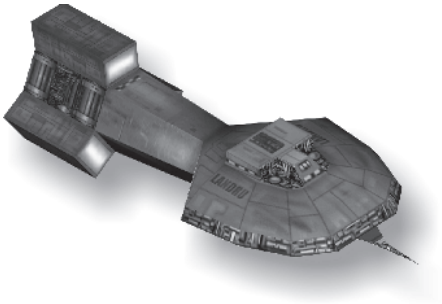
```
fCos = cos(ang);
fSin = sin(ang);

dx =  x * fCos + y * fSin;
dy = -x * fSin + y * fCos;
dz =  z;
```

Listing 6-6: C++

```
void Func(D3DXMATRIX &mtxD, float fRads)
{
  float fSin, fCos;

  FSinCos(&fSin, &fCos, fRads);

  D3DXMatrixIdentity(&mtxD);

  mtxD._12 =               fSin;
  mtxD._22 =  mtxD._11 =   fCos;
  mtxD._21 = -fSin
}
```

DirectX C++ prototype:

```
D3DXMATRIX *D3DXMatrixRotationZ(D3DXMATRIX *pmD, FLOAT fRads);
```



Chapter 7

Quaternions

Before we discuss the ins and outs of a quaternion, maybe we should revisit a standard 3D rotation using Euler angles. As you should know, an object is oriented within 3D space by the specified rotation of its {XYZ} axis rotations. It, in essence, uses an {XYZ} vector to indicate its position within world coordinate space (translation) and is oriented (rotated) based upon a set of {XYZ} rotations. Additional information, such as scaling, etc., is utilized as well. Keep in mind that the x, y, and z rotations must occur in a precise order every time. But there is a problem, and that is gimbal lock!

This is easier to visualize if we first examine a gimbal. Remember that Euler angles need to be rotated in a particular order, such as {XYZ}. Rotating just one axis is not a problem, but when two or three axes are rotated simultaneously, a problem is presented. Keeping in mind that the starting positions of the {XYZ} axis are 90 degrees from each other such as in a left-handed orientation, then first rotate the x ring (axis) and there is no problem. But by then rotating the y axis, such as the middle ring, by 90 degrees, a gimbal lock occurs. The same occurs when moving the z-axis. By rotating it 90 degrees as well, all three rings will be locked into the same position. Just two matching rings is an indicator of a gimbal lock!

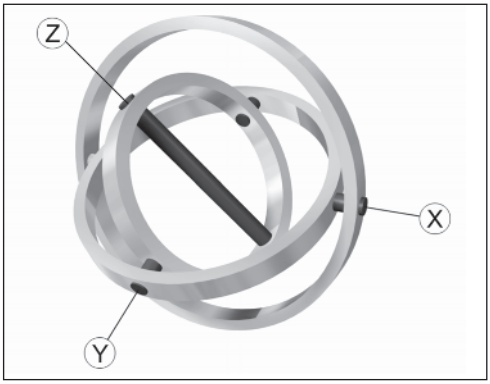


Figure 7-1: Three-axis gimbal (compliments of Ken Mayfield)

Now move one ring, any ring. What are the possible axis angles? Do you see the problem?

If merely using matrices with Euler angles, then the movement of a player needs to be limited. They can be moved (translated) around the playfield with no problem. Rotating them on their y-axis 360° is no problem either. The problem comes when it's time to tip the character over like a top along its x- or z-axis (crawling, swimming, lying down, running down a steep hill, etc.).

The Euler rotation angles each have an associated {XYZ} coordinate {x=pitch, y=yaw, z=roll}. By using a quaternion, this problem is alleviated. Instead of a rotation being stored as separate Euler rotations, it is stored as an {XYZ} vector with a rotation factor.

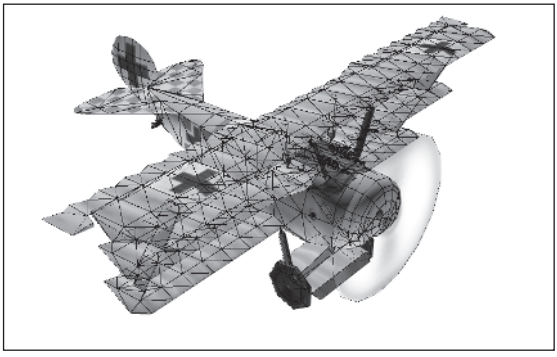


Figure 7-2: Yaw, pitch, and roll Euler rotation angles — Fokker Dr 1 triplane (Red Baron) (compliments of Ken Mayfield)

A rotation, however, involves 360°, which presents a problem. Rotational data is stored in radians and a full circle requires 2π to be stored, but a cosine and/or sine only processes up to the equivalent of π (180°), so half of the rotation information can be lost. A quaternion resolves this by dividing the angle in radians by two (radians * 0.5). When needed to convert back to a full angle, merely multiply by two! This simple solution helps condense the data into a form with no data loss.

But we are developing shader code, not rotating biplanes and stuff. Why do we need it here?

We are drawing those biplanes with a moving camera, as well as other 3D objects with XYZ rotations, and so their visual appearance will become warped as a gimbal lock appears for portions of its polygons. Therefore, quaternion math within shader code will help rectify those situations.

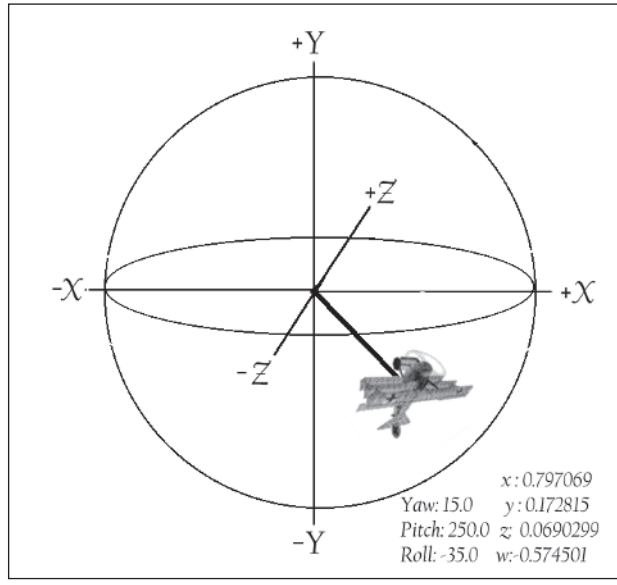


Figure 7-3: Yaw, pitch, and roll quaternion rotation — Fokker Dr 1 triplane (Red Baron) (compliments of Ken Mayfield)

A quaternion uses the same storage type as a quad vector, but to prevent accidental mixing of the two, it is better to define its own data type.

DirectX uses a four-field representation that is different from D3DXVECTOR4.

```
typedef struct D3DXQUATERNION
{
  float x;
  float y;
  float z;
  float w;
} D3DXQUATERNION;
```

A quaternion comes in two forms. The first is a true form, which is the {XYZ} component, and the {W}, which is the rotation.

The second form is a unit vector, which is all four elements with the inclusion of {W}. All are used in the magnitude in calculating a normalized value of 1!

An easy way to think of this is that a true quaternion is {XYZ}W, while a unit quaternion is {XYZW}. A unit sphere has a radius of 1.0, and a unit triangle has a hypotenuse of 1.0. A unit typically means one!

$$q = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

An imaginary is represented by the $\boldsymbol{i}=\sqrt{-1}$.

As such, an identity is defined as {0,0,0,1}. The same rule of identities applies, whereas the product of a quaternion identity is the quaternion.

A quaternion is made up of the imaginaries {*i, j, k*} such that $q=w+x\boldsymbol{i}+y\boldsymbol{j}+z\boldsymbol{k}$. Two other representations would be:

$q = [x\ y\ z\ w]$ and $q = [w, v]$

Whereas *w* is a scalar and *v* is the {XYZ} vector.

In essence, a quaternion is made up of not one but three imaginaries:

$$\boldsymbol{ii} = -1 \qquad \boldsymbol{jj} = -1 \qquad \boldsymbol{kk} = -1$$

The product of imaginary pairs is similar to that of a cross product of axes in 3D space.

$$\boldsymbol{i} = \boldsymbol{jk} = -\boldsymbol{kj} \qquad \boldsymbol{j} = \boldsymbol{ki} = -\boldsymbol{ik} \qquad \boldsymbol{k} = \boldsymbol{ij} = -\boldsymbol{ji}$$

I find that a simple method to remember this is the ordering of imaginary numbers {***i, j, k***}. Think of it as a sequence of {***i, j, k, i***,…} and that the third imaginary results with a positive sign if the multiplicands were sequential and negative if in reverse order. So ***ij***=***k***, ***jk***=***i***, ***ki***=***j***, but ***kj***=–***i***, ***ji***=–***k***, and ***ik***=–***j***. See, simple! Just remember that the products of imaginaries are not commutative.

There are different implementations of a quaternion library function depending on left-handed versus right-handed coordinate systems, level of normalization, flavor of the quaternion instantiation, and overall extra complexity needed if branchless code is attempted.

Quaternion Operations

This section delves into a variety of operations that are possible with quaternions.

Quaternion Copy

The quaternion copy is identical to a quad vector copy, as they both consist of four packed single-precision floating-point values.

Listing 7-1: Vertex shader

```
mov r0, c0
```

Quaternion Addition

The summation of two quaternions would be as follows:

$$\begin{aligned} q_1 + q_2 &= (w_1\ \ x_1\boldsymbol{i}\ \ y_1\boldsymbol{j}\ \ z_1\boldsymbol{k}) + (w_2\ \ x_2\boldsymbol{i}\ \ y_2\boldsymbol{j}\ \ z_2\boldsymbol{k}) \\ &= w_1+w_2 \quad + x_1\boldsymbol{i} + x_2\boldsymbol{i} + y_1\boldsymbol{j} + y_2\boldsymbol{j} + z_1\boldsymbol{k} + z_2\boldsymbol{k} \\ &= (w_1+w_2) + \ (x_1+x_2)\boldsymbol{i} + (y_1+y_2)\boldsymbol{j} + \ (z_1+z_2)\boldsymbol{k} \end{aligned}$$

A simpler method of thinking about this would be as follows:

$$\begin{aligned} q_1 + q_2 &= [w_1\ v_1] + [w_2\ v_2] = [w_1+w_2\ \ v_1+v_2] \\ &= [(w_1+w_2)\ \ (x_1+x_2\ \ y_1+y_2\ \ z_1+z_2)] \\ &= [w_1+w_2\ \ x_1+x_2\ \ y_1+y_2\ \ z_1+z_2] \end{aligned}$$

So in its simple elemental form:

$D_x=A_x+B_x \quad D_y=A_y+B_y \quad D_z=A_z+B_z \quad D_w=A_w+B_w$

Listing 7-2: Vertex shader

```
add r0, c23, r3        // Addition
```

You should hopefully have recognized it as the same as a quad vector floating-point addition and the following as a subtraction.

Quaternion Subtraction

The difference of two quaternions is:

$$q_1 - q_2 = w_1 - w_2 + (x_1 - x_2)\mathbf{i} + (y_1 - y_2)\mathbf{j} + (z_1 - z_2)\mathbf{k}$$

So in its simple elemental form:

$D_x=A_x-B_x \quad D_y=A_y-B_y \quad D_z=A_z-B_z \quad D_w=A_w-B_w$

Listing 7-3: Vertex shader

```
add r0, r0, -c24        // Subtraction
```

Quaternion Dot Product (Inner Product)

$$q = w + x\boldsymbol{i} + y\boldsymbol{j} + z\boldsymbol{k}$$

$$q_1 \bullet q_2 = w_1w_2 + (x_1x_2)\boldsymbol{i} + (y_1y_2)\boldsymbol{j} + (z_1z_2)\boldsymbol{k}$$

$$D = A_xB_x + A_yB_y + A_zB_z + A_wB_w$$

This is very similar to a vector dot product, except it contains the fourth product sum element of {W}.

Listing 7-4: Vertex shader

```
dp4 r2,r0,r1
```

DirectX C++ prototype:

```
FLOAT D3DXQuaternionDot(const D3DXQUATERNION *pqA,
                        const D3DXQUATERNION *pqB);
```

Quaternion Magnitude (Length of Vector)

Just like a dot product, this function is similar to a vector magnitude, except it uses the squared sum of the {W} element before the square root.

$$q = w + xi + yj + zk$$

$$q_1 q_2 = \sqrt{\left(w_1 w_2 + \left(x_1 x_2\right)i + \left(y_1 y_2\right)j + \left(z_1 z_2\right)k\right)}$$

$$q^2 = \sqrt{\left(w^2 + x^2 i + y^2 j + z^2 k\right)}$$

$$D = \sqrt{\left(A_w A_w + A_z A_z + A_y A_y + A_x A_x\right)}$$

Listing 7-5: Vertex shader

```
dp4  r0.w, r1, r1        // r=ax2+ay2+az2+aw2
rsq  r0.x, r0.w          // dx = 1/√r
mul  r0.x, r0.x, r0.w    // dx=r*dx
```

DirectX C++ prototype:

```
FLOAT D3DXQuaternionLength(const D3DXQUATERNION *pqA);
```

Quaternion Normalization

Note that most of the quaternion functions are dependent on the normalization of a number (that is, the dividing of each element by the magnitude of the quaternion). There is only one little condition to watch out for and handle — the divide by zero! But remember your calculus where *as x approaches zero* the result goes infinite, so the effect on the original value is negligible, and thus the solution is the original value.

The same principles of assembly code are applied here, so only the C code is shown!

$$\text{norm}(q) = w + xi + yj + zk$$

$$r = \sqrt{\left(w^2 + x^2 i + y^2 j + z^2 k\right)}$$

$$\text{norm}(q) = \frac{w}{r} + \frac{x\boldsymbol{i}}{r} + \frac{y\boldsymbol{j}}{r} + \frac{z\boldsymbol{k}}{r}$$

norm(q)=q / r

Okay, here is where I tend to get a little confused! If you examine quaternion code by others, they typically have:

$$r = \sqrt{\left(x^2 + y^2 + z^2 + w^2\right)}$$

D_x=x/r D_y=y/r D_z=z/r D_w=w/r

But wait! What happens if the q={0,0,0,0}? Will not divide by zero exist? That cannot possibly happen because a quaternion of no length {0,0,0} will have a rotation of 1, such as the quaternion identity {0,0,0,1}. Well, let's assume that a normalized quaternion is subtracted from itself; that would leave a quaternion of {0,0,0,0}. Using those other algorithms would lead to a problem of the divide by zero as well as it not being normalized. The solution is that if the value is too close to 0, then assigning a 1 to the rotation would make no difference and resolve the normalization issue. One other item is that if the value is very close to being finite and negative, then the value would be off a bit. So by setting the {W} element to 1 but with the same sign as the original {W}, then the value is properly normalized.

Listing 7-6: Vertex shader

```
dp4  r0.w, r1, r1          // dw=ax2+ay2+az2+aw2
rsq  r0, r0.w              // dw = 1/√dw    or   1/+∞ = 1/√0
mul  r0, r1, r0            // {aw(1/dw) az(1/dw) ay(1/dw) ax(1/dw)}
```

DirectX C++ prototype:

```
D3DXQUATERNION * D3DXQuaternionNormalize(
                                    D3DXQUATERNION *pqD,
                                const D3DXQUATERNION *pqA);
```

Quaternion Conjugation $D=\overline{A}$

A conjugate is merely the inverse of a vector. As such, the rotation is unaffected. The inverse of a vector can result from negating the sign of the rotation {W}, but it is preferred to keep the rotation in a positive form, so instead only each axis of the vector is inverted. It is assumed that the rotation is already positive.

$$q = w + xi + yj + zk$$

$$\bar{q} = w - xi - yj - zk$$

$$D_w = A_w \quad D_z = -A_z \quad D_y = -A_y \quad D_x = -A_x$$

Listing 7-7: Vertex shader

```
mov  r0,  –r1          // {–aw  –az  –ay  –ax}
mov  r0.w, r1.w        // { aw  –az  –ay  –ax}
```

DirectX C++ prototype:

```
D3DXQUATERNION * D3DXQuaternionConjugate(
                                  D3DXQUATERNION *pqD,
                            const D3DXQUATERNION *pqA);
```

Quaternion Inversion $D=A^{-1}$

$$q = w + xi + yj + zk$$

$$q^{-1} = \frac{\bar{q}}{\text{norm}(q)} = \frac{w - xi - yj - zk}{w^2 + x^2 + y^2 + z^2}$$

Listing 7-8: Vertex shader

```
dp4  r0.w, r1, r1      // dw=ax²+ay²+az²+aw²
rcp  r0, r0.w          // dxyzw = 1/dw
mul  r0, r1, –r0.w     // {–awdw  –azdw  –aydw  –axdw}
mov  r0.w, –r0.w       // { awdw  –azdw  –aydw  –axdw}
```

DirectX C++ prototype:

```
D3DXQUATERNION * D3DXQuaternionInverse(
                                    D3DXQUATERNION *pqD,
                                const D3DXQUATERNION *pqA);
```

Quaternion Multiplication D=AB

Please note the algebraic laws of commutative do not apply here!

AB <> BA AB != BA

Pseudocode:

$$
\begin{aligned}
q_1q_2 = (&w_1 \;\; x_1\mathbf{i} \;\; y_1\mathbf{j} \;\; z_1\mathbf{k})\;(w_2 \;\; x_2\mathbf{i} \;\; y_2\mathbf{j} \;\; z_2\mathbf{k})\\
&= w_1w_2 + w_1x_2\mathbf{i} + w_1y_2\mathbf{j} + w_1z_2\mathbf{k}\\
&+ x_1w_2\mathbf{i} + x_1x_2\mathbf{i}^2 + x_1y_2\mathbf{ij} + x_1z_2\mathbf{ik}\\
&+ y_1w_2\mathbf{j} + y_1x_2\mathbf{ji} + y_1y_2\mathbf{j}^2 + y_1z_2\mathbf{jk}\\
&+ z_1w_2\mathbf{k} + z_1x_2\mathbf{ki} + z_1y_2\mathbf{kj} + z_1z_2\mathbf{k}^2
\end{aligned}
$$

Remember that the square of an imaginary is –1, and the product of two different imaginaries is a positive third imaginary if the two products are sequential; otherwise they are negative: ***ij*=*k***, ***ji*=–*k***, ***jk*=*i***, ***kj*=–*i***, ***ki*=*j***, ***ik*=–*j***

$$
\begin{aligned}
&= w_1w_2 + w_1x_2\mathbf{i} + w_1y_2\mathbf{j} + w_1z_2\mathbf{k}\\
&+ x_1w_2\mathbf{i} - x_1x_2 + x_1y_2\mathbf{k} - x_1z_2\mathbf{j}\\
&+ y_1w_2\mathbf{j} - y_1x_2\mathbf{k} - y_1y_2 + y_1z_2\mathbf{i}\\
&+ z_1w_2\mathbf{k} + z_1x_2\mathbf{j} - z_1y_2\mathbf{i} - z_1z_2
\end{aligned}
$$

Regrouping into like complex terms:

$$
\begin{aligned}
&= w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2\\
&+ w_1x_2\mathbf{i} + x_1w_2\mathbf{i} - z_1y_2\mathbf{i} + y_1z_2\mathbf{i}\\
&+ w_1y_2\mathbf{j} + y_1w_2\mathbf{j} + z_1x_2\mathbf{j} - x_1z_2\mathbf{j}\\
&+ w_1z_2\mathbf{k} + z_1w_2\mathbf{k} + x_1y_2\mathbf{k} - y_1x_2\mathbf{k}
\end{aligned}
$$

$$
\begin{aligned}
&= w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2\\
&+ (w_1x_2 + x_1w_2 - z_1y_2 + y_1z_2)\mathbf{i}\\
&+ (w_1y_2 + y_1w_2 + z_1x_2 - x_1z_2)\mathbf{j}\\
&+ (w_1z_2 + z_1w_2 + x_1y_2 - y_1x_2)\mathbf{k}
\end{aligned}
$$

Listing 7-9: Vertex shader

```
// dx = (ax * bw) + (ay * bz) – (az * by) + (aw * bx)
// dy = (ay * bw) – (ax * bz) + (aw * by) + (az * bx)
// dz = (az * bw) + (aw * bz) + (ax * by) – (ay * bx)
// dw = (aw * bw) – (az * bz) – (ay * by) – (ax * bx)

mul r3, r1.yxwz, r2.z
mul r4, r1.zyxy, r2.y
mul r5, r1.wzyx, r2.x

mov r3.yw, –r3.yw
mov r4.xw, –r4.xw
mov r5.zw, –r5.zw

mad r0, r1, r2.w, r3
add r0, r0, r4
add r0, r0, r5
```

DirectX C++ prototype:

```
D3DXQUATERNION * D3DXQuaternionMultiply(
                                    D3DXQUATERNION *pqD,
                              const D3DXQUATERNION *pqA,
                              const D3DXQUATERNION *pqB);
```

NOTE: DirectX D=BA It expects a B, A order!

Quaternion Division

A quaternion divides the result of the product of a quaternion, the dividend, into an inverse of a quaternion.

$$q1 = \left(w_1 \quad x_1\boldsymbol{i} \quad y_1\boldsymbol{j} \quad z_1\boldsymbol{k}\right) \qquad q_2 = \left(w_2 \quad x_2\boldsymbol{i} \quad y_2\boldsymbol{j} \quad z_2\boldsymbol{k}\right)$$

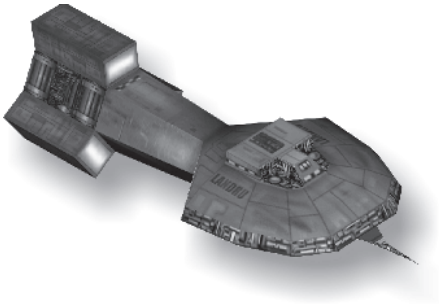
$$\frac{q_1}{q_2} = q_2^{-1} q_1 = \frac{w_2 - x_2\boldsymbol{i} - y_2\boldsymbol{j} - z_2\boldsymbol{k}}{w_2^{\,2} + x_2^{\,2} + y_2^{\,2} + z_2^{\,2}}\left(w_1 + x_1\boldsymbol{i} + y_1\boldsymbol{j} + z_1\boldsymbol{k}\right)$$

Please note that a few steps were skipped here to arrive at the following solution:

$$= \frac{w_2 w_1 + x_2 w_1 + y_2 y_1 + z_2 z_1}{{w_2}^2 + {x_2}^2 + {y_2}^2 + {z_2}^2} + \frac{\left(w_2 x_1 - x_2 w_1 - y_2 z_1 + z_2 y_1\right)\boldsymbol{i}}{{w_2}^2 + {x_2}^2 + {y_2}^2 + {z_2}^2}$$

$$+ \frac{\left(w_2 y_1 + x_2 z_1 - y_2 w_1 - z_2 x_1\right)\boldsymbol{j}}{{w_2}^2 + {x_2}^2 + {y_2}^2 + {z_2}^2} + \frac{\left(w_2 z_1 - x_2 y_1 + y_2 x_1 - z_2 w_1\right)\boldsymbol{k}}{{w_2}^2 + {x_2}^2 + {y_2}^2 + {z_2}^2}$$

There are plenty more quaternion functions supported by DirectX9, but the ones discussed here should be enough to whet your appetite!



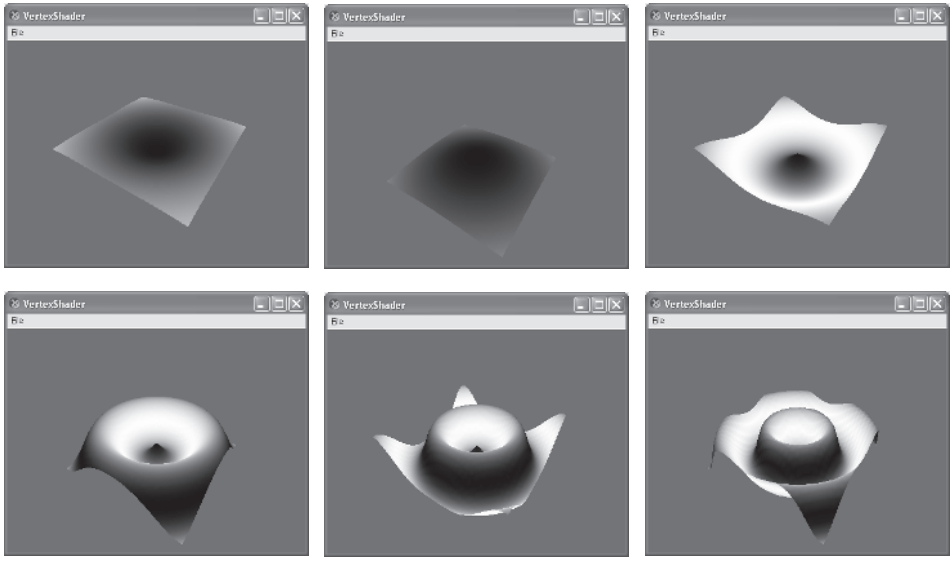
Chapter 8

Vertex Play

In this chapter we play with some of those vertex instructions that were discussed earlier.

D3DX9 Sample: VertexShader.cpp

The VertexShader sample project in the DX9 SDK is an elementary project that demonstrates the basic functionality of a vertex shader. It dynamically builds a 32x32 vertex array and uses 5,766 indices. I have tweaked the code a bit for our purposes to make the frame changes more apparent.



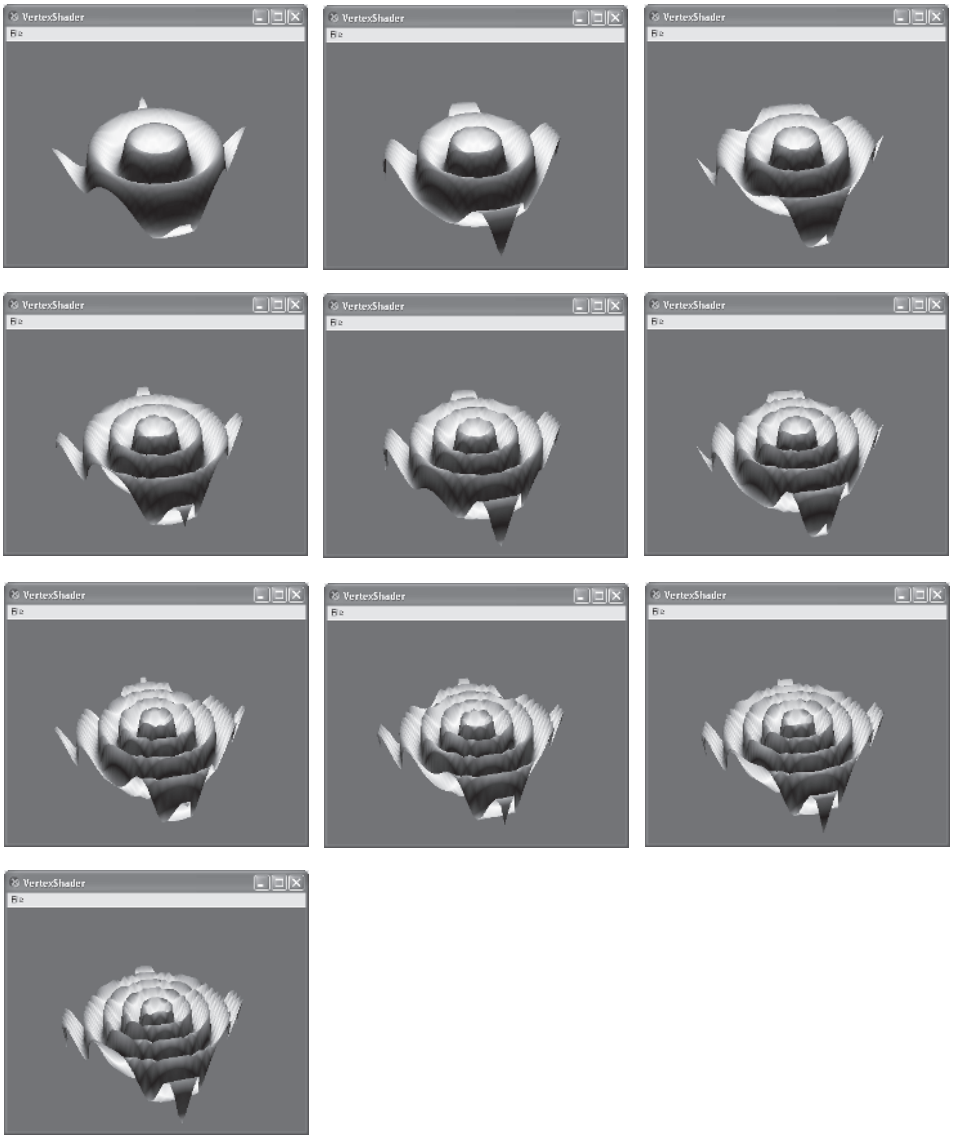


Figure 8-1: The resulting (single-stepped) animation sequence

Listing 8-1: VertexShader.cpp — Indice mapping

```
for(DWORD y=1; y<m_dwSize; y++)
{
  for(DWORD x=1; x<m_dwSize; x++)
  {                                                    // Pattern
    *pIndices++ = (WORD)( (y–1)*m_dwSize + (x–1) );  // 0
    *pIndices++ = (WORD)( (y–0)*m_dwSize + (x–1) );  // 1
    *pIndices++ = (WORD)( (y–1)*m_dwSize + (x–0) );  // 2
    *pIndices++ = (WORD)( (y–1)*m_dwSize + (x–0) );  // 3
    *pIndices++ = (WORD)( (y–0)*m_dwSize + (x–1) );  // 4
    *pIndices++ = (WORD)( (y–0)*m_dwSize + (x–0) );  // 5
  }
}
```

This builds a primitive list of indexed triangles: D3DPT_TRIANGLELIST.

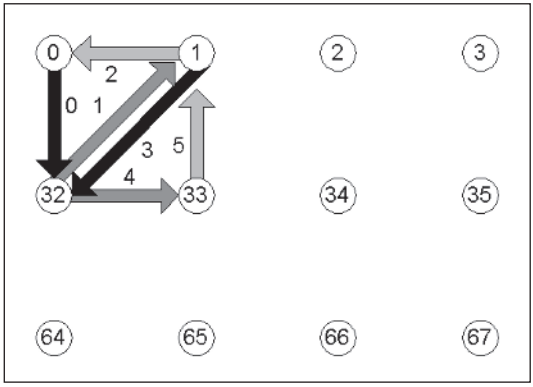


Figure 8-2: Indices resolution mapping

Now build a 32x32 array of vertices, or a mesh. Note that only a set of 2D {X, Y} vertices are generated, ranging from $x \in [-\pi/2, \pi/2]$ to $y \in [-\pi/2, \pi/2]$.

$$X = \left(\frac{N}{N-1} - 0.5\right)\pi \qquad Y = \left(\frac{N}{N-1} - 0.5\right)\pi$$

NOTE: This equation should be recognized as a cosine. Now why would that range of numbers be used here for this example? For simplicity. By using the range of π, it is much easier to cause the ripple effect that this sample demonstrates.

Listing 8-2: VertexShader.cpp — Vertex mapping

```
float f = (float)(m_dwSize–1);

for(uint y=0; y<m_dwSize; y++)
{
    float dy = ((float)y / f – 0.5f) * D3DX_PI;

  for(uint x=0; x<m_dwSize; x++)
  {
    *pVertices++ = D3DXVECTOR2(
       ((float)x / f – 0.5f) * D3DX_PI, dy);
  }
}
```

To manipulate a mesh, one normally alters each vertex within the mesh that is referenced by an index for each frame of the render to display the illusion of movement. This typically involves using a master list of vertices and transforming them and storing the result into a new list of vertices. This new list is passed into the renderer. With shaders, there is no need for this transformation and copy. The vertex shader handles the transformation and pipelines the result to the pixel shader for processing.

The constants needed by the vertex shader code are actually two sets. One set is fixed:

Vertex Shader: Constant Declarations

```
// Taylor series coefficients for sin and cos
D3DXVECTOR4 vD(   D3DX_PI,             // π      180°
                  1.0f/(2.0f*D3DX_PI), // 1/2π    90°
                  2.0f*D3DX_PI,        // 2π     360°
                  0.05f);              // 1/20   π/60

D3DXVECTOR4 vSin( 1.0f,                // 360°
                 –1.0f/6.0f,           // 60°
                  1.0f/120.0f,         // 3°
                 –1.0f/5040.0f);       // (1/14)°

D3DXVECTOR4 vCos( 1.0f,                // 360°
                 –1.0f/2.0f,           // 180°
                  1.0f/24.0f,          // 15°
                 –1.0f/ 720.0f);       // (1/2)°

m_pd3dDevice->SetVertexShaderConstantF(7, (float*)&vD,   1);
```

```
m_pd3dDevice->SetVertexShaderConstantF(10, (float*)&vSin, 1);
m_pd3dDevice->SetVertexShaderConstantF(11, (float*)&vCos, 1);

    D3DXVECTOR4 vSinCos1(D3DSINCOSCONST1);
    D3DXVECTOR4 vSinCos2(D3DSINCOSCONST2);

m_pd3dDevice->SetVertexShaderConstantF(12, (float*)&vSinCos1, 1);
m_pd3dDevice->SetVertexShaderConstantF(13, (float*)&vSinCos2, 1);
```

The other set changes from frame to frame. These are the "dynamic" constants that actually make the mesh appear to animate.

```
D3DXMATRIXA16 mat;
D3DXMatrixMultiply(&mat, &m_matView, &m_matProj);
D3DXMatrixTranspose(&mat, &mat);

D3DXVECTOR4 vA(sinf(m_fTime)*15.0f,  // time
        0.0f, 0.5f, 1.0f);          // {0, ½, 1}

m_pd3dDevice->SetVertexShaderConstantF(0, (float*)&mat, 4);
m_pd3dDevice->SetVertexShaderConstantF(4, (float*)&vA,  1);
```

The following sample code is slightly different from that found with the DX9 SDK, as the code has been optimized and modified slightly to make the effect of the sin-cosine operation more pronounced!

Listing 8-3: Ripple.vsh — version 1.1

```
vs.1.1

;      Constants:
;  c0-c3  - View + Projection matrix
;  c4       {time, 0.0, 0.5, 1.0}
;  c7       {π, π/2, 2π, 0.05}
;  c10    - first four Taylor coefficients for sin(x)
;  c11    - first four Taylor coefficients for cos(x)

dcl_position v0

; Unpack {X,Z} into {X,Y,Z,W}
mov r0.xz, v0.xy      ; {XZ}
mov r0.yw, c4.ww      ; {X,1,Z,1}

; Compute theta from distance and time
mov r1.xz, r0         ; {X,Z,}
```

```
mov r1.y, c4.y        ; {X,0,Y,_}   y = 0
dp3 r1.x, r1, r1      ; r=X²+0²+Z²       d2
rsq r1.x, r1.x        ; {1/d,_,_,_}  1/√r
rcp r1.x, r1.x        ; {d,_,_,_}
mul r1.x, r1.x, c4.x  ; {dt,_,_,_}  scale by time

; Clamp theta to – π .. π
add r1.x, r1.x, c7.x  ; {π +dt, 0t, Yt, _}
mul r1.x, r1.x, c7.y  ; {(π +dt) 0.5π, 0t, Yt, _}
frc r1.xy, r1.x       ; (–1,1)
mul r1.x, r1.x, c7.z  ; 2Nπ
add r1.x, r1.x,–c7.x  ; 2Nπ – π

;   _____________________
;   cos() & sin() calculation
; Compute first four values in  r2   r1   sin and cos series
mov r2.x, c4.w        ; d⁰   {1,_,_,_}
mul r2.y, r1.x, r1.x  ; d²   {1,S²,,}        {S,,,}
mul r1.y, r1.x, r2.y  ; d³   {1,S²,,}        {S,S³,,}
mul r2.z, r2.y, r2.y  ; d⁴   {1,S²,S⁴,}      {S,S³,,}
mul r1.z, r1.x, r2.z  ; d⁵   {1,S²,S⁴,}      {S,S³,S⁵,}
mul r2.w, r2.y, r2.z  ; d⁶   {1,S²,S⁴,S⁶}    {S,S³,S⁵,}
mul r1.w, r1.x, r2.w  ; d⁷   {1,S²,S⁴,S⁶}    {S,S³,S⁵,S⁷}
                      ;          cos            sin
mul r1, r1, c10       ; sin
dp4 r1.y, r1, c4.w    ;      sy =1x+1y+1z+1w

mul r2, r2, c11       ; cos
dp4 r1.x, r2, c4.w    ;      sx =1x+1y+1z+1w

; Set color   r1.x=cos   r1.y=sin
add r1.x, –r1.x, c4.w ; cos() = –sx + 1.0
mul oD0, r1.x, c4.z   ; rx = 0.5 cos()

; Scale height to make more pronounced
mul r0.y, r1.y, c4.z  ; ry = 0.5 sin()

; Transform position  .xyzw
dp4 oPos.x, r0, c0
dp4 oPos.y, r0, c1
dp4 oPos.z, r0, c2
dp4 oPos.w, r0, c3
```

With the release of vertex shader version 2.0 and higher, the sincos instruction became available, and so our shader code actually becomes lighter and thus faster. So the following code actually has

the same functionality but runs more quickly on a hardware-based shader!

Listing 8-4: Ripple2.vsh — version 2.0

```
vs.2.0

; Constants:
;  c0-c3  - View + Projection matrix
;  c4       {time, 0.0, 0.5, 1.0}
;  c7       {π, π/2, 2π, 0.05}
;  c12    - sincos1
;  c13    - sincos2

dcl_position v0

; Decompress position

mov r0.xz, v0.xy     ; {XZ}
mov r0.yw, c4.ww     ; {X,1,Z,1}

; Compute theta from distance and time
mov r1.xz, r0        ; {X_Z_}

mov r1.y, c4.y       ; {X,0,Y,_}   y = 0
dp3 r1.x, r1, r1     ; r=X²+0²+Z²       d2
rsq r1.x, r1.x       ; {1/d,_,_,_}  1 / √r
rcp r1.x, r1.x       ; {d,_,_,_}
mul r1.x, r1.x, c4.x ; {dt,_,_,_}  scale by time

; Clamp theta to – π .. π
add r1.x, r1.x, c7.x ; {π +dt, 0t, Yt, _}
mul r1.x, r1.x, c7.y ; {(π +dt) 0.5π, 0t, Yt, _}
frc r1.xy, r1.x      ; (–1,1)
mul r1.x, r1.x, c7.z ; 2Nπ
add r1.x, r1.x,–c7.x ; 2Nπ – π

mov r2.x,r1.x
sincos r1.xy, r2.x, c12, c13

; Set color   r1.x=cos(r)   r1.y=sin(r)
add r1.x, –r1.x, c4.w     ; `sx = –sx + 1.0
mul oD0, r1.x, c4.z       ; rx = 0.5 `sx

; Scale height
mul r0.y, r1.y, c4.z      ; ry = 0.5 sin()

; Transform position  .xyzw
```

```
dp4 oPos.x, r0, c0
dp4 oPos.y, r0, c1
dp4 oPos.z, r0, c2
dp4 oPos.w, r0, c3
```

A Wee Bit o' Simple Particle Physics

You may recall hearing little equations in school such as:

$E=MC^2$

Or:

Speed = distance/time

Or:

y = sin(t)

Well, we do not care about the speed of light for purposes of this book, but we do care about distance over time. Displacement of vertices is all about distance moved during a period of time. To support motion, every frame adjusts the time before calling the shader-based render. In its simplest form, a vehicle could move across the screen based upon velocity. In a slightly more complex version, taking advantage of the physics of gravity, a particle (of snow, rain, etc.) can start from an off-screen altitude and fall to the ground.

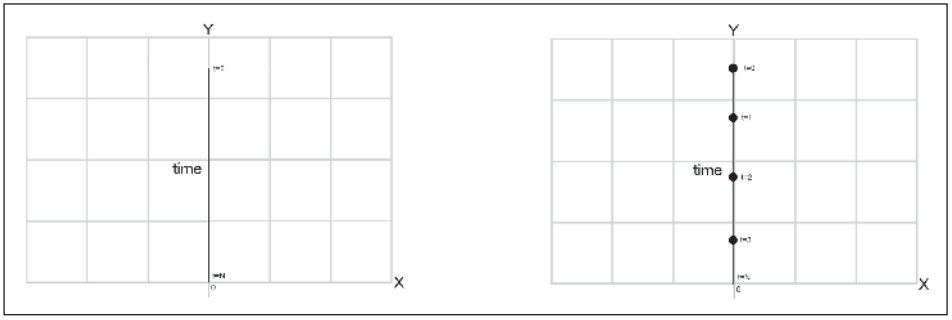


Figure 8-3: The figure on the left is a plot of x=0, y=h–t. With x fixed at 0, a vertical line plot is drawn where y is represented at time t=0, which would be the highest point at h, and t=N would be the resting position at ground level. The plot on the right is slightly more complicated, where a point is accelerating toward the ground using a simple gravity falling algorithm.

You may notice that the points get farther apart as the value of *t* increases. Applying a gentle breeze from east to west (left side of the screen to the right side) causes the points to have increasing displacement as *t* increases as well.

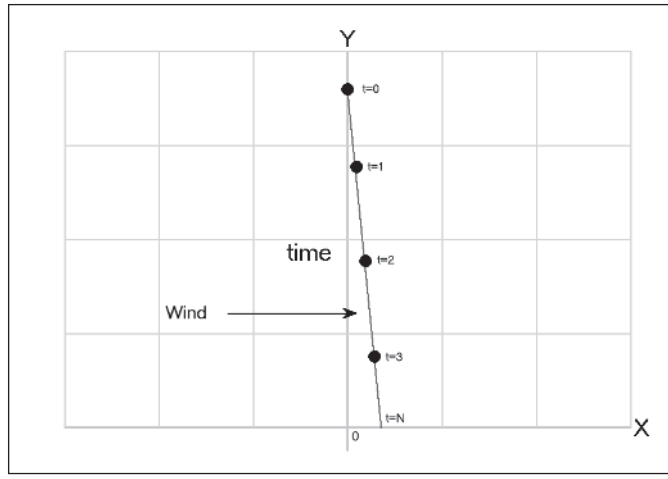


Figure 8-4: This figure shows that same plot over time but with the addition of a (+) displacement as t increases. For example, x = t0.1 So as t increases, x is no longer fixed at 0 but gets larger and thus farther away from 0. This would simulate an environmental factor, such as wind.

These same principles can be applied to other environmental effects, such as water, lava, firework simulations, etc. So let's take a quick look into particle fountains.

NOTE: A particle fountain tends to have a continuous motion such as a waterfall, water fountain, or lava fountain.

First let's look at basic particle paths. If you think in terms of simple geometry…

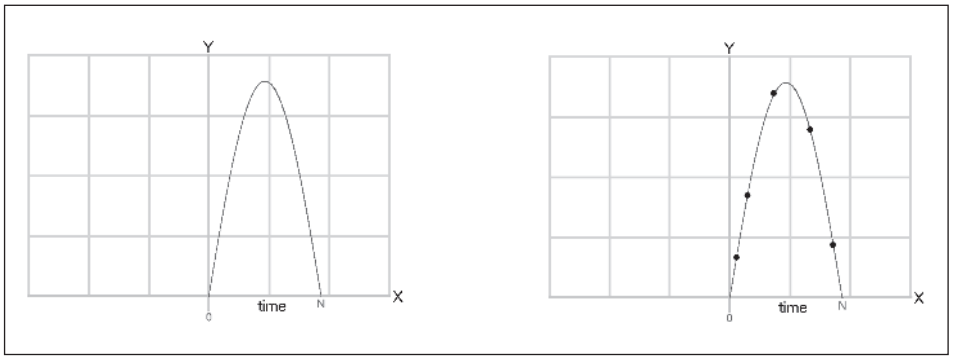


Figure 8-5: The figure on the left plots the path of a simple 2D graph of the equation y = h sin(6t/w). The figure on the right shows that same graph but with the random placement of points along the path.

…there is a position and placement of any time interval along the plot, so if one starts at the beginning t_0 and then progresses toward the end time t_{end}, the position changes along the path dictated by an equation. So now visualize a speck, or particle (a simple triangle or rectangle), starting, for example, at ground level and then accelerating upward, decelerating near the top of the arc, and then accelerating back down to the ground. Now imagine that reoccuring so $t_{end+1} = t_0$. A visually continuous motion of shooting up and falling back down occurs. Now let's imagine that t_0 does not necessarily have to be the starting position. So from mid-air, the items head upward and repeat the same continuous motion. Now instead of just one speck (poly), have a list of them, all starting at different points.

Now set them in motion by changing *t*. Voilà, a simple fountain. Other particle paths are determined by just a simple variation to a formula. For example, Figure 8-6 shows the plot variation just by scaling the Δy result as well as scaling the Δx result.

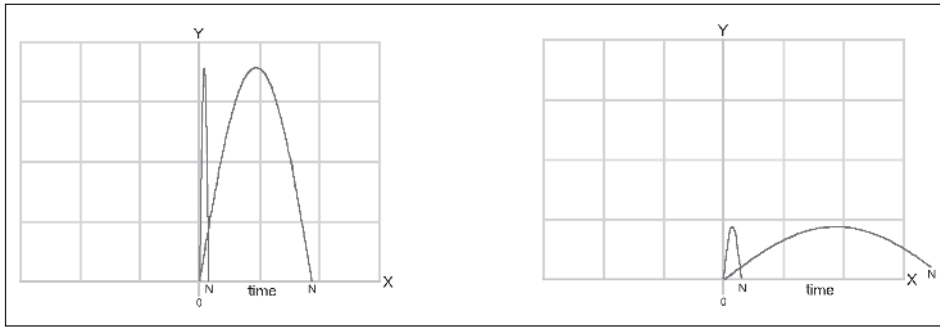


Figure 8-6: The figure on the left plots the path of a simple 2D graph of the equation y = h sin(6t/w) but with the width varied by adjusting w to demonstrate the path a particle would take if it landed closer to the base. The figure on the right scales the plot in size by adjusting h, demonstrating squashing. These demonstrate various paths that a particle might take.

Well, we have been looking at this from a 2D perspective, so now let's look at it from a topographical perspective. As *t* increases, the plot point moves farther away from the center point.

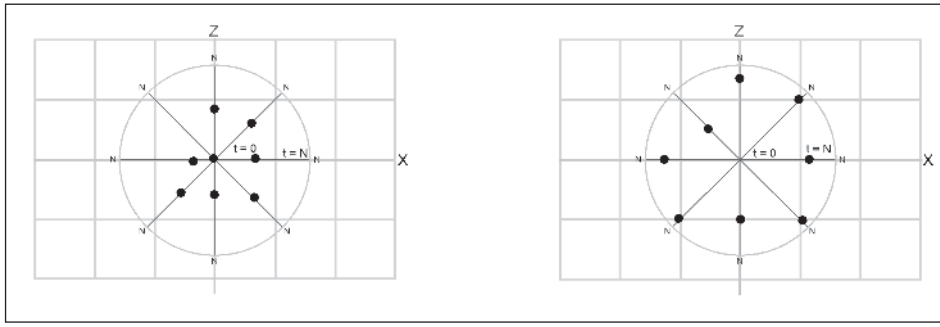


Figure 8-7: The figures demonstrate the points starting at the center at t=0, and radiating farther outward from the center as time increases, eventually coming to rest at the maximum radius as t=N.

So let's examine a simple physics equation:

$$\frac{dr}{dt} = \mathrm{v}$$

In this particular case, a velocity is a vector. Each vertex representing a face within a polgon is manipulated by the vector representing the rate of change over time. As we are dealing with a simple particle fountain with a single formula, only one velocity

is required. If, for example, we were to bounce each polygon upon striking the ground, a change of velocity is represented, thus a second vector representing a new change in direction would be needed. But to get back to our simple fountain, each time slice (rendered frame) deals with a moment (a difference) in time. We can represent this velocity with the formula in a different form:

$$\frac{\Delta r}{\Delta t} = \mathrm{v}$$

But we need to render particles in terms of {XYZ}, and so a form similar to the following is used:

$$\Delta \mathrm{x} = \Delta \mathrm{t} \bullet \mathrm{v}_{\mathrm{x}} \qquad \Delta \mathrm{y} = \Delta \mathrm{t} \bullet \mathrm{v}_{\mathrm{y}} \qquad \Delta \mathrm{z} = \Delta \mathrm{t} \bullet \mathrm{v}_{\mathrm{z}}$$

Not to put it too simply, but the result of a velocity and time calculate a position in 2D and, in our particular case, 3D space.

Well, to some it may seem to be more complicated than need be, but for purposes of smooth motion with characteristics of reality, physics-type equations are needed. Since this book only lightly touches upon some aspects of physics, either dust off your old physics textbooks from college or visit your local university and take a class (or at the very least visit their bookstore and buy one along with the accompanying workbooks). Physics can be fun, especially in conjunction with experimenting with antifriction devices (like an air hockey game), but that is another story.

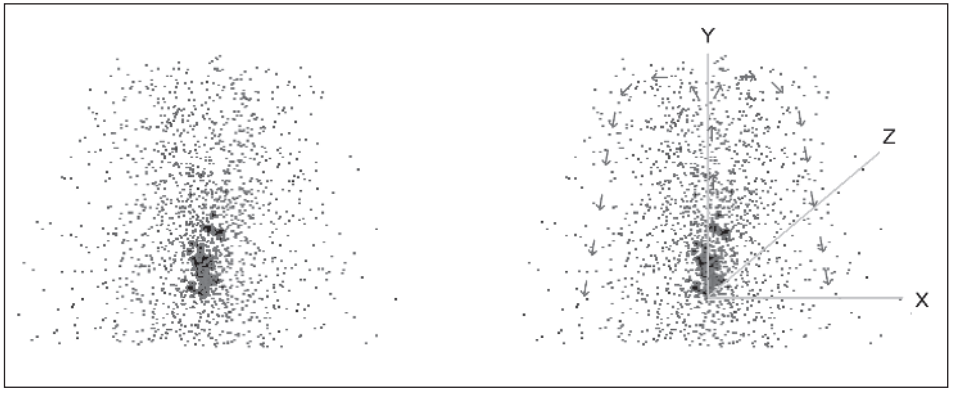


Figure 8-8: A 3D view of a particle fountain

HINT: By putting this all together into a 3D context and adding a *lot more* polygons (particles), a nice full fountain is formed!

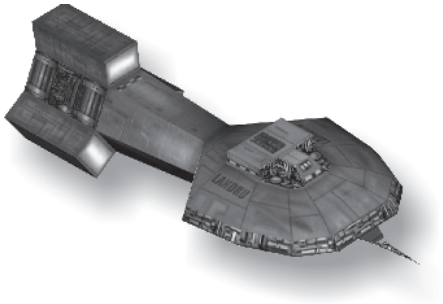
Pretty cool looking, huh? (It looks much better animated though.)

Of course, this can be made even more complicated with extra coding logic and placement of other scene elements, such as lighting, friction, etc. The newer branching instructions can also be utilized to switch between equations, for example, depending on conditions. Quite often, one basic equation is not enough to cover the complete path. Although a random function is not available during the render of a frame, it can be introduced to add a per-frame variability. Keep in mind that any position adjustment is not remembered from frame to frame. So any implementation of this kind of alteration vertex position will be lost after the render. Of course, that does not mean that the original vertex couldn't be altered before beginnning any render or from frame to frame, but this later operation tends to slow down frame rates.

NOTE: All this discussion on particle effects brings back fond memories of Yosemite when I was a child. The Yosemite Firefall was a nightly event in Yosemite for 88 years between 1872 and 1968. Glowing embers of a large bonfire up on the high "glacier point" in Yosemite would be pushed over the side and rain down upon the ground below at Camp Curry. It was a cascade of glowing embers similar to a waterfall.

Other natural effects, such as wind-blown bushes, etc., can use vertex technology. Your exploration into vertex shaders does not need to end here! There are other more advanced books on the subject available as well as Internet downloads, etc.

This page intentionally left blank.



Chapter 9

Texture 101

Normally, a texture is related to pixels, not vertices, but there are a few uses of textures by vertex shaders. One of them is bump mapping, as the elevation of the pixel is actually an adjustment of the vertex position, and so the topic of textures is discussed here and then again in relation to the pixel shader instructions.

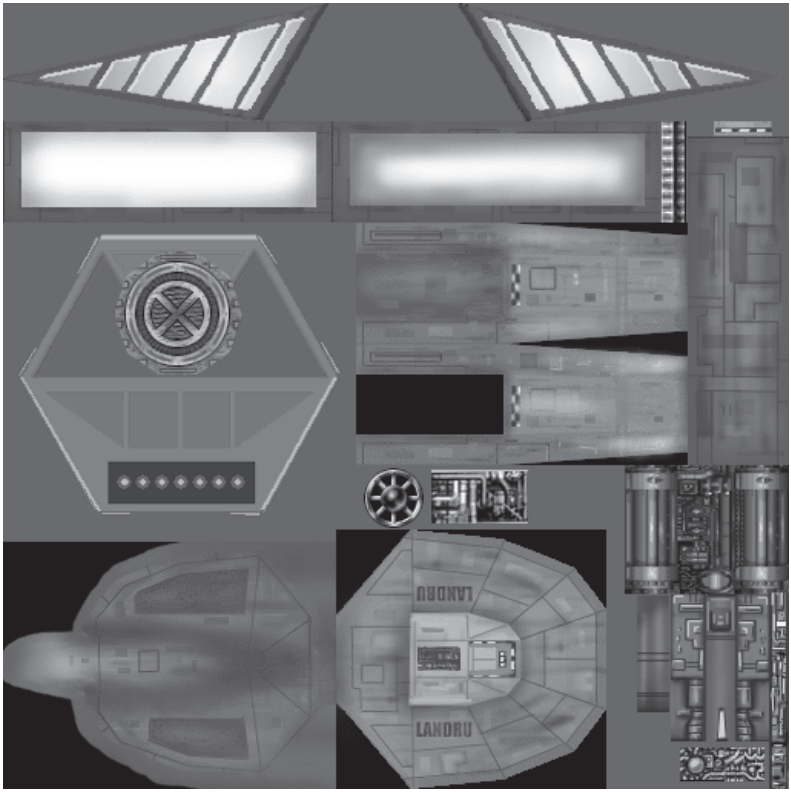


Figure 9-1: Texture for a space freighter (compliments of Ken Mayfield)

Before continuing, the following questions must be asked!

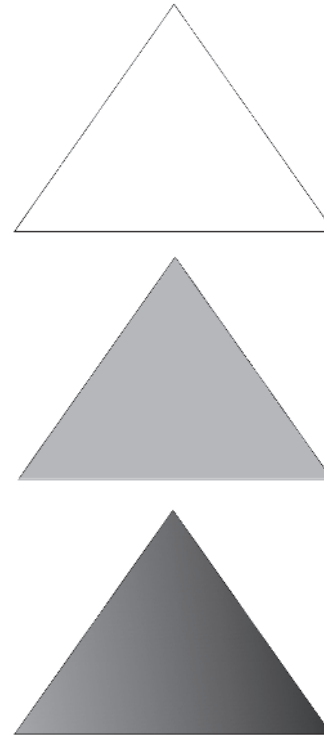
What Is a Polygon?

Originally, 3D graphics involved building 3D representations constructed with a series of polygon shapes using primitives. Our attention within this book is focused on the three-sided polygon (a face). Two of these triangles make up a quadrilateral. In this medium, the properties of the polygon's surfaces (skin) in conjunction with color are the most important. In the early days of 3D rendering, shade-generated images resulting from light sources in conjunction with color had a plastic appearance — fortunate only if you wanted a stained glass window.

What Is Shading?

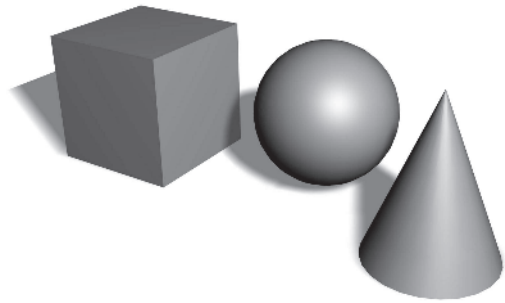
Shading is the method of filling each face of a polygon when it is rendered. There are various levels of shading of increasing complexity. The following are the principal methods of shading.

- **Wireframe**: The basic skeletal structure that makes up the shape
- **Facet shading**: This is also referred to as flat shading. Each face is colored by a single constant color.
- **Gouraud shading**: This is also referred to as color interpolation shading or intensity interpolation shading. It shades using two steps, the first of which is a light intensity calculation for each vertex. The second step is a bilinear



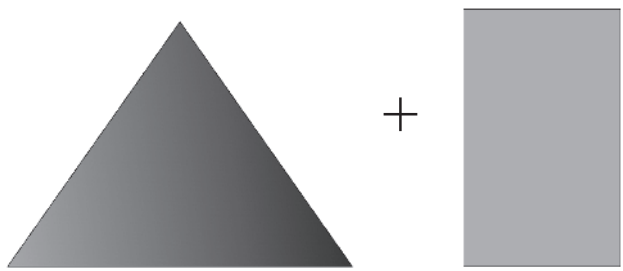
interpolation calculation of the intensity of pixels between the three corner vertices of the face.

- **Phong shading**: This method is more complex than Gouraud shading, as it uses a high number of calculations to achieve a realistic rendering, such as per-pixel lighting, which generates nice specular highlights.



What Is Color Mixing?

Of course there is more to color than just shading. There are issues of color mixing. When mixing colors, such as the blending (summation) of two colors, one needs to be careful of what happens when the sum of the two color values exceeds that of a possible limit.

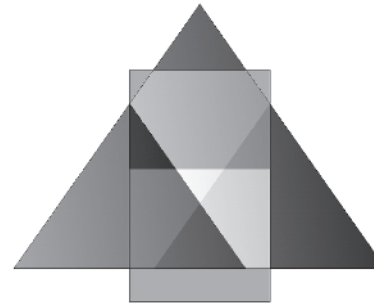


We learned this using saturation to limit values and prevent wrap-around of values. The following samples are based upon RGB 8:8:8.

$D_{Red} = A_R + B_R$ $D_{Green} = A_G + B_G$ $D_{Blue} = A_B + B_B$

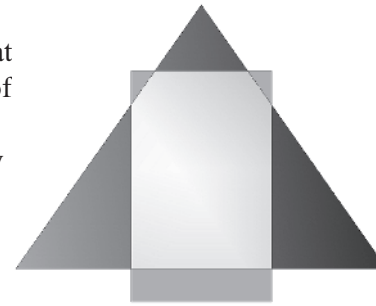
- **Color Clipping**: Data wrap-around. Any overflow resulting from the summation wraps from 255 through 0, so effectively a mod 256 is performed and then stored as the resulting pixel.

  D = (A + B) MOD 256



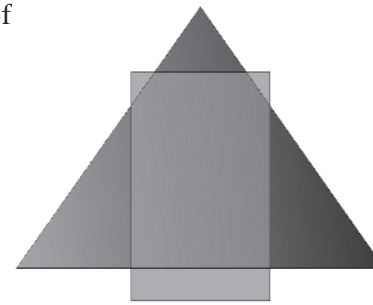
- **Color Saturation**: The result of the summation is saturated at the maximum intensity value of 255 and not allowed to wrap, such as in "clipping." You may remember that the shader instruction modifier _sat engages the saturation mechanism.

  D = MIN(255, (A + B))



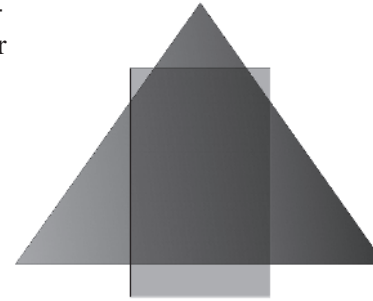
- **Color Averaging**: The result of the averaging of the two pixels

  D = (A + B + 1) ÷ 2



- **Color and Light**: When working with light, the surface color is affected by the intensity and color of the light. Effectively, the product of the two normalized values is used to calculate the pixel color.

  D = ((A÷255) (B÷255)) 255



An interesting artifact of this algorithm is that the light can never add a color that is not in the

original surface color. The algebraic law of dominance is 0N=0. So, for example, if there is no blue surface color and even if a white light source is provided, there will still be no blue pigment in the resulting pixel. Something else that should be kept in mind is the algebraic law of multiplicative identities 1N=N. Thus, the light value never increases the surface color value; it only reduces (darkens) it.

Ah, but we are not done with color yet. There is also vertex color lighting. This is typically used in darkened areas, such as caves or dark shadowy areas. Sometimes individual light sources do not generate quite enough light to illuminate the area, and so adding color to individual vertices helps.

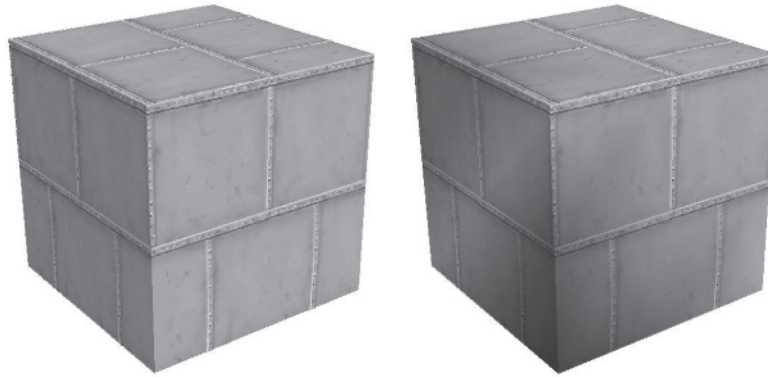


Figure 9-2: Metallic texture-rendered cube without vertex lighting (on the left) and with vertex lighting (on the right). It also helps make great-looking rust. (Compliments of Ken Mayfield)

What Is a Texture?

A texture is in essence a 2D pixel bitmap that gives an image the appearance of having a degree of roughness. A texture can be thought of as a material that wraps the surface of an object (so, in essence, a substitution). Instead of using shading with colored light, a pixel from the source image (the texture) bitmap (a texel) is placed at the destination pixel location. For example, the skin of an orange can be thought of as that orange's texture. It contains the color information that is wrapped around the edible portion,

and the dimpled effect on the surface gives it the appearance of roughness. The texture is stretched to fit. A model of the Earth (a globe) is designed in the same way. The colored atlas of the world is wrapped around the sphere, thus painting the object into 3D space.

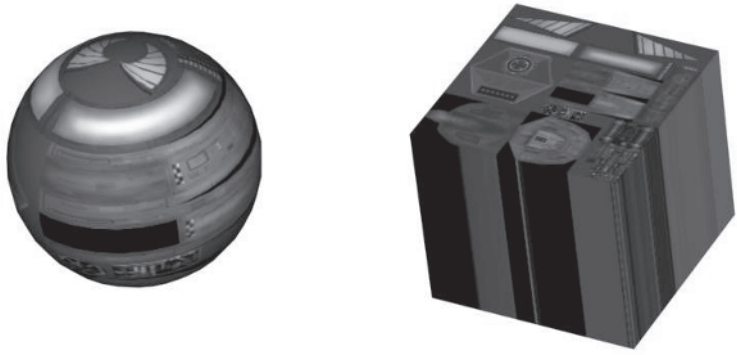


Figure 9-3: Texture mapped onto a sphere and cube (compliments of Ken Mayfield)

Note the continuation of the texture on the edge of the cube and how the edge pixels are stretched along the side of the cube. If you really want to see textures for planet Earth instead of a high-tech ball, check out the Bump Map Earth sample:

DX9SDK\Samples\C++\Direct3D\BumpMapping\BumpEarth

But since we are just discussing textures, let's continue. An object can be literally any shape, even a cube, but for our purposes of rendering, it is made up of a collection of triangles of which each triangle is formed around three positions in space, which are its vertices. This, of course, was discussed in the chapters related to vertex shading. In the case of the following spacecraft, the 2D art is wrapped around a polygon model, creating the illusion of volume and depth.

In the following figure, notice the dramatic difference between just having a wireframe image versus having a texture-mapped image.

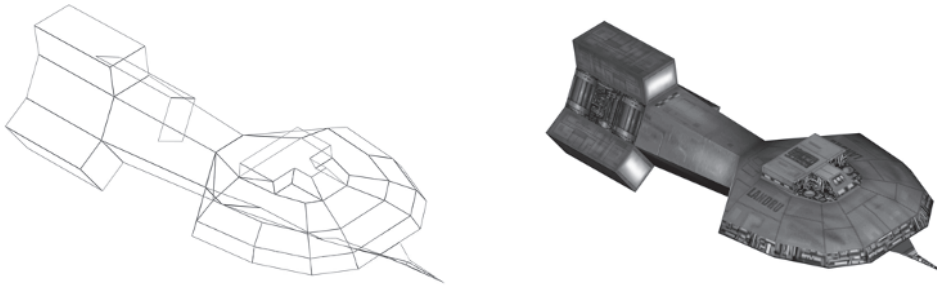


Figure 9-4: Wireframe and texture-based rendered space freighter (compliments of Ken Mayfield)

What Is Texture Filtering?

New filtering methods are being developed from time to time as technological innovations come about. For our purposes, here are some filtering techniques:

- **Point Sampling**: Closest pixel (texel) in the texture map
- **Linear**: An interpolation (average) of the four nearest texels
- **Mipmapping**: Closest texel within a single mipmap
- **Linear Mip**: A single texel from two different mipmaps are averaged together.
- **Linear Mip Nearest**: The four closest texels are used with a bilinear filter on one mipmap.
- **Trilinear Mipmapping**: Interpolates color using the two closest mipmaps, each containing four texels

Texture Dimensional

Textures can come in many shapes and sizes, but for purposes of compatibility across video cards and platforms, a texture should be designed with a power of two, thus 2^n pixels, and with a square pixel resolution. Some of the pixel shader instructions require the texture to be 2^n pixels in order to work properly. This size should be larger than 8x8 and smaller than 256x256 to be compatible with most 3D hardware render cards. But with this book, we are not dealing with most 3D hardware here. We are dealing with

top-of-the-line graphics cards that support pixel shaders, which do not have square texture limitations and have a higher common pixel resolution limit of 2048x2048. Even higher resolutions are available, but this is a good common base. Another plus is that these newer boards have large quantities of graphics memory — 64/128/256 meg — available for textures. Therefore, a lot of functionality is available for a product designed only for graphics cards with shader support. To get the best performance, higher resolution textures must be created. But to support older, less capable cards, lower resolution bitmaps need to be created. (Unfortunately, game consoles tend to fall into this category, where only the Xbox has shader support at the time of publication.) This means a lot more work for the artist, but game company managerial types typically lean toward one set of texture art for time and budgetary cost savings. It can be said that the same kind of decisions might be made for shaders, but fortunately (or unfortunately, depending on your point of view) the different shaders are not compatible and therefore different versions have to be made.

So first let's see determine the largest size texture that is supported by your current graphics card. Remember the version checking in Chapter 2 and how each Direct3DDevice object type used an enumerated D3DCAPS9 data structure? We are now interested in these two data members:

- `DWORD MaxTextureWidth; // maximum texture width`
- `DWORD MaxTextureHeight; // maximum texture height`

These specify the maximum size texture that is supported by the enumerated device object.

The data member *TextureCaps* has various flags of which the following are of interest:

- D3DPTEXTURECAPS_POW2 (=0x00000002)
- D3DPTEXTURECAPS_SQUAREONLY (=0x00000020)
- D3DPTEXTURECAPS_MIPMAP (=0x00004000)

D3DPTEXTURECAPS_POW2 indicates that the texture supported by the device object must be 2^n.

The SQUAREONLY flag, if set, indicates that a texture must be square. Textures come in many color depths, which are really defined by the capabilities of the graphics cards. The following bit combinations are represented by {*Alpha*:Red:Green:Blue}, where the alpha is optional. If only three arguments are shown, then alpha is zero bits. The typical depths are 15 {5:5:5}, 16 {5:6:5}, 24 {8:8:8}, and 32 {8:8:8:8} bits per pixel using a fixed palette and 8 bits using a 256-color palette lookup table. There are other color depths, which may or may not contain alpha (translucency) information, such as the following configurations: {2:10:10:10}, {1:5:5:5}, {4:4:4:4}, {3:3:2}, etc.

Each vertex within a polygon needs to be mapped to a coordinate on the texture. Each texture is mapped with a normalized UV number ranging from 0.0 to 1.0. This allows a texture to be folded along the axis of a line on the plane of the texture. This texture is, in essence, folded and stretched between the three vertices that make up the three points of a face.

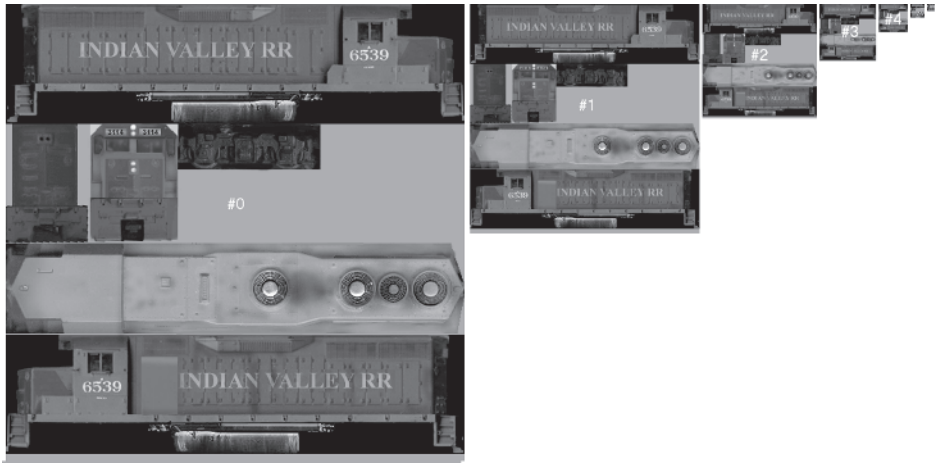


Figure 9-5: Original square locomotive texture mipmap (compliments of Ken Mayfield)

Textures also come in a form known as a mipmap. This is a sequence of textures that allow an image to be progressively scaled. When a single bitmap image used in rendering is scaling from a large to a small size, sparkle effects and shimmer will appear and disappear in the image during the image shrinking or expansion. As a mipmap is progressively scaled, the bitmap 2^n (for example, 512x512 (2^9)) would appear when the rendered

object is closest to the camera to maximize the amount of detail. Each progressive image as the image moves farther away from the camera is reduced in size by one half $2^{(n-1)}$; thus the next farther image would be 256x256 (2^8), then 128x128 (2^7), etc., down to the minimum sized texture supported. Typically, this is a 1x1 bitmap supported by most of the later hardware, but unfortunately not all hardware render-supported video cards actually support that size. Typically, the minimum supported size is an 8x8.

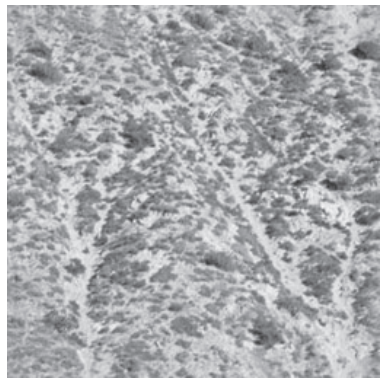
The MIPMAP flag, if set, indicates that a mipmap is supported by the device.

The good news here, however, is that since this book is about video cards with programmable shaders, and if we sort of ignore software shader support, we are not quite as restricted by texture dimensions or minimal texture sizes.

Although mipmaps do not have to be square, for best portability between video cards, they should be kept square. Alternatively, if two sets of art are designed, one that is square for best compatibility and one that is not, then a video card can be used to its best advantage. The only drawback is that for each bitmap, a set of bitmap pixel references (UV) needs to be maintained for each vertex.

Bump Mapping

Bump mapping is a process of combining textures and adjusting each *u* and *v* pixel texture coordinate with an elevation (contour) displacement map to render an image with an illusion of depth. Notice in the following two images that there is a correlation between what appears to be hiking trails on the left and the dark interconnecting veins on the right.



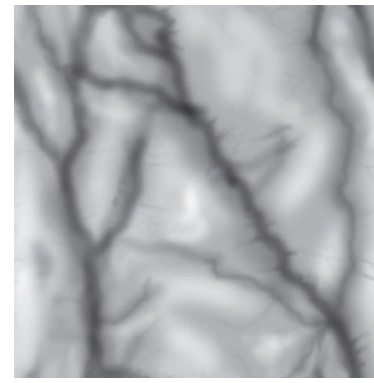
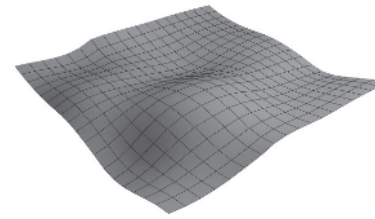
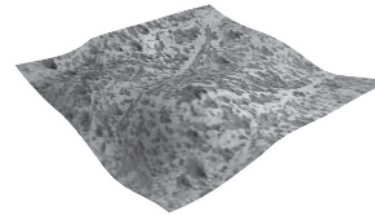


Figure 9-6: Terrain texture map on the left and its accompanying bump map on the right (compliments of Ken Mayfield)

Linear topography defined by polygons and their vertices (right).



You can just imagine a hiker walking around the sagebrush along the trails. The linear topography shown on the left side of Figure 9-6 is mapped by the terrain texture. The image has taken shape.



Those hiking trails turn out to be ravines of various depths resulting from terrain displaced by the bump map, which gives the illusion of realism. Pixels with darkness as the intensity approaches black displace the elevation in a negative height, and when the intensity approaches white, elevation is increased from the mean.

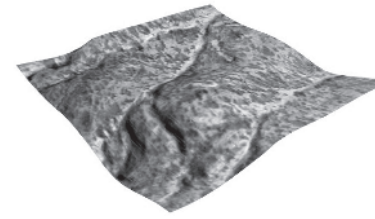


Figure 9-7: Topographical mesh on the top, terrain mapped with texture map in the middle, and rendered with bump map on the bottom (compliments of Ken Mayfield)

The bump map is an image containing a specialized pixel format that includes ($\Delta u\ \Delta v$), delta values for u and v, which range from {–1.0…1.0}, and occasionally a luminance component, L, which ranges from {0…255}. Remember that under DirectX, you need to verify that bump maps are supported by your graphics card. This support is determined if either one of the following flags is set:

- D3DTOP_BUMPENVMAP
- D3DTOP_BUMPENVMAPLUMINANCE

These, in essence, indicate that per pixel bump mapping is supported. This is in conjunction with using an environment map in the following texture stage but one supports luminance and the other doesn't. This is only supported for color operations.

The following function checks for bump map support on the current device:

Listing 9-1: C++

```
bool IsBumpMapSupported(IDirect3DDevice9 *pDev)
{
   D3DCAPS9 d3dCaps;

   ASSERT_PTR(pDev);

   pDev->GetDeviceCaps(&d3dCaps);

          // Check to see if either bump map blending is supported.

   if (0 == d3dCaps.TextureOpCaps
        & (D3DTEXOPCAPS_BUMPENVMAP
          | D3DTEXOPCAPS_BUMPENVMAPLUMINANCE))
   {
      return false;
   }

          // Are at least three blending stages supported?
   return (3 <= d3dCaps.MaxTextureBlendStages);
}
```

As texture memory is typically a limited resource (depending on the graphics card), topographical textures are typically designed to

be seamless when tiled into a checkerboard type pattern, such as in the following figure.

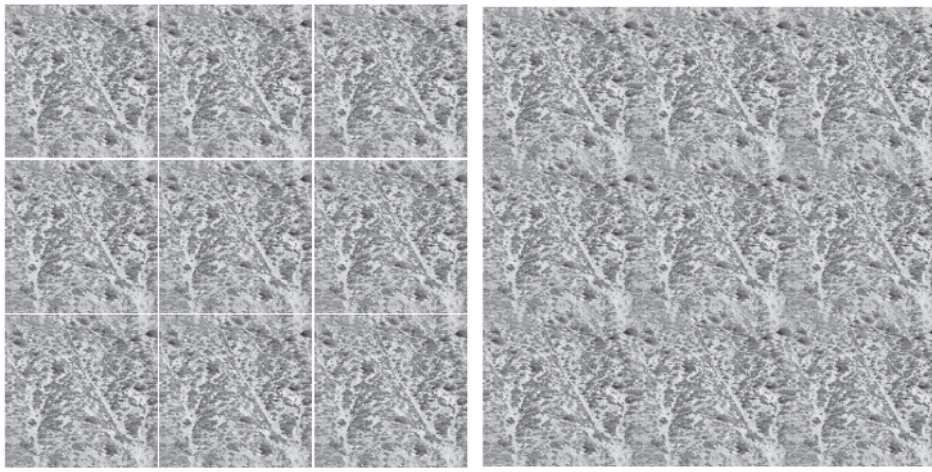


Figure 9-8: On the left, the topographical terrain is arranged as a 3x3 tiled checkerboard with the seams exposed; the right is without the seams. Note that even though a pattern can be detected, there are no seam lines!

On the left of Figure 9-8, note that the the left edge of the middle cell is designed to wrap seamlessly into its own right edge and the top edge to wrap with the bottom so textures can be adjacently placed, such as in a checkerboard grid.

The problem with tiled textures, however, is that you may notice the appearance of a repeating pattern, such as the four diagonal slashes cut into the surface at the left of Figure 9-9. Notice, however, that in the texture map on the right, the single bump map is used to camouflage, thus helping to hide the reused texture and allowing resources to be stretched.

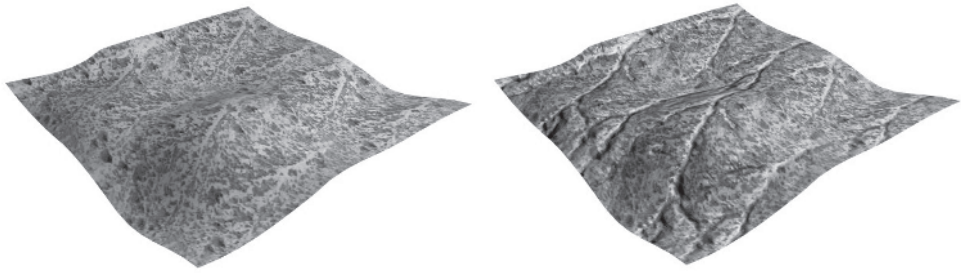


Figure 9-9: Topographical terrain mapped upon the same vector grid, with a 2x2 tiled texture map on the left and rendered with a 1x1 bump map on the right (compliments of Ken Mayfield)

Normally in 3D art (as well as 2D isometric views), the tiling effect is broken up by the insertion of other scene elements, hence other textures.

The following vertex shader code combines a texture and a bump map to produce a bump map.

Listing 9-2: C++

```
D3DVERTEXELEMENT9 decl[] =
{
   {0, 0, D3DDECLTYPE_FLOAT3,   D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_POSITION, 0},
   {0, 12, D3DDECLTYPE_FLOAT3,  D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_NORMAL, 0},
   {0, 24, D3DDECLTYPE_FLOAT2,  D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_TEXCOORD, 0},
   {0, 32, D3DDECLTYPE_FLOAT2,  D3DDECLMETHOD_DEFAULT,
                                D3DDECLUSAGE_TEXCOORD, 1},
   {0, 32, D3DDECLTYPE_FLOAT2,  D3DDECLMETHOD_LOOKUP,
                                D3DDECLUSAGE_SAMPLE, 0},
   D3DDECL_END()
};


        D3DXMATRIX m;
        D3DXMatrixMultiply(&m, &m_matWorld, &m_matView);
        D3DXMatrixMultiplyTranspose(&m, &m, &m_matProj);
        m_pd3dDevice->SetVertexShaderConstantF(4, (float*)&m, 4);
//c4

        float c[4] = {0.15f, 0, 0, 0};
        m_pd3dDevice->SetVertexShaderConstantF(4+4, c, 1);  //c8
```

Listing 9-3: Vertex shader

```
vs.1.1

dcl_position0 v0
dcl_texcoord0 v1            // Texture
dcl_sample0   v2
dcl_normal0   v3
dcl_texcoord1 v4

mul r2.xyz, v3.xyz, c8.xxx  // Scale the normal
mul r2.xyz, r2.xyz, v2.xxx  // Multiply displacement
add r2.xyz, r2.xyz, v0.xyz  // Add normal to pos.
```

```
mov r2.w, v0.w          // Copy input pos.
m4x4 oPos, r2, c4       // xform pos. to the projection
mov oT0, v4             // Copy input texture#1 coord
mov oT1, v4             // Copy input texture#1 coord
mov oT2, v1             // Copy input texture#0 coord
```

Figure 9-10: The image on the left is a spherical render using only the terrain texture. The texture image in the middle is the environmental map. The image on the right is a spherical render using only the texture terrain and the environmental map.

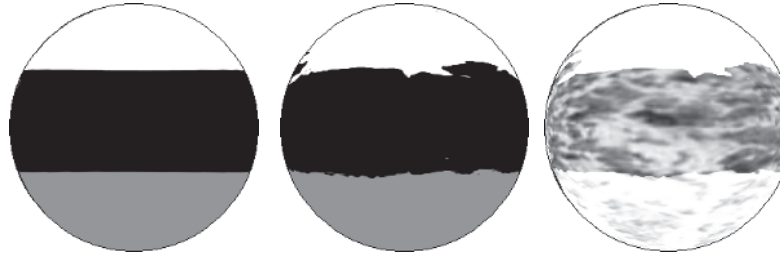
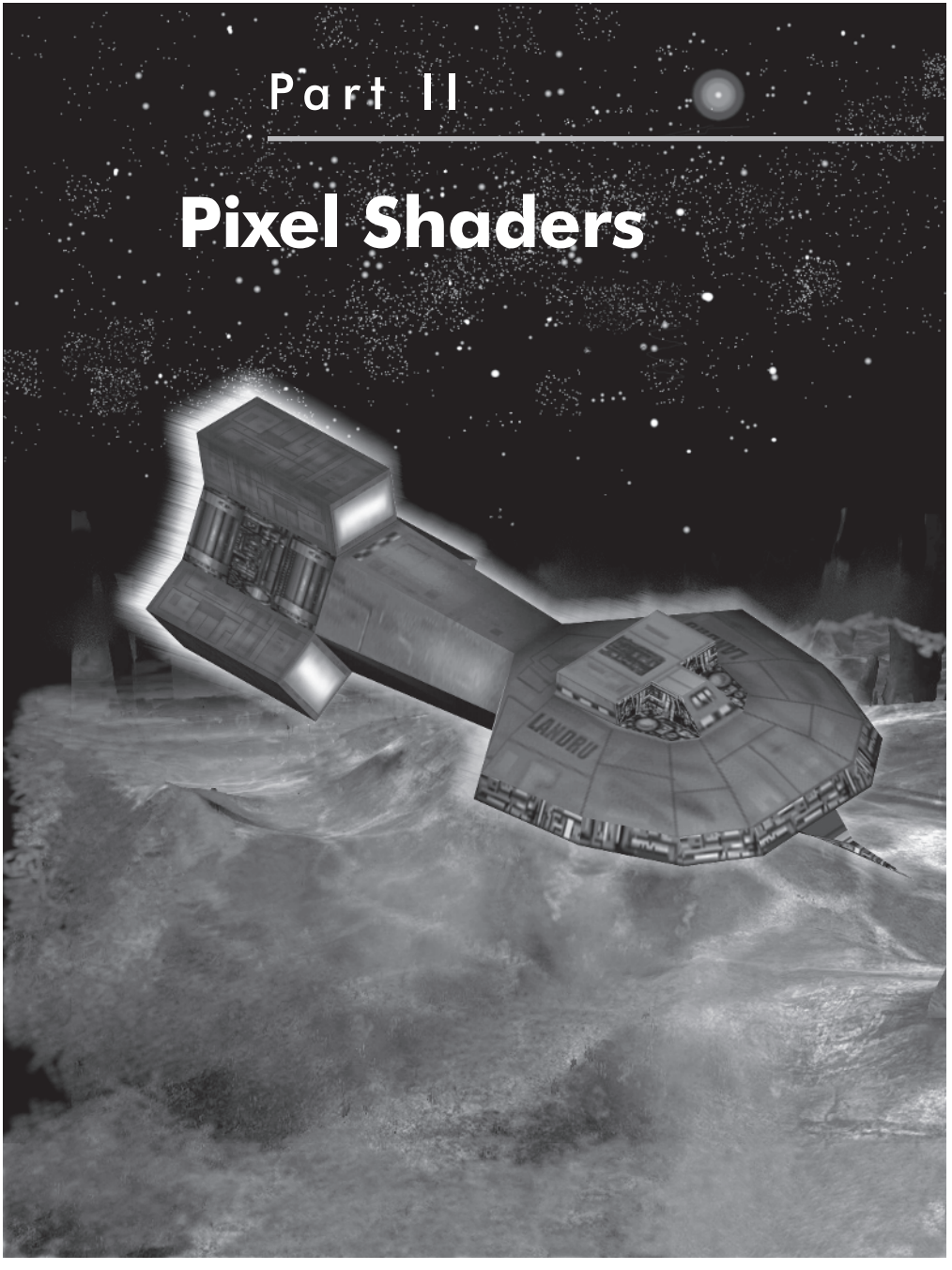


Figure 9-11: These images are the result of a spherical render where the image on the left was rendered using only the environmental map. The middle image is a render using the environmental map and a bump map. The image on the right is a render using the environmental map, bump map, and terrain texture.

Note that where the environmental map is full intensity (white), the destination render is bright white. Where the environmental map had intensity off (black), the full bump mapped texture is rendered. Anywhere a medium intensity environmental map exists, some (washed) bump mapped texture is rendered.

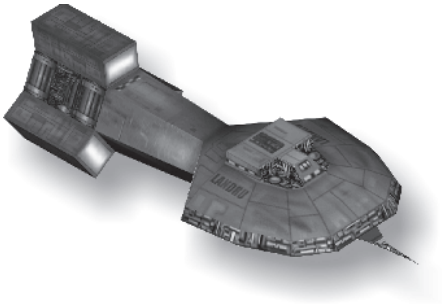
This page intentionally left blank.



Part II

Pixel Shaders

This page intentionally left blank.



Chapter 10

Pixel Shaders

I had a problem writing this chapter. Other books covering this subject had mistakes, a lot of white space, and a lot of duplicate information. But why regurgitate the same information again here? I could be evil and make you flip back and forth between the vertex shader and pixel shader parts of this book. But then again, replicating the information would make it seem as if this book was cheating the reader by filling it with duplicate information. (Wow — two book halves for the price of one! Ah, decisions, decisions.)

Okay, I decided to go with forcing you to flip back and forth. It won't be too bad, so don't worry. If you read and understood the vertex part, then this part should be just a reminder with the need for page flipping kept to a minimum.

This part of the book about pixel shaders is divided into two chapters. This first chapter covers every pixel shader instruction not related to textures, and Chapter 11 covers only textures. The subject of textures is large enough to warrant its own chapter and also made for a nice dividing line for breaking up this subject matter. I should note, however, that occasionally a texture instruction will be referenced as needed but not really discussed in this chapter.

Pixel Shader Version Checking

Let's revisit the "Version(s) Determination" section from Chapter 2 for a moment. It discussed vertex shaders and how to detect and enumerate them for selection as vertex shader types that could be supported. Since we were only focusing on vertex shaders, the pixel shader version detection was sort of glossed over.

Within each Direct3DDevice object type was a D3DCAPS9 data structure with two data members of which the first one, VertexShaderVersion, was of primary interest.

```
DWORD               VertexShaderVersion;    // vertex shader version
DWORD               PixelShaderVersion;     // pixel shader version
```

The second one, PixelShaderVersion, is of interest to us now. The same major/minor version checks are done:

```
Maj = D3DSHADER_VERSION_MAJOR(pCaps->PixelShaderVersion);
Min = D3DSHADER_VERSION_MINOR(pCaps->PixelShaderVersion);

DWORD Version = D3DPS_VERSION(Maj, Min);
```

The version number indicates the highest version level that the pixel shader hardware can support; in essence, pixel versions less than or equal to a specified version are compatible!

This is actually much simpler than vertex shader version detection!

What is not so easy, though, is the programming of pixel shaders for the various hardware shader supported cards that are out there. Many special rules apply depending on what version of card is supported and what version of code you are writing. One thing to remember is that early version numbers are not software emulated as they are for the vertex shader, and so care must be taken as to which shader code is being accessed for which version of hardware. This means that for your shipped application, you could possibly generate different versions of shader code, where those with newer, higher performance cards have more capabilities of the hardware taken advantage of compared to those with older, less capable cards.

Pixel Shader Registers

From Chapter 3, "Vertex Shaders," you should find a familiarity between the basic functionality of vertex shaders and pixel shaders. They are essentially the same. Therefore, for the most part, the registers should be similar whenever the same functionality exists. There is still a need for constant floats, integers, and Booleans, as well as temporary storage registers and a predicate, etc. There is, however, a difference, as the vertex-specific registers do not apply to pixel shaders, and pixel-specific registers do not apply to vertex shaders. So there are some differences. Please note that the constant access from a C/C++ application is also slightly different, since even though they both have constant registers and they are the same type of registers, they are kept separate from each other and thus need an alternate API method of access.

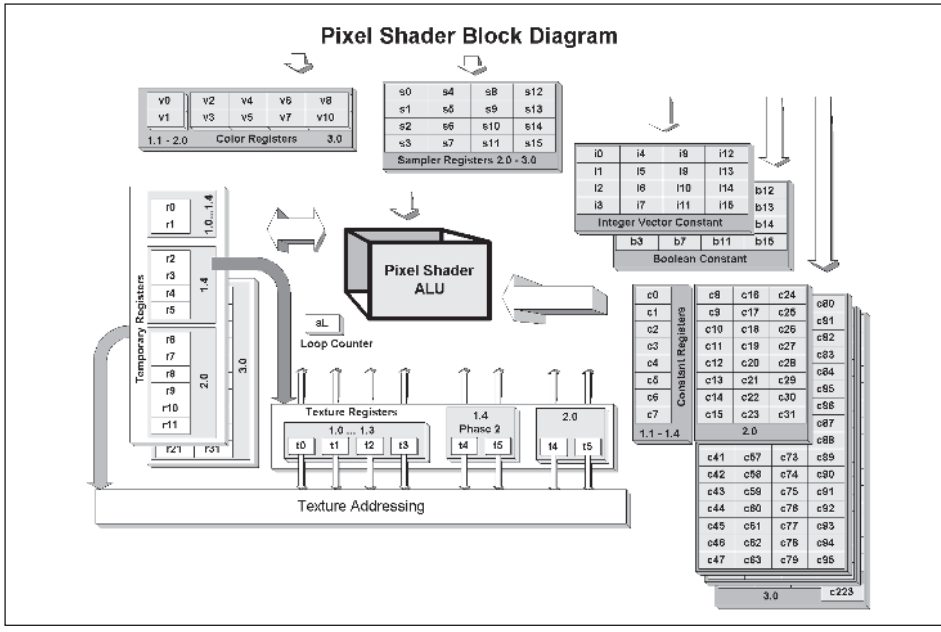


Figure 10-1: Pixel shader block diagram

c0 ... c7 *c8 ... c31* *c32 ... c223*	The constant read-only registers are each read-only quad single-precision floating-point vectors. Depending on the version, software emulation, and hardware factors, the highest register index can range from {c0…c223}. [1.1 … 1.4]=c{0…7}, [2.0]=c{0…31}, [2.0$_x$, 3.0]=c{0…223}. They are set either with the use of the *def* instruction or by calling an external function from an application. Only one vector constant can be used per instruction, but the elements can be negated and/or swizzled. These can only be read by the pixel shader code or from the game application through an API interface. Access is through c[#]. For Direct3D, see SetPixelShaderConstantF() under *def* later in this chapter.
b0 ... b15	The standard 16 constant registers are each read-only quad Booleans. They are set either with the use of the *defb* instruction or by calling an external function from an application. These can only be read by the pixel shader code by a version [2.0$_x$, 3.0] or higher or from the game application through a DX9 API interface. Access is through b[#]. For Direct3D, see SetPixelShaderConstantB() under *defb* later in this chapter.
i0 ... i15	The standard 16 constant registers are each read-only quad integer vectors. They are set either with the use of the *defi* instruction or by calling an external function from an application. These can only be read by the pixel shader code by a version [2.0$_x$, 3.0] or higher or from the game application through an DX9 API interface. Access is through i[#]. For Direct3D, see SetPixelShaderConstantI() under *defi* later in this chapter.
aL	Loop count register, ps 3.0
p0	Predicate, ps 2.0$_x$, 3.0

v0 ... v1 *v2 ... v9*	The read-only color registers; *v0* is the diffuse, and *v1* is the specular. [1.1 … 1.4, 2.0]=v{0…1}., [3.0]=v{0…9}.
vFace	Face register, ps 3.0
vPos	Position register, ps 3.0
s0 ... s15	Read-only sampler registers, ps [2.0 … 3.0].
t0 ... t3 *t4 ... t5* *t6 ... t7*	Texture registers for pixel shaders [1.1 … 1.3]=t{0…3}, [1.4]=t{0…5}, [2.0 … 2.0_x]=t{0…7}, [3.0 … 3.0_{sw}]= Not supported.
r0 ... r1 *r2 ... r5* *r6 ... r11* *r12...r31*	The temporary registers {r0 … r31} are used as scratch read/write registers to temporarily save vertex data in various stages of processing; for version [1.0 … 1.3]=r{0…1}, [1.4]=r{0…5}, [2.0]= r{0…11}, [2.0_x, 3.0]={r0…r31}. For version 1.4 or earlier, *r0* is used for color output.
oC0 ... oC3	Write-only output color register, ps [2.0 … 3.0_{sw}].
oDepth	Write-only output depth register, ps [2.0 … 3.0_{sw}].

Only a single register type may be used per instruction except in the case of the temporary register (r#), where up to three can be used for the same instruction and in most cases can be used as source and destination.

Pixel Shader Instructions

As discussed in the chapters on vertex shader instructions, it should be noted that only pixel shader instructions for 1.0 through 1.4 are supported for the older version of Direct3D version 8.0 or 8.1. In Direct3D version 9.0 and beyond, instructions up to 2.0 or 3.0 are supported. Again, as version 9.0 is readily available, do not use previous versions of 8.1 or older, as it will have restrictions as to what it can do. That, and this book was written specifically for version 9.0 and thus samples may fail.

NOTE: For each new pixel version type, the number of instructions supported is increased! The number of instructions supported can be checked by examining the *MaxPShaderInstructionsExecuted* data member of D3DCAPS9.

Pixel shader version	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
Maximum number of instructions	8	8	8		96	512			

In the following table there is a difference between versions 1.0 through 1.3. This was originally supported by nVidia and Microsoft, and the ATI chipset as well as others cover that same range of versions. Textures, which are discussed in the next chapter, have a variety of special handling requirements depending on the version number.

There are similarities as well as multiple differences between the vertex shader and pixel shader instruction sets. The same techniques shown in the vertex shader part of this book can apply directly to this part of the book related to pixel shaders. A large subsection of the pixel instructions behave virtually identically to what was discussed in the vertex shader chapters. Instructions in these pixel shader chapters that also appeared in the vertex shader chapters are programmable in virtually the same manner. Any new differences are noted as they are presented. To prevent this from being the kind of book where the second half is virtually a repeat of the first half, the repetition of information has been minimized. In fact, for DirectX 9, numerous new instructions were added to make the pixel shader much more robust.

NOTE: For versions up to 1.3, only eight instructions slots were available! Other rules applied, such as for version 1.3: Texture registers t0, t1, t2 must be used in order (t0 before t1 or t2 but not t1 before t0, etc.).

Table 10-1: Programmable pixel instructions and their relationship with the version of Direct3D and versions.

Direct3D		8.0		8.1		9.0				
Instruction Version		1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
Assembly (Scripting) Commands										
dcl	Declaration					☻	☻			
dcl_usage	Declaration of usage								☻	
def	Floating-point definition	☻	☻	☻	☻	☻	☻	☻	☻	☻
defb	Boolean definition						☻	☻	☻	☻
defi	Integer definition								☻	☻
label	Address length						☻	☻	☻	☻
ps	Version (pix shader)	☻	☻	☻	☻	☻	☻	☻	☻	☻
Data Conversions										
frc	Fractional float comp.					☻	☻	☻	☻	☻
mov	Copy	☻	☻	☻	☻	☻	☻	☻	☻	☻
Add/Sub/Mul/Div										
add	Addition	☻	☻	☻	☻	☻	☻	☻	☻	☻
crs	*Cross product*					☻	☻	☻	☻	☻
dp2add	2D dot product (scaled)					☻	☻		☻	
dp3	Dot product (XYZ)	☻	☻	☻	☻	☻	☻	☻	☻	☻
dp4	Dot product (XYZW)		☻	☻	☻	☻	☻	☻	☻	☻
mad	Multiply-add	☻	☻	☻	☻	☻	☻	☻	☻	☻
mul	Multiply	☻	☻	☻	☻	☻	☻	☻	☻	☻
rcp	Reciprocal					☻	☻		☻	
rsq	Reciprocal square root					☻	☻		☻	
sub	Subtraction	☻	☻	☻	☻	☻	☻	☻	☻	☻
Special Functions										
exp	Exponential					☻	☻		☻	
log	$Log_2(x)$					☻	☻		☻	
lrp	Linear interpolation	☻	☻	☻	☻	☻	☻	☻	☻	☻
nop	No operation	☻	☻	☻	☻	☻	☻		☻	
pow	2^x					☻	☻		☻	
sincos	Sine and cosine					☻	☻		☻	

Flow Control (Branchless)

abs	Absolute					☻	☻	☻	☻	☻
cmp	Cmp. (# >= 0)		☻	☻	☻	☻	☻	☻	☻	☻
cnd	Cmp. (r0.a > 0.5) ? b:c Cmp. (a > 0.5) ? b:c	☻	☻	☻	☻ ☻	☻ ☻				
max	Maximum					☻	☻	☻	☻	☻
min	Minimum					☻	☻	☻	☻	☻
nrm	Normalize					☻	☻	☻	☻	☻
setp	Set predicate register						☻	☻	☻	☻

Flow Control (Branching)

break	Break out of loop						☻	☻	☻	☻
break_comp	Conditional loop						☻	☻	☻	☻
break_pred	Predicate break						☻	☻	☻	☻
call	Function call						☻	☻	☻	☻
callnz	Function call if ≠ 0						☻	☻	☻	☻
callnz_pred	Predicate call if ≠ 0						☻	☻	☻	☻
dsx	X rate of change						☻	☻	☻	☻
dsy	Y rate of change						☻	☻	☻	☻
else	If-**else**-endif						☻	☻	☻	☻
endif	If-else-**endif**						☻	☻	☻	☻
endloop	End of a loop								☻	☻
endrep	End of a repeat						☻	☻	☻	☻
if	If Boolean						☻	☻	☻	☻
if_comp	If comparison						☻	☻	☻	☻
if_pred	If predicate						☻	☻	☻	☻
loop	Start of loop								☻	☻
rep	Start of repeat						☻	☻	☻	☻
ret	Return from function						☻	☻	☻	☻

Matrices

m3x2	Apply m3x2 matrix					☻	☻	☻	☻	☻
m3x3	Apply m3x3 matrix					☻	☻	☻	☻	☻
m3x4	Apply m3x4 matrix					☻	☻	☻	☻	☻
m4x3	Apply m4x3 matrix					☻	☻	☻	☻	☻
m4x4	Apply m4x4 matrix					☻	☻	☻	☻	☻

Texture

tex	RGBA from tex.	☻	☻	☻						
texcoord	Intrp. UVW1	☻	☻	☻						

texdp3	Tex. DP		☻	☻						
texdp3tex	Tex. DP for 1D tex tbl		☻	☻						
texbem	Fake bump xform	☻	☻	☻						
texbeml	Bump xform w/lum	☻	☻	☻						
texcrd	Copy UVW1				☻					
texdepth	Calc. dept values				☻					
texkill	Cull pixel if UVW zero	☻	☻	☻	☻	☻	☻	☻	☻	☻
texreg2ar	Intrp. alpha and red	☻	☻	☻						
texreg2gb	Intrp. green and blue	☻	☻	☻						
texreg2rgb	Intrp. red, green, blue		☻	☻						

Texture Matrices

texm3x2depth	Calc. depth			☻						
texm3x2pad	1st row × of 2x2	☻	☻	☻						
texm3x2tex	Last row × of 3x2	☻	☻	☻						
texm3x3pad	1st or 2nd row 3x3	☻	☻	☻						
texm3x3	3x3 ×. w/3x3 pad		☻	☻						
texm3x3tex	3x3 ×. tex tbl	☻	☻	☻						
texm3x3spec	3x3 ×. spec reflt	☻	☻	☻						
texm3x3vspec	3x3 ×. vspec. reflt	☻	☻	☻						

Texture Referencing

bem	Bump environ. xform				☻	☻				
phase	Phase 1 to phase 2				☻	☻				
texld	Load RGBA				☻	☻	☻			
texldb	Load RGBA (bias)					☻	☻	☻	☻	☻
texldd	Load RGBA (user)						☻	☻	☻	☻
texldl	Load RGBA (lod)								☻	☻
texldp	Load RGBA (proj.)					☻	☻	☻	☻	☻

The ☻ indicates the pixel instruction is supported for that version.

Interesting table, don't you think? There are a lot of retired instructions replaced with instructions very similar to the vertex shader instructions — sort of the merging of the two technologies. Also to support this, the number of instruction slots was largely increased. Something to keep in mind, however, is that the more slots used, the slower the render. But you will soon discover that!

So let's start breaking these down. Those instructions described in the vertex shader section that are similar in functionality will

only be painted in broad strokes here and have a reference similar to the following:

> See the equivalent instructions in the vertex shader chapters for additional information as to usage.

Assembly (Scripting) Commands

Similar to vertex programming, the following are assembly language definitions and not instructions. They are, in essence, scripting commands to the assembler that do not generate code instructions but control the building of that code. They are also used to generate constant data (that is, read-only data!).

- ***ps***: Definition for the version of the code written for the pixel shader

ps.*MajVer.MinVer*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This is an assembly (scripting) definition and not an instruction for setting the version for which the code was written. It *must* be the first declaration in a code fragment. *MajVer* is the major version number, and *MinVer* is the minor version number of the vertex shader. Current range is {1.0, 1.1, 1.2, 1.3, 1.4, 2.0, 2_x, 2_sw, 3.0, 3_sw}. See *vs* (in Chapter 3) for additional information.

Watch out for version levels because some assemblers try to be helpful, such as version 1.42 of nVidia's nvasm.exe. If 1.4 is specified, it will print a warning about an unknown version and default to 1.1, which can easily break 1.3 code. Also, the same rules of dot versus underscore apply.

Listing 10-1: Pixel shader

```
ps.1.3        // Uses 1.3 pixel shader code
ps.1.4        // Uses 1.4 pixel shader code
ps.2.0
ps.3.0
ps.3.sw       // Version 3sw software (emulated) pixel shader
ps_3_sw
```

■ ***label***: A code address location

label l#	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This statement is used to mark a location in code for purposes of branching the program counter of the vertex processor of a particular pipe. A *label* may occur following a *ret* instruction to mark the beginning of a new block of code.

For version 2_x, the label number (#) must be in a range of {0…15} and for versions 2_{sw} and 3.0 {0…2047}. See *label* (in Chapter 3) for additional information.

Listing 10-2: Pixel shader

```
label l1
```

■ ***def***: Definition of a single-precision floating-point vector constant

def *Dst*, *aSrc*, *bSrc*, *cSrc*, *dSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This statement is used to define values within the constant registers before the pixel shader code is executed. This instruction must occur after the version instruction but before any arithmetic or texture instructions. This is not a programming instruction but a definition, and so it does not use up any of the instruction programming slots. The constant value read can only be read by the shader code. Values that can be defined with this definition range from {–1.0, …, 1.0}. See *def* (in Chapter 3) for additional information.

Listing 10-3: Pixel shader

```
ps   1.3                          // Version 1.3
def  c0, 1.0f, 0.0f, 1.0f, 0.0f   // Set c0 register {1,0,1,0}
```

An alternative to this is writing or reading the value directly from a C/C++ application by using the provided API for access from the application.

IDirect3DDevice9::SetPixelShaderConstantF()

```
HRESULT SetPixelShaderConstantF(
    UINT StartRegister,         // Register c#
    CONST float *pConstantData, // Pointer to array of float vectors
    UINT Vector4fCount          // # of four float vectors
);
```

If the function succeeds, a return value of D3D_OK will result. If there is an error, then D3DERR_INVALIDCALL results.

■ ***defb***: Definition of a Boolean constant

defb *Dst, aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This statement is used to set a single Boolean value {true : false} within the specified constant register (b#) used by the pixel shader code before it is executed. See *defb* (in Chapter 3) for additional information.

Note that the Boolean values are used only for conditional branching.

Listing 10-4: Pixel shader

```
ps.3.0                          // Version 3.0
defb  b3, true                  // Set b3 register {true : false}
defb  b1, false
```

An alternative to this is writing or reading the value directly from a C/C++ application by using the provided API for access from the application.

IDirect3DDevice9::SetPixelShaderConstantB()

```
HRESULT SetPixelShaderConstantB(
    UINT StartRegister,         // Register b#
    CONST BOOL *pConstantData, // Pointer to array of Booleans
    UINT BoolCount              // # of Boolean values in an array
);
```

If the function succeeds, a return value of D3D_OK will result. If there is an error, then D3DERR_INVALIDCALL results.

■ ***defi***: Definition of an integer vector constant

defi *Dst, aSrc, bSrc, cSrc, dSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
								☻	☻

This statement is used to define vector integer values within the constant registers (i#) by the code of the pixel shader code before it is executed. See *defi* (in Chapter 3) for additional information.

Listing 10-5: Pixel shader

```
ps.3.0                      // Version 2.0
defi  i3, 2, 3, 1, 0        // Set i3 register {2,3,1,0}
```

An alternative to this is writing or reading the value directly from a C/C++ application by using the provided API for access from the application.

IDirect3DDevice9::SetPixelShaderConstantI()

```
HRESULT SetPixelShaderConstantI(
    UINT StartRegister,        // Register i#
    CONST int *pConstantData,  // Pointer to array of integer vectors
    UINT Vector4iCount         // # of four integer vectors
);
```

If the function succeeds, a return value of D3D_OK will result. If there is an error, then D3DERR_INVALIDCALL results.

■ ***dcl***: Source input declaration

dcl *Dst*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻		

For version 2.0 and above, registers t(#) and v(#) have to declare an association between the individual components of the vertex shader outputs and the pixel shader inputs that will be accessed by the shader code.

Pseudocode:

```
dcl t(m)
dcl v(n)
```

So if, for example, the following is needed in the pixel shader code:

```
mov r0, t0.x      // replicate x
mov oC0, r0       // {xyza} = {xxxx}
```

…then the *x* component of *t0* needs to be declared!

```
dcl t0.x
```

Listing 10-6: Pixel shader

```
dcl t0.x
dcl t0.xy
dcl t1.xy
dcl v0.rgba
```

■ ***dcl_2d***: Source sampler declarations

dcl *Dst*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

For versions 2.0 and above, register s(#) has to declare a sampler input that will be accessed by the shader code.

Pseudocode:

```
dcl_2d s(m)
```

Listing 10-7: Pixel shader

```
dcl_2d s0
dcl_2d s1
dcl_2d s11

texld r0,r0,s0
mov oC0, r0
```

■ ***dcl_?(usage)?***: Source sampler declarations

dcl *Dst*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
								☻	☻

For versions 3.0 and above, register v(#) has to declare a vertex input usage based upon the D3DDECLUSAGE enumeration type. This specifies how the vertex input data is being used by the shader code. This is virtually the same statement declaration that you have been making in your vertex shader code since using the vertex assember from DX9. See *dcl_?usage?* (in Chapter 3) for additional information.

Pseudocode:

```
dcl_normal v(m)
dcl_blendweight v(m)
dcl_texcoord0 v(m)
dcl_texcoord1 v(m)
```

Listing 10-8: Pixel shader

```
dcl_normal      v0.xyz
dcl_blendweight v0.w
dcl_texcoord0   v1.zw
dcl_texcoord1   v1.y
```

Pixel Shader Instructions (Data Conversions)

■ ***mov***: Copy register data to register d = a

mov *Dst, aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction copies the referenced source register from *aSrc* to the destination register *Dst*. This instruction has the ability to write

to any of the destination registers. See *mov* (in Chapter 3) for additional information.

Listing 10-9: Pixel shader

```
mov r0,r1
mov_x4 r1.a, r1.a
```

Pixel Shader Assembly

Well, congratulations! You have now learned enough for your first pixel shader program.

For most of the pixel instructions in this chapter (but not for all pixel instructions), the following register usage table applies. Most registers can be used as source arguments, and the register r(#) is the only destination register.

NOTE: The shaded cells indicate something that's impossible. The empty cells indicate "possible but illegal."

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
☻	☻		☻	☻	2.0 aSrc
☻	☻			☻	3.0 aSrc

First, a vertex data structure needs to be declared. This was learned in the earlier vertex shaders chapters. This example first uses a simple custom vertex data structure that allows for each vector to have its own diffuse color:

```
struct CUSTOMVERTEX
{
   FLOAT x, y, z;          // Standard {XYZ} vector
   DWORD diffuseColor;     // The vector's diffuse color
}
```

Then, of course, an FVF macro is defined for that custom vertex.

```
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE)
```

Then the shader is passed a vertex data as a stream of two triangles using a triangle fan draw primitive, thus rendering a rectangle:

```
CUSTOMVERTEX vBoxAry[] =
{   // x      y     z     A R G B
  { -1.0f, -1.0f, 0.0f, 0xffff0000 },  // #0 red - LL
  {  1.0f, -1.0f, 0.0f, 0xff00ff00 },  // #1 green - LR
  {  1.0f,  1.0f, 0.0f, 0xff0000ff },  // #2 blue - UR
  { -1.0f,  1.0f, 0.0f, 0xffffffff },  // #3 white - UL
};
```

Indices for:

- Triangle #0: {0,1,2}
- Triangle #1: {1,2,3}

Then finally there is the pixel shader file *simple.psh*.

So let's peek at the file architecture for this graphics processor assembly language. A pixel shader script (PSH) exists as a *.psh file. As such, it would be ordered similar to the following:

Listing 10-10: Pixel shader version 1.1 – 1.4

```
// Note the version (PS) at the top of the file!
ps.1.1          // Version 1.1
mov r0,v0       // Output the diffused vertex color rgba
```

The vertex and color diffuse information from the source register *v0* components {rgba} is moved to the output register *r0*, and the render interpolates between the individual assigned diffuse colors associated with each corner of the polygon. All four components {rgba} are needed for output.

Did you note the version 1.1-1.4 part? The rules change from pixel shader to pixel shader. For example, the output for pixel shader version 2.0 is not register *r0* anymore; it is *oC0*, as the following example demonstrates.

Listing 10-11: Pixel shader version 2.0

```
// Note the version (PS) at the top of the file!
ps.2.0          // Version 2.0

dcl v0.rgba     // Declare – vertex and color diffuse data rgba
mov oC0,v0      // Output the diffused vertex color rgba
```

Did you also notice that *v0* was declared? For versions 2.0 and above, registers t(#) and v(#) have to be declared the source components that will be accessed using the *dcl* statement.

Listing 10-12: Pixel shader version 3.0

```
// Note the version (PS) at the top of the file!
ps.3.0                // Version 3.0

dcl_normal  v0.rgba   // Declare – vertex data rgb

mov oC0,v0            // Output the diffused vertex color rgba
```

■ ***frc***: Return fractional component of each source input

frc *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)					☻	☻	☻	☻	☻

This macro removes the integer component from the source *aSrc*, leaving the fractional component of the elements $\{0.0 \le x < 1.0\}$, which is stored in the destination *Dst*. See *frc* (in Chapter 3) for additional information.

Listing 10-13: Pixel shader

```
frc r0,v0
```

Instruction Modifiers

Now would probably be a good time to discuss instruction modifiers. These are filters applied to the result of the calculation before it is stored to the destination.

	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
_x2 = n << 1 = 2n	☻	☻	☻	☻					
_x4 = n << 2 = 4n	☻	☻	☻	☻					
_x8 = n << 3 = 8n				☻					
_d8 = n >> 3 = n ÷ 8				☻					
_d4 = n >> 2 = n ÷ 4				☻					
_d2 = n >> 1 = n ÷ 2	☻	☻	☻	☻					
_sat = Saturate (0.0 = n =1.0)	☻	☻	☻	☻					

These are applied to the individual instructions.

Listing 10-15: Pixel shader

```
add_sat r0.a, r0.a, c0.x  // (0.0 = n = 1.0)
add_x4  r1.a, t0, t1      // 4n
add_x4_sat  r3, r2, r1    // (0.0 = 4n = 1.0)
```

Co-Issued Instruction

	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
+	☻	☻	☻	☻					

A co-issued instruction allows two functions to be simultaneously executed for purposes of optimization. This involves the vector pipeline and the scalar pipeline being executed in parallel.

The following is an example of this:

```
  add r0.rgb, r0, v0
+ add r0.a,   r0, v0
```

However, this is exactly the same functionality as:

```
add r0,     r0, v0
```

An example instruction pairing that is more efficient would be something similar to:

```
  add r0.rgb, r0, v0
+ add r0.a,   r1, c0
```

Pixel Shader Instructions (Mathematics)

■ ***add***: Addition d = a + b

add *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction sums the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*. See *add* (in Chapter 3) for additional information.

Pseudocode:

```
da=aa+ba    db=ab+bb    dg=ag+bg    dr=ar+br
```

Listing 10-14: Pixel shader

```
add r0,r0,t0
```

■ ***sub***: Subtraction d = a – b

sub *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction subtracts the source *bSrc* from the source *aSrc* and stores the result in the destination *Dst*. See *sub* (in Chapter 3) for additional information.

Pseudocode:

```
da=aa–ba    db=ab–bb    dg=ag–bg    dr=ar–br
```

Listing 10-16: Pixel shader

```
sub r0,v0,t0

sub r1.rgb,t1,t0
+sub_x4 r1.a,t0,t1
```

■ ***mul***: Multiply d = ab

mul *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction results in the product of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*. See *mul* (in Chapter 3) for additional information.

Pseudocode:

$d_a=a_ab_a$ $d_b=a_bb_b$ $d_g=a_gb_g$ $d_r=a_rb_r$

Listing 10-17: Pixel shader

```
mul r0,v0,t0
+ mul_sat r0.a, t0_bx2, t1_bx2

mul_x2 r0, r0, v0
mul r0.a, r0.r, r0.r
```

■ ***mad***: Multiply add d = ab + c

mad *Dst, aSrc, bSrc, cSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction results in the product of the source *aSrc* and the source *bSrc,* sums the source *cSrc*, and stores the result in the destination *Dst*. See *mad* (in Chapter 3) for additional information.

Pseudocode:

$d_a=a_ab_a+c_a$ $d_b=a_bb_b+c_b$ $d_g=a_gb_g+c_g$ $d_r=a_rb_r+c_r$

Listing 10-18: Pixel shader

```
mad r0,v0,t0,v0
mad_d4 r0.rgb, r1, r0.r, r2
```

■ ***crs***: Cross product (outer product) d = a × b

crs *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)					☻	☻	☻	☻	☻

This two-slot macro instruction results in the cross product (outer product) of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*. The destination cannot be the same as the source register. The destination is only allowed to be one of the following: {.x, .y, .z, .xy, .xz, .yz, .xyz, .xyza, or the rgba equivalents}. See *crs* (in Chapter 3) for additional information.

Listing 10-19: Pixel shader

```
crs r1.x, r0, c0
```

■ ***dp3***: Dot product d = a • b

dp3 *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction results in the dot product of the source *aSrc.xyz* and the source *bSrc.xyz* and stores the replicated scalar result in each element of the destination *Dst.xyzw.* See *dp3* (in Chapter 3) for additional information.

Pseudocode:

$d_a=d_b=d_g=d_r= \quad a_rb_r + a_gb_g + a_bb_b$

Listing 10-20: Pixel shader

```
dp3  r0, v0, v0
dp3  r0.rgb, t0, v0
dp3_ sat r0, t0_bx2, v0_bx2
dp3  t0.rgba, r1, c4
```

■ ***dp4***: Dot product d = a • b

dp4 *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
		☻	☻	☻	☻	☻	☻	☻	☻

This instruction results in the dot product of the source *aSrc.rgba* and the source *bSrc.rgba* and stores the replicated scalar result in each element of the destination *Dst.rgba*. See *dp4* (in Chapter 3) for additional information.

Pseudocode:

$d_a=d_b=d_g=d_r= \quad a_rb_r + a_gb_g + a_bb_b + a_ab_a$

Listing 10-21: Pixel shader

```
dp4 r0,t0,v0
dp4 r5.y,v0,c3
```

■ ***dp2add***: 2D dot product d = a • b + scalar

dp2add *Dst, aSrc, bSrc, cSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction occupies two slots and results in the 2D dot product of the source *aSrc.xy* and the source *bSrc.xy,* sums the specified swizzled scalar from *cSrc,* and stores the replicated scalar result in each element specified by the mask of the destination *Dst* in conjunction with an optional saturation specifier. Any of the source arguments may be optionally negated.

The allowed masks for *aSrc* and *bSrc* are: {.rgba, .r, .g, .b, .a, .gbra, .brga, .abgr}.

Pseudocode:

```
d_m=   a_r b_r + a_g b_g + c_?   m={replicated element}.  ?={r,g,b,a}
```

Listing 10-22: Pixel shader

```
dp2add r1, r0.r, c0.b, r3.x
```

■ ***rcp***: Reciprocal of the source scalar d = 1/a

rcp *Dst, aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction results in the reciprocal of the source *aSrc* and stores the replicated scalar result in each specified element of the destination. Special case handling is utilized if a source is equal to 1.0 or 0.0. The default is *Dst.xyzw*, *Src.x*. See *rcp* (in Chapter 3) for additional information.

Listing 10-23: Pixel shader

```
rcp r0, c1.x
```

■ ***rsq***: Reciprocal square root of the source scalar $d = 1/\sqrt{abs(a)}$

rsq *Dst, aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction results in the reciprocal square root of the source *aSrc* specified by only one element {.x .y .z .w} and stores the replicated scalar result in each element of the destination. Special case handling is utilized if the source is equal to 1.0 or 0.0. The default is *Dst.xyzw*, *aSrc.x*. See *rsq* (in Chapter 3) for additional information.

Listing 10-24: Pixel shader

```
rsq r0, v0.z
```

Special Functions

We are now nearing the end, with the few remaining non-branch and matrix instructions that have special functionality.

■ ***nop***: No operation

nop	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction performs no operation. See *nop* (in Chapter 3) for additional information.

Listing 10-25: Pixel shader

```
nop
```

■ ***pow***: Power $d = |a|^b$

pow *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)					☻	☻	☻	☻	☻

This instruction occupies three slots. It calculates the scalar value using the source *aSrc* as a base value and the source *bSrc* as an exponent value and stores the replicated result in each component of the destination *Dst*. This is a scalar instruction, thus both the *aSrc* and *bSrc* source arguments require the swizzle of a single replicated component {x, y, z, w}. See *pow* (in Chapter 3) for additional information.

The *Dst* register should be a temporary register (r#) and not the same register as *bSrc*.

Pseudocode:

$d_x = d_y = d_z = d_w = |a|^b$

Listing 10-26: Pixel shader

```
pow r3, r4.x, r4.y
pow r2, r0.z, c0.w
```

■ ***exp***: Exponential 2^x — 19-bit precision

exp *Dst, aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction calculates the exponential number using the single element of the source *aSrc* and stores the result in the destination *Dst*. See *exp* (in Chapter 3) for additional information.

Pseudocode:

$d_x = d_y = d_z = d_w = pow(2, a_w)$

Listing 10-27: Pixel shader

```
exp r0, r1.y
```

■ ***lrp***: Linearly interpolate by proportion

lrp *Dst, aSrc, bSrc, cSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction occupies two slots. It results in the linear interpolation of the product between source *aSrc,* source *bSrc, and* source *cSrc* and stores the result in the destination *Dst*. See *lrp* (in Chapter 3) for additional information.

Pseudocode:

```
d = a * b + (1 – a) * c
// which is the same as
d = c + a * (b – c)
```

Listing 10-28: Pixel shader

```
lrp r2.rgb, r3.a, r4, c0
lrp r3, r2.y, c2, v0
lrp_sat r0, r1.a, r1, r5
```

■ ***sincos***: Sine – cosine calculation

sincos (Macro)	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
sincos *Dst, aSrc, bSrc, cSrc*					☻	☻	☻		
sincos *Dst, aSrc*								☻	☻

This (eight-slot) macro instruction calculates both the sine and cosine of the source arguments in radians. See *sincos* (in Chapter 3) for additional information.

Listing 10-29: Pixel shader

```
sincos r1.xy, r0.x, c1, c2             // version 2.0, 2x, 2sw
sincos r1.xy, r0.x, c1.xyzw, c2.xyzw   // version 2.0, 2x, 2sw
sincos r1.xy, c0.z, c1.xyzw, c2.xyzw   // version 2.0, 2x, 2sw

sincos r1.xy, r0.x                     // version 3.0, 3sw
sincos r0.xy, v0.z                     // version 3.0, 3sw
```

Branchless Code

■ ***abs***: Absolute d = |a|

abs *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction results in the positive conversion of a negative value in the source *aSrc* and stores the result in each specified element of the destination. Please note that this instruction is only valid if your version is set to 2.0 or above. See *abs* (in Chapter 4) for additional information.

Pseudocode:

```
dx=|ax|   dy=|ay|   dz=|az|   dw=|aw|
```

Listing 10-30: Pixel shader

```
abs r0, r1
```

■ ***min***: Minimum d = (a < b) ? a : b

min *Dst*, *aSrc*, *bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction results in the selection of the lower value from each element of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*. See *min* (in Chapter 4) for additional information.

Pseudocode:

```
dx= (ax < bx)  ? ax : bx
dy= (ay < by)  ? ay : by
dz= (az < bz)  ? az : bz
dw= (aw < bw)  ? aw : bw
```

Listing 10-31: Pixel shader

```
min r0, c0, r1
min r0, r1, c0
```

■ ***max***: Maximum d = (a > b) ? a : b

max *Dst*, *aSrc*, *bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction results in the selection of the higher value from each element of the source *aSrc* and the source *bSrc* and stores the result in the destination *Dst*. See *max* (in Chapter 4) for additional information.

Pseudocode:

$d_x = (a_x > b_x) \ ? \ a_x : b_x$
$d_y = (a_y > b_y) \ ? \ a_y : b_y$
$d_z = (a_z > b_z) \ ? \ a_z : b_z$
$d_w = (a_w > b_w) \ ? \ a_w : b_w$

Listing 10-32: Pixel shader

```
min r1, c3, r2
min r0, r2, v0
```

■ ***nrm***: 3D vector normalization

nrm *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction occupies three slots and calculates the normalization of a 3D vector. See *nrm* (in Chapter 4) for additional information.

Pseudocode:

$r = 1/\sqrt{(a_x a_x + a_y a_y + a_z a_z)}$
$d_x = ra_x \quad d_y = ra_y \quad d_z = ra_z$

Listing 10-33: Pixel shader

```
nrm r0, c0
```

■ *dsx*: Calculate the rate of change in the x direction

dsx *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This instruction calculates the rate of change in the x direction from the source *aSrc* and stores the result in the destination *Dst*.

Listing 10-34: Pixel shader

```
dsx r0, c0
```

■ *dsy*: Calculate the rate of change in the y direction

dsy *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This instruction calculates the rate of change in the y direction from the source *aSrc* and stores the result in the destination *Dst*.

Listing 10-35: Pixel shader

```
dsy  r1, c3
```

■ *setp*: Set predicate

setp_?? *aSrc*, *bSrc*		1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
setp_gt	(a > b)						☻	☻	☻	☻
setp_ge	(a ≥ b)						☻	☻	☻	☻
setp_eq	(a = b)						☻	☻	☻	☻
setp_ne	(a <>b) (a ≠ b)						☻	☻	☻	☻
setp_le	(a ≤ b)						☻	☻	☻	☻
setp_lt	(a < b)						☻	☻	☻	☻

This instruction results in a per-channel comparison between the source register *aSrc* and the source register *bSrc* and stores the result in the destination *Dst*. See *setp* (in Chapter 4) for additional information.

Listing 10-36: Pixel shader

```
setp_gt p0, r1,c1
(!p0) add r0, r3,r4
```

■ ***cmp***: Compare less than d = (a < 0) ? c : b

cmp *Dst, aSrc, bSrc, cSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻	☻	☻	☻	☻	☻

This instruction results in a comparison of each element of the source *aSrc* to zero. If less than zero (negative), the source *cSrc* is selected, or if greater than or equal to zero, the source *bSrc* is selected. The result is returned in *Dst*.

Pseudocode:

$d_x = (a_x < 0) \quad ? \; c_x : b_x)$
$d_y = (a_y < 0) \quad ? \; c_y : b_y)$
$d_z = (a_z < 0) \quad ? \; c_z : b_z)$
$d_w = (a_w < 0) \quad ? \; c_w : b_w)$

Listing 10-37: Pixel shader

```
cmp r0, c0, c1, c2
```

■ ***cnd***: Compare greater than 0.5 d = (a > 0.5) ? b : c

cnd *Dst, aSrc, bSrc, cSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻					

This instruction results in a comparison of each element of the source *aSrc* to zero. If less than zero (negative), the source *cSrc* is selected, and if greater than or equal to zero, the source *bSrc* is selected. The result is returned in *Dst*.

Pseudocode:

```
dx= (ax < 0)   ? cx : bx)
dy= (ay < 0)   ? cy : by)
dz= (az < 0)   ? cz : bz)
dw= (aw < 0)   ? cw : bw)
```

Listing 10-38: Pixel shader

```
cnd r0, r0, c1, c2
```

Branching Code

These were introduced with vertex version 2.0 and have exactly the same functionality for the pixel shaders. The source argument *aSrc* is a Boolean register (b#), and if set, the code following the *if* instruction is executed. If not, the optional *else* code is executed. In either case, the shader code resumes execution with the first instruction after the *endif*. See *if*, *else*, and *endif* (in Chapter 4) for additional information.

The if-else-endif conditional:

- if-else-endif (Boolean)

if aSrc	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

The *if* conditional occupies three instruction slots.

else	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻
endif	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

The *if* compare conditional occupies three instruction slots, and the *if* predicate conditional occupies three instruction slots.

Listing 10-39: Pixel shader

```
defb b1, TRUE

if b1
mul r0.xyz, v0, c2.x   // Executed if b1 is true!
mad r2.xyz, v1, c2.y, r0
else
mul r0.xyz, v0, c3.x   // Executed if b1 is false!
mad r2.xyz, v1, c3.y, r0
endif
```

■ ***rep***-endrep: Repeat

rep *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

The *rep* instruction uses three instruction slots and is the beginning marker of a block of repeating code. The source integer register *aSrc* only uses the constant integers (i#), which contain the non-swizzled number of iterations (loops) that occur between the *rep* and *endrep* instructions. The maximum number of loops allowed is 255. See *rep* and *endrep* (in Chapter 4) for additional information.

■ rep-***endrep***: End repeat

endrep	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

An *endrep* instruction occupies two instruction slots, must be used in conjunction with the *rep* instruction, and occurs at the end of the looped code block.

The repeat loops are not allowed to be nested. When used in conjunction with *if* statements, the repeat loop must either be a container for the *if* code block or reside within the *if* block. The *rep* and *endrep* both occupy one instruction slot each. This can be thought of as a while loop.

Listing 10-40: Pixel shader

```
defi i3, 5, 5, 5, 5

rep i3
        // Insert your code here!
endrep

rep i3.xyzw
        // Insert your code here!
endrep
```

■ ***loop***-endloop

loop *aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
								☻	☻

This is the beginning instruction of a looped section of code and occupies three instruction slots. The source register *aSrc* is the loop counter *aL* register. The source register *bSrc* is an integer register i#, where the {*x*} component contains the iteration count, the {*y*} component contains the initial value of the loop counter, and the {*z*} component contains the incremental value. See *loop* and *endloop* (in Chapter 4) for additional information.

■ loop-***endloop***

endloop	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
								☻	☻

The *endloop* statement indicates the bottom of a loop and occupies two instruction slots. The pixel code between *loop* and *endloop* will cycle up to the value of the loop counter.

Listing 10-41: Pixel shader

```
defi i3, 5, 2, 1, 0   // 5 loops, i2...i6, +1

loop aL, i3
          // Insert your code here using aL index!
endloop
```

■ ***break***: Break out of loop

break	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This instruction is used to break out of repeats and loops and allow the function to execute the instruction just below the *endloop* or *endrep* looping block that it resides in. It has an identical functionality to that of the break used in while loops in C. See *break* (in Chapter 4) for additional information.

Listing 10-42: Pixel shader

```
break
// Warning: This usage may not be correct for your
// version of psa.exe.
```

■ ***break_??*** (compare break)

break_?? *aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
break_gt if (a > b) break						☻	☻	☻	☻
break_ge if (a ≥ b) break						☻	☻	☻	☻
break_eq if (a = b) break						☻	☻	☻	☻
break_ne if (a <>b) break (a ≠ b)						☻	☻	☻	☻
break_le if (a ≤ b) break						☻	☻	☻	☻
break_lt if (a < b) break						☻	☻	☻	☻

This is a combination of an *if* conditional and a *break* contained within a single instruction that occupies three instruction slots. Just like the *if* conditional, an element of both the source *aSrc* and *bSrc* needs to be selected for the individual scalar compare. See *break_??* (in Chapter 4) for additional information.

Listing 10-43: Pixel shader

```
rep i1
break_gt c3.x, r0.x
endrep
```

■ ***break*** (predicate)

break_pred *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This predicate conditional break occupies three instruction slots and uses the predicate register *p0* as the conditional to break out of a loop. The ! symbol indicates a NOT condition and therefore 1's complements the value in the predicate register. See *break_pred* (in Chapter 4) for additional information.

Listing 10-44: Pixel shader

```
break   p0.x
break  !p0.y

// Warning: This usage may not be correct for your
// version of psa.exe.
```

■ ***call***-ret: Call function and then return

call *label*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This instruction occupies two slots and is a function call to a code block whose entry point is marked by a label and exit point is marked by a *ret* instruction. Labels were discussed in Chapter 3. Similar to how general-purpose processors function, the effective address of the next executable instruction is pushed on a stack and execution is branched (jumped) to the address marked by a label. Execution continues from that point forward until a return is encountered. See *call* and *ret* (in Chapter 4) for additional information.

An indicator is needed to mark the end of the called function code, which is the *ret* instruction.

■ call-***ret***: Return from called function

ret	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This is the exit point of a code block accessed by a *call* instruction. The address pushed onto the stack by the *call* instruction is popped off the stack, and then the execution is branched (jumped) to.

Listing 10-45: Pixel shader

```
call l1
```

■ ***callnz***-ret

callnz *label, aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This is similar to a call instruction, except it is a conditional call and occupies three instruction slots. That is, if the Boolean constant referenced by the source input *aSrc* is *not* zero, thus True, then the function is called. See *callnz* (in Chapter 4) for additional information.

Listing 10-46: Pixel shader

```
callnz l1, b2
```

■ ***callnz*** (predicate)

This instruction calls if the predicate is not zero; it occupies three instruction slots.

Listing 10-47: Pixel shader

```
callnz l1, p0.r
callnz l2, !p0.g
```

Matrices

These pixel shader matrix instructions are nearly identical to the matrix instructions explained in Chapter 5, "Matrix Math."

■ ***m4x4***: Apply a 4x4 matrix to a four-component vector d = aB

m4x4 *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)					☻	☻	☻	☻	☻

This macro instruction occupies four instruction slots. It applies a 4x4 matrix referenced by the four sequential registers beginning with the source *bSrc* {+0, …, +3} to the {XYZW} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*. See *m4x4* (in Chapter 5) for additional information.

Negation and swizzle are only allowed for the source vector *aSrc* and not the four consecutive *bSrc* registers. The destination must have the four components specified.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
☻	☻		☻	☻	2.0 aSrc
☻	☻		☻		2.0 bSrc
☻	☻			☻	3.0 aSrc, bSrc

Listing 10-48: Pixel shader

```
m4x4  r1.xyzw, r3, c4    ; Multiply r3 *  C4...C7
```

■ ***m4x3***: Apply a 4x3 matrix to vector d = aB

m4x3 *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)					☻	☻	☻	☻	☻

This macro instruction occupies three instruction slots. It applies a 4x3 matrix referenced by the three sequential registers beginning with the source *bSrc* {+0, +1, +2} to the {XYZW} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*. See *m4x3* (in Chapter 5) for additional information.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
☻	☻		☻	☻	2.0 aSrc
☻	☻		☻		2.0 bSrc
☻	☻			☻	3.0 aSrc, bSrc

Listing 10-49: Pixel shader

```
m4x3  r1.xyz, r3, c4    ; Multiply r3 *  C4...C6
```

■ ***m3x2***: Apply a 3x2 matrix to vector d = aB

m3x2 *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)					☻	☻	☻	☻	☻

This macro instruction occupies two instruction slots. It applies a 3x2 matrix of the two sequential registers beginning with the source *bSrc* {+0, +1} to the {XYZ} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*. See *m3x2* (in Chapter 5) for additional information.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
☻	☻		☻	☻	2.0 aSrc
☻	☻		☻		2.0 bSrc
☻	☻			☻	3.0 aSrc, bSrc

Listing 10-50: Pixel shader

```
m3x2  r1.xy, r3, c4    ; Multiply r3 *  C4...C5
```

■ ***m3x3***: Apply 3x3 matrix to vector d = aB

m3x3 *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)					☻	☻	☻	☻	☻

This macro instruction occupies three instruction slots. It applies a 3x3 matrix referenced by the three sequential registers beginning with the source *bSrc* {+0, +1, +2} to the {XYZ} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*. See *m3x3* (in Chapter 5) for additional information.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
☻	☻		☻	☻	2.0 aSrc
☻	☻		☻		2.0 bSrc
☻	☻			☻	3.0 aSrc, bSrc

Listing 10-51: Pixel shader

```
m3x3  r1.xyz, r3, c4    ; Multiply r3 *  C4...C6
```

■ ***m3x4***: Apply 3x4 matrix to vector d = aB

m3x4 *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
(Macro)					☻	☻	☻	☻	☻

This macro instruction occupies two instruction slots. It applies a 3x4 matrix referenced by the four sequential registers beginning with the source *bSrc* {+0, …, +3} to the {XYZ} vector referenced by the source *aSrc* and stores the result in the destination vector *Dst*. See *m3x4* (in Chapter 5) for additional information.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
☻	☻		☻	☻	2.0 aSrc
☻	☻		☻		2.0 bSrc
☻	☻			☻	3.0 aSrc, bSrc

Listing 10-52: Pixel shader

```
m3x4  r1.xy, r3, c4    ; Multiply r3 *  C4...C5
```

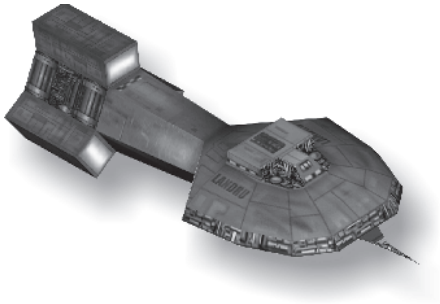
There is no software emulation for pixel shaders under Direct3D (*well, almost!*).

This, to me, is sort of a joke and brings back a fond memory. I was working on a 3D rendered children's game project that was supposed to be ported to the PS2, Nintendo 64, and PC using Direct3D. Since children tend to inherit "hand-me-down" computers from their parents, they typically do not have hardware render cards, and software emulation under Direct 3D is extremely slow. This execution speed was so fast, that it ran at one frame per second on a high-performance machine. This tremendous lack of

speed required me to write my own software render that actually jumped the frame rate almost 1000 percent before I started to convert the C code to assembly.

The point here is that you can do software emulation by using the REF (reference device) on a 3D graphics card, but unfortunately, like the D3D software emulation mode, your shaders become more like a slide show!

So now let's move on to one of the most important portions of pixel shaders — the texture!



Chapter 11

Textures

This chapter is the second half of the pixel shader instructions but is specifically related to textures and their handling. In the previous chapter we learned the instructions needed for manipulation of pixel shader information, but in this chapter we discuss how it relates to the texture itself.

Texture Registers

Recall from the last chapter that there are registers specifically used for accessing texture data.

t0 ... t3	Texture registers for pixel shaders
t4 ... t5	[1.1 … 1.3] = *t*{0…3}
t6 ... t7	[1.4] = t{0…5}
	[2.0 … 2.0_x] = *t*{0…7}
	[3.0 … 3.0_x]= Not supported

The fact that those texture registers are to be associated with texture data in a sequential (index) order was not mentioned. That is, texture coordinates are moved into *t0*, and then *t1* (if needed) is calculated based upon *t0* (then *t2* using *t0…t1*, and so forth).

Based upon those registers, the maximum number of texture samples that can be supported by that version of pixel shader hardware is represented in the following table.

Pixel shader version	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
Maximum texture samples	4	4	4	6	16	16	16	16	16

That ratio can also be calculated for fixed function pipelines using the data members *MaxTextureBlendStages* and *MaxSimultaneous-Textures* from the D3DCAPS9 data structure.

Samples = MaxTextureBlendStages/MaxSimultaneousTextures

The texture instructions can be the most confusing of all the shader instructions due to the fact that certain instructions are only supported by limited versions of the pixel shaders. In essence, some instructions only work with some versions. Typically, when designing your 3D graphical application for the consumer market, you need to keep in mind that you will definitely need to write different versions to effectively generate similar effects, depending on which pixel shader versions that a particular card supports. This is especially true since in conjunction with hardware shaders, only the newer cards support branching, and all shader cards support different texture methodologies.

Before getting into textures, we should highlight at least one use of pixel shaders without textures. The following shader is sometimes used to draw character shadows into a shadow texture. Prior to this shader being called, the shadow texture is set up with a color surface and a depth surface.

Listing 11-1: Character shadow pixel shader from Paul Stapley

```
//  v0 = distance from light [0.0, 1.0]     range: 0.0 = v0 = 1.0
//     = when 1.0f = CV_CASTING_SHADOW_LIGHT.w distance

ps.1.1

def c0, 0.0f, 0.0f, 0.0f, 1.0f

mov r0, v0           ; Vertex color
mov r0.a, c0.a       ; Set to 1.0
```

One should also keep in mind that certain texture instructions only work with certain versions and care should be taken when developing shader code. There is also a special case of the *texld* instruction, where the number of arguments changes depending on the version of the shader.

■ ***tex***: Load destination register with sampled RGBA

tex *Dst*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻					

This instruction loads the color data (RGBA) sampled from the texture and stores the result in the destination *Dst.rgba*. Note that the indexed texture register t(#) is actually the texture stage number.

o_n	r_n	t_n	Dst
		☻	

Listing 11-2: Pixel shader

```
tex  t0
tex  t1
tex  t2
tex  t3
```

The following is a simple object pixel shader where the color data from the corresponding pixel position in the texture is rendered.

Listing 11-3: Object pixel shader with lighting from Paul Stapley

```
ps.1.1

tex t0                ; Fetch texture base
mov r0, v0
```

In the following case, the texture color is blended with the vertex color.

Listing 11-4: Object pixel shader with lighting from Paul Stapley

```
ps.1.1

tex t0                ; Fetch texture base
mul r0, t0, v0        ; Texture color × vertex color
```

This effect is similar to the following image:

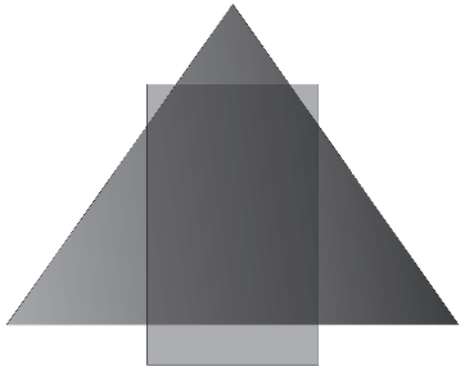


Figure 11-1: Result of a color product mixing a texture color with a vertex color

■ ***texld***: Loads RGBA using texture coords

texld	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
texld *Dst, aSrc*				☻					
texld *Dst, aSrc, bSrc*				☻	☻	☻	☻	☻	☻

This instruction samples textures from source *t(#)* or *r(#)* depending on phase to destination *r(#)*. When source is r(#), then the elements {XYZ} must have been assigned in phase1.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
			☻		1.4 phase 1, 2.0 aSrc
	☻		☻		1.4 phase 2, 2.0 aSrc
		☻			2.0, 3.0 bSrc
☻	☻			☻	3.0 aSrc

Pseudocode:

```
texld r(d), t(a).xyz     // Version 1.4
texld r(d), t(a)
texld r0, t0

texld r(d), t(a), s(b)   // Version 2.0
```

If a cube-mapped texture is utilized, the instruction occupies 1+3 cube slots.

Listing 11-5: Pixel shader

```
texld r0, t0.xyz       // Version 1.4
texld r0, t0,    s0    // Version 2.0, 3.0
```

Listing 11-6: ATI ShadowMap sample — showmap.psh

```
// Note that since this is version 2.0, not 1.4, the phase
// instruction is not needed!

ps.2.0

dcl t0.xy
dcl_2d s0

texld r0, t0, s0      // Get sample shadow map

mov r0, r0.x
mov oC0, r0           // Output shadow map
```

■ ***texldb***: Loads mipmap level of detail with bias

texldb *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction occupies six slots. It loads a projected texture sample using the signed fourth element (.a or .w) as a bias for the level of detail.

Positive bias source values result in smaller mipmaps being selected, and negative values result in larger mipmaps being selected.

Values outside the ranges ps 2.0: {–3.0 … +3.0}, ps 3.0: {–16.0 … 15.0} have an undetermined result. The source *aSrc* is unaffected by the result.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
	☻		☻		2.0 aSrc
☻	☻			☻	3.0 aSrc
		☻			bSrc

Pseudocode:

```
texldb r(d), r(a), s(b)
texldb r(d), t(a), s(b)
```

Listing 11-7: Pixel shader

```
texldb r1, r0, s0
```

■ ***texldd***: Loads texture using texture coords, *dsx*, and *dsy*

texldd *Dst, aS, bS, cS, dS*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
						☻	☻	☻	☻

This three-slot instruction loads a texture in which the texture coordinates are supplied in source *aSrc*, the sampler register (s#) in source *bSrc*, the x-gradient (*dsx*) from *cSrc*, and the y-gradient (*dsy*) from *dSrc* and stores the result in *Dst*.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
	☻		☻		2_x 2_{sw} aSrc
☻	☻		☻	☻	2_x 2_{sw} cSrc, dSrc
		☻			bSrc
☻	☻			☻	3.0 aSrc, cSrc, dSrc

Listing 11-8: Pixel shader

```
texldd r1, r0, s0, r0, r0
texldd r2, r0, s0, c2, v0
```

■ ***texldl***: Loads mipmap level of detail

texldl *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
								☻	☻

This instruction loads a texture in which the texture coordinates are supplied in source *aSrc* and the sampler register (s#) in source *bSrc.w* and stores the result in *Dst*. If a cube-mapped texture is utilized, the instruction occupies 2+3 cube slots.

The texture coordinates may not be scaled.

Dst must be a temporary register (r#) and can use a swizzled element.

If the source texture is unsigned, then the result will be {0.0 … 1.0}. If signed, then {–1.0 … 1.0} will result.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
☻	☻			☻	aSrc
		☻			bSrc

Listing 11-9: Pixel shader

```
texldl r2, c0, s0
```

■ ***texldp***: Loads mipmap level of detail

texldp *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
					☻	☻	☻	☻	☻

This instruction loads a texture in which the texture coordinates are supplied in source *aSrc*, divides by the source *bSrc.w* (.a or .w), and stores the result in *Dst*. If a cube-mapped texture is utilized, the instruction occupies 3+1 cube slots.

o_n	r_n	t_n	Dst
	☻		

c_n	r_n	s_n	t_n	v_n	Src
	☻		☻		2.0 aSrc
		☻			bSrc
☻	☻			☻	3.0 aSrc

Listing 11-10: Pixel shader

```
texldp r0, r0, s0
```

■ ***texcoord***: Interpret coordinate (UVW1) as color data (RGBA)

texcoord *Dst*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction interprets the texture coordinate (UVW1) referenced by the destination register into color data (RGBA) and

stores the result in the destination *Dst.rgb.* A 1.0 is stored in the alpha element.

o_n	r_n	t_n	Dst
		☻	

Pseudocode:

```
texcoord t(d)
```

Listing 11-11: Pixel shader

```
texcoord t0
```

■ ***texdp3***: Texture dot product d = a • b

texdp3 *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
		☻	☻						

This instruction results in the dot product of the source *aSrc.xyz* and the source *bSrc.xyz* and stores the replicated scalar result in each element of the destination *Dst.xyz*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

It cannot be used after a non-texture-based instruction. Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
texdp3 t(d), t(a)      when d > a
```

Listing 11-12: Pixel shader

```
tex    t0
texdp3 t1, t0
texdp3 t2, t0
```

■ ***texdp3tex***: Texture dot product for 1D tex table d = a • b

texdp3tex *Dst*, *aSrc*, *bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
		☻	☻						

This instruction results in the dot product of the source *aSrc.xyz* and the source *bSrc.xyz* and stores the replicated scalar result in each element of the destination *Dst.xyzw*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

It cannot be used after a non-texture-based instruction. Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
texdp3tex t(d), t(a)     when d > a
```

Listing 11-13: Pixel shader

```
tex          t0
texdp3tex t1, t0
texdp3tex t2, t0
```

■ ***texbem***: Apply a fake bump environment-map transform

texbem *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction transforms the red and green components of the source *aSrc* using the 2D bump environmental-mapping matrix, summing the result to the destination *Dst*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

It cannot be used after a non-texture-based instruction. Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
texbem t(d), t(a)     when d > a
```

Listing 11-14: Pixel shader

```
tex        t0
texbem t1, t0
texbem t2, t0
```

■ ***texbeml***: Apply a fake bump map xform with luminance correction

texbeml *Dst, aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction transforms the red and green components of the source *aSrc* using the 2D bump environmental-mapping matrix, sums the result to the destination *Dst,* applies a luminance correction, and stores the result in the destination *Dst*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
texbeml t(d), t(a)     when d > a
```

Listing 11-15: Pixel shader

```
tex         t0
texbeml t1, t0
texbeml t2, t0
```

■ ***texcrd***: Copy coordinate data from source as RGBA

texcrd *Dst, aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
				☻					

This instruction copies the source coordinate data from the source *aSrc*, copies the referenced RGBA color data, and stores the result in the destination *Dst*.

o_n	r_n	t_n	Dst
	☻		
	☻		

c_n	r_n	s_n	t_n	v_n	Src
			☻		phase 1
			☻		phase 2

Pseudocode:

```
texcrd r(d).rgb, t(a).xyz
texcrd r(d).rgb, t(a)
texcrd r(d).rg, t(a)_dw.xyw
```

Listing 11-16: Pixel shader

```
texcrd r2.rgb, t0.xyz
texcrd r3.rgb, t3
texcrd r5.rg,  t5_dw.xyw
```

■ ***texdepth***: Calculate depth for pixel buffer comparison

texdepth *Dst*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
				☻					

This instruction occupies two slots and is used in the depth buffer comparison test of the texture specified by *Dst*.

o_n	r_n	t_n	Dst
	☻		phase 2 only

Pseudocode:

```
texdepth r(d)
```

Listing 11-17: Pixel shader

```
texdepth r5   // Output new depth
```

■ ***texkill***: Kill rendered pixel

texkill *Src*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻	☻					

This instruction occupies two slots and cancels the render if any of the first three elements {UVW} of the source *Src* of the texture coordinates are negative.

c_n	r_n	s_n	t_n	v_n	Src
			☻		1.1, 1.2, 1.3
	☻		☻		1.4 phase 2 only

Results of this instruction must *not* be read!

Pseudocode:

```
If (Src.x < 0.0 | Src.y < 0.0 | Src.z < 0.0)   ; ((u < 0) ∨ (v < 0)
                                                ; ∨ (w < 0))
   Halt pixel render.
         (Cull pixel by stopping this shader execution.)

texkill t(a)       // 1.3 or 1.4
texkill r(a)       // 1.4 only
```

Listing 11-18: Pixel shader

```
texkill t0

texkill r0
```

Listing 11-19: Pixel shader — diffuse

```
ps.1.1
texkill t0   // cull pixel if t0{xyzw} < 0

mov r0, v0   // Diffuse output
             // r0.w is used as alpha-blending factor
```

■ ***texreg2ar***: Interpret the alpha and red components with {UV}

texreg2ar *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction interprets the red and alpha elements of the source register *aSrc* as (u,v) texture address data and stores the result in *Dst*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex              t(a)        // Assign the stage with the texture
texreg2ar  t(d), t(a)        when d > a
```

Listing 11-20: Pixel shader

```
tex           t0
texreg2ar t1, t0
```

■ ***texreg2gb***: Interpret the green and blue components with {UV}

texreg2gb *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
		☻	☻						

This instruction interprets the green and blue elements of the source register *aSrc* as (u,v) texture address data and stores the result in *Dst*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex              t(a)        // Assign the stage with the texture
texreg2gb   t(d), t(a)        when d > a
```

Listing 11-21: Pixel shader

```
tex           t0
texreg2gb t1, t0
```

■ ***texreg2rgb***: Interpret the red, green, and blue components with {UV}

texreg2rgb *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
		☻	☻						

This instruction interprets the red, green, and blue elements of the source register *aSrc* as (u,v) texture address data and stores the result in *Dst*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex                t(a)       // Assign the stage with the texture
texreg2rgb     t(d), t(a)     when d > a
```

Listing 11-22: Pixel shader

```
tex           t0
texreg2rgb t1, t0
```

■ ***phase***: Phase1 to phase2 transition marker

phase	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
				☻					

This instruction is a transition marker to indicate that phase1 has completed, and it is now time to execute phase2.

Listing 11-23: Pixel shader

```
phase
texld r1, t0      // base map
texld r2, r2      // from calc. environment map
```

- ***bem***: Apply a fake bump environmental map transform

bem *Dst, aSrc, bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
				☻					

This (two-slot) instruction occurs in phase 1 of the pixel shader and can only be called once per shader. The destination red and green component masks must be of the form {.rg or .xy}, keeping in mind that they are the same elements {rgba vs. xyzw}. See *phase*. This instruction cannot be co-issued (+).

o_n	r_n	t_n	Dst
	☻		1.4 phase 1 .rg .xy

c_n	r_n	s_n	t_n	v_n	Src
☻	☻				1.4 phase 1 (aSrc)
	☻				1.4 phase 1 (bSrc)

Pseudocode:

```
N == Dst(#)

Dst_r = aSrc_r + bSrc_r · D3DTSS_BUMPENVMAT00(stage N)
               + bSrc_g · D3DTSS_BUMPENVMAT10(stage N)
Dst_g = aSrc_g + bSrc_r · D3DTSS_BUMPENVMAT01(stage N)
               + bSrc_g · D3DTSS_BUMPENVMAT11(stage N)
```

Keep in mind that the 2x2 texture bump-mapping matrix stage states are:

```
D3DTSS_BUMPENVMAT00 = 7,   [0][0] in Bump Map Matrix
D3DTSS_BUMPENVMAT01 = 8,   [0][1]      "      "
D3DTSS_BUMPENVMAT10 = 9,   [1][0]      "      "
D3DTSS_BUMPENVMAT11 = 10,  [1][1]      "      "
```

Listing 11-24: Pixel shader

```
ps.1.4

texld r1, t1        ; bump map
texcrd r2.rgb, t2
bem  r2.rg, r2, r1  ; convert from tex coords...
                    ; ...to environmental map.
phase

texld r0, t0        ; Color map
texld r2, r2        ; Environmental map
add   r0, r0, r2

; Alternate
bem r3.xy, c0, r0
```

Pixel Shader Instructions (Texture Matrices)

- ***texm3x2pad***: First row multiply of two-row matrix multiply

texm3x2pad *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction applies the first row of the matrix specified by the source *aSrc*:

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

Results of this instruction must *not* be read! Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex               t(a)          // Assign the stage with the texture
texm3x2pad  t(d), t(a)          // d+0 > a     Matrix first row

Followed by either a...
texm3x2depth, texm3x2tex
...with a t(d+1).

texm3x2???  t(d+1), t(a)    // d+1 > a   Matrix second row + FUNCTION
```

Listing 11-25: Pixel shader

```
tex            t0
texm3x2pad t1, t0
```

- ***texm3x2depth***: Calculate the depth to test pixel

texm3x2depth *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
			☻						

This instruction is the second stage of a texture matrix instruction. It accepts the texture source *aSrc*, which represents the second row, and calculates and stores the result to destination *Dst*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

This instruction must be used in conjunction with a *texm3x2pad* instruction, which calculates the product of the first matrix row. Results of this instruction must *not* be read! Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex                t(a)          // Assign the stage with the texture
texm3x2pad   t(d), t(a)          // d+0 > a     Matrix first row
texm3x2depth t(d), t(a)          // d+1 > a     Matrix second row
```

Listing 11-26 Pixel shader

```
tex              t0
texm3x2pad   t1, t0
texm3x2depth t2, t0
mov          r0, t2
```

- ***texm3x2tex***: Last row multiply of 3x2 row matrix multiply

texm3x2tex *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction applies the last row of the matrix specified by the source *aSrc*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

This instruction must be used in conjunction with a *texm3x2pad* instruction, which calculates the product of the first matrix row. Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex              t(a)          // Assign the stage with the texture
texm3x2pad  t(d), t(a)         // d+0 > a     Matrix first row
texm3x2tex  t(d), t(a)         // d+1 > a     Matrix second row
```

Listing 11-27: Pixel shader

```
tex            t0
texm3x2pad t1, t0
texm3x2tex t2, t0
mov        r0, t2
```

■ ***texm3x3pad***: First or second row multiply of three-row matrix multiply

texm3x3pad *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction applies the first row or second row of the matrix specified by the source vector *aSrc*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

Results of this instruction must *not* be read! Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex                t(a)    // Assign the stage with the texture
texm3x3pad   t(d), t(a)    // d+0 > a     Matrix first row
```

```
texm3x3pad    t(d+1), t(a)   // d+1 > a     Matrix second row

Followed by either a...
texm3x3, texm3x3tex, texm3x3spec, texm3x3vspec
...with a t(d+2).

texm3x3???   t(d+2), t(a)    // d+2 > a   Matrix third row + FUNCTION
```

Listing 11-28: Pixel shader

```
tex            t0
texm3x3pad t1, t0
texm3x3pad t2, t0
// Insert...    texm3x3 ???  t3, t0    ...instruction here!
```

■ ***texm3x3***: 3x3 matrix multiply

texm3x3 *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
		☻	☻						

This instruction multiplies a 3x3 matrix specified by the source *aSrc*.

The instruction is similar to *texm3x3tex* but without the texture lookup.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

This instruction must be used in conjunction with two *texm3x3pad* instructions, each calculating the product of a matrix row. Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex                  t(a)      // Assign the stage with the texture
texm3x3pad   t(d),   t(a)      // d+0 > a     Matrix first row
texm3x3pad   t(d+1), t(a)      // d+1 > a     Matrix second row
texm3x3      t(d+2), t(a)      // d+2 > a     Matrix third row
```

Listing 11-29: Pixel shader

```
tex           t0
texm3x3pad t1, t0
texm3x3pad t2, t0
texm3x3    t3, t0
mov        r0, t3
```

■ ***texm3x3tex***: A 3x3 matrix multiply with texture lookup

texm3x3tex *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction is the third stage of a texture matrix multiply instruction using a texture source *aSrc,* which represents the third row, and calculates and stores the result to destination *Dst*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

This instruction must be used in conjunction with two *texm3x3pad* instructions, each calculating the product of a matrix row. Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex                 t(a)     // Assign the stage with the texture
texm3x3pad   t(d),  t(a)     // d+0 > a     Matrix first row
texm3x3pad   t(d+1), t(a)    // d+1 > a     Matrix second row
texm3x3tex   t(d+2), t(a)    // d+2 > a     Matrix third row
```

Listing 11-30: Pixel shader

```
tex           t0
texm3x3pad t1, t0
texm3x3pad t2, t0
texm3x3tex t3, t0
mov        r0, t3
```

- ***texm3x3spec***: A 3x3 matrix multiply with texture lookup for spec map

texm3x3spec *Dst*, *aSrc*, *bSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction is the third stage of a texture matrix multiply instruction using a texture specular map source *aSrc,* which represents the third row, and calculates and stores the result to destination *Dst*.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		aSrc
☻					bSrc

This instruction must be used in conjunction with two *texm3x3pad* instructions, each calculating the product of a matrix row. Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex                t(a)          // Assign the stage with the texture
texm3x3pad   t(d),   t(a)        // d+0 > a     Matrix first row
texm3x3pad   t(d+1), t(a)        // d+1 > a     Matrix second row
texm3x3spec  t(d+2), t(a), c(b)  // d+2 > a     Matrix third row
```

Listing 11-31: Pixel shader

```
tex             t0
texm3x3pad  t1, t0
texm3x3pad  t2, t0
texm3x3spec t3, t0, c3
mov         r0, t3
```

- ***texm3x3vspec***: A 3x3 matrix multiply with texture lookup and eye-ray vec

texm3x3vspec *Dst*, *aSrc*	1.1	1.2	1.3	1.4	2.0	2_x	2_{sw}	3.0	3_{sw}
	☻	☻	☻						

This instruction is the last step of a 3x3 matrix multiply in which a third row matrix multiply is performed and the resulting three-vector result is reflected by the eye-ray vector with the result used as an address for a texture lookup.

o_n	r_n	t_n	Dst
		☻	

c_n	r_n	s_n	t_n	v_n	Src
			☻		

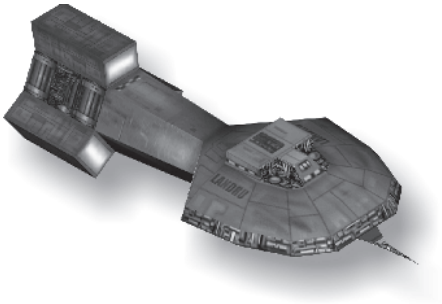
This instruction must be used in conjunction with two *texm3x3pad* instructions, each calculating the product of a matrix row. Texture indexing rules apply, whereas the index for *Dst* > *aSrc*.

Pseudocode:

```
tex                 t(a)        // Assign the stage with the texture
texm3x3pad   t(d),  t(a)        // d+0 > a     Matrix first row
texm3x3pad   t(d+1), t(a)       // d+1 > a     Matrix second row
texm3x3vspec t(d+2), t(a)       // d+2 > a     Matrix third row
```

Listing 11-32: Pixel shader

```
ps.1.1
tex              t0   // {xyz}
texm3x3pad   t1, t0   // first row of matrix multiply
texm3x3pad   t2, t0   // second row of matrix multiply
texm3x3vspec t3, t0   // third row of matrix multiply
mov          r0, t3   // store the reflection calculation
```



Chapter 12

Rendering Up, Up 'n Away

Pixel Shading for Fun and Profit

So are you ready to start your own game company or get a job at an existing one? Or maybe you are a game tester and general coding programmer and looking to enhance your programming skills to move on to bigger and better things. Within this book, you should have found examples for almost every shader instruction (something I have found lacking in the books that I have reviewed). Within those examples, you should have found and developed the foundations that you need for moving on to advanced shader programming, such as understanding the ShaderX books by Wolfgang Engel. With an understanding of the principles of this book, you should now be able to move forward.

Do you have an old computer you are thinking about retiring? Drop a shader-capable render card into it, and give it to the kids. You will get a few more years of life out of it and will save money in the long run.

All joking aside, there *is* more to cover!

Where Do You Go From Here?

Macros, of course!

```
#include "..\ShaderDefs.h"
#include "..\ShaderMacros.h"

vs.1.1
```

Definitions and macros can both be contained within common files to be used by different shader algorithms.

Listing 12-1: Portion of ShaderDefs.h from Paul Stapley

```
// Eye position in world space
  #define CV_EYE_POS_WORLD       0

// Eye direction normal
  #define CV_EYE_VECTOR          1

// Model to world matrix
  #define CV_WORLD               2
  #define CV_WORLD_0        CV_WORLD + 0
  #define CV_WORLD_1        CV_WORLD + 1
  #define CV_WORLD_2        CV_WORLD + 2
  #define CV_WORLD_3        CV_WORLD + 3

// Transpose of model to world matrix
  #define CV_WORLD_TRANSPOSE      6
  #define CV_WORLD_TRANSPOSE_0 CV_WORLD_TRANSPOSE + 0
  #define CV_WORLD_TRANSPOSE_1 CV_WORLD_TRANSPOSE + 1
  #define CV_WORLD_TRANSPOSE_2 CV_WORLD_TRANSPOSE + 2
  #define CV_WORLD_TRANSPOSE_3 CV_WORLD_TRANSPOSE + 3

  #define CV_LOCAL_TO_WORLD_INVERSE  10

// Definitions can be used in place of registers as well!
  #define VERT_POSITION       v0 // Position
  #define VERT_COLOR          v1 // Color
  #define VERT_NORMAL         v2 // Normal
  #define VERT_TEX1           v3 // Texture 1 Coordinates
  #define VERT_TEX2           v4 // Texture 2 Coordinates
  #define VERT_TEX3           v5 // Texture 3 Coordinates
  #define VERT_TEX4           v6 // Texture 4 Coordinates

  #define TEMP_REG            r5
  #define VERTEX_TO_LIGHT_ATT r6
```

```
  #define VERTEX_TO_LIGHT_COS r7
  #define FINAL_COLOR         r8
  #define SPECULAR_COLOR      r9
  #define SPECULAR_INTENSITY  r4.x

// Constant values (start at 100)
  #define CV_CONSTANT_ZERO    95 // x, y, z, w = 0.0f
  #define CV_CONSTANT_ONE     94 // x, y, z, w = 1.0f
  #define CV_CONSTANT_TWO     93 // x, y, z, w = 2.0f
  #define CV_CONSTANT_HALF    92 // x, y, z, w = 0.5f
```

Listing 12-2: Portion of ShaderMacros.h from Paul Stapley

```
// TransformVector3 transforms ArgVector by matrix
// starting in register ArgMatrix and places the results
// in ArgDest.  ArgDest != ArgVector     Only transforms
// the x, y, and z coordinates

macro TransformVector3 ArgDest, ArgMatrix, ArgVector
  dp3 %ArgDest.x, %ArgMatrix,             %ArgVector
  dp3 %ArgDest.y, %inc(%ArgMatrix),       %ArgVector
  dp3 %ArgDest.z, %inc(%inc(%ArgMatrix)), %ArgVector
  mov %ArgDest.w, c[ CV_CONSTANT_ONE ]
endm

// NormalizeVector normalizes the vector in ArgVector and
// places the results in ArgVector.  ArgVector.w holds
// the length of the vector after this is called.

macro NormalizeVector ArgVector
  dp3 %ArgVector.w, %ArgVector, %ArgVector
                                    // d*d = (x2)+(y2)+(z2)
  rsq %ArgVector.w, %ArgVector.w
                                    // 1/dd = 1/v(d*d)
  mul %ArgVector.xyz, %ArgVector.xyz, %ArgVector.www
endm                                // x/d, y/d, z/d
```

So, in a typical usage:

Listing 12-3: Portion of macro usage from Paul Stapley

```
#include "..\ShaderDefs.h"
#include "..\ShaderMacros.h"

vs.1.1

; Transform normal to {r0} world space normal
TransformVector3 r0, c[CV_WORLD_TRANSPOSE], VERT_NORMAL
```

This expands into:

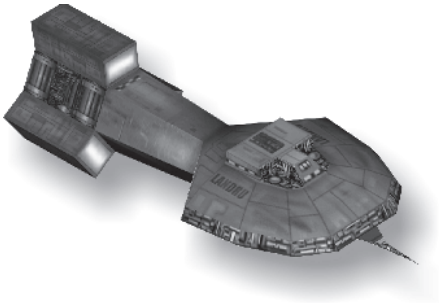
```
dp3 r0.x, c[CV_WORLD_TRANSPOSE+0], VERT_NORMAL
dp3 r0.y, c[CV_WORLD_TRANSPOSE+1], VERT_NORMAL
dp3 r0.z, c[CV_WORLD_TRANSPOSE+2], VERT_NORMAL
mov r0.w, c[ CV_CONSTANT_ONE ]
```

…and with definition remapping:

```
dp3 r0.x, c6, v2
dp3 r0.y, c7, v2
dp3 r0.z, c8, v2
mov r0.w, c92
```

This makes programming shader assembly code for large projects much easier. Common definitions between shaders are defined in a game-specific file, the way that ShaderDefs.h does. Macros with reusable shader snippets can be shared between game applications, the way that ShaderMacros.h does.

But it does not end here! This book was written so that the beginner can build an understanding of how shaders work. But there is something that this book does not discuss — the Cg shader language. This is a compiler for graphics using a C-type programming language and shaders. It has some pluses and some minuses, but these are more of a personal preference. For example, for a pixel shader, using versions 1.4 and below did not make much sense (at least to me) due to the severe instruction count limitation to which one is subjected. Now with version 2.0, pixel shaders have a large number of instructions, and so this method of shader construction actually makes a lot more sense!



Epilogue

In this book we saw a brief overview of the various shader-capable graphics cards, vertex shaders, pixel shaders, and the instruction sets needed to program them. This book practically ignored version 1.0, as it is no longer supported by DirectX 9. It was discovered that even when working with DirectX 9, it is warranted to work in as low a shader version as possible. This makes the code useable across more video cards, and higher version functionality only works in hardware on the newer, more expensive cards.

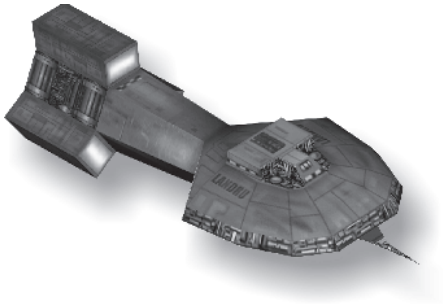
So where do you go from here? Is there more that can be done with these instructions? Check out the document "Where Is That Instruction? How to Implement 'Missing' Vertex Shader Instructions," available for download from the nVidia web site.

Download all the SDKs and tools and experiment. Above all, practice, practice, practice! Make this one of your specialties when you go for job interviews. Or, then again, don't! Forget everything you have read so far in this book. Being a 3D programmer is nothing more than downloading shaders and calling APIs that someone else wrote anyway, right? You do not need to know shader programming because someone else can always construct the rendering sections of the game code. Good for you! (Thank you, as that leaves the possibility of another job opportunity in the future the next time I need to go job hunting!)

Are you still reading this chapter? Did you actually read the entire book? Great (if not, for shame). Why are you still reading? It is time to start doing. Get up out of that chair, get to your computer and get to it! Oh, and for those of you money savers, buy a copy of this book as a reference manual on the way out of the bookstore. You just might find that it will come in handy!

Well, I guess I can now install some of my top-of-the-line video cards into my kids' computers so they can play Toon Town and other 3D games!

Happy shading!



Appendix A

Shaders — Opcode Ordered

These are the opcodes associated with programmable vertex and pixel shaders. Note that they both share the same opcodes, as they both typically use the same assembler. In some cases, pixel shader 1.0 through 1.3 uses an opcode for one instruction, and 1.4 uses the opcode for a different instruction. This is typically because the number of operands changes between versions!

```
D3DSIO_INSTRUCTION_OPCODE_TYPE
```

D3DSIO_xxxxx (Defined within D3D9Types.h)

OpCode	Operand	V	P	W	I	Description
0x0000	nop	☺	☻	1	1	No operation
0x0001	mov	☺	☻	3	1	Move
0x0002	add	☺	☻	4	1	Addition
0x0003	sub	☺	☻	4	1	Subtraction
0x0004	mad	☺	☻	5	1	Multiplication-addition
0x0005	mul	☺	☻	4	1	Multiplication
0x0006	rcp	☺	☻	3	1	Reciprocal
0x0007	rsq	☺	☻	3	1	Reciprocal square root
0x0008	dp3	☺	☻	4	1	Dot product (xyz)
0x0009	dp4	☺	☻	4	1	Dot product (xyzw)
0x000a	min	☺	☻	4	1	Minimum
0x000b	max	☺	☻	4	1	Maximum
0x000c	slt	☺		4	1	Set if <

OpCode	Operand	V	P	W	I	Description
0x000d	sge	☺		4	1	Set if ≥
0x000e	exp	☺	☻	3	1,10	Exponential 2^x full precision
0x000f	log	☺	☻	3	1,10	$Log_2(x)$ full precision
0x0010	lit	☺		3	1,3	Calc. light coefficients
0x0011	dst	☺		4	1	Calc. distance vector
0x0012	lrp	☺	☻	5	1,2	Linear interpolation
0x0013	frc	☺	☻	3	1,3	Get fractional comp.
0x0014	m4x4	☺	☻	4	4	Vec product matrix 4x4
0x0015	m4x3	☺	☻	4	3	Vec product matrix 4x3
0x0016	m3x4	☺	☻	4	4	Vec product matrix 3x4
0x0017	m3x3	☺	☻	4	3	Vec product matrix 3x3
0x0018	m3x2	☺	☻	4	2	Vec product matrix 3x2
0x0019	call	☺	☻	2	2	Function call
0x001a	callnz callnz pred	☺	☻	3	3	Func. call if ≠ 0 Func. call if ≠ 0 with pred.
0x001b	loop	☺	☻	3	3	Loop begin
0x001c	ret	☺	☻	1	1	Return from subroutine
0x001d	endloop	☺	☻	1	2	End of loop
0x001e	*label*	☺	☻	2	0	Function label
0x001f	*dcl* *dcl_2d* *dcl_usage*	☺ ☺	☻ ☻ ☻	3	0	Declaration " "
0x0020	pow	☺	☻	4	3	Power
0x0021	*crs*	☺	☻	4	2	Cross product
0x0022	*sgn*	☺		5	3	Set sign
0x0023	abs	☺	☻	3	1	Absolute
0x0024	*nrm*	☺	☻	3	3	Normalize
0x0025	sincos	☺	☻	5 3	8	2.0 sine/cosine 3.0 sine/cosine
0x0026	rep	☺	☻	2	3	Repeat begin
0x0027	endrep	☺	☻	1	2	End repeat loop
0x0028	if if_pred	☺	☻	2	3	If-else-endif If-else-endif with pred.
0x0029	ifc	☺	☻	3	3	If-else-endif with comp.
0x002a	else	☺	☻	1	1	If-else-endif
0x002b	endif	☺	☻	1	1	If-else-endif
0x002c	break	☺	☻	1	1	Break from loop/rep
0x002d	breakc	☺	☻	3	3	Break from loop with comp.

OpCode	Operand	V	P	W	I	Description
0x002e	mova	☺		3	1	Move reg to aL
0x002f	*defb*	☺	☻	2	0	Bool const. definition
0x0030	*defi*	☺	☻	6	0	Int const. definition
0x0040	texcoord		☻	2	1	(<=1.3) uvw1 to rgba
	texcrd		☻	3	1	(1.4) Copy tex as color
0x0041	texkill		☻	2	$1,1_T,2_T$	Cancel render of pixel
0x0042	tex		☻	2	1	(<=1.3) RGBA load
	texld		☻	3	$1,1+3_3$	(1.4) RGBA load
	texldb		☻	4	$1_T,6,6_T$	
	texldp		☻	4	$1_T,3+1_3$	
0x0043	texbem		☻	3	1	Fake bump map xform
0x0044	texbeml		☻	3	$1+1_T$	Fake bump map with lum.
0x0045	texreg2ar		☻	3	1	Alpha-red to tex. addr.
0x0046	texreg2gb		☻	3	1	Green-blue to tex. addr.
0x0047	texm3x2pad		☻	3	1	Mul. matrix 3x2 (first row)
0x0048	texm3x2tex		☻	3	1	Mul. matrix 3x2 (last row)
0x0049	texm3x3pad		☻	3	1	Mul. matrix 3x3 (first, second)
0x004a	texm3x3tex		☻	3	1	Mul. matrix 3x3 tex. idx
0x004b	texm3x3diff		☻			
0x004c	texm3x3spec		☻	4	1	Mul. matrix 3x3 spec
0x004d	texm3x3vspec		☻	3	1	Mul. matrix with eye-ray
0x004e	expp	☺		3	1	Exp. 2^x partial precision
0x004f	logp	☺		3	1	$Log_2(x)$ partial precision
0x0050	cnd		☻	5	1	Cond. upon 0.5 factor
0x0051	*def*	☺	☻	6	0	Vec const definition
0x0052	texreg2rgb		☻	3	1	Interp. RGB to tex. addr.
0x0053	texdp3tex		☻	3	1	Texture dot product (xyz)
0x0054	texm3x2depth		☻	3		Texture matrix 3x2 depth
0x0055	texdp3		☻	3	1	Texture dot product (xyz)
0x0056	texm3x3		☻	3	1	Mul. matrix 3x3
0x0057	texdepth		☻	2	1	Calc. depth values
0x0058	cmp		☻	5	1	Conditional choose
0x0059	bem		☻	4	2	Bump env. map xform
0x005a	dp2add		☻	5	1,2	Dot product (xy)
0x005b	dsx		☻	3	2	X rate of change
0x005c	dsy		☻	3	2	Y rate of change
0x005d	texldd		☻	6	3	

OpCode	Operand	V	P	W	I	Description
0x005e	setp	☺	☻	4	1	Set predicate with cmp.
0x005f	texldl	☺	☻	4	$2+3_3$	
0x0060	breakp	☺	☻		3	Break from loop with predicate
0xfffd	*phase*		☻	1	0	Phase one to phase two
0xfffe	comment					
0xffff	end		☻	1		End of code

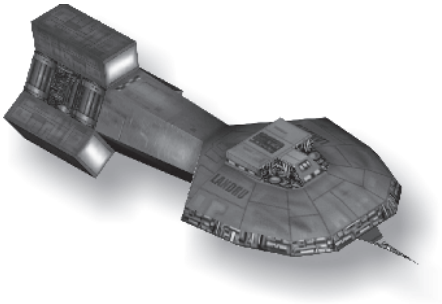
NOTE: *The table entries left empty are not mistakes*. They were left available so that those of you who are Xbox developers can fill them in yourselves with that proprietary information.

The operands are a combination of *statements*, non-programmable statements, and instructions. The opcode is the opcode encoded into the *.vso and *.pso files for the shader decoder to process.

The ☺ and ☻ indicate whether that operand is supported by the vertex or pixel shader.

The W represents the number of 32-bit words that the operand packs into at assembly time.

The I represents the number of instruction slots needed. Note that if a dual set of numbers delineated by a comma is specified, then the larger number is typically to represent a version 1.x when it is a macro and the smaller number from version 2.0 or newer when it is implemented as an instruction. A value of 0 indicates a statement, not an instruction. Therefore, an instruction slot is not consumed. A value such as $1+3_3$ indicates one plus three if a cube map is used.



Appendix B

Shaders — Mnemonic Ordered

These are the opcodes associated with programmable vertex and pixel shaders. Note that they both share the same opcodes, as they both typically use the same assembler. In some cases, pixel shader 1.0 through 1.3 uses an opcode for one instruction and 1.4 uses the opcode for a different instruction. This is typically because the number of operands changes between versions!

```
D3DSIO_INSTRUCTION_OPCODE_TYPE
```

D3DSIO_xxxxx (Defined within D3D9Types.h)

Operand	OpCode	V	P	W	I	Description
abs	0x0023	☺	☻	3	1	Absolute
add	0x0002	☺	☻	4	1	Addition
bem	0x0059		☻	4	2	Bump env. map xform
break	0x002c	☺	☻	1	1	Break from loop/rep
breakc	0x002d	☺	☻	3	3	Break from loop with comp.
breakp	0x0060	☺	☻		3	Break from loop with predicate
call	0x0019	☺	☻	2	2	Function call
callnz	0x001a	☺	☻	3	3	Func. call if ≠ 0
callnz_pred	0x001a	☺	☻	3	3	Func. call if ≠ 0 with pred.
cmp	0x0058		☻	5	1	Conditional compare
cnd	0x0050		☻	5	1	Cond. upon 0.5 factor
comment	0xfffe					
crs	0x0021	☺	☻	4	2	Cross product

Operand	OpCode	V	P	W	I	Description
dcl	0x001f	☺	☻	3	0	Declaration
dcl_2d	0x001f		☻	3	0	"
dcl_usage	0x001f	☺	☻	3	0	"
def	0x0051	☺	☻	6	0	Vec const. definition
defb	0x002f	☺	☻	2	0	Bool const. definition
defi	0x0030	☺	☻	6	0	Int const. definition
dp2add	0x005a		☻	5	1,2	Dot product (xy)
dp3	0x0008	☺	☻	4	1	Dot product (xyz)
dp4	0x0009	☺	☻	4	1	Dot product (xyzw)
dst	0x0011	☺		4	1	Calc. distance vector
dsx	0x005b		☻	3	2	X rate of change
dsy	0x005c		☻	3	2	Y rate of change
else	0x002a	☺	☻	1	1	If-else-endif
end	0xffff		☻	1		End of code
endif	0x002b	☺	☻	1	1	If-else-endif
endloop	0x001d	☺	☻	1	2	End of loop
endrep	0x0027	☺	☻	1	2	End repeat loop
exp	0x000e	☺	☻	3	1,10	Exponential 2^x full precision
expp	0x004e	☺		3	1	Exponent 2^x partial precision
frc	0x0013	☺	☻	3	1,3	Get fractional component
if	0x0028	☺	☻	2	3	If-else-endif
ifc	0x0029	☺	☻	3	3	If-else-endif w/comp.
if_pred	0x0028	☺	☻		3	If-else-endif w/predicate
label	0x001e	☺	☻	2	0	Function label
lit	0x0010	☺		3	1,3	Calc. lighting coefficients
log	0x000f	☺	☻	3	1,10	$Log_2(x)$ full precision
logp	0x004f	☺		3	1	$Log_2(x)$ partial precision
loop	0x001b	☺	☻	3	3	Loop begin
lrp	0x0012	☺	☻	5	1,2	Linear interpolation
m3x2	0x0018	☺	☻	4	2	Vec product matrix 3x2
m3x3	0x0017	☺	☻	4	3	Vec product matrix 3x3
m3x4	0x0016	☺	☻	4	4	Vec product matrix 3x4
m4x3	0x0015	☺	☻	4	3	Vec product matrix 4x3
m4x4	0x0014	☺	☻	4	4	Vec product matrix 4x4
mad	0x0004	☺	☻	5	1	Multiplication-addition
max	0x000b	☺	☻	4	1	Maximum
min	0x000a	☺	☻	4	1	Minimum

Operand	OpCode	V	P	W	I	Description
mov	0x0001	☺	☻	3	1	Move
mova	0x002e	☺		3	1	Move reg to aL
mul	0x0005	☺	☻	4	1	Multiplication
nop	0x0000	☺	☻	1	1	No operation
nrm	0x0024	☺	☻	3	3	Normalize
phase	0xfffd		☻	1	0	Phase one to phase two
pow	0x0020	☺	☻	4	3	Power
ps			☻	1	0	Pixel version
rcp	0x0006	☺	☻	3	1	Reciprocal
rep	0x0026	☺	☻	2	3	Repeat begin
ret	0x001c	☺	☻	1	1	Return from call
rsq	0x0007	☺	☻	3	1	Reciprocal square root
setp	0x005e	☺	☻	4	1	Set predicate with comp.
sge	0x000d	☺		4	1	Set if greater than or equal to
sgn	0x0022	☺		5	3	Set sign
sincos	0x0025	☺	☻	5 3	8	2.0 sine/cosine 3.0 sine/cosine
slt	0x000c	☺		4	1	Set if less than
sub	0x0003	☺	☻	4	1	Subtraction
tex	0x0042		☻	2	1	(<=1.3) RGBA load
texbem	0x0043		☻	3	1	Fake bump map xform
texbeml	0x0044		☻	3	$1+1_T$	Fake bump map with lum.
texcoord	0x0040		☻	2	1	(<=1.3) UVW1 to RGBA
texcrd	0x0040		☻	3	1	(1.4) Copy tex as color
texdepth	0x0057		☻	2	1	Calc. depth values
texdp3	0x0055		☻	3	1	Tex. dot product (3 xyz)
texdp3tex	0x0053		☻	3	1	Tex. dot product (3 xyz)
texkill	0x0041		☻	2	$1,1_T,2_T$	Cancel render of pixel
texld	0x0042		☻	3	$1,1+3_3$	(1.4) RGBA load
texldb	0x0042		☻	4	$1_T,6,6_T$	
texldd	0x005d		☻	6	3	
texldl	0x005f	☺	☻	4	$2+3_3$	
texldp	0x0042		☻	4	$1_T,3+1_3$	
texm3x2depth	0x0054		☻	3		Texture matrix 3x2 depth
texm3x2pad	0x0047		☻	3	1	Mul. matrix 3x2 (first row)
texm3x2tex	0x0048		☻	3	1	Mul. matrix 3x2 (last row)
texm3x3	0x0056		☻	3	1	Mul. matrix 3x3

Operand	OpCode	V	P	W	I	Description
texm3x3diff	0x004b		☻	?		
texm3x3pad	0x0049		☻	3	1	Mul. matrix 3x3 (first, second)
texm3x3spec	0x004c		☻	4	1	Mul. matrix 3x3 spec. reflect
texm3x3tex	0x004a		☻	3	1	Mul. matrix 3x3 tex. idx
texm3x3vspec	0x004d		☻	3	1	Mul. matrix with eye-ray
texreg2ar	0x0045		☻	3	1	Alpha-red to tex. addr.
texreg2gb	0x0046		☻	3	1	Green-blue to tex. addr.
texreg2rgb	0x0052		☻	3	1	Interp. RGB to tex. addr.
vs		☺		1	0	Vertex version

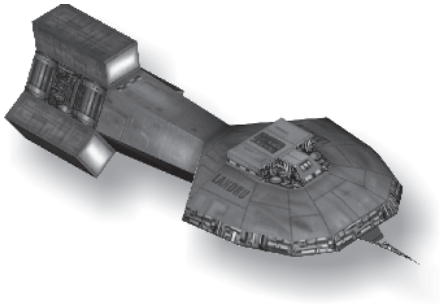
NOTE: *The table entries left empty are not mistakes*. They were left available so that those of you who are Xbox developers can fill them in yourselves with that proprietary information.

The operands are a combination of *statements*, non-programmable statements, and instructions. The opcode is the opcode encoded into the *.vso and *.pso files for the shader decoder to process.

The ☺ and ☻ indicate whether that operand is supported by the vertex or pixel shader.

The W represents the number of 32-bit words that the operand packs into at assembly time.

The I represents the number of instruction slots needed. Note that if a dual set of numbers delineated by a comma is specified, then the larger number is typically to represent a version 1.x when it is a macro and the smaller number from version 2.0 or newer when it is implemented as an instruction. A value of 0 indicates a statement, not an instruction. Therefore, an instruction slot is not consumed. A value such as $1+3_3$ indicates one plus three if a cube map is used.



Appendix C

Instruction Dissection

Some of you may find this of great interest, while others couldn't care less (that is why it is buried way back here in Appendix C). But I am one of those people who likes to see how things work and always takes things apart! Those of you still confused about data swizzling and source negation of {XYZW} elements have probably skipped here from the front of this book to try to get a better understanding. The material contained here should definitely fill that void.

- nvasm.exe: nVidia – V&P Macro Assembler (no longer developed)
- psa.exe: Direct3D9 Pixel Shader Assembler
- vsa.exe: Direct3D9 Vertex Shader Assembler
- xsasm.exe: Xbox Shader Assembler
- cg.exe: Cg (C for graphics) Compiler

All assemblers and compilers compile various versions of shader code but have different options and export different kinds of files of different little-endian formats. But the most important thing to understand is how the ASCII mnemonics get converted into machine opcodes. With this understanding, the concept of swizzled data, as well as the power and limitations it presents, should become second nature to you!

The opcode ordered instructions are listed in Appendix A, "Shaders — Opcode Ordered," and Appendix B, "Shaders — Mnemonic Ordered."

All instructions and their parameters are 32-bit double word-based (dword-based) bit encoding. Note that the term *dword* is used here to represent a 32-bit value, as that is the term used for that size data on an X86-based processor. This book is written for DirectX on a Win32-based machine, so it is important to keep with its adopted terminology. On other processors, the term *word* would be used to represent 32-bit values and *dword* for 64-bit values, as a *half-word* represents a 16-bit *short*, unlike the X86 word instruction.

With that said, it means an instruction that takes no arguments, such as a no operation (*nop*), would use one dword and that is to contain the instruction opcode:

```
nop
```

A texture remove (*texkill*) takes a single argument, thus it would use two dwords:

```
texkill t0
```

A move (*mov*) instruction would take two arguments, thus it would use three dwords:

```
mov r3, c1
```

An addition (*add*) uses four dwords:

```
add r0, v0, c1
```

With that understood, we know that the first dword contains the opcode signifying the instruction.

So what happens if you append more arguments than the instruction requires? You encounter either an assembly error or some very nasty code export!

Any trailing parameters would be encoded into an equal number of trailing dwords. Now we come to the really interesting part — the bit encoding and thus the visualization part.

We are getting a little ahead of ourselves, so let's examine a simple register-to-register copy known as the move instruction (*mov*). We viewed some images earlier, but let's examine the individual bits using the following particular case:

```
mov r3.xy, -c1.zy
```

Please note that all of the following use a little-endian representation.

The Opcode Dword

The first encoded dword contains the 16 opcode bits for the represented instruction.

```
(MSB) 31                  15                  0 (LSB)
      xxxx xxxx xxxx xxxx ???? ???? ???? ????
                                    ↑
```

The opcode occupies the lower 16 bits, thus a value between 0x0000 and 0xffff (65,535) is supported. Since the move instruction uses the opcode value of 2, the instruction bit encoding will appear as the following:

```
mov r3.xy, -c1.zy

(MSB) 31                  15                  0 (LSB)
      0000 0000 0000 0000 0000 0000 0000 0010    = 0x0002
                                    ↑
```

The Parameter (Argument) Dword

Okay, I am getting a little ahead of myself, but there are basically six kinds of registers used by the shaders and certain behavioral rules for each as to their usage. Let's ignore that for now, as it is discussed in the appropriate shader chapters. Our only concern for now is with their bit encoding.

Register Identifier Code

With the current release of technology, these are the base registers used by the shaders:

Table C-1: Shader register labels and the base codes.

Register:	r#	v#, s#	c#	oD#	t# oT#	o(utput)
Base:	0x80	0x90	0xa0	0xd0	0xe0	0xc0
Bits:	1000	1001	1010	1101	1110	1100

Note the # represents a numerical index value from 0 to n–1 (the range of indices that the type of register supports).

```
(MSB) 31       24                    7       0 (LSB)
      1??? XXXS XXXX XXXX XXXX XXXX #### ####
  Base  ↑     ↑ Inverse                  ↑ Index
```

The uppermost four bits represent these current values (8…F). The bits (7…0) represent the numerical index of that register. For example, for register *c3*, the upper four bits would be set to 1010, and the lower eight bits would be set to 00000011.

There is an extra bit of functionality, and that is the inverse flag on bit 24. If set, then the values in the selected source elements are negated before being processed by the instruction. Each source parameter to the instruction has the ability to set this flag. The following table of scalar multiplication is used as a product (multiply) for clarification. I hope you remember the following from grade school:

xy = x × y	–xy = x × –y	–xy = –x × y	xy = –x × –y
+ = (+) × (+)	– = (+) × (–)	– = (–) × (+)	+ = (–) × (–)

So again, we examine the source register components of our instruction:

```
mov r3.xy, -c1.zy      (Source)

(MSB) 31       24                    7       0 (LSB)
      1010 XXX1 XXXX XXXX XXXX XXXX 0000 0001
  Base  ↑(c)  ↑ (-) Inverse              (1)↑ Index
```

Please remember that only a source register can be negated, as the negation of the destination is an illegal operation and will result in an error!

```
mov r3.xy, -c1.zy     (Destination)

(MSB) 31       24                    7       0 (LSB)
      1000 XXX0 XXXX XXXX XXXX XXXX 0000 0011
  Base  ↑(r)   ↑                     (3)↑ Index
```

Destination {XYZW} Elements

As {XYZW} element ordering for the destination register is the simplest to understand and most important, it is discussed first!

```
(MSB) 31             19  16                0 (LSB)
      XXXX XXXX XXXX ???? XXXX XXXX XXXX XXXX
                     wzyx
```

It was mentioned earlier that the destination is sequenced in an {XYZW} order. There are only four bits {16:x, 17:y, 18:z, 19:w} to represent those elements, so if the bit is set to 1, then the corresponding element is altered; if cleared to 0, it is left alone. The source information is used to alter the destination in sequence.

```
mov r3.xy, -c1.zy     (Destination)

(MSB) 31             19  16                0 (LSB)
      XXXX XXXX XXXX 0011 XXXX XXXX XXXX XXXX
                     wzyx
```

There can be gaps in the ordering, such as the {Z} missing in the following {XYW}, but no swizzling {YXW} can occur, such as the reversal of the {X} and {Y} elements.

Source Swizzled {XYZW} Elements

The source elements can be individually swizzled so that they can be arranged into any combination!

```
(MSB) 31        23       16                0 (LSB)
      XXXX XXXX ???? ???? XXXX XXXX XXXX XXXX
                ↑  ↑ ↑  ↑
                D3 D2 D1 D0  Source element
```

This concept is going to be a little tricky, as selected destination elements D_{xyzw} (alias D_{0123}) need to be set to store each individual elemental result of the instruction. Each destination indexed element to be altered needs to have a corresponding indexed source element. In this previously shown bit encoding, D_0 is the first

element result in a sequence up to D_3, where D_3 would be the last element result (provided that four elements were being processed).

D_0: bits 17,16 D_1: bits 19,18 D_2: bits 21,20 D_3: bits 23,22

Each of those two bits decodes to one of the following as to the source element:

0 0 .x
0 1 .y
1 0 .z
1 1 .w

So using our same move instruction:

```
mov r3.xy, -c1.zy     (Source)

(MSB) 31        23      16                  0 (LSB)
      XXXX XXXX 0101 0110 XXXX XXXX XXXX XXXX
                ↑Y↑Y ↑Y↑Z
                D3 D2 D1 D0  Source element
```

This is just a note for you, but this particular instruction fortunately had two source elements and two destination elements. It could have just as easily been two sources and three or four destination elements. But where would the source for the others have come from? The solution is simple! If the source uses less than the full four possible elements $D_0 \ldots D_3$, then the last specified element is merely replicated into the remaining slots. In this way, the destination will have all available. So in our particular case, the {Y} was replicated into the D_2 and D_3 slots and is denoted by the underlined {**Y**}.

Here are a couple of samples:

mov D.xyzw, A.xyzw	D_3 D_2 D_1 D_0	D_0 = A.x	D.x = D_0
0123	w z y x	D_1 = A.y	D.y = D_1
D.xyzw A.0123	11 10 01 00	D_2 = A.z	D.z = D_2
xyzw xyzw		D_3 = A.w	D.w = D_3
mov D.xyzw, A.x	D_3 D_2 D_1 D_0	D_0 = A.x	D.x = D_0
0	x→x→x→x	D_1 = A.x	D.y = D_1
D.xyzw A.0000	00 00 00 00	D_2 = A.x	D.z = D_2
xyzw xxxx		D_3 = A.x	D.w = D_3

Keep in mind that in the following, the second {X} of the source *A* is redundant, as it is replicated for D_2 and D_3 to fill the remaining source element slots!

mov D.xyw, A.zxx	D_3 D_2 D_1 D_0	D_0 = A.z	D.x = D_0
012	x→x x z	D_1 = A.x	D.y = D_1
D.xyw A.012	00 00 00 10	D_2 = A.x	
xyw zxx		D_3 = A.x	D.w = D_2

Some quick bit encodings:

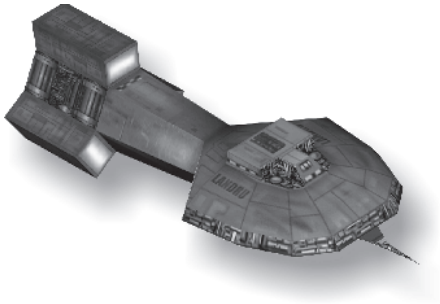
{00 00 00 00} 00=.x	{11 10 01 00} e4=.xyzw
{01 01 01 01} 55=.y	{01 01 01 00} 54=.xy
{10 10 10 10} aa=.z	{00 00 00 11} 03=.wx
{11 11 11 11} ff=.w	{11 11 10 01} 01=.yzw

Some of you may at this point be thinking, "Ahem, excuse me. Where's the rest? I mean, what happened to the bit encodings for all those other little features related to DirectX 9. This only covers all the features of DirectX 8!"

The answer is simple: This particular knowledge isn't needed to build shader code. As you aren't building an assembler or compiler, it really isn't necessary to know. And as I mentioned at the beginning of this appendix, only some of you would be interested in this (as was I when the assemblers for DirectX version 8 weren't as helpful during their operation as I would have liked and was thus building my macro-based multi-manufacturer supported version!). So I leave it to you (if you have the time and are inquisitive enough) to reverse-engineer those instructions further for yourself.

Besides, I have games to write. So there!

This page intentionally left blank.



Appendix D

Floating-Point 101

You may wonder, "What do details about floating-point have to do with this book?" This book is about shaders, not floating-point numbers, especially those that support single precision. Well, to put it briefly, this is really about precision, and to understand precision one must have an understanding of the foundations of how floating-point values are stored. Most programmers tend to have an integer type mentality, and some have an idea that there is some precision loss — but not how much, or why!

Remember, this is not rocket science, and so minor deviations will occur in the formulas since, for example, a single-precision float is only 32 bits in size, which is the data size that this book predominately uses. For higher precision, 64-bit double-precision or 80-bit double extended-precision floating-point should be used instead. These floating-point numbers are based upon a similarity to the IEEE 754 standards specification. Unfortunately, the 80-bit version is only available in a scalar form on an X86's FPU, and the 64-bit *packed* double-precision is only available on the SSE2 processor.

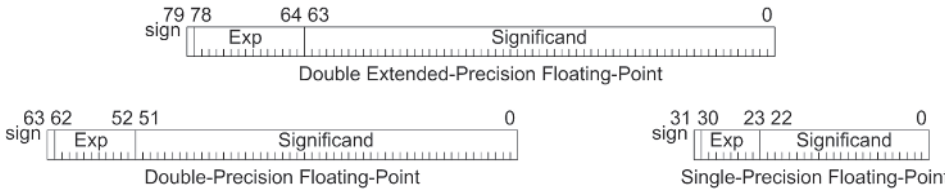


Figure D-1: Floating-point bit configurations

Most programmers only know a floating-point value from using a declaration, such as a float, double, real4, or real8, etc. They know that there is a sign bit that if set indicates the value is negative and if clear means the value is positive. That is about it to them, as it is pretty much a black box, and they have never had a need to dig into it further.

I felt that there needed to be a deeper understanding within this book, but it is off topic from shaders and thus is in the back of the book in an appendix.

The sign bit is the MSB (most significant bit) just like the integer, but that is where the similarity stops.

To invert an integer y=–x (take the absolute value of), one only needs a 2's complement. That consists of a 1's complement (a NOT) followed by an increment (addition by one):

```
    00000000000000001010010110100101b   00000a5a5h (42405)
NOT 11111111111111101011010010110101b   0ffff5a5ah (-42406)
INC 11111111111111101011010010110110b   0ffff5a5bh (-42405)
```

…and back:

```
    11111111111111101011010010110110b   0ffff5a5bh (-42405)
NOT 00000000000000001010010110100100b   00000a5a4h (42404)
INC 00000000000000001010010110100101b   00000a5a5h (42405)
```

But for a floating-point number, regardless of the type of precision {single, double, double-extended}, only the MSB needs to have a 1's complement...

```
    00111111100000000000000000000000b   03f800000h ( 1.0)
NOT 10111111100000000000000000000000b   0bf800000h (-1.0)
```

It's a much easier operation, but that is where the simplicity stops.

The exponent is a base-2 power representation stored as a binary integer. The significand (mantissa) really consists of two components, a J-bit and a binary fraction.

For the single-precision value, there is a hidden "1." leading the 23 bits of the mantissa, making it a 24-bit significand. The exponent is 8 bits, thus it has a bias value of 127. The magnitude of the supported range of numbers is 2×10^{-38} to 2×10^{38}.

For double-precision values, there is a hidden "1." leading the 52 bits of the mantissa, making it a 53-bit significand. The exponent is 11 bits, thus it has a bias value of 1023. The magnitude of the supported range of numbers is 2.23×10^{-308} to 1.8×10^{308}.

For the 80-bit version, the extra bits are primarily for protection against precision loss from rounding and over/underflows. The leading "1." is the 64th bit of the significand. The exponent is 15 bits, thus it has a bias value of 32767. The magnitude of the supported range of numbers is 3.3×10^{-4932} to 1.21×10^{4932}.

The product of the exponent and significand result in the floating-point value.

Table D-1: Single-precision floating-point to hex equivalent

Value	Hex	Sign Exp Sig.
–1.0	0xBF800000	1 7F 000000
0.0	0x00000000	0 00 000000
0.0000001	0x33D6BF95	0 67 56BF95
1.0	0x3F800000	0 7F 000000
2.0	0x40000000	0 80 000000
3.0	0x40400000	0 80 800000
4.0	0x40800000	0 81 000000

Programmers are also usually aware that floats cannot be divided by zero or process a square root of negative one because an exception error would occur.

But as discussed in Chapter 3, "Vertex Shaders," in regard to the *rcp* (reciprocal) and *rsq* (reciprocal square root) instructions, that is not the case with a shader. The calculation that normally results in an exception error is adjusted so that since:

$$y = 1 / x \qquad y = \infty \text{ iff } x = 0 \quad \text{(divide by zero)}$$

…is in reality infinity, a valid floating-point value as close to infinity as possible is substituted. For the reciprocal square root:

$$y = y' = 1 / \sqrt{x} \quad y = 1 / \sqrt{|x|} \quad \text{iff } x < 0 \quad \text{since } y' = 1 / \boldsymbol{i}\sqrt{x} = 1 / \sqrt{-x}$$
$$y = \text{FLT\_MAX} \quad \text{iff } x = 0 \quad \text{since } y' = 8 = 1 / \sqrt{0}$$

If *x*, it is negative. Then it is merely negated to a positive value by always taking the square root of the absolute value. But if *x* is zero, the same rule as for a reciprocal applies.

Now that floating-point values have been examined, we can move on to comparisons of floating-point values.

Floating-Point Comparison

Do not expect the resulting values from different calculations to be identical. For example, 2.0 x 9.0 is about 18.0, and 180.0 / 10.0 is about 18.0. But the two 18.0 values are not guaranteed to be identical.

Let's examine a range of values 10^n and compare a displacement of ±0.001 versus ±0.0000001.

Table D-2

Base Number	–0.001	+0.0	+0.001
1.0	0x3F7FBE77	0x3F800000	0x3F8020C5
10.0	0x411FFBE7	0x41200000	0x41200419
100.0	0x42C7FF7D	0x42C80000	0x42C80083
1000.0	0x4479FFF0	0x447A0000	0x447A0010
10000.0	0x461C3FFF	0x461C4000	0x461C4001
100000.0	**0x47C35000**	**0x47C35000**	**0x47C35000**
1000000.0	**0x49742400**	**0x49742400**	**0x49742400**
10000000.0	**0x4B189680**	**0x4B189680**	**0x4B189680**
100000000.0	**0x4CBEBC20**	**0x4CBEBC20**	**0x4CBEBC20**

Note the single-precision loss between the ±0.001 displacement as the number of digits goes up in the base number. As the base number gets larger, fewer decimal places of precision can be supported. The hexadecimal numbers in bold are where the precision was totally lost!

Table D-3

Base Number	–0.0000001	+0.0	+0.0000001
1.0	0x3F7FFFFE	0x3F800000	0x3F800001
10.0	**0x41200000**	**0x41200000**	**0x41200000**
100.0	**0x42C80000**	**0x42C80000**	**0x42C80000**
1000.0	**0x447A0000**	**0x447A0000**	**0x447A0000**
10000.0	**0x461C4000**	**0x461C4000**	**0x461C4000**
100000.0	**0x47C35000**	**0x47C35000**	**0x47C35000**
1000000.0	**0x49742400**	**0x49742400**	**0x49742400**
10000000.0	**0x4B189680**	**0x4B189680**	**0x4B189680**
100000000.0	**0x4CBEBC20**	**0x4CBEBC20**	**0x4CBEBC20**

This is a similar single-precision table, except the displacement is between ±0.0000001. Note the larger number of hexadecimal numbers in bold indicating a loss of precision.

Okay, one more table for more clarity.

Table D-4

Base Number	+0.001	+0.002	+0.003
1.0	0x3F8020C5	0x3F804189	0x3F80624E
10.0	0x41200419	0x41200831	0x41200C4A
100.0	0x42C80083	0x42C80106	0x42C80189
1000.0	0x447A0010	0x447A0021	0x447A0031
10000.0	0x461C4001	0x461C4002	0x461C4003
100000.0	0x47c35000	0x47c35000	0x47c35000
1000000.0	0x49742400	0x49742400	0x49742400

Note that the accuracy of the precision of the numbers diminishes as the number of digits increases!

This means that smaller numbers, such as those that are normalized $x \in [-1, 1]$ and have a numerical range from –1.0 to 1.0, allow for higher precision values. But those with larger values are inaccurate, thus, they are not very precise. For example, the distance between 1.001 and 1.002, 1.002 and 1.003, etc., is about 0x20c4 (8388). This means that about 8,387 numbers exist between those two samples. A number with a higher digit count, such as 1000.001 and 1000.002, supports about 0x11 (17), so only about 16 numbers exist between those two numbers. A number around 1000000 identifies 1000000.001 and 1000000.002 as the same number. This makes comparisons of floating-point numbers with nearly the same value very tricky. This is one of the reasons why floating-point numbers are not used for currency, as they tend to lose pennies. Binary Coded Decimal (BCD) and fixed-point (integer) are used instead.

So when working with normalized numbers {–1.0 … 1.0} in C/C++, a comparison algorithm with a precision slop factor (accuracy) of around 0.0000001 should be utilized. When working with estimated results, a much smaller value should be used. Normally, one would not compare two floating-point values except to see if one is greater than the other for purposes of clipping. *It is almost never a comparison for equality.*

With shaders, there is no slop factor adjustment, and so whenever possible, exact comparisons of $(a = b)$ and $(a \neq b)$ should be avoided.

This page intentionally left blank.

Glossary

— A number.

alpha channel — A field within an RGBW (red, green, blue, alpha) color value representing the level of opacity and/or transparency.

ALU — Algorithmic logic unit.

AoS — Array of structures {XYZW}[4]

bump map — Textures representing bump maps are used to add surface detail to an image to give it the illusion of depth. This is done by displacing the lighting level of pixels based upon the bump map texture.

compiler — A software tool that converts symbolic source code into object code.

coprocessor — A secondary processor that adds enhanced functionality to a primary processor.

CPU — Central processing unit.

culling — A process of reducing the number of polygons needed to be passed to the rendering engine.

delta frame — The compression information to alter the current frame to be similar to the next frame in an animated sequence.

diffuse reflection — A component of reflected light that is diffused in all directions.

floating-point — A number in which the decimal point is floating and thus can be in any position. It is typically stored in sign, exponent, and mantissa components.

fogging — The blending of color with an image to increasingly obscure distant objects.

FPU — Floating-point unit.

GPU — Graphics processor unit.

GRDB — Game relational database.

IDE — Integrated development environment.

little-endian — The byte ordering used by most modern computers. For purposes of this book, that would include the X86 and MIPS processor. Although a MIPS processor can be configured for big-endian, for game consoles it is used in a little-endian configuration.

	0	1	2	3
0x1A2B3C4D	4D	3C	2B	1A

LOD — Level of detail.

LSB — Least significant bit. The related bit depends upon the endian orientation.

mipmap — A set of indexed bitmaps of which each n+1 bitmap is typically 2^n–1 smaller width by height than the previous one.

MSB — Most significant bit. The related bit depends upon the endian orientation.

normal vector — A vector that is perpendicular to a face and typically represents the visible side of a face.

Phong shading — A process by which the color of the vertex normals is interpolated across the face.

polygon — In the context of 3D rendering, a graphical primitive within a closed plane consisting of a three-sided (triangle) or four-sided (quadrilateral) shape representing a face typically covered by a texture.

RGB — Red-green-blue

scalar processor — A processor that can perform only one instruction on one data element at a time. See vector processor.

SIMD — Single instruction multiple data. A computer instruction that performs the same instruction in parallel for a series of isolated packed data blocks.

SoA — Structure of arrays {X[4], Y[4], Z[4], W[4]}.

specular reflection — A component of reflected light at a point on a surface regulated by the direction of the incidental light source in conjunction with the viewing angle in relation to the normal of the surface.

texture — A 2D image that is mapped upon a 3D wireframe polygon to represent its surface.

vector — (1) A pointer to code or data typically used in a table (vector table). (2) A one-dimensional array. (3) A line defined by starting and ending points.

vector processor — A processor that performs an instruction on an entire array of data in a single step. See scalar processor.

vertex — The intersection of two vectors used to define a corner of a polygon. Example: three corners of a triangle, eight corners of a cube.

vertex normal — A direction vector perpendicular to the plane intersecting the three vertices of a triangle.

w-buffer — A rectangular representation of the image buffer used to store the distance of each pixel of the image from the camera. The range of possible z values is linearly distributed between the camera and a point in 3D space depicted as infinity. The distances from the camera are finer in resolution than those closer to infinity, allowing for a more refined depth of view.

z-buffer — A rectangular representation of the image buffer used to store the distance of each pixel of the image from the camera. The range of possible z values is uniformly distributed between the camera and a point in 3D space depicted as infinity.

Some algebraic laws used in this book:

Additive identity	$n + 0 = 0 + n = n$
Multiplicative identity	$n1 = 1n = n$

Additive inverse	$a - b = a + (-b)$

Commutative law of addition	$a + b = b + a$
Commutative law of multiplication	$ab = ba$

Distributive	$a(b+c) = ab+ac$	$(b+c)/a = b/a + c/a$

This page intentionally left blank.

References

Color space tool:
http://www.couleur.org

Intel:
http://www.intel.com

RGB cube tool:
http://www.couleur.org

Game Development Links

Gamasutra — game developers:
http://www.gamasutra.com

Game Developer Magazine:
http://www.gdmag.com

Game Development Magazine:
http://www.gignews.com

Game Development search engine:
http://www.game-developer.com

GDC — Game Developers Conference:
http://www.gdconf.com

Online community for game developers of all levels:
http://www.gamedev.net

Vertex and Pixel Shaders

ATI:
http://www.ati.com or http://www.ati.com/developer

CgShaders.org:
www.cgshaders.org

Cg Sample Shaders:
http://developer.nvidia.com/object/cg_shaders.html

Cg and Programmable GPUs:
http://tim.plush.org/~grosa/cgtalk/shaders.php

Engel, Wolfgang F. (editor). *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks.* Wordware Publishing, Inc. www.shaderx.com, www.shaderx2.com

Engel, Wolfgang F. (editor). *ShaderX^2: Introductions and Tutorials with DirectX 9*. Wordware Publishing, Inc.

Engel, Wolfgang. "Advanced Shader Programming — Diffuse and Specular Lighting with Pixel Shaders." GameDev.net: http://www.gamedev.net/columns/hardcore/dxshader5/

Engel, Wolfgang. "Shader Programming — Part 1: Fundamentals of Vertex Shaders." GameDev.net: http://www.gamedev.net/columns/hardcore/dxshader1/

Engel, Wolfgang. "Shader Programming — Part 2: Programming Vertex Shaders." GameDev.net: http://www.gamedev.net/columns/hardcore/dxshader2/

Engel, Wolfgang. "Shader Programming — Part 3: Fundamentals of Pixel Shaders." GameDev.net: http://www.gamedev.net/columns/hardcore/dxshader3/

Engel, Wolfgang. "Shader Programming — Part 4: Programming Pixel Shaders." GameDev.net: http://www.gamedev.net/columns/hardcore/dxshader4/

Fosner, Ron. *Real-Time Shader Programming — Covering DirectX 9*. Morgan Kaufmann Publishers.

Hallingstad, Arne Olav. "Introduction on Vertex and Pixel Shaders." http://www.geocities.com/arne9184/articlestutorials/Introduction_on_vertex_and_pixel_shaders.htm

Leiterman, James. *Vector Game Math Processors*. Wordware Publishing, Inc.

nVidia: http://www.nvidia.com/developer

Penfold, Tom. "Tom's Hardware Guide — Vertex Shaders and Pixel Shaders." http://www.tomshardware.com/graphic/20020116/

Rage 3D: www.rage3d.com or http://www.directx.com/shader/vertex/instructions.htm#defb

RmanNotes — Writing RenderMan shaders: http://www.accad.ohio-state.edu/~smay/RManNotes/WritingShaders/intro.html

SHADERS — An example of light distribution methods: http://www.cinegrfx.com/newpages/tour/prman-opbox-descriptions/Shaders.html

Synthesizing Patches Using Vertex Shaders: http://www.mvps.org/directx/articles/shadeland/

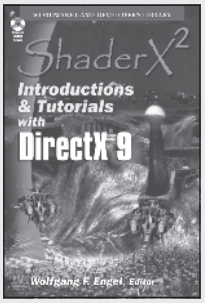
This page intentionally left blank.

Index

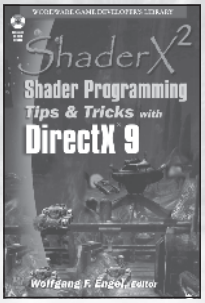
Looking

Check out Wordware's market-featuring the following new

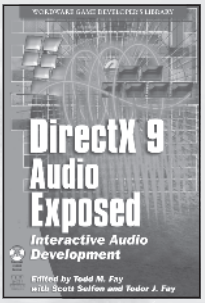
DirectX Game Development



ShaderX²: Introductions & Tutorials with DirectX 9
1-55622-902-X • $44.95
6 x 9 • 384 pp.

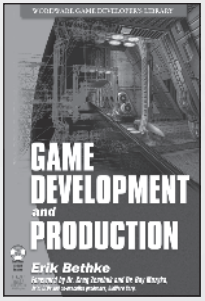


ShaderX²: Shader Programming Tips & Tricks with DirectX 9
1-55622-988-7 • $59.95
6 x 9 • 728 pp.

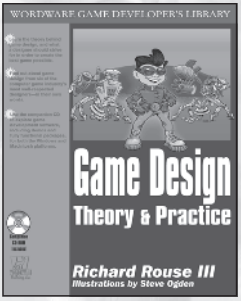


DirectX 9 Audio Exposed: Interactive Audio Development
1-55622-288-2 • $59.95
6 x 9 • 568 pp.

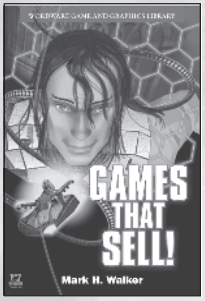
Game Development



Game Development and Production
1-55622-951-8 • $49.95
6 x 9 • 432 pp.

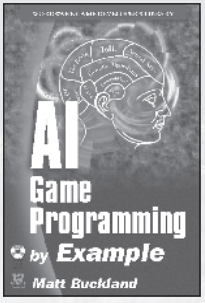


Game Design: Theory and Practice
1-55622-735-3 • $49.95
7½ x 9¼ • 608 pp.



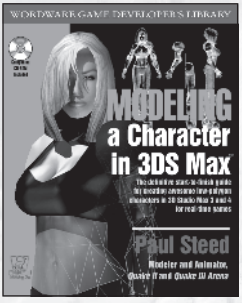
Games That Sell!
1-55622-950-X • $34.95
6 x 9 • 336 pp.

Coming Soon

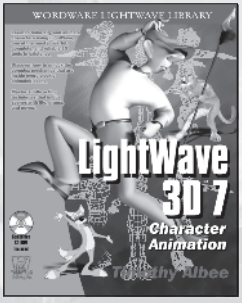


AI Game Programming by Example
1-55622-078-2 • $59.95
6 x 9 • 500 pp.

Game Graphics



Modeling a Character in 3DS Max
1-55622-815-5 • $44.95
7½ x 9¼ • 544 pp.

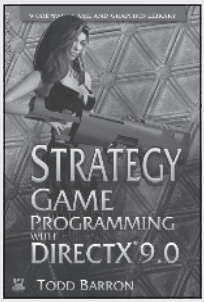


LightWave 3D 7 Character Animation
1-55622-901-1 • $49.95
7½ x 9¼ • 360 pp.

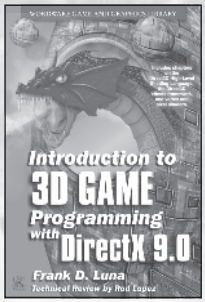
Visit us online at www.wordware.com for more information.

for more?

leading Game Developer's Library releases and backlist titles.



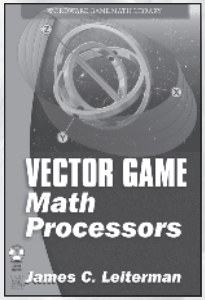
Strategy Game Programming with DirectX 9.0
1-55622-922-4 • $59.95
6 x 9 • 560 pp.



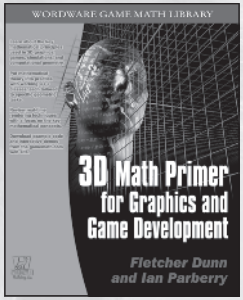
Introduction to 3D Game Programming with DirectX 9.0
1-55622-913-5 • $49.95
6 x 9 • 424 pp.

Advanced 3D Game Programming with DirectX 9.0
1-55622-968-2 • $59.95
6 x 9 • 552 pp.

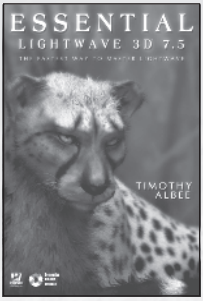
Java 1.4 Game Programming
1-55622-963-1 • $59.95
6 x 9 • 672 pp.



Vector Game Math Processors
1-55622-921-6 • $59.95
6 x 9 • 528 pp.



3D Math Primer for Graphics and Game Development
1-55622-911-9 • $49.95
7½ x 9¼ • 448 pp.



Essential LightWave 3D 7.5
1-55622-226-2 • $44.95
6 x 9 • 424 pp.

LightWave 3D 7.5 Lighting
1-55622-354-4 • $69.95
6 x 9 • 496 pp.

Named Best Production Utility by Game Developer magazine, FileMaker Pro continues to set the standard for databases. Be sure to check out Wordware's Developer Library for FileMaker.

Learn FileMaker Pro 6
1-55622-974-7 • $39.95
6 x 9 • 504 pp.

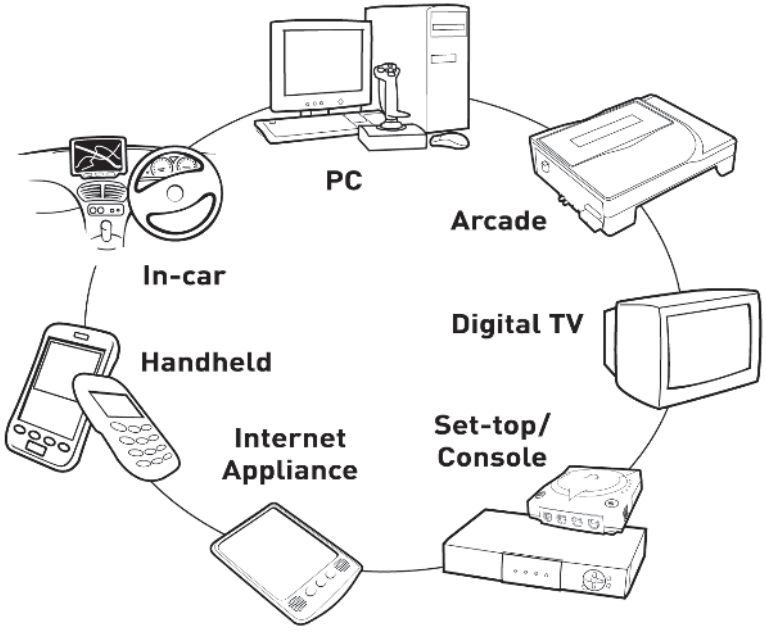
FileMaker Pro 6 Developer's Guide to XML/XSL
1-55622-043-X • $49.95
6 x 9 • 416 pp.

Advanced FileMaker Pro 6 Web Development
1-55622-860-0 • $59.95
6 x 9 • 464 pp.

Use the following coupon code for online specials: Shader2874

POWERVR

Visionary IP



www.powervr.com

www.pvrdev.com

Get **Deep** into 3D!

DEEP EXPLORATION.

Search, View, Translate, Animate, Render and Publish!

Deep Exploration™ provides easy navigation tools that let you search and view your 2D graphics and 3D models. It gives artists everywhere a production edge, including the ability to quickly translate 2D and 3D file formats with animation included.
Deep Exploration creates high quality fast renderings of 3D objects and scenes for use in many graphic applications. It is also a powerful web publication tool, ideal for creating interactive 3D content for web-based presentations.

DEEP PAINT 3D.

3D Painting and Texturing

Deep Paint 3D® provides artists with an intuitive, easy to use tool to paint and texture 3D models interactively in 3D! It uses textures or natural media paints which can be brushed directly or projected onto 3D models and scenes. This creative environment supports an integrated workflow with 3ds max®, Maya®, SoftImage® and LightWave 3D®. Deep Paint 3D comes complete with a bi-directional interface to Photoshop® and special support for the Wacom® Intuos™ pressure sensitive tablet.

DEEP UV.

Ultimate UV Mapping

Deep UV™ is a superior set of tools for the creation and modification of UV mapping for polygonal and sub-division surface models within an interactive 2D and 3D UV mapping environment. Deep UV includes fast and easy to use tools such as soft selection, relax and 3D UV selection. Whether you're a beginner or a professional 3D artist, you'll have the most efficient UV mapping possible with both automatic and advanced features. Deep UV can be used standalone, or with Deep Paint 3D.

RIGHT HEMISPHERE.

www.righthemisphere.com