



Real-Time Shader Programming

By Ron Foster?

ISBN:1558608532

[Morgan Kaufmann Publishers](#) ?2003 (406 pages)

An indispensable reference for the game developer, graphics programmer, game artist, or visualization programmer, to create countless real-time 3D effects.

[Companion Web Site](#) ?

▼[Table of Contents](#) ►[Back Cover](#) ►[Comments](#)

Table of Contents

[Real-Time Shader Programming—Covering DirectX 9.0](#)

[Preface](#)

[Chapter 1](#) - Introduction

[Chapter 2](#) - Preliminary Math

[Chapter 3](#) - Mathematics of Lighting and Shading

[Chapter 4](#) - Introduction to Shaders

[Chapter 5](#) - Shader Setup in DirectX

[Chapter 6](#) - Shader Tools and Resources

[Chapter 7](#) - Shader Buffet

[Chapter 8](#) - Shader Reference

[Part I](#) - Vertex Shader Reference

[Part II](#) - Pixel Shader Reference

[References](#)

[About the CD-Rom](#)

[Index](#)

[List of Figures](#)

[List of Tables](#)

[List of Sidebars](#)

Real-Time Shader Programming— Covering DirectX 9.0

Ron Fosner

MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER SCIENCE

Publishing Director: Diane D. Cerra

Publishing Services Manager: Edward Wade

Project Management/Proofreading: Yonie Overton

Editorial Coordinator: Mona Buehler

Cover Design: Ross Carron Design

Text Design: Frances Baca Design

Composition: Proctor-Willenbacher

Technical Illustration: Dartmouth Publishing, Inc.

Copyeditor: Robert Fiske

Indexer: Richard Shrout

Interior Printer: Quebecor World Book Services

Cover Printer: Phoenix Color Corporation

About the cover: The cover image, "Pipe Dream" Demo, is taken from a real-time DirectX 9 Radeon 9700 version of "Pipe Dream" from Animusic's DVD entitled, *ANIMUSIC: A Computer Animation Video Album* (<http://www.animusic.com>). This was shown in offline-rendered form in the Electronic Theatre at Siggraph 2001. All of the animation is data-driven from the original data from the Animusic demo. However, the motion blur is done using shaders to dynamically alter the shape and lighting of the balls (which are really a cylinder covered by two hemispheres) and the strings (which have a static string shape and a plucked vibrating shape). For example, a ball's shape was distorted in the vertex shader by stretching the length of the cylinder's axis from the ball's apparent velocity. The velocity was used to calculate a vertex blurriness factor in the vertex shader, which was then passed to the pixel shader, where the blurriness was used to spread out the specular highlights and make the highlight spread in the direction of travel. The full details can be found on ATI's website. Thanks to David Gosselin of ATI Research for being a good sport about providing me images on short notice.

Figure credits: (1) Figure 3–39 from "Digital Facial Engraving" by Victor Ostromoukhov, *Proceedings of SIGGRAPH '99*, page 421. © 1999 Association for Computing Machinery. Reprinted with permission. (2) Figure 3–40 from "Real-Time Hatching" by Emil Praun, Hugues Hoppe, Matthew Webb, and Adam Finkelstein, *Proceedings of SIGGRAPH '01*, page 583. © 2001 Association for Computing Machinery. Reprinted with permission. (3) Figure 3–42 from "A Non-Photorealistic Lighting Model for Automatic Technical Illustration" by Amy Gooch, Bruce Gooch, Peter Shirley, and Elaine Cohen, *Proceedings of SIGGRAPH '98*, page 452. © 1998 Association for Computing Machinery. Reprinted with permission. (4) Figure 3–43 from Jet Set Radio Future. © 2002 Sega Enterprises, Inc. All rights reserved.

Designations used by companies to distinguish their products are often claimed as trademarks or registered trademarks. In all instances in which Morgan Kaufmann Publishers is aware of a claim, the product names appear in initial capital or all capital letters. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

Morgan Kaufmann Publishers
An Imprint of Elsevier Science
340 Pine Street, Sixth Floor
San Francisco, CA 94104–3205
<http://www.mkp.com>

Copyright © 2003 Elsevier Science (USA)

All rights reserved.

07 06 05 04 03 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means—electronic, mechanical, photocopying, or otherwise—without the prior written permission of the publisher.

Library of Congress Control Number: 2002112069
1-55860-853-2

*To my three exciting women—
Sue, Rachael, and Olivia.
Thank you.*

About the Author

Ron Fosner has worked as a graphics programmer since the mid-1980s, first doing rudimentary 3D graphics in assembly language, then learning OpenGL when it became available on Windows systems. The lack of any good information on programming OpenGL on Windows led him to write the first widely successful introductory book on OpenGL programming, *OpenGL Programming for Windows 95 and Windows NT*. In addition to writing books, Ron has published numerous articles on 3D graphics and code optimization and is a frequent lecturer at the Windows Developer (WinDev) and Game Developers (GDC) conferences.

In the mid-1990s, the leading edge of real-time 3D graphics programming shifted from data visualization to game programming and so did Ron, forming a graphics programming company. Since that time, Ron has programmed four rendering engines, one facial animation engine, a stock market visualization tool, and a video editing tool. Recently, he has been involved in programming internet tools for performance and unit testing. In addition to programming, Ron tries to spread his knowledge on graphics and programming by writing articles for numerous magazines, including *Microsoft Systems Journal*, *Dr.*

Dobb's, *Game Developer Magazine*, and *Gamasutra*, as well as articles for the Microsoft Developer Network CDs.

Preface

One of the greatest things about being in the computer graphics industry is the sheer helpfulness of people. Almost without fail people are willing to help each other out, overcome difficulties, and take the time to share experiences, successes, and failures.

I got the idea for this book when I was chatting with Phil Taylor,^[1] the product manager of the DirectX SDK team at Microsoft. He was his usual ebullient self, going on about the new "shader" stuff that was coming out with the next version of DirectX. He was also lamenting about how tough it was to get people to understand what the advantage of shaders was, particularly when using textures as data. At one point, he said something like, "I mean, it's not an image, it's just a matrix of four element values—it could hold *anything*." It was at that point that I decided he was on to something. Only a small minority of graphics programmers have actually done tricks with data masquerading as textures, and fewer possess a solid understanding of how specifying a light, a material, and a triangle gets turned into a rendered object. Shaders let you do that and more, but in order to use them to good effect, you have to understand the power they place in your hands. To do that, you need knowledge.

My original vision for this book was along the lines of "a little book of shaders," but it didn't turn out that way. The further I got involved in the writing, the more I discovered that you did need extensive knowledge of not only the commands that were available to you and what they did, but how to assemble them to suit your needs. I knew that I'd have to include a little section to discuss the math, but when I started writing about the math involved with shading and illumination, with materials and lights, I knew I had to back up and start from the beginning.

My goal is to provide you with the foundations that you need to write your own shaders. There's not only a lot of theoretical stuff out there, but there's a golden opportunity to do your own thing and create your own custom style. My goal is to give you enough knowledge so that you can feel comfortable writing your own shaders. You should be able to feel free to go beyond the bounds of creating diffuse and specular lighting, and write your own with three kinds of diffuse and five kinds of specular and an emissive term based upon the proximity of the object to a power source. *That* is exactly what writing a shader is all about. There's no mystery to it. There are a few conventions that you should follow, but the one thing that I hope you'll learn is that what's "accepted," is just someone else's hack that got perpetuated.

There's a story that Mike Abrash related once while he was working on Quake II. He and John Carmak had gone to a convention and saw a competitor's game. The game had dynamic lighting effects—something they hadn't thought necessary to implement at the time. Seeing it in action, however, made

them change their minds. So it got put on Mike's "to-do" list. A few weeks later, John asked if Mike had gotten around to putting it in yet. "Nope. I bet I can do it in an hour," says John, and he goes away. (John is known for some ferocious coding habits.) According to Mike, 54 minutes later, John says he's done. He brings up the game, and fires a rocket down a long corridor. The corridor illuminates and it follows the rocket down the length, and it looks great! John had taken the rocket's location, cast rays up, down, left, and right, and pasted a spotlight texture at the intersection of the ray with any wall it found. It wasn't realistic, but in the short time you got to see it, it was enough to give you the illusion of real lighting. It was a hack, it looked good, and they shipped it. The moral of graphics programming is once it looks good enough, it's good enough.

As my deadline for this book approaches, there's been no small upheaval in the shader world. Higher level shading languages came too late for me to write about with any depth, so I focused on the low-level language and how to construct shaders. In the last weeks before I finished writing, there's not one but three competing high-level shader languages. It's still very much up in the air as to what the standard language will look like, and there's considerable pressure building to make the language independent of the hardware—that is, there'll be a software fallback—so that any shader program will run on any modern hardware. We'll be free from "checking the caps bits" to see if we can use our favorite shader. But now's not the time to wait! No matter what language you learn to write shaders in, it's the experience and understanding from writing shaders that will carry over into any shading language of the future. So let's get started!

^[1]My first contact with Phil was when someone was (I thought) heckling me during an OpenGL course at the Computer Game Developers Conference. Even in those early days of the API wars, Phil was one of the few with hearty integrity.

Chapter 1: Introduction

OVERVIEW

This is the true joy in life, being used for a purpose recognized by yourself as a mighty one. Being a force of nature instead of a feverish, selfish little clod of ailments and grievances complaining that the world will not devote itself to making you happy. I am of the opinion that my life belongs to the whole community and as I live it is my privilege—my *privilege*—to do for it whatever I can. I want to be thoroughly used up when I die, for the harder I work the more I love. I rejoice in life for its own sake. Life is no brief candle to me; it is a sort of splendid torch which I've got a hold of for the moment and I want to make it burn as brightly as possible before handing it on to future generations.

--George Bernard Shaw

Shader: A custom shading and lighting procedure that allows the motivated artist or programmer to specify the rendering of a vertex or pixel.

"Shader" comes from Pixar's RenderMan, which is a program that takes an entire description of a scene, from camera positions through object geometry, to a final rendering. RenderMan was introduced in 1989, but it wasn't really until the 1995 release of the movie *Toy Story* that the general public was introduced to the power of RenderMan. About this same time, there was a revolution taking place on the graphics boards of PCs; the boards were evolving at a faster and faster clip, and the features that were showing up on "commodity" boards were rivaling those previously found only on workstations.

As Pixar continued to make hit after hit using RenderMan, soon other movie studios joined in. Meanwhile, the PC games community was finding new uses for the powerful graphics cards with which new PCs were now equipped. Light maps in particular were soon finding their way into games, followed by bump maps and procedural vertex generation. In fact, it was the games community that soon started clamoring for more features, and in order to differentiate themselves from the pack, some graphics card vendors heeded this call and soon started layering more and more features onto their cards. This had a snow-ball effect of creating a larger and larger installed base of fairly sophisticated PCs that had a good selection of graphics features.

The latter part of the century also saw the graphics API war—OpenGL (the "established" standard) vs. Direct3D (the "upstart" API from Microsoft). While two fractious camps sided off leaving many agnostics in the middle, it soon became clear that the committee-based architecture board that governs OpenGL's evolution was too mired in the plodding workstation world, whereas the nimble and ferocious commodity PC graphics board industry was continually trying to outdo each other with every release, which was better suited to Direct3D's sometimes annoying yearly release cycle.

With only a few graphics boards on the market, it was easy to program to OpenGL and ignore the many growing pains of Direct3D, but 1999 saw some accelerated graphics hardware dedicated *only* to games start to show up, and by 2001 that was the norm. By this time, the games development community had discovered the traditional graphics researchers (the "Siggraph crowd") and vice-versa. The game developers discovered that many of the problems they faced had already been solved in the 1970s and 1980s as workstations were evolving and the graphics researchers liked the features that could be found on \$100 video cards.

Thus we come into the era of the shading language. Both OpenGL and Direct3D use the "traditional" lighting equations, and these have served quite well for many years. But game programmers are continually striving to create the most gorgeous "eye candy" that they can (ignoring the actual game play for the moment). The traditional lighting equations are fine for common things, but if you are trying to create some stunning visual effects, Goraud-shaded objects are just not going to cut it. And here we turn full circle and return to *Toy Story*. For a number of years, it was bandied about in the graphics community that we were approaching *Toy Story* levels of *real-time* animation. Now *Toy Story* was emphatically *not* rendered in real time. Many months running on 117 Sun Sparc Stations grinding out over 130,000 frames of animation is what rendered *Toy Story* (plus 1300 unique shaders!). But both the

computer's CPU and the graphics card's GPU (geometry processing unit) were getting continually more powerful. GPUs were becoming capable of offloading many of the things traditionally done by the CPU.

The first thing that GPUs took over from the CPU was some of the simple, repetitive math that the CPU used to have to do, the object transformation and lighting, or TnL [Fosner 2000]. This was a simple change, and in many cases, applications that used the graphics API for transformation and lighting saw a speedup with no code changes. On the other hand, there were many games—or games programmers who decided that they could do a better job than the API and did it themselves—and these games got no benefit from better hardware. But by this time, the writing was on the wall. No matter how good a programmer you were, your software implementation would never run as fast as the generic all-purpose hardware implementation.

However, there was one mantra that graphics hardware developers were continually hearing from graphics software developers, and that was they wanted a bigger choice in deciding how pixels got rendered. Of course, it still had to be hardware accelerated, but *they* wanted a chance to put their own interpretation on the rendered object. In fact, they wanted something like RenderMan shaders.

RENDERMAN VS. REAL TIME

RenderMan is a continually evolving product, but it's never been one that could be considered real time. A good reason for this is the flexibility needed by the artists. In 1989, there was no hardware that could render a complex ray-traced scene in real time. You can think of ray tracing as tracing each pixel through a scene—through transparent and translucent objects—back to the location where the pixel is saturated. This works nicely for scenes that have lots of complex behavior, such as scenes with reflecting surfaces, water, shadows, glass, smoke, etc. And while that is the direction 3D consumer graphics is heading, we're not quite there yet.

In the consumer 3D graphics arena, we're dealing with objects that reflect light, and we can deal with objects that are some degree of transparent. But there's no support for the effect of other objects in a scene. There's support for the concept of "lights," but a "light" in OpenGL or Direct3D is just a source of light, not an object itself. An object only "knows" about the lights in a scene, but not if it's occluded from these lights by anything else in a scene.

WHAT YOU'LL LEARN FROM THIS BOOK

This book is intended for someone familiar with 3D graphics who wants a better understanding about just how some math can turn a bunch of data into a photorealistic scene. This book not only covers the basics of setting up and using vertex and pixel shaders, it also discusses in depth the very equations (and simplifications) that were used in the default graphics pipeline. You'll learn what the basic equations are for lighting and shading, what the standards are, and most important, how to toss them out and program your own.

The emphasis is on giving you the knowledge to understand the equations that are used in lighting and shading, what the alternatives are, and how you can write your own. The equations are discussed in detail, and a shader editing tool from ATI called RenderMonkey is included. RenderMonkey is a powerful tool with an easy to use interface with which you can dynamically edit vertex and pixel shaders and their parameters in real time and see the changes upon a variety of objects. Not only will you understand the differences between the standard Blinn-Phong lighting equations and others like Torrance-Sparrow and Oren-Nayar, but you'll learn how to make real-time shaders that implement these models. Anisotropic lighting, cartoon shading, Fresnel effects; it's all covered in this book.

Chapter 2: Preliminary Math

OVERVIEW

I'm very well acquainted too with matters mathematical, I understand equations, both the simple and quadratical. About binomial theorem I'm teeming with a lot of news—With many cheerful facts about the square of the hypotenuse.

--Gilbert and Sullivan, The Pirates of Penzance

Well, there's no getting around it. If you want to *really* understand shader programming and be able to program some wicked shaders, you have to have a firm understanding of the basic math required for 3D graphics and how it ties together. Understanding the math is a prerequisite for setting up shaders. If you're a programmer, you really should have a good grasp on the mathematics involved in taking a vertex from a file and getting it rendered as pixels on the screen. If you're an artist, you might be able to slide on some of the math, but to fully use a shading language requires knowing what data is coming through in a particular coordinate system and knowing how to change it as needed. Expertise lies in knowing how to shift data from one range to another, how to maintain as much precision as possible, and how to spoof the system to do what you want it to do.

If you've already programmed your own 3D rendering engine, have written a tri-stripping routine, know what a projection matrix is, and understand the usefulness of negative color values, then you can probably skip this section. However, if you feel that a quick and painless refresher on the basics of useful 3D math, terminology, and the mathematics of color would be useful, read on!

CONVENTIONS AND NOTATION

First, let's agree to some notation since in latter parts of the book I'll be getting into some pretty heavy notation. Generally, any single-valued quantity is written in a normal, unbolded font, whereas any multivalued quantity like a vector or a matrix is written in bold font. It gets a little confusing since one of the things a shader is generally trying to compute is color and, in current graphics hardware, that means a three- or four-element value representing the rgb or rgba values that the graphics hardware uses to represent a color. Traditionally, this color value is treated as a vector value, but it's manipulated as individual elements ([Table 2.1](#)).

Table 2.1: OPERAND NOTATION

TYPE	EXAMPLE	DESCRIPTION
Scalars	<i>a b</i>	A single floating point number. Represented by a lowercase italicized letter.
Vectors	a _s v	A three- or four-element array of floating point numbers representing a direction. Represented by a lowercase bold letter. A subscript indicates an individual element.
Unit vector	<i>â</i> <i>ĉ</i> <i>ê</i> <i>ĝ</i>	A vector with a little hat character (a "circumflex") represents a normalized or unit vector. A unit vector is just a vector that has a length of one.
Points	p x	A three- or four-element array of floating point numbers representing a position. Represented by a lowercase bold letter. A subscript indicates an individual element.
Matrices	M T	A three- or four-element array of floating point numbers representing a position. Represented by an uppercase bold letter.

In addition to notation for the various data types we will be manipulating, we'll also need notation for the various types of mathematical operations we'll be performing on the data ([Table 2.2](#)).

Table 2.2: OPERATOR NOTATION

TYPE	EXAMPLE	DESCRIPTION
Addition	$a + b$	Addition or subtraction between similar operand types.
Subtraction	$c - d$	
Multiplication	MA pv	There is no explicit operator typically used in this text for multiplication between operands that can be multiplied together. The absence of an operator indicates implied multiplication.
Division	$1/2$ a/b	Division is represented by either a stroke or a dividing line.
Dot product	$a \cdot b$	Also called the inner product. Represented by the \cdot symbol.

Table 2.2: OPERATOR NOTATION

TYPE	EXAMPLE	DESCRIPTION
Cross product	$b \times a$	Represented by the \times symbol.
Absolute value or magnitude	$ a $	The absolute value is the positive value of a scalar quantity. For a vector, it's the length of the vector. In either case, a value divided by its absolute value is ± 1 .
Piecewise multiplication	$i_d \otimes c_s$	Element-by-element multiplication. Used in color operations, where the vector just represents a convenient notation for an array of scalars that are operated on simultaneously but independently.
Piecewise addition	$i_d \oplus c_s$	Element-by-element addition. Used in color operations, where the vector just represents a convenient notation for an array of scalars that are operated on simultaneously but independently.

You'll also see text in a paragraph in a monospaced font. This is used to indicate particular aspects of shader programming. For example, you might need to check the capabilities of the device you're currently rendering by checking the D3DCAPS structure, or we might want to render a triangle list with a call to `DrawIndexedPrimitive()`.

Finally, you'll see colored blocks of code, either C or C++ or shader code:

```
// shader file
vs.1.0
//transform vertices by view/projection matrix

m4x4 oPos, v0, c0

mov r0, c5          // load color5
mov r1, c6          // load color6
dp3 r2, r0, r1      // combine using a dot product
mov oD0, r2         // output color
```

VERTICES

When you say "vertex," most people think that you're speaking of a location in 3-space—that is, an x, y, z triplet. In this book, that's called a *point*. A vertex in 3D graphics typically means *all* the properties that are used to describe that particular vertex—that is, all the information needed to describe a vertex so that it can be rendered. Of course, the most obvious one is its location—its "point." However, if I give you a vertex description consisting of just a position and tell you to draw it, what do you draw? No, it's obvious that you need more information in order to draw a vertex, be it the vertex color, material properties, texture coordinates, whatever. So in 3D graphics, a vertex is the vertex position and whatever other information is required to render that vertex. I should note that vertices themselves aren't rendered; they are typically grouped in threes and rendered as triangles—a triangle having enough information to describe a surface area.

This "other information" could be anything about that point that can be used to describe or calculate its final color value. In the simplest case, it's just the color information for that point (e.g., the point is red). Or the point may be part of a textured surface, in which case it might have texture coordinates instead of a color, in which case the color of the point is looked up from the texture. A vertex that's part of an illuminated surface (a "lit" surface in the vernacular) could have a set of material properties that are unique color values for diffuse, ambient, and specular properties plus the point's normal value. Thus when we talk about a vertex, we're not simply talking about the vertex's location, but also that we've specified enough information to draw it according to the situation we want to render it in, be that a textured surface, a lit surface, etc. In some parts of this text, I'll talk about a vertex and mean just the position part of the vertex; other times, I might be focused on the resultant color information, but the important thing is that a vertex is specified by more than just a point.

POINTS

A point is a location in space, typically described by an x, y, z location—in other words, a location coordinate. It has no other properties. It can be translated to different locations or translated by application of a transformation matrix. Points (or coordinates) in 3D graphics are usually represented by homogenous coordinates, which represent the point as a location in 4D space through the addition of the w coordinate. For our purposes, this is just a semantic nicety that is taken care of by the graphics API, so we usually don't have to worry about it. Points are represented by the same structure we use for vectors, and many of the same mathematical operators can be performed on each, but be aware that though they look the same, they are very different.

VECTORS

In the mathematical sense, a vector is an array of numbers. We're going to use them in the 3D graphics shorthand, where vector means a direction vector. In this section, I want to cover some basic mathematical properties of vectors. One of the reasons that vectors and points are important is that in order to understand how to program shaders, you'll need to understand the math that's required to program some of the basics of shader programming. In order to do that, you need to have a basic

understanding of vector and point math and to understand how these types of values are used in 3D graphics.

A vector is written a columnwise matrix, with elements x , y , and z .

$$a = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Although this is fine for performing element-by-element analyses of what's going on, it's a bit tedious to read and write, so we'll usually just represent a vector by a bold lowercase letter.

Since both points and vectors are represented by a three- or four-element array of floating point numbers, we'll occasionally use the same math for them—for example, multiplying a point or a vector by a transformation matrix will transform either one. The math is identical, but it's good to keep in mind that though they might look the same, and can usually be treated the same, they aren't strictly interchangeable.

Vector Magnitude and Unit Vectors

Not only do direction vectors contain a direction, but they also encode a length or magnitude, which, depending upon what your vector represents, you can think of as the force of the vector in that direction. To compute the length of a vector, you take the sum of the squares of the vector's elements, and then compute the square root of the sum. Thus, the length or magnitude, $|a|$, of our vector, a , would be computed as

$$|a| = \sqrt{x^2 + y^2 + z^2}$$

Another way of writing vector v is to break it into the magnitude (which is a scalar) and then the normalized or *unit* vector. A unit vector is a vector in which the magnitude of the vector is one. We'll write unit vectors in this book as a vector with a hat on it, for example, \hat{a} . A unit vector is computed (or *normalized*) from an unnormalized one by dividing each element of the vector by the vector's magnitude.

$$\hat{a} = \frac{a}{|a|} = \begin{bmatrix} x/|a| \\ y/|a| \\ z/|a| \end{bmatrix}$$

In most cases in 3D graphics, we just want a unit vector since we're usually interested in direction and not magnitude. Many of the lighting equations are simplified by using normal vectors.

Dot Product

The dot product is one of the more common things that you'll be using in shaders (hence the reason they are part of the shader language), so it's important to understand what they can tell us. I won't go into the derivation of the equations^[2], but it's very useful to know how the dot product can be used to derive information about the relationship between two vectors. Now, given that we have two vectors, we can examine what the dot product tells us about the relationship between these vectors. Note that although the vectors in the illustrations are shown having a shared origin, this really isn't a requirement since a vector is a *direction* that's unrelated to any origin. This way it's easier to visualize the angle between the vectors, θ . To calculate a dot product, first we need two vectors, **a** and **b** ([Figure 2.1](#)).

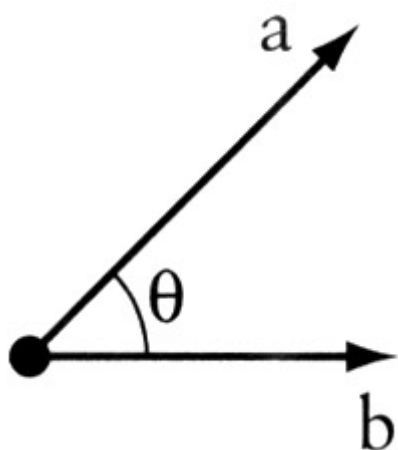


Figure 2.1: The dot product can be used to calculate the angle between two vectors.

For any two vectors, the dot product describes a function of the magnitudes of the vectors times the cosine of the angle between them.

$$\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos(\theta)$$

Thus the dot product (also called the *inner* product or the *scalar* product) gives us information about the angle between the two vectors and can be computed as the sum of the product of each element of the vectors. Note that the result from the dot product is a scalar value. Even more interesting is if $\hat{\mathbf{a}}$ and $\hat{\mathbf{b}}$ are unit vectors (which is typically the way we try to set things up for 3D graphics calculations), then the magnitudes of the vectors are one (and can be factored out), and the dot product gives us the cosine of the angle between the vectors directly.

$$\hat{\mathbf{a}} \bullet \hat{\mathbf{b}} = \cos(\theta)$$

You can also use trigonometry to show that the dot product can also be computed by computing the sum of the product of the individual elements of the vectors.

$$\mathbf{a} \cdot \mathbf{b} = \sum a_i b_i = x_a x_b + y_a y_b + z_a z_b$$

The preceding equation also illustrates that the dot product is order independent, or commutative, $\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$. Getting the angle between two vectors allows you to calculate some very important information. [Figure 2.2](#) shows you some of the information that the dot product tells you.

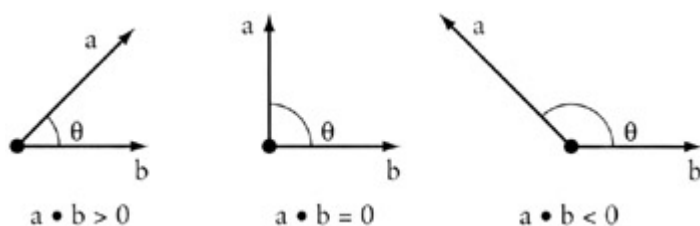


Figure 2.2: The dot product lets you determine the relationship of two vectors.

Thus you can glean information about how vectors relate to one another. For example,

- If the dot product is zero, then the vectors are perpendicular.
- If vector \mathbf{b} is the view direction, and vector \mathbf{a} is the surface normal for a triangle, and if the angle is greater than zero, then you know the triangle is facing away from you—thus you could cull it or apply your back face material to it.
- Another example would be if vector \mathbf{b} were again the view direction, with the view origin at point \mathbf{o} , and you have an object at point \mathbf{c} . You can easily create a direction vector from the view origin to the object (assuming they are in the same coordinate system) by $\mathbf{a} = \mathbf{o} - \mathbf{c}$. Thus if $\mathbf{a} \cdot \mathbf{b} < 0$, then the object is behind the viewpoint and you can cull it.
- The dot product is used repetitively in lighting calculations to compute the intensity of a light shining on a surface.

Cross Product

The dot product is useful because it tells us about the relationship between two angles, but it doesn't tell us anything about how those angles are oriented in 3-space. If we have two nonidentical vectors, they define a plane in 3-space, and it's sometimes useful to know something about that plane. This is where the cross product comes in. Unlike the dot product, which gives a scalar value as a result, the cross product gives a vector as its result—hence it's also called the *vector product*.

The cross product of two vectors results in a third vector.

$$\mathbf{c} = \mathbf{a} \times \mathbf{b}$$

And this third vector **c** has the property that it's perpendicular to both **a** and **b** ([Figure 2.3](#)).

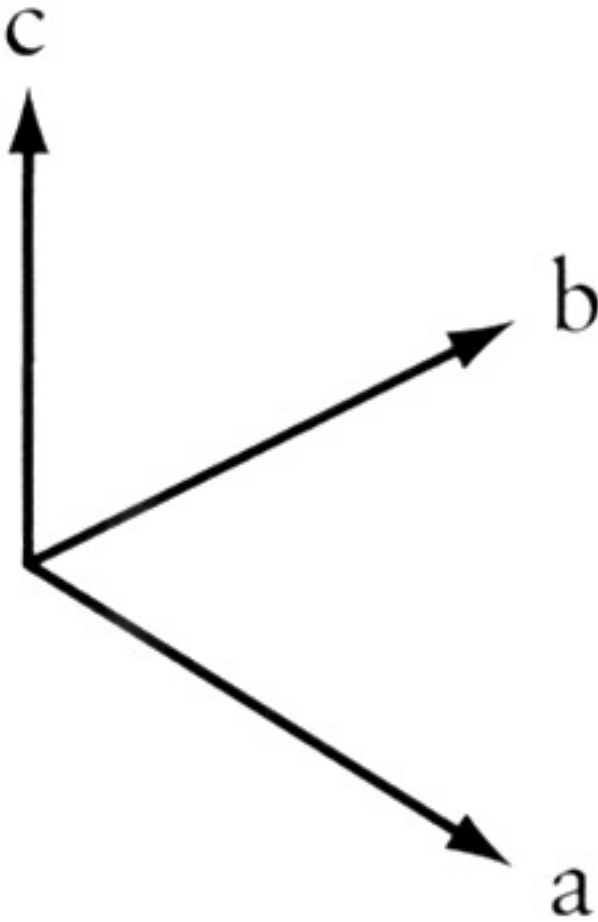


Figure 2.3: The cross-product of $\mathbf{a} \times \mathbf{b}$ is vector **c**, perpendicular to both.

This means that the dot products of vector **c** with the original two vectors are zero.

$$\begin{aligned} \text{If } c &= a \times b \\ \text{Then } 0 &= c \cdot b = c \cdot a \end{aligned}$$

When I first learned to use the cross product, I was taught in the determinant notation, which looks like this.

$$a \times b = \begin{vmatrix} x & y & z \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

This gives you another way of calculating the cross product by multiplying it out.

$$a \times b = a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x$$

The cross product is also antisymmetric, meaning that the direction of the result is order dependent. If you reverse the order of the cross product, you'll get two vectors that are equal in magnitude, but point in opposite directions.

This is another way of stating that the cross product follows the right-hand rule. In [Figure 2.4](#), we'd say that we're taking "**a** cross **b**." Thus if you take the fingers on your right hand and curl them from the first vector to the second—that is, sweep them from **a** toward **b** with your wrist at the junction—your thumb will point toward the **c** direction.

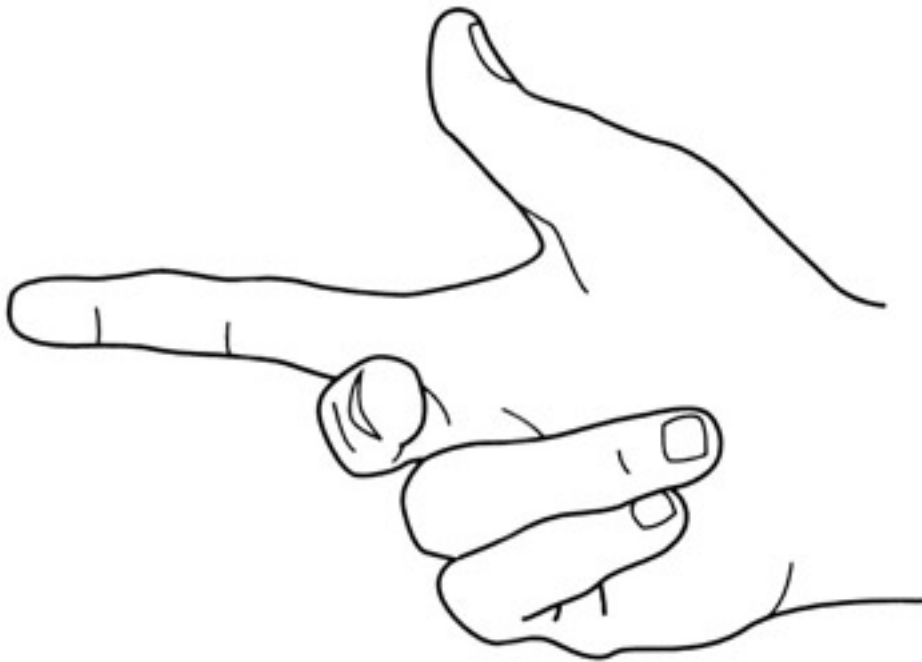


Figure 2.4: The right-hand rule.

Or in math notation

$$c = a \times b = -b \times a$$

The right-hand rule also means that you can generate the third vector given the other two.

$$\begin{array}{ll} \text{if} & a = b \times c \\ \text{then} & b = c \times a \\ \text{and} & c = a \times b \end{array}$$

If you take the length of the cross product of two vectors, you get the area of the parallelogram formed by those two vectors. Though we're usually interested in the area of the triangle, we can just take half that value to get the area of the triangle.

Thus if we have two vectors from point o on a triangle, we can compute the area of the parallelogram (the area in light blue in [Figure 2.5](#)) from the magnitude of the cross product of the vectors.

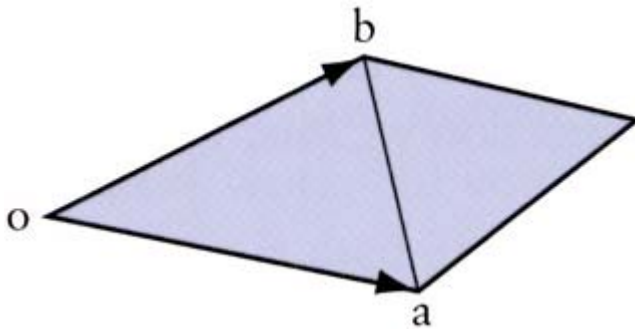


Figure 2.5: The cross product lets you determine the area of the parallelogram formed by two vectors.

$$\begin{aligned} \text{if } b &= [1,1,0] \text{ and } a = [2,0,0] \\ \text{then } b \times a &= [2,0,0] \end{aligned}$$

The area of the parallelogram is $|b \times a| = 2$. (The area of the triangle formed by oab is thus 1.) It's also possible to calculate the area of the parallelogram by using the lengths of the vectors and the sine of the angle between them to get the following relationship:

$$|a \times b| = |a||b|\sin(\theta)$$

which, if you square it and use the law of cosines, gives you what's called *Lagrange's identity*, which you'll note, now relates a cross product term to a dot product term.

$$|a \times b|^2 = |a|^2|b|^2 - (a \cdot b)^2$$

Other useful formulas are the vector triple product

$$(a \times b) \times c = b(a \cdot c) - a(b \cdot c)$$

and the scalar triple product, v ,

$$v = a \cdot (b \times c)$$

which gives you the area of the parallelepiped defined by vectors a , b , c ([Figure 2.6](#)). If $v = 0$, then this tells you that at least two of the three vectors lie in the same plane (assuming that none are zero). Note that the order is unimportant since the volume will remain the same no matter how you perform the multiplications.

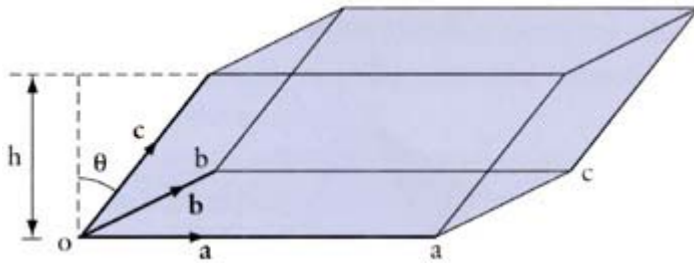


Figure 2.6: The scalar triple product lets you determine volumes.

For relating vectors using cross products, you have *Jacobi's identity*.

$$a \times (b \times c) = b \times (c \times a) = c \times (a \times b)$$

^[2]For a good explanation of the basics of the dot product, see [FARIN 1998], p. 27, or [HILL 1990] p. 152.

CREATING NORMALS OUT OF GEOMETRY

One practical aspect is to use the cross product to create normals for your objects if they don't have them. Suppose you have a triangle with three vertices: a , b , and c ([Figure 2.7](#)). You can create two direction vectors by calculating $a - b$, and $a - c$, and then calculate the normal vector, \mathbf{n} , by taking the cross product of the two vectors.

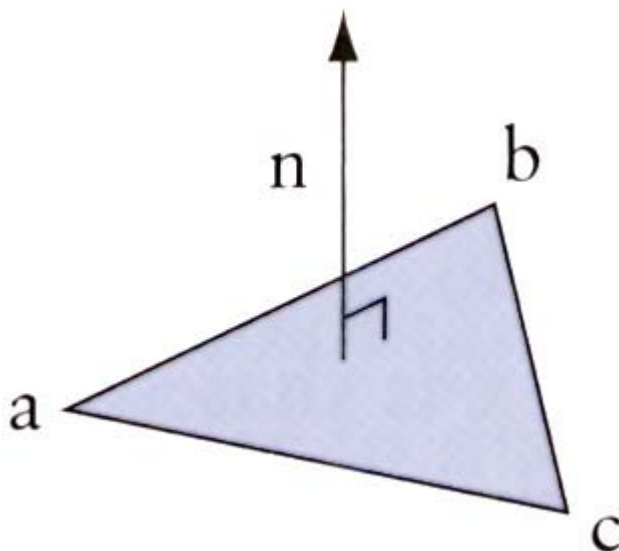


Figure 2.7: The normal \mathbf{n} of a triangle.

So, first we create two direction vectors, v_1 and v_2 , from the three points.

$$v_1 = a - b$$

$$v_2 = a - c$$

Then we create the normal from the cross product of these two vectors. Note that if we cross from **ac** to **ab**, we'll get the normal pointing out of the triangle, as shown in [Figure 2.7](#). If we did it the other way, we'd get the normal pointing out of the bottom of the triangle.

$$n = v_2 \times v_1$$

Finally, you'll probably want to normalize the normal vector.

Vertex Normals vs. Face Normals

When creating normal vectors from geometry, you frequently don't want the normal that we've calculated here, which is called a face normal (since it's the normal of the triangle's surface or the *face*). Instead, you want normals for the individual vertices. This is a bit more work, but you'd start with face normals, then find all the faces to which a vertex is used in, then average all the normals from those faces, possibly with some sort of weighting function thrown in. Once you get an averaged normal, you store that as the vertex's normal ([Figure 2.8](#)).

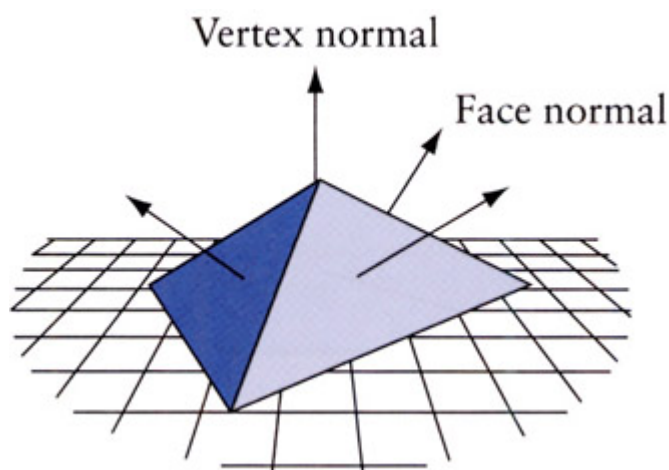


Figure 2.8: Vertex normals and face normals.

This is fine for models with smoothly varying surfaces, but for models with sharp angles, there's more work to do. Typically, there's some sort of crease angle cutoff, which means that if a vertex is shared between faces in which the angle between the faces is too large, such as a crease or a point (imagine the vertex at the corner of a cube), then the vertex needs to be assigned to one set of faces. Or, the vertex needs to be duplicated into two or more vertices that share the same position, but are assigned normals from one set of face averages. Then others use the other normals from the faces that were over the crease angle.

Another creative example of using a cross product is given by [VERTH 2001]. If you are an object traveling on a reasonably level **xy** plane in direction **d1** and you want to turn to direction **d2**, then you can examine the cross product's **z** value—if it's positive you'll turn left; if it's negative, you'll turn right.

Though you might not be using the cross product in a shader, you'll typically use it to calculate other vectors that are required at some point.

MATHEMATICS OF COLOR IN COMPUTER GRAPHICS

Now you might think it strange that I've added a section on the mathematics of color, but it's important to understand how color is represented in computer graphics so that you can manipulate it effectively. A color is usually represented in the graphics pipeline by a three-element vector representing the intensities of the red, green, and blue components, or for a more complex object, by a four-element vector containing an additional value called the *alpha* component that represents the opacity of the color. Thus we can talk about **rgb** or **rgba** colors and mean a color that's made up of either three or four elements. There are many different ways of representing the intensity of a particular color element, but shaders use floating point values in the range [0,1].

I should also point out that when dealing with colors, particularly with some of the subtleties that we'll be getting into with shaders, you should understand the gamut of the target device. This is the nasty edge where our beautiful clean mathematics meets the real world. The gamut of a device is simply the physical range of colors the device can display.

Typically, a high-quality display has a better gamut than a cheap one. A good printer has a gamut that's significantly different from a monitor's. One of the issues that I had to deal with in generating the images for this book was getting the printed materials looking like the displayed images generated by the shaders. If you're interested in getting some color images for printing, you'll have to do some manipulation on the color values to make the printed image look like the one your program generated on the screen. You should also be aware that there are color spaces other than the RGB color space shaders use. HSV (hue, saturation, and value) is one that's typically used by printers, for example.

WHY YOU MIGHT WANT 128-BIT COLOR

One of the gods of 3D graphics is a guy named Mike Abrash. He's the guy to blame for sending many of us on the road to 3D graphics as a career. In one of his early magazine articles [ABRASH 1992], he tells a story about going from a 256-color palette to hardware that supported 256 levels for each RGB color—16 million colors! What would we do with all those colors? He goes on to tell of a story by Sheldon Linker at the eighth Annual Computer Graphics Show on how the folks at the Jet Propulsion Lab back in the 1970s had a printer that could print over 50 million distinct colors. As a test, they printed out words on paper where the background color was only one color index from the word's color. To their surprise, it was easy to discern the words—the human eye is very sensitive to color graduations and edge detection. The JPL team then did the same tests on color monitors and discovered that only about 16 million colors could be distinguished. It seems that the eye is (not too surprisingly) better at perceiving detail from reflected light (such as from a printed page) than from emissive light (such as from a CRT). The moral is that the eye is a lot more perceptive than you

might think. Twentyfour bits of color really isn't that much range, particularly if you are performing multiple passes. Round-off error can and will show up if you aren't careful!

An example of the various gamuts is shown in [Figure 2.9](#). The CIE diagrams are the traditional way of displaying perceived color space, which, you should note, is very different from the linear color space used by today's graphics hardware. The colored area is the gamut of the human eye. The gamuts of printers and monitors are subsets of this gamut.

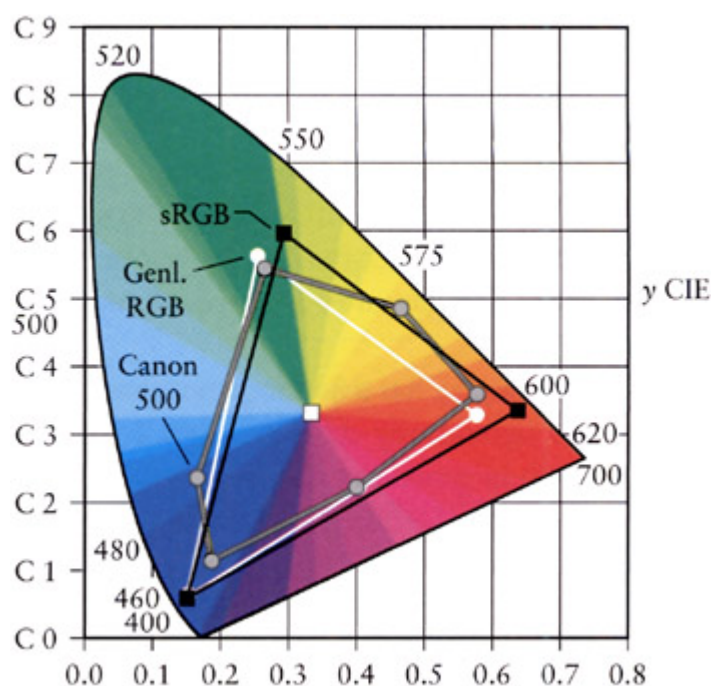


Figure 2.9: The 1931 CIE diagram shows the gamuts of the eye and the lesser gamuts of output devices.

Multiplying Color Values

Since shaders allow you to do your own color calculations, you need to be aware of how to treat colors. The calculation of the color of a particular pixel depends, for example, on the surface's material properties that you've programmed in, the color of the ambient light, the color of any light shining on the surface (perhaps of the angle of the light to the surface), the angle of the surface to the viewpoint, the color of any fog or other scattering material that's between the surface and the viewpoint, etc. No matter how you are calculating the color of the pixel, it all comes down to color calculations, at least on current hardware, on **rgb** or **rgba** vectors where the individual color elements are limited to the $[0,1]$ range. Operations on colors are done piecewise—that is, even though we represent colors as **rgb** vectors, they

aren't really vectors in the mathematical sense. Vector multiplication is different from the operation we perform to multiply colors. We'll use the \otimes symbol to indicate such piecewise multiplication.

Colors are multiplied to describe the interaction between a surface and a light source. The colors of each are multiplied together to estimate the reflected light color—this is the color of the light that this particular light reflects off this surface. The problem with the standard **rgb** model is just that we're simulating the entire visible spectrum by three colors with a limited range.

Let's start with a simple example of using reflected colors. In the section on lighting, we'll discover how to calculate the intensity of a light source, but for now, just assume that we've calculated the intensity of a light, and it's a value called i_d . This intensity of our light is represented by, say, a nice lime green color. Thus

$$\text{light color } i_d = [0.34765, 0.92578, 0.24609]$$

Let's say we shine this light on a nice magenta surface given by c_s .

$$\text{surface color } c_s = [0.86719, 0.00000, 0.98828]$$

So, to calculate the color contribution of this surface from this particular light, we perform a piecewise multiplication of the color values.

$$\begin{aligned} i_d \otimes c_s &= [0.34765, 0.92578, 0.24609] \otimes [0.86719, 0.00000, 0.98828] \\ &= [(0.34765)(0.86719), (0.92578)(0), (0.24609)(0.98828)] \\ &= [0.30148, 0.00000, 0.243210] \end{aligned}$$

This gives us the dark plum color shown in [Figure 2.10](#). You should note that since the surface has no green component, that no matter what value we used for the light color, there would *never* be any green component from the resulting calculation. Thus a pure green light would provide no contribution to the intensity of a surface if that surface contained a zero value for its green intensity. Thus it's possible to illuminate a surface with a bright light and get little or no illumination from that light. You should also note that using anything other than a full-bright white light $[1, 1, 1]$ will involve multiplication of values less than one, which means that using a single light source will only illuminate a surface to a maximum intensity of its color value, never more. This same problem also happens when a texture is modulated by a surface color. The color of the surface will be multiplied by the colors in the texture. If the surface color is anything other than full white, the texture will become darker. Multiple texture passes can make a surface very dark very quickly.

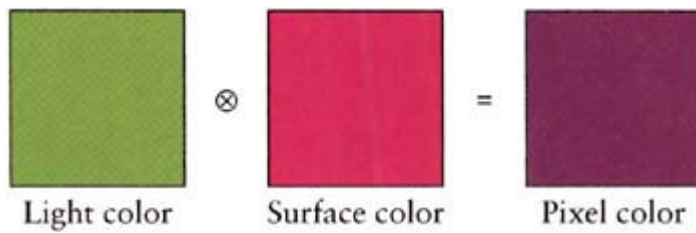


Figure 2.10: Multiplying (modulating) color values results in a color equal to or less than (darker) the original two.

Given that using a colored light in a scene makes the scene darker, how do you make the scene brighter? There are a few ways of doing this. Given that color multiplication will never result in a brighter color, it's offset a bit since we end up summing all the light contributions together, which, as we'll see in the next section, brings with it its own problems. But if you're just interested in increasing the brightness on one particular light or texture, one way is to use the API to artificially brighten the source—this is typically done with texture preprocessing. Or, you can artificially brighten the source, be it a light or a texture, by adjusting the values after you modulate them.

Dealing with Saturated Colors

On the other hand, what if we have *too* much contribution to a color? While the colors of lights are modulated by the color of the surface, *each* light source that illuminates the surface is added to the final color. All these colors are summed up to calculate the final color. Let's look at such a problem. We'll start with summing the reflected colors off a surface from two lights. The first light is an orange color and has *rgb* values [1.0,0.49,0.0], and the second light is a nice light green with *rgb* values [0.0,1.0,0.49].

Summing these two colors yields [1.0, 1.49, 0.49], which we can't display because of the values larger than one ([Figure 2.11](#)).

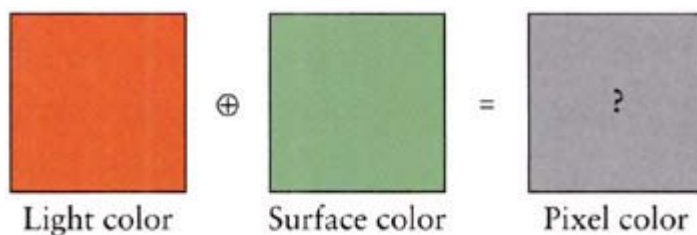


Figure 2.11: Adding colors can result in colors that are outside the displayable range.

So, what can be done when color values exceed the range that the hardware can display? It turns out that there are three common approaches [HALL 1990]. Clamping the color values is implemented in hardware, so for shaders, it's the default, and it just means that we clamp any values outside the [0,1] range. Unfortunately, this results in a shift in the color. The second most common approach is to scale the colors by the largest component. This maintains the color but reduces the overall intensity of the color. The third is to try to maintain the intensity of the color by shifting (or clipping) the color toward pure bright white by reducing the colors that are too bright while increasing the other colors and

maintaining the overall intensity. Since we can't see what the actual color for [Figure 2.11](#) is, let's see what color each of these methods yields ([Figure 2.12](#)).

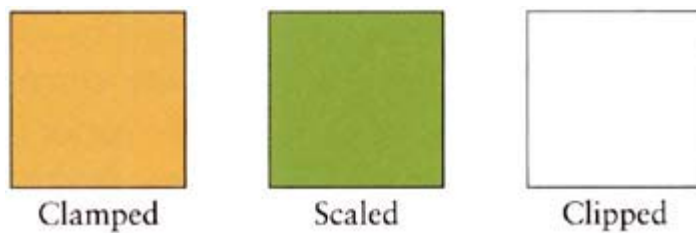


Figure 2.12: The results of three strategies for dealing with the same oversaturated color.

As you can see, we get three very different results. In terms of perceived color, the scaled is probably the closest though it's darker than the actual color values. If we weren't interested in the color but more in terms of saturation, then the clipped color is closer. Finally, the clamped value is what we get by default, and as you can see, the green component is biased down so that we lose a good sense of the "greenness" of the color we were trying to create.

Clamping Color Values

Now it's perfectly fine to end up with an oversaturated color and pass this result along to the graphics engine. What happens in the pipeline is an implicit clamping of the color values. Any value that's greater than one is clamped to one, and any less than zero are clamped to zero. So this has the benefit of requiring no effort on the part of the shader writer. Though this may make the rendering engine happy, it probably isn't what you want. Intuitively, you'd think that shining orange and green lights on a white surface would yield a strong green result. But letting the hardware clamp eradicates any predominant effect from the green light.

Clamping is fast, but it tends to lose fidelity in the scene, particularly in areas where you would want and expect subtle changes as the light intensities interact, but end up with those interactions getting eradicated because the differences are all getting clamped out by the graphics hardware.

Scaling Color Values by Intensity

Instead of clamping, you might want to scale the color by dividing by the largest color value, thus scaling the **rgb** values into the $[0,1]$ range. In the example from [Figure 2.11](#), the final color values were $[1.0, 1.49, 0.49]$ meaning our largest color value was the green, at 1.49. Using this approach, we divide each element by 1.49, yielding a scaled color of $[0.671, 1.0, 0.329]$. Thus any values greater than one are scaled to one, while any other values are also scaled by the same amount. This maintains the hue and saturation but loses the intensity. This might not be acceptable because the contrast with other colors is lost, since contrast perception is nonlinear and we're applying a linear scaling. By looking at the three results, you can see there's a large difference between the resulting colors.

Shifting Color Values to Maintain Saturation

One problem with clamping or scaling colors is that they get darker (lose saturation). An alternative to scaling is to maintain saturation by shifting color values. This technique is called clipping, and it's a bit more complicated than color scaling or clamping. The idea is to create a gray-scale vector that runs along the black-white axis of the color cube that's got the same brightness as the original color and then to draw a ray at right angles to this vector that intersects (i.e., clips) the original color's vector. You need to check to make sure that the grayscale vector is itself within the $[0,1]$ range and then to check the sign of the ray elements to see if the color elements need to be increased or decreased. As you are probably wondering, this can result in adding in a color value that wasn't in the original color, but this is a direct result of wanting to make sure that the overall brightness is the same as the original color. And, of course, everything goes to hell in a handbasket if you've got overly bright colors, which leave you with decisions about how to nudge the gray-scale vector into the $[0,1]$ range, since that means you can't achieve the input color's saturation value. Then we're back to clamping or scaling again.

ColorSpace Tool

The ColorSpace tool is a handy tool that you can use to interactively add two colors together to see the effects of the various strategies for handling oversaturated colors. You simply use the sliders to select the **rgb** color values for each color. The four displays in [Figure 2.13](#) show the composite, unmodified values of the resulting color (with no color square) and the clamped, clipped, and scaled color **rgb** values along with a color square illustrating those color values.



Figure 2.13: The ColorSpace tool interface.

Negative Colors and Darklights

You may be wondering, if I can have color values greater than the range in intermediate calculations, can I have negative values? Yes, you can! They are called "darklights" after their description in an article [GLASSNER 1992] in *Graphic Gems III*. Since this is all just math until we pass it back to the graphics hardware, we can pretty much do anything we want, which is pretty much the idea behind

programmable shaders! Darklights are nothing more than lights in which one or more of the color values are negative. Thus instead of contributing to the overall lighting in a scene, you can specify a light that diminishes the overall lighting in a scene. Darklights are used to eliminate bright areas when you're happy with all the lighting in your scene *except* for an overly bright area. Darklights can also be used to affect a scene if you want to filter out a specific **rgb** color. If you wanted to get a *night vision* effect, you could use a darklight with negative red and blue values, for example, which would just leave the green channel.

Chapter 3: Mathematics of Lighting and Shading

OVERVIEW

In mathematics you don't understand things. You just get used to them.

--Johann von Neumann

Nothing is really work unless you would rather be doing something else.

--James M. Barrie

One of the nice things about shaders is that you can create your own for whatever special effects you are looking for. In fact, one of the reasons that shaders have finally made it into mainstream 3D computer graphics is the flexibility that they provide, which can finally be realized in real time on consumer grade hardware. Unfortunately, with power comes responsibility—the responsibility to understand how lighting and shading in computer graphics is traditionally done and how you can do it yourself (or do it differently) in a shader. But first, you'll need an understanding of the mathematics behind lighting and shading.

LIGHTS AND MATERIALS

In order to understand how an object's color is determined, you'll need to understand the parts that come into play to create the final color. First, you need a source of illumination, typically in the form of a light source in your scene. A light has the properties of color (an **rgb** value) and intensity. Typically, these are multiplied to give scaled **rgb** values. Lights can also have attenuation, which means that their intensity is a function of the distance from the light to the surface. Lights can additionally be given other properties such as a shape (e.g., spotlights) and position (local or directional), but that's more in the implementation rather than the math of lighting effects.

Given a source of illumination, we'll need a surface on which the light will shine. Here's where we get interesting effects. Two types of phenomena are important lighting calculations. The first is the interaction of light with the surface boundary, and the second is the effect of light as it gets absorbed, transmitted, and scattered by interacting with the actual material itself. Since we really only have tools for describing surfaces of objects and not the internal material properties, light—surface boundary

interactions are the most common type of calculation you'll see used, though we can do some interesting simulations of the interaction of light with material internals.

Materials are typically richer in their descriptions in an effort to mimic the effects seen in real light—material surface interactions. Materials are typically described using two to four separate colors in an effort to catch the nuances of real-world light—material surface interactions. These colors are the ambient, diffuse, specular, and emissive colors, with ambient and specular frequently grouped together, and emissive specified only for objects that generate light themselves. The reason there are different colors is to give different effects arising from different environmental causes. The most common lights are as follows:

- Ambient lighting: The overall color of the object due to the global ambient light level. This is the color of the object when there's no particular light, just the general environmental illumination. That is, the ambient light is an approximation for the global illumination in the environment, and relies upon no light in the scene. It's usually a global value that's added to every object in a scene.
- Diffuse lighting: The color of the object due to the effect of a particular light. The diffuse light is the light of the surface if the surface were perfectly matte. The diffuse light is reflected in all directions from the surface and depends only on the angle of the light to the surface normal.
- Specular lighting: The color of the highlights on the surface. The specular light mimics the shininess of a surface, and its intensity is a function of the light's reflection angle off the surface.
- Emissive lighting: When you need an object to "glow" in a scene, you can do this with an emissive light. This is just an additional color source added to the final light of the object. Don't be confused just because we're simulating an object giving off its own light; you'd still have to add a real "light" to get an effect on objects in a scene.

Before we get into exactly what these types of lighting are, let's put it in perspective for our purpose of writing shader code. Shading is simply calculating the color reflected off a surface (which is pretty much what shaders do). When a light reflects off a surface, the light colors are modulated by the surface color (typically, the diffuse or ambient surface color). Modulation means multiplication, and for colors, since we are using **rgb** values, this means component-by-component multiplication. So for light source l with color (r_l, g_l, b_l) shining on surface s with color (r_s, g_s, b_s) , the resulting color r would be:

$$r = l \otimes s$$

or, multiplying it out, we get

$$r = [l_r s_r, l_g s_g, l_b s_b]$$

where the resulting **rgb** values of the light and surface are multiplied out to get the final color's **rgb** values.

The final step after calculating all the lighting contributions is to add together all the lights to get the final color. So a shader might typically do the following:

1. Calculate the overall ambient light on a surface.
2. For each light in a scene, calculate the diffuse and specular contribution for each light.
3. Calculate any emissive light for a surface.
4. Add all these lights together to calculate the final color value.

This is pretty much what the FFP does, and it's fairly simple to do as long as you don't let the number of lights get too large. Of course, since you're reading this, you're probably interested not only in what the traditional pipeline does, but also in ways of achieving your own unique effects. So let's take a look at how light interacts with surfaces of various types.

In the real world, we get some sort of interaction (reflection, etc.) when a photon interacts with a surface boundary. Thus we see the effects not only when we have a transparent—opaque boundary (like air-plastic), but also a transparent—transparent boundary (like air-water). The key feature here is that we get some visual effect when a photon interacts with some boundary between two different materials.

The conductivity of the materials directly affects how the photon is reflected. At the surface of a conductor (metals, etc.), the light is mostly reflected. For dielectrics (nonconductors), there is usually more penetration and transmittance of the light. For both kinds of materials, the dispersion of the light is a function of the roughness of the surface ([Figures 3.1](#) and [3.2](#)).

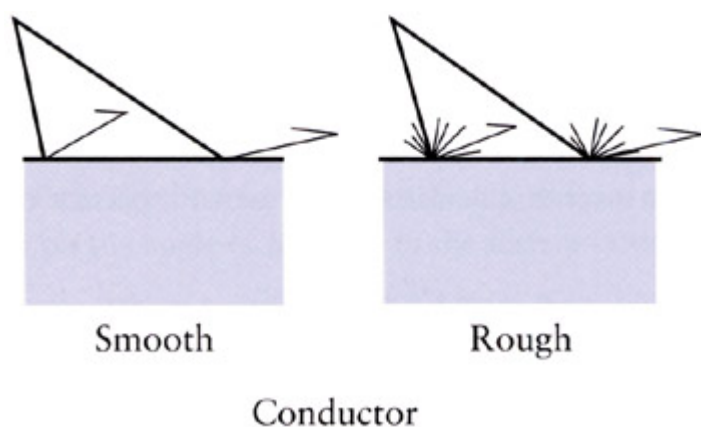


Figure 3.1: Light reflecting from a rough and smooth surface of a conductor.

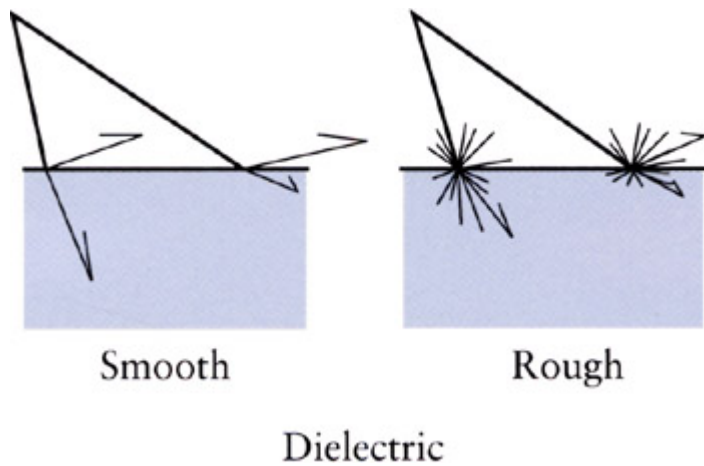


Figure 3.2: Light reflecting from a rough and smooth surface of a dielectric showing some penetration.

The simplest model assumes that the roughness of the surface is so fine that light is dispersed equally in all directions as shown in [Figure 3.1](#), though later we'll look at fixing this assumption.

A generalization is that conductors are opaque and dielectrics are transparent. This gets confusing since most of the dielectric surfaces that we are interested in modeling are mixtures and don't fall into the simple models we've described so far. Consider a thick colored lacquer surface. The lacquer itself is transparent, but suspended in the lacquer are reflective pigment off of which light gets reflected, bounced, split, shifted or altered before perhaps reemerging from the surface. This can be seen in [Figure 3.3](#), where the light rays are not just reflected but bounced around a bit inside the medium before getting retransmitted to the outside.

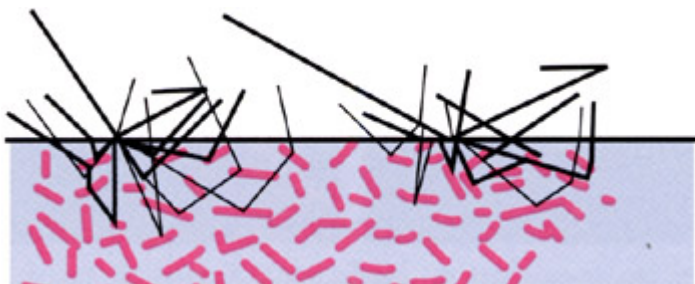


Figure 3.3: Subsurface scattering typical of pigment-saturated translucent coatings.

Metallic paint, brushed metal, velvet, etc. are all materials for which we need to examine better models to try to represent these surfaces. But with a little creativity in the modeling, it's possible to mimic the effect. [Figure 3.4](#) shows what you get when you use multiple broad specular terms for multiple base colors combined with a more traditional shiny specular term. There's also a high-frequency normal perturbation that simulates the sparkle from a metallic flake pigment. As you can see, you can get something that looks particularly striking with a fairly simple model.

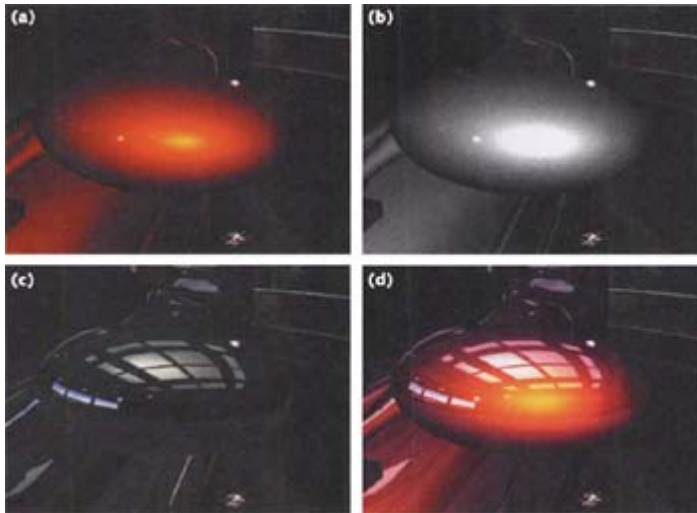


Figure 3.4: A simple shader to simulate metallic paint: (a) shows the two-tone paint shading pass; (b) shows the specular sparkle shading pass; (c) shows the environment mapping pass; (d) shows the final composite image

The traditional model gives us a specular term and a diffuse term. We have been able to add in texture maps to give our scenes some uniqueness, but the lighting effects have been very simple. Shaders allow us to be much more creative with lighting effects. As [Figure 3.4](#) shows, with just a few additional specular terms, you can bring forth a very interesting look. But before we go off writing shaders, we'll need to take a look at how it all fits together in the graphics pipeline. And a good place to start is by examining the traditional lighting model that has been around for the last two decades.

TRADITIONAL 3D HARDWARE-ACCELERATED LIGHTING MODELS

Before we get into the more esoteric uses of shaders, we'll first take a look at the traditional method of calculating lighting in hardware—a method that you'll find is sufficient for most of your needs.

The traditional approach in real-time computer graphics has been to calculate lighting at a vertex as a sum of the ambient, diffuse, and specular light. In the simplest form (used by OpenGL and Direct3D), the function is simply the sum of these lighting components (clamped to a maximum color value). Thus we have an ambient term and then a sum of all the light from the light sources.

$$i_{\text{total}} = k_a i_a + \sum (k_d i_d + k_s i_s)$$

where i_{total} is the intensity of light (as an **rgb** value) from the sum of the intensity of the global ambient value and the diffuse and specular components of the light from the light sources. This is called a *local lighting model* since the only light on a vertex is from a light source, not from other objects. That is, lights are lights, not objects. Objects that are brightly lit don't illuminate or shadow any other objects.

I've included the *reflection coefficients* for each term, **k** for completeness since you'll frequently see the lighting equation in this form. The reflection coefficients are in the [0,1] range and are specified as part

of the material property. However, they are strictly empirical and since they simply adjust the overall intensity of the material color, the material color values are usually adjusted so the color intensity varies rather than using a reflection coefficient, so we'll ignore them in our actual color calculations.

This is a very simple lighting equation and gives fairly good results. However, it does fail to take into account any gross roughness or anything other than perfect isotropic reflection. That is, the surface is treated as being perfectly smooth and equally reflective in all directions. Thus this equation is really only good at modeling the illumination of objects that don't have any "interesting" surface properties. By this I mean anything other than a smooth surface (like fur or sand) or a surface that doesn't really reflect light uniformly in all directions (like brushed metal, hair, or skin). However, with liberal use of texture maps to add detail, this model has served pretty well and can still be used for a majority of the lighting processing to create a realistic environment in real time. Let's take a look at the individual parts of the traditional lighting pipeline.

Ambient Light

Ambient light is the light that comes from all directions—thus all surfaces are illuminated equally regardless of orientation. However, this is a big hack in traditional lighting calculations since "real" ambient light really comes from the light reflected from the "environment." This would take a long time to calculate and would require ray tracing or the use of radiosity methods, so traditionally, we just say that there's x amount of global ambient light and leave it at that. This makes ambient light a little different from the other lighting components since it doesn't depend on a light source. However, you typically *do* want ambient light in your scene because having a certain amount of ambient light makes the scene look natural. One large problem with the simplified lighting model is that there is no illumination of an object with reflected light—the calculations required are enormous for a scene of any complexity (every object can potentially reflect some light and provide some illumination for every other object in a scene) and are too time consuming to be considered for real-time graphics.

So, like most things in computer graphics, we take a look at the real world, decide it's too complicated, and fudge up something that kinda works. Thus the ambient light term is the "fudge factor" that accounts for our simple lighting model's lack of an inter-object reflectance term.

$$i_a = m_a \otimes s_a$$

where i_a is the ambient light intensity, m_a is the ambient material color, and s_a is the light source ambient color. Typically, the ambient light is some amount of white (i.e., equal **rgb** values) light, but you can achieve some nice effects using colored ambient light. Though it's very useful in a scene, ambient light doesn't help differentiate objects in a scene since objects rendered with the same value of ambient tend to blend since the resulting color is the same. [Figure 3.5](#) shows a scene with just ambient illumination. You can see that it's difficult to make out details or depth information with just ambient light.



Figure 3.5: Ambient light provides illumination, but no surface details.

Ambient lighting is your friend. With it you make your scene seem more realistic than it is. A world without ambient light is one filled with sharp edges, of bright objects surrounded by sharp, dark, harsh shadows. A world with too much ambient light looks washed out and dull. Since the number of actual light sources supported by hardware FFP is limited (typically to eight simultaneous), you'll be better off to apply the lights to add detail to the area that your user is focused on and let ambient light fill in the rest. Before you point out that talking about the hardware limitation of the number of lights has no meaning in a book on shaders, where we do the lighting calculations, I'll point out that eight lights were typically the maximum that the hardware engineers created for *their* hardware. It was a performance consideration. There's nothing stopping you (except buffer size) from writing a shader that calculates the effects from a hundred simultaneous lights. But I think that you'll find that it runs much too slowly to be used to render your entire scene. But the nice thing about shaders is *you can*.

Diffuse Light

Diffuse light is the light that is absorbed by a surface and is reflected in all directions. In the traditional model, this is *ideal* diffuse reflection—good for rough surfaces where the reflected intensity is constant across the surface and is independent of viewpoint but depends only upon the direction of the light source to the surface. This means that regardless of the direction from which you view an object with a stationary diffuse light source on it, the brightness of any point on the surface will remain the same. Thus, unlike ambient light, the intensity of diffuse light is directional and is a function of the angle of the incoming light and the surface. This type of shading is called *Lambertian shading* after Lambert's cosine law, which states that the intensity of the light reflected from an ideal diffuse surface is proportional to the cosine of the direction of the light to the vertex normal.

Since we're dealing with vertices here and not surfaces, each vertex has a normal associated with it. You might hear talk of per-vertex normals vs. per-polygon normals. The difference being that per-polygon has one normal shared for all vertices in a polygon, whereas per-vertex has a normal for each

vertex. OpenGL has the ability to specify per-polygon normals, and Direct3D does not. Since vertex shaders can't share information between vertices (unless you explicitly copy the data yourself), we'll focus on per-vertex lighting. [Figure 3.6](#) shows the intensity of reflected light as a function of the angle between the vertex normal and the light direction.

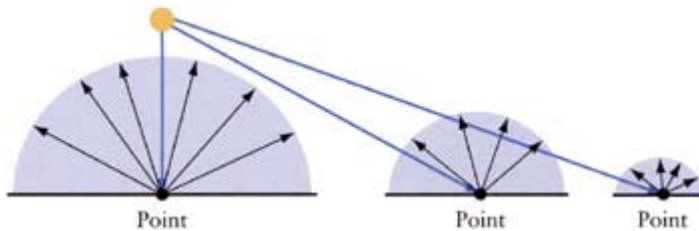


Figure 3.6: Diffuse light decreases as the angle between the light vector and the surface normal increases.

The equation for calculating diffuse lighting is

$$i_d = (\hat{n} \cdot \hat{l})(m_d \otimes s_d)$$

which is similar to the ambient light equation, except that the diffuse light term is now multiplied by the dot product of the unit normal of the vertex \hat{n} and the unit direction vector \hat{l} to the light from the vertex (not the direction *from* the light). Note that the m_d value is a color vector, so there are **rgb** or **rgba** values that will get modulated.

Since $(\hat{n} \cdot \hat{l}) = |\hat{n}||\hat{l}| \cos(\theta)$, where θ is the angle between vectors, when the angle between them is zero, $\cos(\theta)$ is 1 and the diffuse light is at its maximum. When the angle is 90° , $\cos(\theta)$ is zero and the diffuse light is zero. One calculation advantage is that when the $\cos(\theta)$ value is negative, this means that the light isn't illuminating the vertex at all. However, since you (probably!) don't want the light illuminating sides that it physically can't shine on, you want to clamp the contribution of the diffuse light to contribute only when $\cos(\theta)$ is positive. Thus the equation in practice looks more like

$$i_d = \text{MAX}(0, (\hat{n} \cdot \hat{l})(m_d \otimes s_d))$$

where we've clamped the diffuse value to only positive values. [Figure 3.7](#) was rendered with just diffuse lighting. Notice how you can tell a lot more detail about the objects and pick up distance cues from the shading.



Figure 3.7: Diffuse shading brings out some surface details.

The problem with just diffuse lighting is that it's independent of the viewer's direction. That is, it's strictly a function of the surface normal and the light direction. Thus as we change the viewing angle to a vertex, the vertex's diffuse light value never changes. You have to rotate the object (change the normal direction) or move the light (change the light direction) to get a change in the diffuse lighting of the object.

However, when we combine the ambient and diffuse, as in [Figure 3.8](#), we can see that the two types of light give a much more realistic representation than either does alone. This combination of ambient and diffuse is used for a surprisingly large number of items in rendered scenes since when combined with texture maps to give detail to a surface you get a very convincing shading effect.

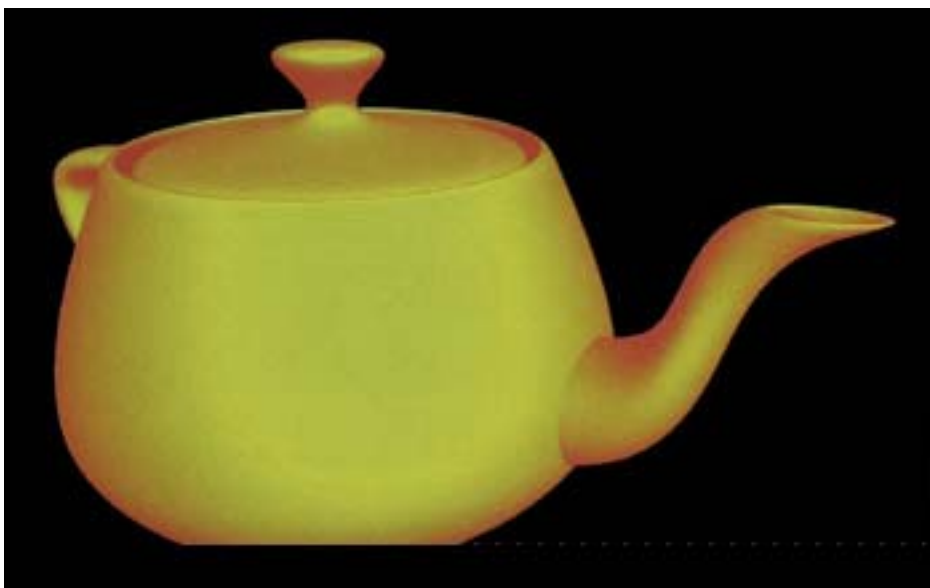


Figure 3.8: When diffuse and ambient terms are combined, you get more detail and a more natural-looking scene. The final color is the combination of the ambient and diffuse colors.

Specular Light

Ambient light is the light that comes from the environment (i.e., it's directionless); diffuse light is the light from a light source that is reflected by a surface evenly in all directions (i.e., it's independent of the viewer's position). Specular light is the light from a light source that is reflected by a surface and is reflected in such a manner that it's both a function of the light's vector and the viewer's direction. While ambient light gives the object an illuminated matte surface, specular light is what gives the highlights to an object. These highlights are greatest when the viewer is looking directly along the reflection angle from the surface. This is illustrated in [Figure 3.9](#).

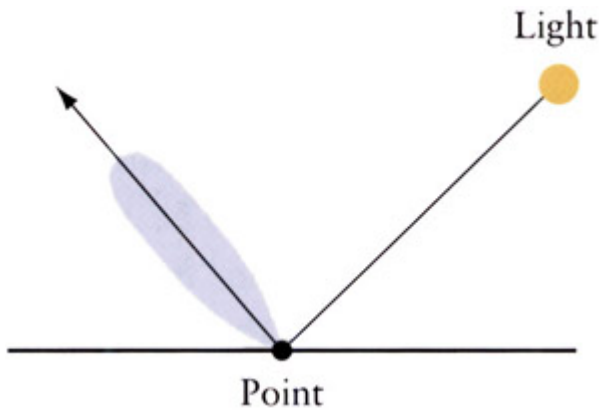


Figure 3.9: Specular light's intensity follows the reflection vector.

Most discussions of lighting (including this one) start with Phong's lighting equation (which is not the same as Phong's shading equation). In order to start discussing specular lighting, let's look at a diagram of the various vectors that are used in a lighting equation. We have a light source, some point the light is shining on, and a viewpoint. The light direction (from the point to the light) is vector \mathbf{l} , the reflection vector of the light vector (as if the surface were a mirror) is \mathbf{r} , the direction to the viewpoint from the point is vector \mathbf{v} . The point's normal is \mathbf{n} .

Phong's Specular Light Equation

Warnock [WARNOCK 1969] and Romney [ROMNEY 1969] were the first to try to simulate highlights using a $\cos^n(\theta)$ term. But it wasn't until Phong Bui-Tong [BUI 1998] reformulated this into a more general model that formalized the power value as a measure of surface roughness that we approach the terms used today for specular highlights. Phong's equation for specular lighting is

$$i_s = (m_s \otimes s_s)(\hat{r} \bullet \hat{v})^{m_s} \quad (\text{Phong})$$

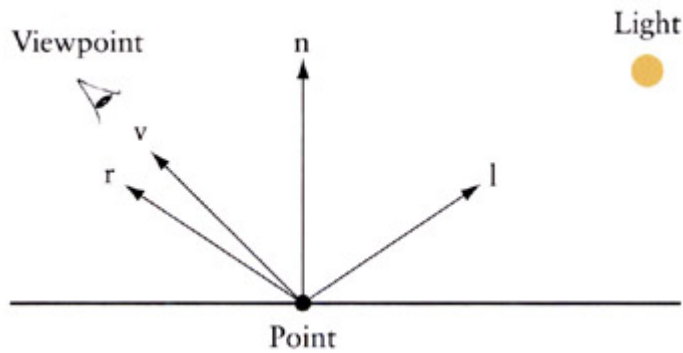


Figure 3.10: The relationship between the normal \mathbf{n} , the light vector \mathbf{l} , the view direction \mathbf{v} , and the reflection vector \mathbf{r} .

It basically says that the more the view direction, \mathbf{v} , is aligned with the reflection direction, \mathbf{r} , the brighter the specular light will be. The big difference is the introduction of the \mathbf{m}_s term, which is a power term that attempts to approximate the distribution of specular light reflection. The \mathbf{m}_s term is typically called the "shininess" value. The larger the \mathbf{m}_s value, the "tighter" (but not brighter) the specular highlights will be. This can be seen in the [Figure 3.11](#), which shows values of $(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^{\mathbf{m}_s}$ for values of \mathbf{m} ranging from 1 to 128. As you can see, the specular highlights get narrower for higher values, but they don't get any brighter.

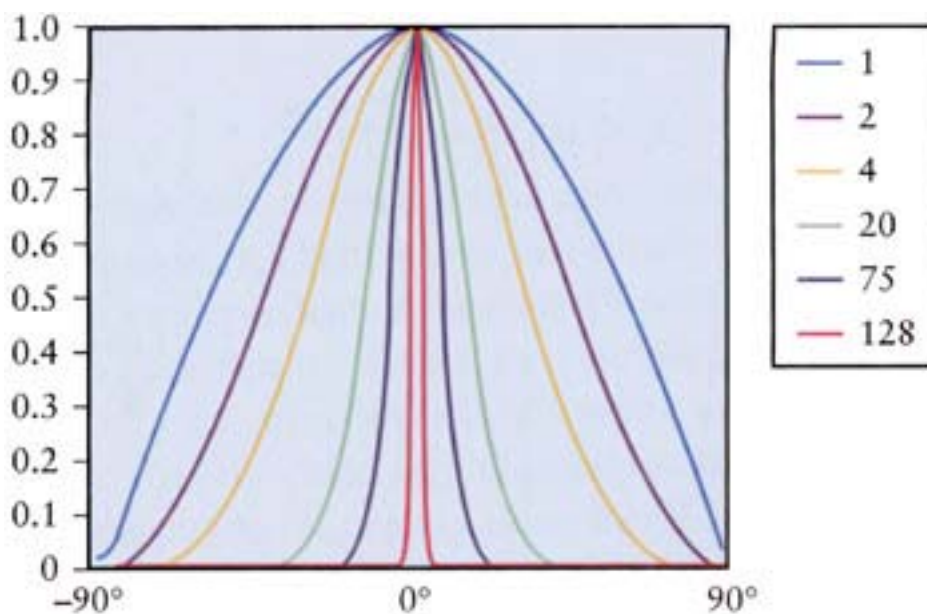


Figure 3.11: Phong's specular term for various values of the "shininess" term. Note that the values never get above 1.

Now, as you can see, this requires some calculations since we can't know \mathbf{r} beforehand since it's the \mathbf{v} vector reflected around the point's normal. To calculate \mathbf{r} we can use the following equation:^[3]

$$r = \frac{2(n \bullet l)n - l}{|n|^2}$$

If \mathbf{l} and \mathbf{n} are normalized, then the resulting \mathbf{r} is normalized and the equation can be simplified.

$$\hat{r} = 2(\hat{n} \bullet \hat{l})\hat{n} - \hat{l}$$

And just as we did for diffuse lighting, if the dot product is negative, then the term is ignored.

$$i_s = \text{MAX}\left(0, (\hat{r} \bullet \hat{v})^{m_s} (m_s \otimes s_s)\right) \quad (\text{Phong})$$

[Figure 3.12](#) shows the scene with just specular lighting. As you can see, we get an impression of a very shiny surface.

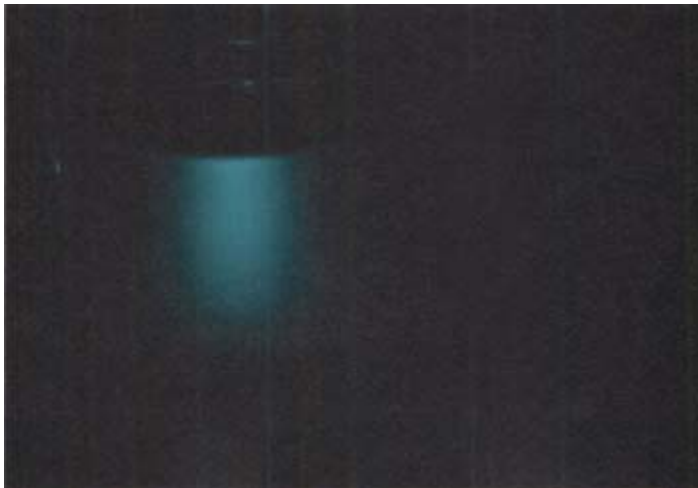


Figure 3.12: A specular term just shows the highlights.

When we add the ambient, diffuse, and specular terms together, we get [Figure 3.15](#). The three terms all act in concert to give us a fairly good imitation of a nice smooth surface that can have a varying degree of shininess to it.

You may have noticed that computing the reflection vector took a fair amount of effort. In the early days of computer graphics, there was a concerted effort to reduce anything that took a lot of computation, and the reflection vector of Phong's equation was one such item.

Blinn's Simplification: OpenGL and DirectX Lighting

Now it's computationally expensive to calculate specular lighting using Phong's equation since computing the reflection vector is expensive. Blinn [BLINN 1977] suggested, instead of using the

reflection and view vectors, that we create a "half" vector that lies between the light and view vectors. This is shown as the \mathbf{h} vector in [Figure 3.13](#). Just as Phong's equation maximizes when the reflection vector is coincident with the view vector (thus the viewer is looking directly along the reflection vector), so does Blinn's. When the half vector is coincident with the normal vector, then the angle between the view vector and the normal vector is the same as between the light vector and the normal vector. Blinn's version of Phong's equation is:

$$i_s = (m_s \otimes s_s)(\hat{n} \cdot \hat{h})^{m_s} \quad (\text{Blinn-Phong})$$

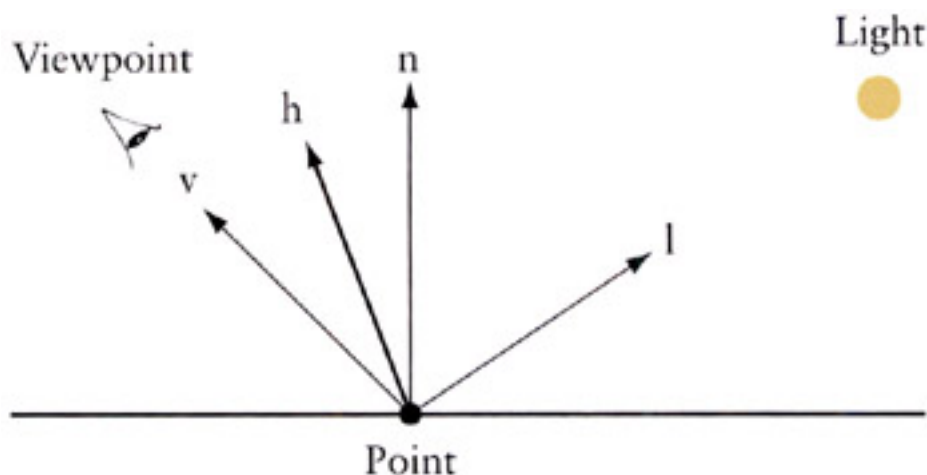


Figure 3.13: The half-angle vector is an averaging of the light and view vectors.

where the half vector is defined as

$$h = \frac{l + v}{|l + v|}$$

The advantage is that no reflection vector is needed; instead, we can use values that are readily available, namely, the view and light vectors. Note that both OpenGL and the DirectX FFP use Blinn's equation for specular light.

Besides a speed advantage, there are some other effects to note between Phong's specular equation and Blinn's.

- If you multiply Blinn's exponent by 4, you approximate the results of Phong's equation.
- Thus if there's an upper limit on the value of the exponent, Phong's equation can produce sharper highlights.
- For $\mathbf{l} \cdot \mathbf{v}$ angles greater than 45° (i.e., when the light is behind an object and you're looking at an edge), the highlights are longer along the edge direction for Phong's equation.
- Blinn's equation produces results closer to those seen in nature.

For an in-depth discussion of the differences between the two equations, there's an excellent discussion in [FISHER 1994]. [Figure 3.14](#) shows the difference between Phong lighting and Blinn—Phong lighting.



Figure 3.14: Blinn-Phong specular on the left, Phong specular on the right.

The Lighting Equation

So now that we've computed the various light contributions to our final color value, we can add them up to get the final color value. Note that the final color values will have to be made to fit in the $[0,1]$ range for the final **rgb** values.

$$i_{\text{total}} = i_a + \sum (i_d + i_s)$$

Our final scene with ambient, diffuse, and (Blinn's) specular light contributions (with one white light above and to the left of the viewer) looks like [Figure 3.15](#).



Figure 3.15: A combination of ambient, diffuse, and specular illumination.

It may be surprising to discover that there's more than one way to calculate the shading of an object, but that's because the model is empirical, and there's no correct way, just different ways that all have

tradeoffs. Until now though, the only lighting equation you've been able to use has been the one we just formulated.

Most of the interesting work in computer graphics is tweaking that equation, or in some cases, throwing it out altogether and coming up with something new.

The next sections will discuss some refinements and alternative ways of calculating the various coefficients of the lighting equation. We hope you'll get some ideas that you'll be able to use to create your own unique shaders.

Light Attenuation

Light in the real world loses its intensity as the inverse square of the distance from the light source to the surface being illuminated. However, when put into practice, this seemed to drop off the light intensity in too abrupt a manner and then not to vary too much after the light was far away. An empirical model was developed that seems to give satisfactory results. This is the attenuation model that's used in OpenGL and DirectX. The f_{atten} factor is the attenuation factor. The distance d between the light and the vertex is always positive. The attenuation factor is calculated by the following equation:

$$f_{\text{atten}} = 1 / (k_c + k_l d + k_q d^2)$$

where the k_c , k_l , and k_q parameters are the constant, linear, and quadratic attenuation constants, respectively. To get the "real" attenuation factor, you can set k_q to one and the others to zero.

The attenuation factor is multiplied by the light diffuse and specular values. Typically, each light will have a set of these parameters for itself. The lighting equation with the attenuation factor looks like this.

$$i_{\text{total}} = i_a + \sum f_{\text{atten}}(i_d + i_s)$$

[Figure 3.16](#) shows a sample of what attenuation looks like. This image is the same as the one shown in [Figure 3.15](#), but with light attenuation added.



Figure 3.16: A scene with light attenuation. The white sphere is the light position.

Schlick's Simplification for the Specular Exponential Term

Real-time graphics programmers are always looking for simplifications. You've probably gathered that there's no such thing as the "correct" lighting equation, just a series of hacks to make things look right with as little computational effort as possible. Schlick [SCHLICK 1994] suggested a replacement for the exponential term since that's a fairly expensive operation. If we define part of our specular light term as follows:

$$(S)^{m_s}$$

where S is either the Phong or Blinn-Phong flavor of the specular lighting equation, then Schlick's simplification is to replace the preceding part of the specular equation with

$$\frac{S}{m_s - m_s S + S}$$

which eliminates the need for an exponential term. At first glance, a plot of Schlick's function looks very similar to the exponential equation ([Figure 3.17](#)).

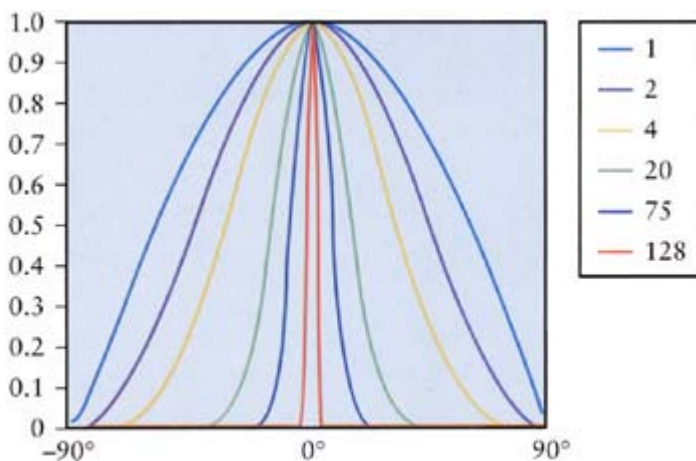


Figure 3.17: Schlick's term for specular looks very much like the more expensive Phong term.

If we plot both equations in the same graph ([Figure 3.18](#)), we can see some differences and evaluate just how well Schlick's simplification works. The blue values are Schlick's, and the red are the exponential plot. As the view and light angles get closer (i.e., get closer to zero on the x axis), we can see that the values of the curves are quite close. (For a value of zero, they overlap.) As the angles approach a grazing angle, we can see that the approximation gets worse. This would mean that when there is little influence from a specular light, Schlick's equation would be slightly less sharp for the highlight.

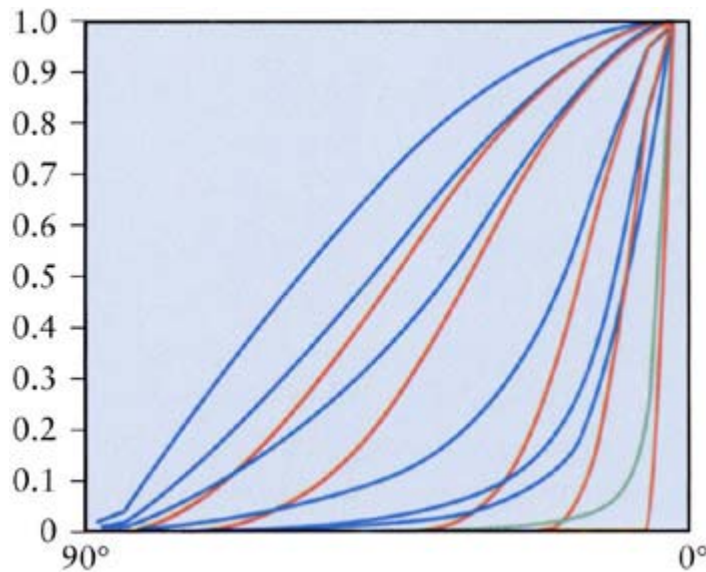


Figure 3.18: Schlick's vs. Phong's specular terms.

You might notice the green line in [Figure 3.18](#). Unlike the limit of a value of 128 for the exponential imposed in both OpenGL and DirectX FFP, we can easily make our values in the approximation any value we want. The green line is a value of 1024 in Schlick's equation. You may be thinking that we can make a very sharp specular highlight using Schlick's approximation with very large values—sharper than is possible using the exponential term. Unfortunately, we can't since you really need impractically large values (say, around 100 million) to boost it significantly over the exponential value for 128. But that's just the kind of thinking that's going to get your creative juices flowing when writing your own shaders! If the traditional way doesn't work, figure out something that will.

Oren—Nayar Diffuse Reflection

Though there's been a lot of research on specular reflection models, there's been less research on diffuse reflection models. One of the problems of the standard Lambertian model is that it considers the surface as a smooth diffuse surface. Surfaces that are really rough, like sandpaper, exhibit much more of a backscattering effect, particularly when the light source and the view direction are in the same direction.

The classic example of this is a full moon. If you look at the picture of the moon shown in [Figure 3.19](#), it's pretty obvious that this doesn't follow the Lambertian distribution—if it did, the edges of the moon would be in near darkness. In fact, the edges look as bright as the center of the moon. This is because the moon's surface is rough—the surface is made of a jumble of dust and rock with diffuse reflecting surfaces at all angles—thus the quantity of reflecting surfaces is uniform no matter the orientation of the surface; hence no matter the orientation of the surface to the viewer, the amount of light reflecting off the surface is nearly the same.



Figure 3.19: The full moon is an good example of something that doesn't show Lambertian diffuse shading.

The effect we're looking at is called backscattering. Backscattering is when a rough surface bounces around a light ray and then reflects the ray in the direction the light originally came from. Note that there is a similar but different effect called retroreflection. Retroreflection is the effect of reflecting light toward the direction from which it came, no matter the orientation of the surface. This is the same effect that we see on bicycle reflectors. However, this is due to the design of the surface features (made up of ∇ -shaped or spherical reflectors) rather than a scattering effect.

In a similar manner, when the light direction is closer to the view direction, we get the effect of forward scattering. Forward scattering is just backscattering from a different direction. In this case, instead of near uniform illumination though, we get near uniform loss of diffuse lighting. You can get the same effects here on Earth. [Figures 3.20](#) and [3.21](#) show the same surfaces demonstrating backscattering and forward scattering. Both the dirt field in [Figure 3.20](#) and the soybean field in [Figure 3.21](#) can be considered rough diffuse reflecting surfaces.



Figure 3.20: The same dirt field showing wildly differing reflection properties.



Figure 3.21: A soybean field showing differing reflection properties.

Notice how the backscattering image shows a near uniform diffuse illumination, whereas the forward scattering image shows a uniform dull diffuse illumination. Also note that you can see specular highlights and more color variation because of the shadows due to the rough surface, whereas the backscattered image washes out the detail.

In an effort to better model rough surfaces, Oren and Nayar [OREN 1992] came up with a generalized version of a Lambertian diffuse shading model that tries to account for the roughness of the surface. They applied the Torrance—Sparrow model for rough surfaces with isotropic roughness and provided parameters to account for the various surface structures found in the Torrance—Sparrow model. By comparing their model with actual data, they simplified their model to the terms that had the most significant impact. The Oren—Nayar diffuse shading model looks like this.

$$i_d = \frac{\rho}{\pi} E_0 \cos(\theta_i) (A + B \max[0, \cos(\phi_r - \phi_i)] \sin(\alpha) \tan(\beta))$$

where

$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09}$$

Now this may look daunting, but it can be simplified to something we can appreciate if we replace the original notation with the notation we've already been using. ρ/π is a surface reflectivity property, which we can replace with our surface diffuse color. E_o is a light input energy term, which we can replace with our light diffuse color. And the θ_i term is just our familiar angle between the vertex normal and the light direction. Making these exchanges gives us

$$i_d = (m_d \otimes s_d)(\hat{n} \cdot \hat{l})(A + B \max[0, \cos(\phi_r - \phi_l)] \sin(\alpha) \tan(\beta))$$

(Oren-Nayar)

which looks a lot more like the equations we've used. There are still some parameters to explain.

- σ is the surface roughness parameter. It's the standard deviation in radians of the angle of distribution of the microfacets in the surface roughness model. The larger the value, the rougher the surface.
- θ_r is the angle between the vertex normal and the view direction.
- $\phi_r - \phi_l$ is the circular angle (about the vertex normal) between the light vector and the view vector.
- α is $\max(\theta_i, \theta_r)$.
- β is $\min(\theta_i, \theta_r)$.

Note that if the roughness value is zero, the model is the same as the Lambertian diffuse model. Oren and Nayar also note that you can replace the value 0.33 in coefficient A with 0.57 to better account for surface interreflection.

^[3] An excellent explanation of how to compute the reflection vector can be found in [RTR].

PHYSICALLY BASED ILLUMINATION

In order to get a more realistic representation of lighting, we need to move away from the simplistic models that are found hard coded in most graphics pipelines and move to something that is based more in a physical representation of light as a wave with properties of its own that can interact with its environment. To do this, we'll need to understand how light passes through a medium and how hitting the boundary layer at the intersection of two media can affect light's properties. In [Figure 3.22](#), there's an incident light hitting a surface. At the boundary of the two media (in this case, air and glass), there

are two resulting rays of light. The reflected ray is the one that we've already discussed to some extent, and the other ray is the refracted or transmitted ray.

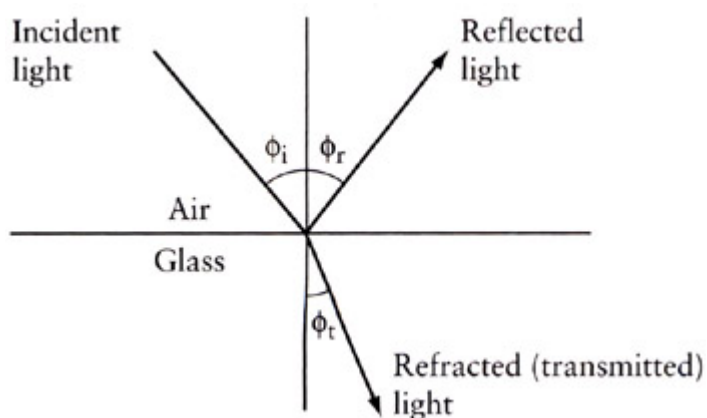


Figure 3.22: Light being reflected and refracted through a boundary.

In addition to examining the interaction of light with the surface boundary, we need a better description of real surface geometries. Until now, we've been treating our surfaces as perfectly smooth and uniform. Unfortunately, this prevents us from getting some interesting effects. We'll go over trying to model a real surface later, but first let's look at the physics of light interacting at a material boundary.

Reflection

Reflection of a light wave is the change in direction of the light ray when it bounces off the boundary between two media. The reflected light wave turns out to be a simple case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now). Thus for a light wave reflecting off a perfectly smooth surface

$$\phi_{\text{incident}} = \phi_{\text{reflected}}$$

Until now, we've treated all of our specular lighting calculations as essentially reflection off a perfect surface, a surface that doesn't interact with the light in any manner other than reflecting light in proportion to the color of the surface itself. Using a lighting model based upon the Blinn—Phong model means that we'll always get a uniform specular highlight based upon the color of the reflecting light and material, which means that all reflections based on this model will be reminiscent of plastic. In order to get a more interesting and realistic lighting model, we need to add in some nonlinear elements to our calculations. First, let's examine what occurs when light is reflected off a surface. For a perfect reflecting surface, the angle of the incoming light (the angle of incidence) is equal to that of the reflected light. Phong's equation just blurs out the highlight a bit in a symmetrical fashion. Until we start dealing with nonuniform smooth surfaces in a manner a bit more realistic than Phong's in the section on surface geometry, this will have to do.

Refraction

Refraction happens when a light wave goes from one medium into another. Because of the difference in the speed of light of the media, light bends when it crosses the boundary. Snell's law gives the change in angles.

$$n_a \sin(\phi_a) = n_b \sin(\phi_b)$$

where the n 's are the material's index of refraction. Snell's law states that when light refracts through a surface, the refracted angle is shifted by a function of the ratio of the two material's indices of refraction. The index of refraction of vacuum is 1, and all other material's indices of refraction are greater than 1.

What this means is that in order to realistically model refraction, we need to know the indices of refraction of the two materials that the light is traveling through. Let's look at an example ([Figure 3.23](#)) to see what this really means. Let's take a simple case of a ray of light traveling through the air ($n_{\text{air}} \cong 1$) and intersecting a glass surface ($n_{\text{glass}} \cong 1.5$). If the light ray hits the glass surface at 45° , at what angle does the refracted ray leave the interface?

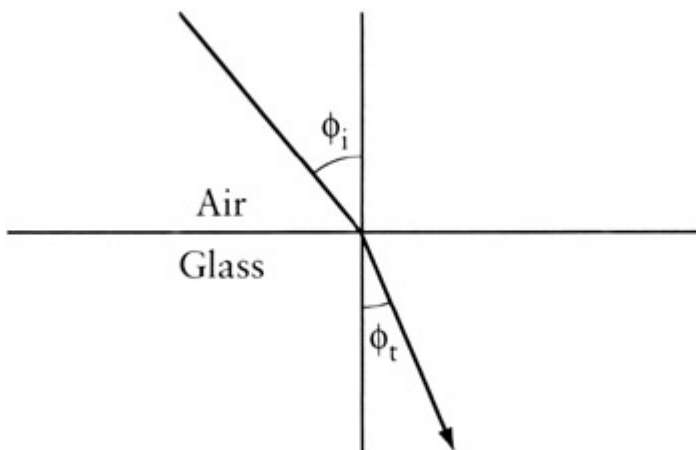


Figure 3.23: The refracted ray's angle is less than the incoming ray's.

The angle of incidence is the angle between the incoming vector and the surface. Rearranging Snell's law, we can solve for the refracted angle.

$$\left(\frac{n_{\text{air}}}{n_{\text{glass}}}\right) \sin(\phi_{\text{air}}) = \sin(\phi_{\text{glass}})$$

Inserting the values give us

$$\begin{aligned} \left(\frac{1.0}{1.5}\right) \sin(45^\circ) &= \sin(\phi_{\text{glass}}) \\ \phi_{\text{glass}} &= 28.1^\circ \end{aligned}$$

which is a fairly significant change in the angle! If we change things around so that we are following a light ray emerging from water into the air, we can run into another phenomenon. Since the index of refraction is just a measure of the change in speed that light travels in a material, we can observe from Snell's law (and the fact that the index of refraction in a vacuum is 1) that light bends toward the normal when it slows down (i.e., when the material it's intersecting with has a higher index of refraction). Consequently, when we intersect a medium that has a lower index of refraction (e.g., going from glass to air), then the angle will increase. Ah, you must be thinking, we're approaching a singularity here since we can then easily generate numbers that we can't take the inverse sine of! If we use Snell's law for light going from water to air, and plug in 90° for the refracted angle, we get 41.8° for the incident angle. This is called the critical angle at which we observe the phenomenon of *total internal reflection*. At any angle greater than this, light will not pass through a boundary but will be reflected internally.

One place that you get interesting visual properties is in the diamond—air interface. The refractive index of a diamond is fairly high, 2.24, which means that it's got a very low critical angle, just 24.4° . This means that a good portion of the light entering a diamond will bounce around the inside of the diamond hitting a number of air—diamond boundaries, and as long as the angle is 24.4° or greater, it will keep reflecting internally. This is why diamonds are cut to be relatively flatish on the top but with many faceted sides, so that light entering in one spot will bounce around and exit at another, giving rise to the sparkle normally associated with diamonds.

Another place where a small change in the indices of refraction occurs is on a road heated by the sun when viewed from far away (hence a glancing incident angle). The hot air at the road's surface has a slightly smaller index of refraction than the denser, cooler air above it. This is why you get the effect of a road looking as though it were covered with water and reflecting the image above it—the light waves are actually reflected off the warm air—cold air interface.

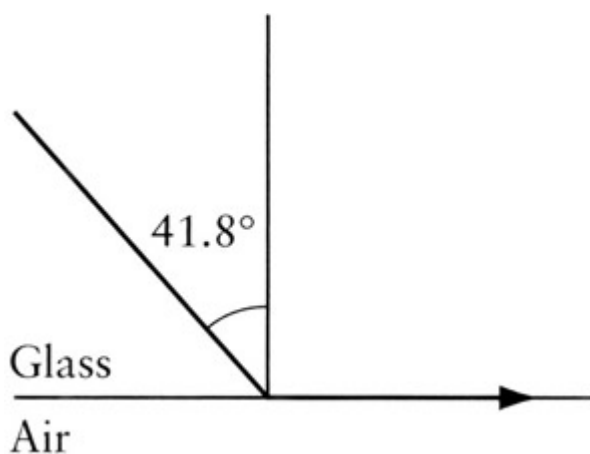


Figure 3.24: The critical angle.

What makes this really challenging to model is that the index of refraction for most materials is a *function of the wavelength of the light*. This means that not only is there a shift in the angle of refraction,

but that the shift is different for differing wavelengths of light. [Figure 3.25](#) and [3.26](#) show the index of refraction for fused quartz and sapphire plotted against the wavelength. You can see the general trend that shorter wavelength light (bluish) tends to bend more than the longer (reddish) wavelengths.

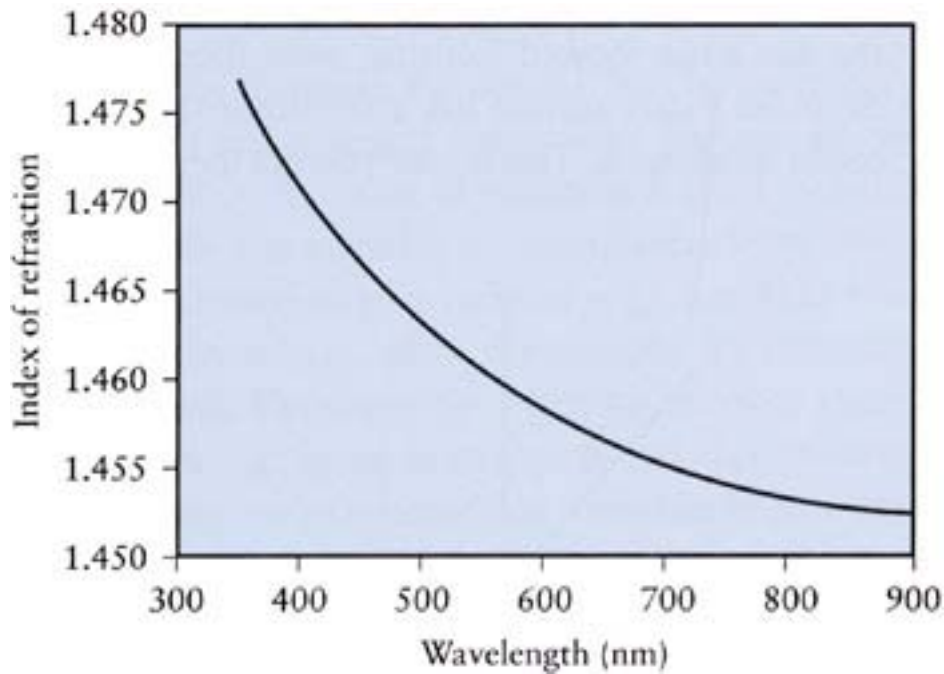


Figure 3.25: Index of refraction as a function of wavelength for quartz.

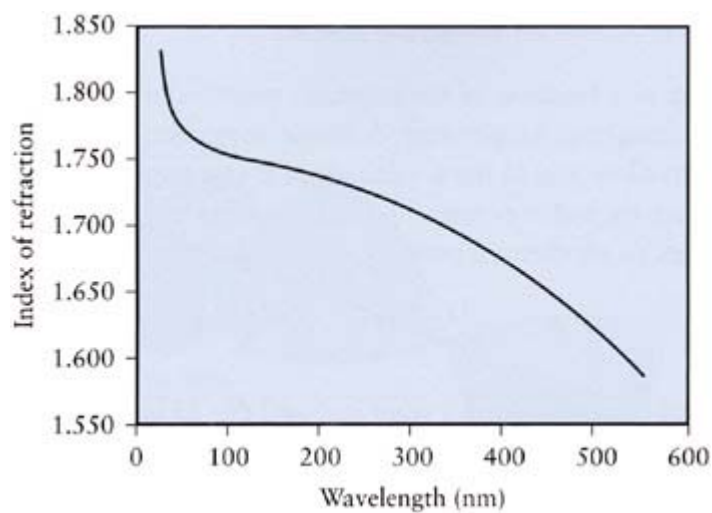


Figure 3.26: Index of refraction as a function of wavelength for sapphire.

This is the phenomenon that's responsible for the spectrum that can be seen when white light is passed through a prism ([Figure 3.27](#)). It's refraction that will break apart a light source into its component colors, not reflection.

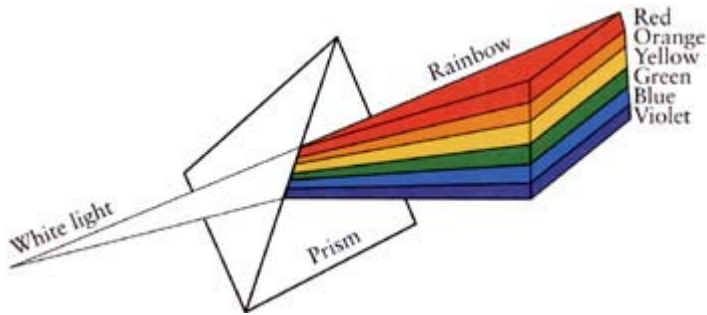


Figure 3.27: The wavelength dependence of the index of refraction in action.

This is one area where our simplistic model of light breaks down since we're not computing an entire spectrum of light waves, but we're limited to three primary colors. For reference, the **rgb** values can be assigned to a range of wavelengths as follows:

$$\begin{aligned}\text{blue} &= 492 - 455\text{nm} \\ \text{green} &= 577 - 492\text{nm} \\ \text{red} &= 780 - 622\text{nm}\end{aligned}$$

There's a lot more to color science than just determining wavelengths, but that's beyond the scope of this book.

While the spectrum spreading effect of refraction is interesting in itself, the **rgb** nature of computer color representation precludes performing this spreading directly—we can't break up a color value into multiple color values. However, with some work, you can compute the shade of the color for a particular angle of refraction and then use that as the material color to influence the refracted color.

Temperature Correction for Refractive Index

Refractive index is a function of temperature, mostly due to density changes in materials with changes in temperature. A simple correction can be applied in most circumstances to allow you to use a value given at one temperature at another. For example, suppose the index of refraction value you have is given at 25°C: η^{25} . To convert the index to another temperature, η^t , you can use the following equation:

$$\eta^t = \eta^{25} + (25.0 - t)(0.00045)$$

where the actual temperature you want is t , and the 25 is the temperature (both in °C) of the actual index you have, η^{25} .

The Fresnel Equations

The Fresnel (pronounced *Freh-nel*) equations are used to calculate the percentage of energy in the refracted and the reflected parts of the wave (ignoring adsorption). In order to understand what the equations calculate, we'll have to take a look at what happened when a light wave (as opposed to a photon) interacts with a surface. We have to do it this way since this is the only way to describe the subtle (and realistic) visual effects we are looking for. A wave of energy has both an electric field and a magnetic field that travel in perpendicular phase, as shown in [Figure 3.28](#).

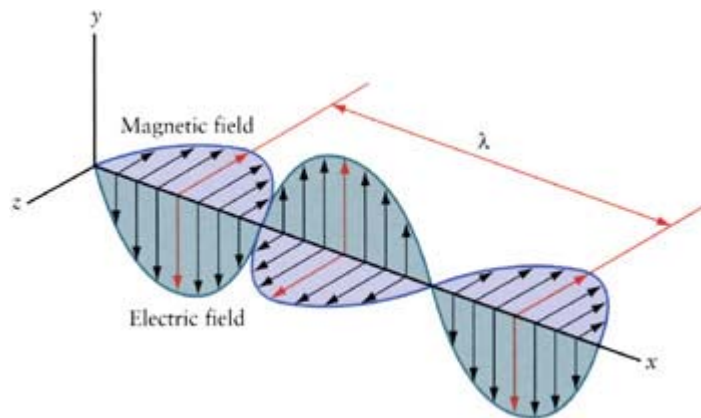


Figure 3.28: The perpendicular nature of the magnetic and electrical fields of a light wave.

In general, when a wave reaches a boundary between two different dielectric constants, part of the wave is reflected and part is transmitted, with the sum of the energies in these two waves equal to that of the original wave.

The Fresnel equations are a solution to Maxwell's equations for electromagnetic waves at the interface. If you are really interested in seeing how this works see [HECHT 1987], but I'll spare you the actual derivation and get to the important part. What Fresnel did (besides proving once and for all that light can behave like a wave) is figure out that for the two extrema of the light wave—the light wave with the electric field parallel to the surface and the light wave with the electric field perpendicular to the surface, the energy transmitted and reflected—are functions of the angle of incidence and the indices of refraction for the two media. This is for a nonconductive (*dielectric*) medium like plastic or glass.

For a conductive media, there's actually some interaction between the free electrons in the conductor (or else it wouldn't be a conductor) and the magnetic field of the light wave. Being a conductor, there are free electrons in the material. When the light wave interacts with the material, the electrons in the material oscillate with the magnetic field of the light wave, matching its frequency. These oscillations radiate (and, in effect, reflect) the light wave. In addition, a conductor has some resistance to electron movement, so the material absorbs some of the energy that would have been reradiated. Fresnel equations for conductive interfaces (since they absorb light they are also generally opaque) usually involve a dielectric (like air) and a conductor since you don't have a light interface between two opaque materials. The parallel and perpendicular components are sometimes referred to as the *p*-polarized and *s*-polarized components, respectively.

Fresnel Equations for Dielectrics

The simplest form of the Fresnel equations are for dielectrics.

$$F_{r_{\parallel}} = \frac{n_t(n \bullet l) + n_i(n \bullet t)}{n_t(n \bullet l) - n_i(n \bullet t)}$$

$$F_{r_{\perp}} = \frac{n_i(n \bullet l) + n_t(n \bullet t)}{n_i(n \bullet l) - n_t(n \bullet t)}$$

$$F_{t_{\parallel}} = \frac{2n_i(n \bullet l)}{n_t(n \bullet l) - n_i(n \bullet t)}$$

$$F_{t_{\perp}} = \frac{2n_i(n \bullet l)}{n_i(n \bullet l) - n_t(n \bullet t)}$$

where r and t are broken into parallel and perpendicular segments. n_t and n_i are the indices of refraction for the transmitting (reflecting) and incident materials respectively.

Now we can simplify these equations by assuming normalized vectors and multiplying out the dot products, noting that $\mathbf{n} \bullet \mathbf{l} = \cos(\phi_i)$ and $\mathbf{n} \bullet \mathbf{t} = -\cos(\phi_r)$, and using Snell's law to get rid of the indices of refraction. ϕ_i is the angle of incidence and ϕ_t the reflected angle. These simplified equations are

$$F_{r_{\parallel}} = \frac{\tan^2(\phi_i - \phi_r)}{\tan^2(\phi_i + \phi_r)}$$

$$F_{r_{\perp}} = \frac{\sin^2(\phi_i - \phi_r)}{\sin^2(\phi_i + \phi_r)}$$

$$F_{t_{\parallel}} = \frac{2 \sin(\phi_t) \cos(\phi_i)}{\sin(\phi_i + \phi_t) \cos(\phi_i - \phi_t)}$$

$$F_{t_{\perp}} = \frac{2 \sin(\phi_t) \cos(\phi_i)}{\sin(\phi_i + \phi_t)}$$

Using these equations, we can calculate the percent of energy transmitted or reflected. Since these equations represent the maxima and minima of the interaction between the media depending upon the orientation (polarization) of the light wave's fields, we'll just take the average to compute the amount transmitted and reflected. Thus

$$F_r = \frac{1}{2}(F_{r_{\parallel}} + F_{r_{\perp}})$$

$$F_t = \frac{1}{2}(F_{t_{\parallel}} + F_{t_{\perp}})$$

gives us the fraction of unpolarized light transmitted and reflected. Note that this average is for unpolarized light. If your light source was polarized, you could pick one equation instead of the average. Also note that due to the conservation of energy, we could also write

$$F_r = 1 - F_t$$

POISSON'S BRIGHT SPOT

In 1818, the French Academy of Science held a competition for best essay covering the wave theory of light. At this time, there was great debate about whether light was a particle or a wave. A 30-year-old civil engineer by the name of Augustin Fresnel submitted a monster 135-page paper on the theory of light waves and their diffraction for the competition, which completely went against the particle theory since particles don't bend. Though an excellent theorist, Fresnel couldn't solve some of his equations in his paper. Unfortunately for him, one of the judges of the competition **was** an excellent mathematician: Siméon Denis Poisson. Poisson also happened to be a very strong believer in Newton's particle theory of light and was able to solve Fresnel's equations. He then set out to "prove" that the wave theory was wrong using Fresnel's own equations. Poisson stated that Fresnel's equations gave the absurd prediction that a disk casting a shadow would show a bright spot at the center of the shadow due to the diffraction waves all being in phase. The waves would all contribute to make the spot just as bright as if the disk weren't there. A simple experiment would prove it so. Another judge of the competition, François Arago, arranged to have the experiment performed, and the bright spot was seen! Fresnel was awarded the prize in 1819, and Poisson greatly, though unintentionally, advanced the wave theory of light. The bright spot is usually referred to as Poisson's bright spot, though Poisson refused to believe in the wave theory up to his death 30 years later. You can perform this experiment yourself—it works best with laser light. The spot shows up a little over a disk's width behind the disk. You can also see it during a solar eclipse as a bright ring around the moon's disk. [Figure 3.29](#) shows the bright spot and the other waves of interference.

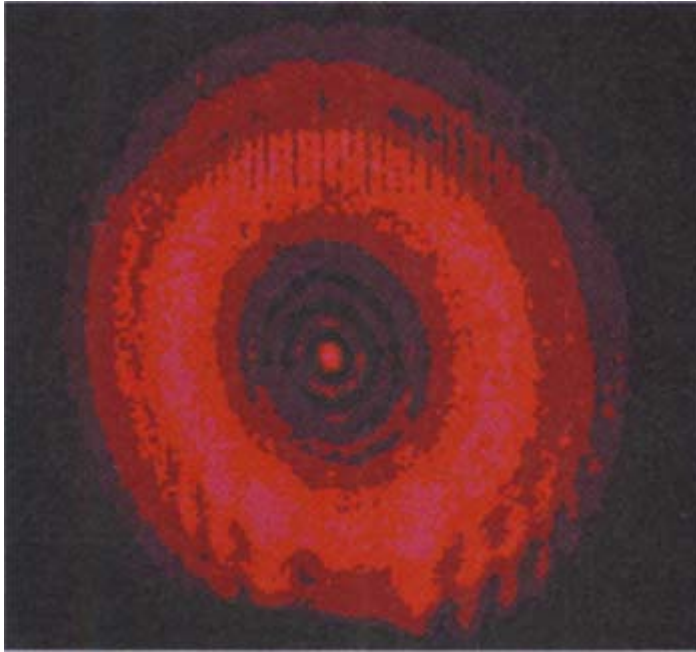


Figure 3.29: Poisson's bright spot.

Let's take a look at what this means in a practical application. Let's plot the values for the Fresnel equation for the air-glass interface ([Figure 3.30](#)). The first thing you should notice is that even when the incident angle is at 0° —that is, the light is shining along the surface normal—there is still some loss in transmittance.

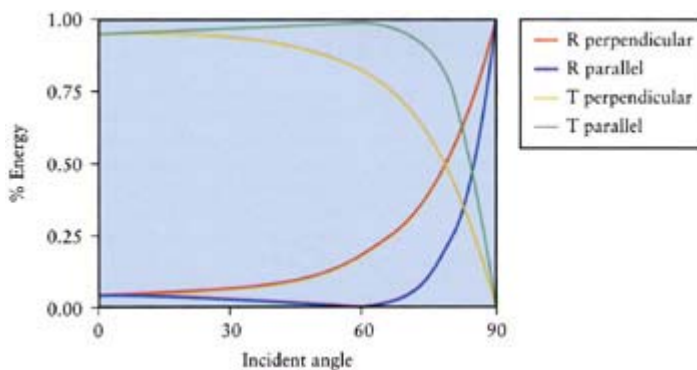


Figure 3.30: The reflection and transmission curves for the parallel and perpendicular waves in the air-glass interface.

If we do the math, we will discover that there is no distinction between the parallel and perpendicular components when the angle is 0° , and in this case, the Fresnel equations simplify to the following:

$$r(0^\circ) = \left(\frac{n_t - n_i}{n_t + n_i} \right)^2$$

$$t(0^\circ) = \frac{4n_t n_i}{(n_t + n_i)^2}$$

Thus at 0° for the air-glass interface, we can see that about 4% of the light is reflected. This means that if you have a glass window, you will get only 92% of the light transmitted through (you have the air-glass interface, 4% reflected, *and* the glass-air interface, another 4%, when the ray comes out of the glass). Glass used in optics is usually coated with a thin film of antireflective coating, which reduces the reflectivity to something around 1%. You might also note that at an angle of about 56° , the parallel reflectance drops to zero. This is called Brewster's angle or the polarization angle (ϕ_p) and is the effect on which polarized lenses work. You can use Snell's law and the observation that for polarized light $\phi_t = 90^\circ - \phi_p$ to derive the following equation for Brewster's angle:

$$\tan(\phi_p) = \frac{n_t}{n_i}$$

Let's plot the average reflected and transmitted energy and take a look at the plot ([Figure 3.31](#)).

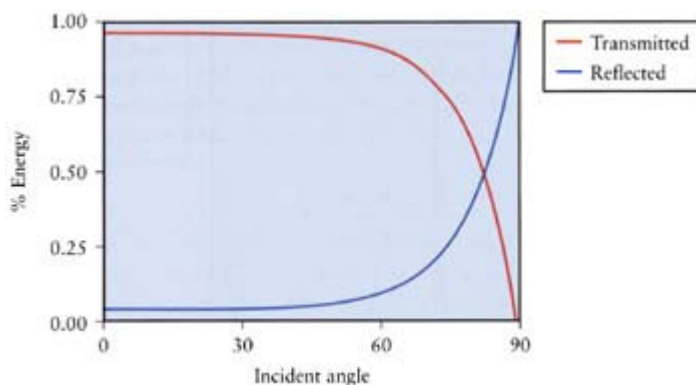


Figure 3.31: The averaged reflectance and transmittance values for the air-glass interface.

We can see that as the incident angle approaches 90° , the reflectivity approaches 100%. This is one of the more important aspects of the Fresnel equations—*at a glancing angle, all surfaces become perfect reflectors, regardless of what the surface is made*. In fact, this is one way in which x-rays are focused. The only fly in the ointment is that this is true only for perfectly flat surfaces, but this will be covered in the next section.

To be thorough, we should also take a look at the other side of the interface, when looking through a medium of higher refractive index to that of a lower one. In this case, we will reach an incident angle where we reach total internal reflection, and no light will be transmitted through the interface. If we

reverse the indices of refraction and take a look at the glass-air interface, we get the plots shown in [Figure 3.32](#).

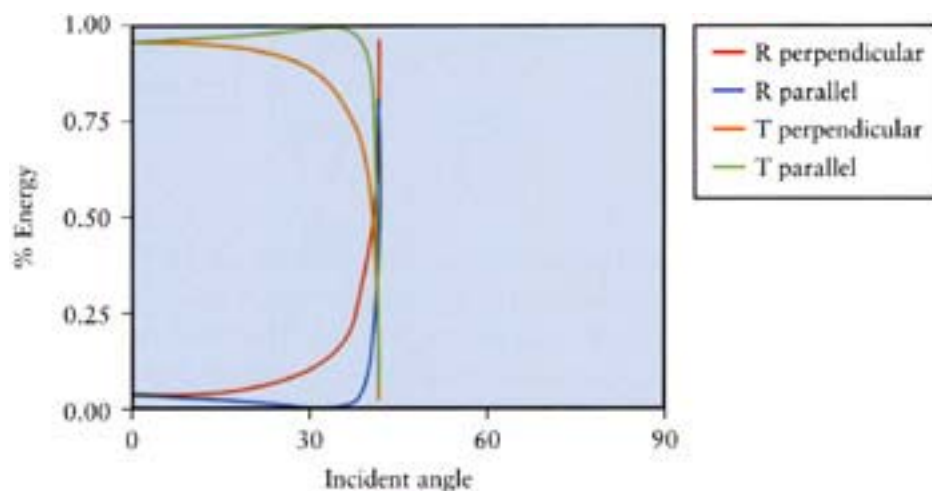


Figure 3.32: A glass-air interface shows that we reach a point where all light is reflected internally.

At the critical angle of 41.8° , we get total internal reflection, and no light is transmitted through the interface. Thus if you wanted to model something like a scene underwater, you'd need to treat the water surface as a mirror surface at any angles over the critical angle.

The Fresnel Term in Practice

For real materials, the Fresnel term depends upon the incoming angle of light and the reflected angle. The reflected angle is a function of the indices of refraction of both materials, which in turn, are dependant upon light wavelength and density of the materials. The density, in turn, typically is a function of temperature. The index of refraction usually increases with the density of the medium, and usually decreases with increasing temperature. This is all very fine if you happen to have data for the index of refraction for the wavelengths of interest over the temperatures you'll need. Since this kind of data is difficult to find, we'll do what innumerable computer graphics researchers have done before us—we'll fake it.

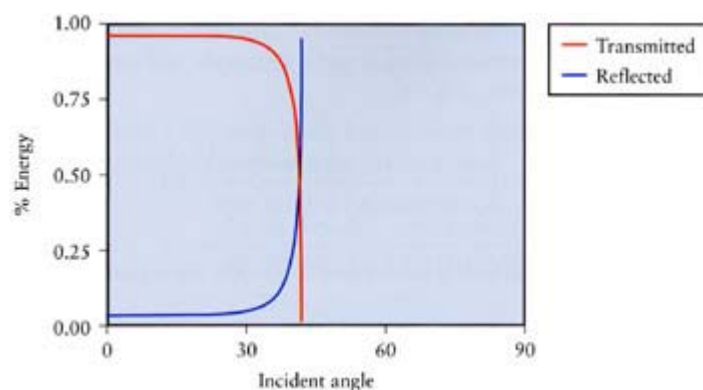


Figure 3.33: The averaged reflectance and transmittance curves for the glass-air interface.

To get a reasonable estimate on the Fresnel term, Cook and Torrance [COOK 1982] note that the values of reflectance at normal angles of incidence are more commonly found. You can get the Fresnel value at this angle (called F_0) and then calculate the angular dependence by back-substituting for the index of refraction using F_0 and then plugging this value for the index of refraction back into the original Fresnel equation. This gives you the Fresnel value as a function of the angle of incidence. In order to perform this calculation, we need to reformat the Fresnel equation for reflectance into a form derived by Blinn [BLINN 1977]. This is easier if we do it in steps. The parallel reflectance part of Fresnel's equation is

$$F_{r_{\parallel}} = \frac{\tan^2(\phi_i - \phi_r)}{\tan^2(\phi_i + \phi_r)}$$

If we get rid of the tangent using $\tan^2(\phi) + 1 = \sec^2(\phi) = 1/\cos^2(\phi)$, then we can rework the equation into terms that involve only cosines. The new equation is

$$F_{r_{\parallel}} = \frac{\cos^2(\phi_i + \phi_r)}{\cos^2(\phi_i - \phi_r)} + 1$$

And then we can use the $\cos(\phi \pm \theta) = \cos(\phi) \cos(\theta) \pm (-1) \sin(\phi) \sin(\theta)$ identity to break it down into terms involving θ_i and ϕ_t separately, and use Snell's law as $\eta_\lambda \sin(\phi_r) = \sin(\phi_i)$, where $\eta_\lambda = \eta_r/\eta_i$.

$$F_{r_{\parallel}} = \left[\frac{\cos(\phi_i) \cos(\phi_r) - \eta_\lambda \sin^2(\phi_r)}{\cos(\phi_i) \cos(\phi_r) + \eta_\lambda \sin^2(\phi_r)} \right]^2 + 1$$

Blinn introduced the term g in his paper to simplify the equations a bit, and we'll do the same here.

$$\text{Let } g = \eta_\lambda \cos(\phi_r) = \sqrt{\eta_\lambda^2 + \cos^2(\phi_i) - 1}$$

First multiplying through by $\eta_\lambda/\eta_\lambda$ and then replacing with g yields

$$F_{r_{\parallel}} = \left[\frac{g \cos(\phi_i) - \eta_\lambda^2 \sin^2(\phi_r)}{g \cos(\phi_i) + \eta_\lambda^2 \sin^2(\phi_r)} \right]^2 + 1$$

Then we can use the $\sin^2(\phi) + \cos^2(\phi) = 1$ identity to remove the sine term and replace it with a cosine term, which will allow a further replacement with the g term to further simplify the equation to

$$F_{r_{\parallel}} = \left[\frac{g \cos(\phi_i) - \eta_\lambda^2 + g^2}{g \cos(\phi_i) + \eta_\lambda^2 - g^2} \right]^2 + 1$$

Finally, we use the term $g^2 = \eta_\lambda^2 + \cos^2(\phi_i) - 1$ and factor out common terms to get

$$F_{r_{\parallel}} = \left[\frac{\cos(\phi_i)(g + \cos(\phi_i) - 1)}{\cos(\phi_i)(g - \cos(\phi_i) + 1)} \right]^2 + 1$$

Whew! What we've now got is an expression for a Fresnel term that is only a function of η_λ and ϕ_i . Let's do the same with the perpendicular reflection term.

$$F_{r_{\perp}} = \frac{\sin^2(\phi_i - \phi_r)}{\sin^2(\phi_i + \phi_r)}$$

Using the $\sin(\phi \pm \theta) = \sin(\phi) \cos(\theta) \pm \cos(\phi) \sin(\theta)$ identity breaks out the equation into terms involving ϕ_i and ϕ_r separately.

$$F_{r_{\perp}} = \left[\frac{\sin(\phi_i) \cos(\phi_r) - \cos(\phi_i) \sin(\phi_r)}{\sin(\phi_i) \cos(\phi_r) + \cos(\phi_i) \sin(\phi_r)} \right]^2$$

Then use Snell's law to get rid of the sine terms.

$$F_{r_{\perp}} = \left[\frac{\eta_\lambda \cos(\phi_r) - \cos(\phi_i)}{\eta_\lambda \cos(\phi_r) + \cos(\phi_i)} \right]^2$$

and finally replace the $\eta_\lambda \cos(\phi_r)$ with g to get

$$F_{r_{\perp}} = \left[\frac{g - \cos(\phi_i)}{g + \cos(\phi_i)} \right]^2$$

Finally, add the two pieces of the Fresnel reflection terms and average, and we get

$$F_r = \frac{1}{2} \left[\frac{g - \cos(\phi_i)}{g + \cos(\phi_i)} \right]^2 \left[\left[\frac{\cos(\phi_i)(g + \cos(\phi_i) - 1)}{\cos(\phi_i)(g - \cos(\phi_i) + 1)} \right]^2 + 1 \right]$$

Since

$$g = \sqrt{\eta_\lambda^2 + \cos^2(\phi_i) - 1}$$

we now have a Fresnel equation that's dependant upon only three values—the two indices of refraction and the incident angle.

Well, that's great, but why have we gone through all that? It's to further simplify the equation. If we look at the equation at normal incidence, when $\theta = 0$, then $\cos(\theta) = 1$ and $g = \eta_\lambda$, then most of the equation cancels out, and we're left with

$$\underline{F_r(\phi = 0) = \left(\frac{\eta_\lambda - 1}{\eta_\lambda + 1} \right)^2}$$

Ok, what's the advantage of this? Well, this lets us assume a value for η_λ when the light is normal to the surface. In other words, when we shine a light directly at the surface and look from the same direction (normal incidence), then this equation lets us solve for η_λ at this angle. Thus we can rearrange the equation to read

$$\eta_\lambda = \left(\frac{1 + \sqrt{F(\phi = 0)_r}}{1 - \sqrt{F(\phi = 0)_r}} \right)^2$$

You can then use the values of η_λ generated in this way to plug into the equation to generate F_r at other angles.

PHYSICALLY BASED SURFACE MODELS

The most widely used model of surfaces that are not perfectly smooth and uniform is the Cook-Torrance [COOK 1982] model. What they did was to assume that

- The geometry of the roughness of the surface is larger than that of the wavelength of the light.
- The geometry is considered to be made up of ∇ -shaped facets.
- The facets are randomly oriented.
- The facets are mirrorlike.

Using such a model, there are three different ways that a light ray can interact with the entire internal reflecting surface of the ∇ -shaped geometry (termed "microfacets") depending on the angle of the ∇ .

- The ray can reflect with no interference.
- The ray could be partially shadowed by other geometry.
- The ray could get blocked by part of the geometry.

These three cases are shown in [Figure 3.34](#).

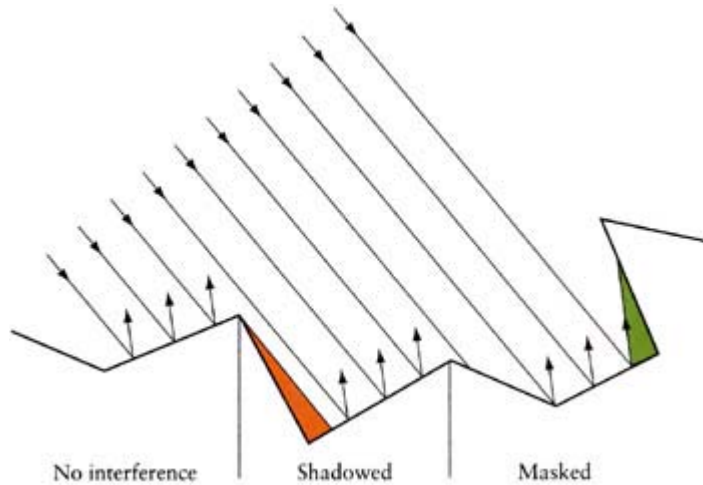


Figure 3.34: The three types of surface reflections that can occur in the Cook-Torrance surface model.

Roughness Distribution Function

When using this model, we need a way to specify the distribution of the slopes of the facets. This is termed the slope distribution function, D. Blinn [BLINN 1977] used a Gaussian distribution function to model the slope distribution.

$$D = ce^{-(\alpha/m)^2}$$

where c is some arbitrary constant and $\cos(\alpha) = \mathbf{n} \cdot \mathbf{h}$. The parameter m is the RMS slope parameter, for which smaller values (0.2) indicate a smooth surface, whereas larger values (0.8) indicate a rougher surface.

Cook and Torrance [COOK 1982] used a Beckmann distribution function, which they state can successfully model both rough and smooth dielectrics and conductors.

$$D = \frac{e^{-(\tan(\alpha)/m)^2}}{m^2 \cos^4(\alpha)}$$

This model has the advantage of not requiring a constant but just relies on one parameter m to specify the surface roughness.

There are other models such as the Trowbridge-Reitz model [TROWBRIDGE 1975], which models the microfacets as ellipsoids. You can even consider the Phong specular term to be a distribution function with the roughness specified as the exponential power value.

Geometric Attenuation Function

In addition to describing how the geometry of a rough surface is laid out, the Cook-Torrance model can be used to calculate the amount of light actually hitting on the microfacets. Blinn [BLINN 1977] has a very nice derivation of the geometry involved. Basically, there are three different cases to consider.

1. There is no interference in any of the light.
2. Some of the incoming light is blocked (shadowed).
3. Some of the reflected light is blocked (masked).

Blinn calculated the amount of light that would get blocked in each case. These are basically functions of the light direction and the facet normal. You then calculate these three values and select the minimum as the geometric attenuation function, G .

$$G = \min \left[1, (n \cdot v) \left(\frac{2n \cdot b}{v \cdot b} \right), (n \cdot l) \left(\frac{2n \cdot b}{v \cdot b} \right) \right]$$

THE BIDIRECTIONAL REFLECTANCE DISTRIBUTION FUNCTION (BRDF)

We're trying to accurately simulate reflectance from light traveling through a dielectric (something transparent, air, vacuum, etc.) and hitting the surface of a conductor (or something nontransparent) and reflecting off that surface.

The bidirectional reflectance distribution function (BRDF) takes into account the structure of the reflecting surface, the attenuation of the incident light by that structure, and the optical properties of the surface. It relates the incident light energy with the outgoing light energy. The incoming and outgoing light rays need to have not only their angles with the surface normal considered, but also the orientation of the rays with the surface orientation. This allows surfaces that reflect light differently, depending on their orientation around the surface normal (i.e., an isotropic surface), to be modeled.

[Figure 3.35](#) shows how the BRDF parameters relate to the surface normal and orientation of the surface. So a BRDF depends upon a total of four angles. Of course, you can simplify these by making assumptions about which terms are important to get the effect you want.

$$\rho(\theta_r, \phi_r, \theta_i, \phi_i)$$

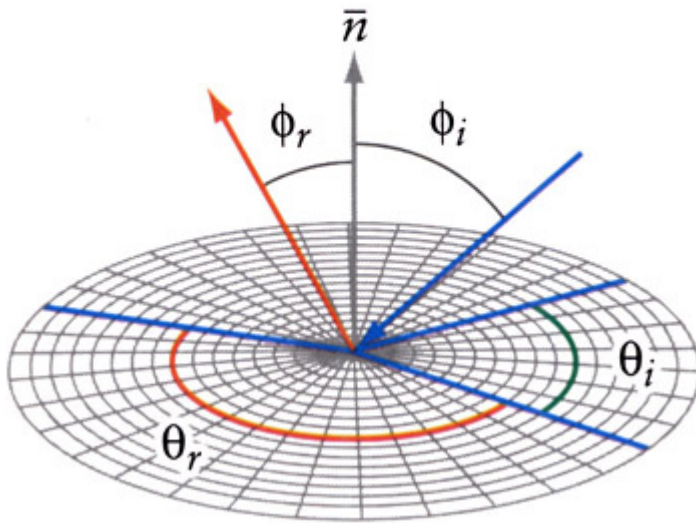


Figure 3.35: A BRDF is a function of an incident and reflection angle and two rotational angles.

BRDFs in Practice

The problem with BRDFs is that they are tough to implement in a practical manner. It's perfectly fine to have a BRDF that's expensive to calculate if you are doing some non-real-time imagery, but for performing BRDFs in shaders, you frequently have to simplify the model. The Lambertian model for diffuse reflection, for example, can be considered a BRDF that's just a constant. Phong's illumination model takes the (typical) approach of breaking a BRDF into diffuse and specular parts. It uses the Lambertian model for the diffuse and a $\cos\theta$ term for the specular, treating the specular BRDF as a function of the incident light vertex normal angle only.

However, we've already tried these models and found them wanting, so we'll take a look at a model for specular reflection developed by Blinn [BLINN 1977] that is still popular. Blinn proposed that, using the Cook-Torrance model for surface geometry, the specular reflection is composed of four parts.

- The distribution function, D .
- The Fresnel reflection law, F .
- The geometric attenuation factor, G .
- The fraction of the microfacets that are visible to the light and the viewer by a $(n \cdot v)(n \cdot l)$ term.

The BRDF specular function is then

$$\rho_s = \frac{FDG}{\pi(\hat{n} \cdot \hat{v})(\hat{n} \cdot \hat{l})}$$

Now we've already gone through the Fresnel term, the distribution function, and the geometric attenuation factor. You're free to make these as complicated as you like. For example, Blinn leaves the Fresnel term = 1, whereas Cook-Torrance doesn't.

It's possible to precompute the BRDF by judiciously choosing some of the parameters, and then generating one or more textures to account for the other terms. Typically, you might generate a texture where the u, v values of the texture are mapped to the $\mathbf{n} \cdot \mathbf{v}$ and $\mathbf{n} \cdot \mathbf{l}$ terms. For more information on BRDF factorization, you can refer to [ENGEL 2002] and [LENGYEL 2002]. [Figure 3.36](#) shows a BRDF reduced to those two terms.

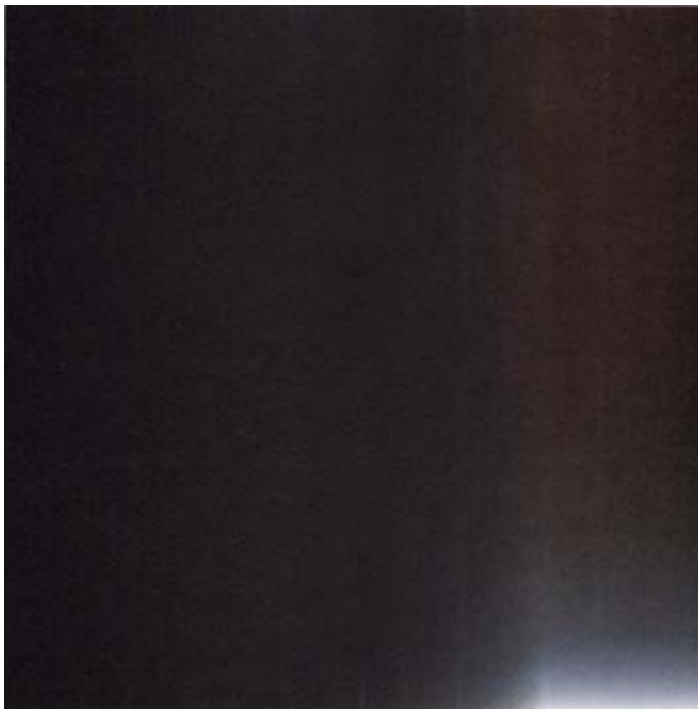


Figure 3.36: A precomputed BRDF texture using the values for gold.

Anisotropic Reflection

One interesting feature of the BRDF is that it supports *anisotropic reflection*, that is, reflection that varies in strength depending upon the orientation of the material's surface. Many surfaces exhibit this type of shading: hair, brushed metal, grooved surfaces (CDs, records), some fabrics, etc. Some of the more complex BRDFs can model the effects.

Poulin-Fournier [POULIN 1990] wrote one of the earliest papers on anisotropic reflection models. The Poulin-Fournier model replaced the randomly oriented ∇ -shaped grooves of the Torrance-Sparrow model with aligned cylindrical shapes (either grooves or protrusions). He, Torrance, Sillion, and Greenberg [HE 1991] proposed a model where they broke the specular term into two parts, a diffuse specular and a directional specular. Their full model is quite complex and can be used with polarized light. Even the unpolarized simplified equations are quite complex and would be nearly impossible to fit inside a

shader. In a latter paper [HE 1992], they address issues of computational speed, and in a time-honored tradition, compute a lookup table from which they can closely reproduce the values in their original paper. A different approach was taken by Ward [WARD 1992], who proposed finding the simplest empirical mode that would fit the data. This is still a very active area of research. [Figure 3.37](#) shows some work on anisotropic BRDFs from [ASHIKHMIN 2000]. Many papers in Siggraph Proceedings of recent years are worth looking into if you are interested in seeing further details and research. [BRDF] lists some online BRDF databases.

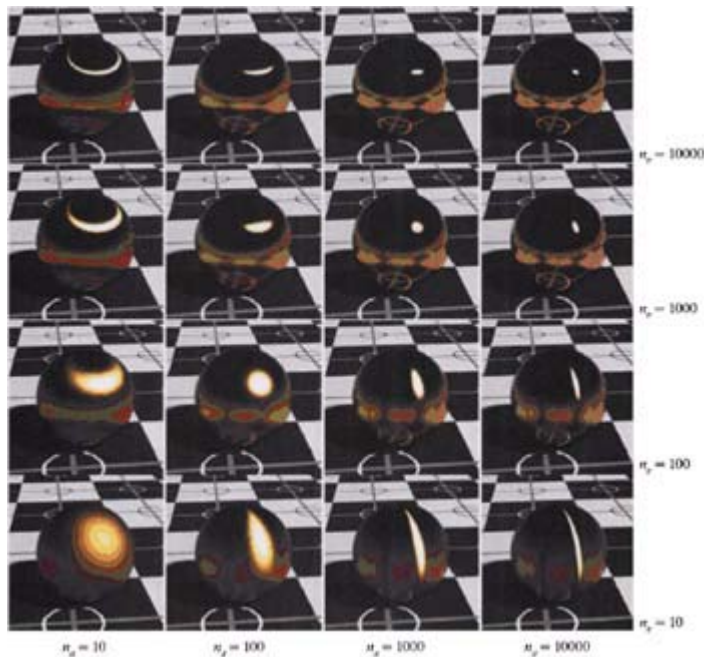


Figure 3.37: Illustrating different anisotropic features.

NONPHOTOREALISTIC RENDERING (NPR)

On the other end of the rendering spectrum is nonphotorealistic rendering (NPR). This style of rendering throws out most of the attempt to simulate real-world reflection models to achieve a different artistic goal. This can be the simplification of a scene to make it easier to understand, the modification of the scene to highlight an aspect of the scene, or the simulation of some other method of illustration such as watercolor or pen and ink drawing. Using shaders, it's possible to create your own method of illustration along any one of these styles. Unfortunately, much of the research cited in these examples was done before hardware shaders existed (though there are quite a few RenderMan shaders out there), so ready-made hardware shader examples are few and far between for some of these techniques, but they are starting to become available. The basic technique is to use the lighting equations for specular light (basically, some $\mathbf{n} \cdot \mathbf{l}$ term) to modulate the intensity of the effect produced. Look at any of the Siggraph Proceedings since 1995, and you'll typically find a couple of papers on these techniques. You can also find some good reviews of the latest research in [GOOCH 2001] and [STROTHOTTE 2002]. Craig Reynolds maintains an excellent Web page that's up to date, with a ton of links to various papers at

<http://www.red3d.com/cwr/npr/>. There's now an annual Nonphotorealistic Animation and Rendering Conference (NPAR). You can get more information about it at <http://www.npar.org>.

NPR Styles in 3D Rendering

NPR in 3D rendering can be roughly broken into a few different styles.

Painterly Rendering

This style is intended to simulate the results from brush-applied media. It's characterized by having a virtual brush apply media to the object's surface. Brush attributes include stroke weight, media loading, stroke attack and media attenuation over time, etc. Application of the media is sometimes done by calculating the particle flow of the media. You can find further examples of this style in [HAEBERLI 1990] and [MEIER 1996].

Pen and Ink, Engraving, and Line Art

This style is a high-contrast style where you're limited to an unvarying color intensity and can only adjust the line width. It's a simulation of using only an instrument of constant color intensity such as a pen to apply color. [Figure 3.38](#), a rendering of Frank Lloyd Wright's Robie House, uses the technique described in [WINKENBACH 1994].

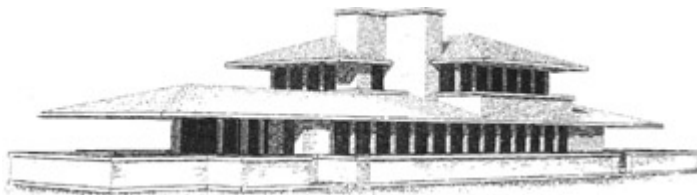


Figure 3.38: The pen-and-ink style.

A nice example of creating digital facial engraving from 2D images, which attempts to imitate traditional copperplate engraving using the techniques from [OSTROMOUKHOV 1999], can be seen in [Figure 3.39](#). This technique uses multiple layers to lay down the different areas of the face in order to provide a sharp demarcation of the different facial regions.

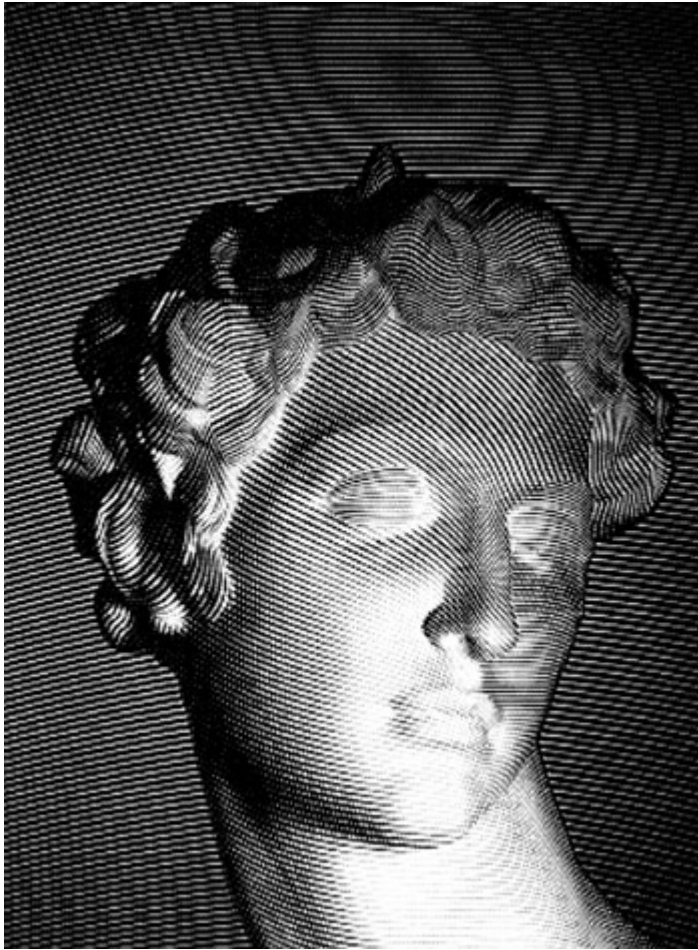


Figure 3.39: Engraving style [OSTROMOUKHOV 1999].

Sketching and Hatching

This style is similar to the preceding one but allows the use of tone as well. It imitates the look of charcoal and pencil with strokes that are hatched images scaled to approximate stroke density. Some nice examples of this work can be found in [WEBB 2002] and [PRAUN 2001]. The use of tonal art maps (TAMs) as textures that are used to control the degree of shading on an object gives some particularly nice results. The TAM in [Figure 3.40](#) shows how the first map starts with a light degree of shading and the shading is built up till a fairly dark map is reached.

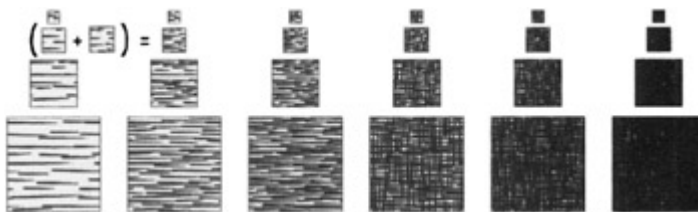


Figure 3.40: Tonal art maps [PRAUN 2001].

When these are applied in place of the traditional shading equations, you get some nice effects. [Figure 3.41](#) shows a teapot rendered with TAMs.



Figure 3.41: Rendered using tonal art maps.

Halftoning, Dithering, and Artistic Screening

These techniques use digital halftoning to get range and material. Rather than simulate a brush or pen stroke, the simulation is of the halftoning technique typically used by printers to achieve density through the use of dots or lines of varying density [OSTROMOUKHOV 1999], [STREIT 1999].

‘Toon Shading, Cel Shading, Outlining, and Stylized Rendering

This is probably one of the best-known areas of NPR. These styles use a combination of simple gradient shading and edge outlining to get some visually stunning results. A particularly striking example found in [GOOCH 1998] shows how using changes in hue and saturation to show changes in the orientation of the model's surface clarify structure. They present an alternative lighting model to traditional Phong shading. [Figure 3.42](#) shows the progression: (a) traditional Phong shading; (b) using their shading model with highlights and a cool-to-warm hue shift; (c) adding edge lines and highlights; (d) using Phong shading, edge lines, and two-colored lights.

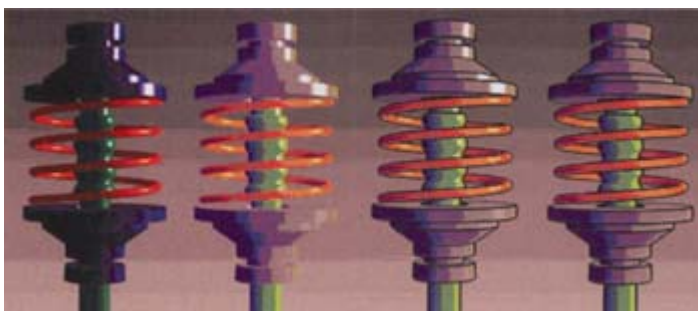


Figure 3.42: The stylized rendering [GOOCH 1998].

Another popular look is to use cel shading to get a cartoonlike feel. It's found a lot on computer-rendered scenes because it's fairly easy to generate automatically. A popular way of cel shading nontextured objects is to shade according to two areas (lit and unlit)-also called hard shading because of the hard delineation-or three areas (brightly lit, lit, and unlit). The vertex color is then set to one of these dark or bright colors according to the amount of light that's falling on the vertex. [Figure 3.43](#) is

from the game Jet Set Radio Future, and you can see two tones on the character's arms and three on the skates.



Figure 3.43: Cel shading found in the game Jet Set Radio Future. The cel shading on the character gives it a unique look.

Other Styles

Of course, there are many styles that don't quite fit into one of the previous types. These range from rendering fur and/or grass based upon a procedural texture placed near silhouette edges (graftals) by Kowalski [KOWALSKI 1999], which creates scenes rendered in a Dr. Seuss-like style. [Figure 3.44](#) shows the traditional 3D scene.



Figure 3.44: The 3D scene before the rendering of graftals [Kowalski 1999].

The edges of objects are determined and then an algorithm is used to graft on textured geometry, the edges are outlined, and all else is rendered with flat shading. The results look like [Figure 3.45](#). With some attention to graftal coherency, it's possible to actually move around the scene.



Figure 3.45: The 3D scene after the addition of graftals [Kowalski 1999].

Chapter 4: Introduction to Shaders

OVERVIEW

Shaders are intended to replace the fixed matte-plastic look that has been a part of consumer level graphics programs and games for the past few years. Shaders are intended to give greater freedom to artists and programmers so they may create a unique look and style for their application. Shaders are designed to run on the hardware (assuming a suitable interface is there) in real time, with control over just how involved and complicated the shader is in the hands of the programmer and artist. With shaders you can

- Perform basic geometry transformations
- Warp the geometry
- Blend or skin geometry
- Generate color information (specular and diffuse)
- Tween vertices between transformation matrices
- Generate texture coordinates
- Transform texture coordinates
- Size point sprites
- Use a custom illumination model
- Perform nonphotorealistic rendering
- Perform bump, environment, and specular mapping
- Perform your own texture-blending operations
- Clip pixels

In fact, your options are limited only by your ingenuity and the size of the shader buffer (which limits how complicated your shader can be).

Writing a shader is a lot like programming PCs back in the old days, when the *real* programmers didn't use any fancy higher-level language. They programmed in assembler. While that may cause you to groan, it really shouldn't since programming in assembly (or in shader instructions) makes programming very simple and obvious though a bit tedious at times. The nice thing is that you know *exactly* what's happening in your shader. Errors are fairly obvious, both in finding them in your code and in seeing them on the screen. The other advantage is that you can toss out anything that doesn't help you get the results you want. Shaders are generally very linear, and culling unnecessary instructions is the way you optimize shaders.

A Note on Higher Level Language Shaders

As I write this, DirectX 9 is in beta and has a specification for higher level shading language. NVIDIA has started promoting Cg, its higher level language. OpenGL 2.0 has a shading language. It's a confusing mess right now. The good thing is that everyone seems to agree that making it difficult is the worst thing that could happen. I chose not to include these languages because I think that sometime in mid- to late 2003, they will be standardized and reworked in such a way that you don't have to worry about shader versions, instruction limits, register assignments, etc. You should just program what you want, and the shader compiler will do the right thing, hiding the limits of the underlying hardware as much as possible. However, as anyone who's written time-critical software knows, sometimes you just have to fall back to assembler. Keep an eye on the <http://www.directx.com> Web site for updates.

SHADERS AND THE EXISTING GRAPHICS PIPELINE

There are currently (as of DirectX 9) two flavors of shaders: vertex and pixel shaders. Vertex shaders operate on vertices, or in more precise terms, the output is assumed to be a vertex in homogenous clip space coordinates. A vertex shader (usually) produces one vertex as its output, (usually) from a set of vertices provided as input. A pixel shader produces the color of the pixel as its sole output, (usually) from color and/or texture coordinates provided as inputs.

I've added the (usually)'s since this is the way that the shaders are intended to be used when replacing the functionality of the fixed-function pipeline (FFP). The point is that when you're using either a vertex shader or a pixel shader (but not both), you're limited by the way that you can provide data to the other part. However, when you're using both a vertex and pixel shader, the vertex shader output is still sampled by whatever interpolating method is set as a render state (flat or Gouraud), and those values are provided as input for the pixel shader. There's nothing wrong with using this pipeline as a conduit for your own data. You could generate some intermediate values in the vertex shader (along with a valid vertex position), and those values will make their way to the pixel shader. The only thing the pixel shader has to do is specify the final color for the pixel. We'll go over this in depth when we talk about using shaders with the FFP.

[Figure 4.1](#) shows how the vertex and pixel shaders fit into the current hardware pipeline. It's been designed so that the current graphics pipeline can be replaced by the new functionality without changing the overall flow through the pipeline. Pixel and vertex shaders can be independently implemented; there is no requirement that they both be present. In fact, the first generation of hardware sometimes only implemented pixel shaders. Vertex shaders were available only as part of the software driver—a very highly optimized part, but still one that doesn't run on the hardware. This illustration also shows the addition of the higher order primitive section. This is going to become an ever-increasing part of the graphics pipeline in the near future since a higher order primitive means less data sent over the graphics memory bus. Why send over a few thousand perturbed vertices when you can just send over a few dozen perturbed control points and have the hardware generate the vertices for you?

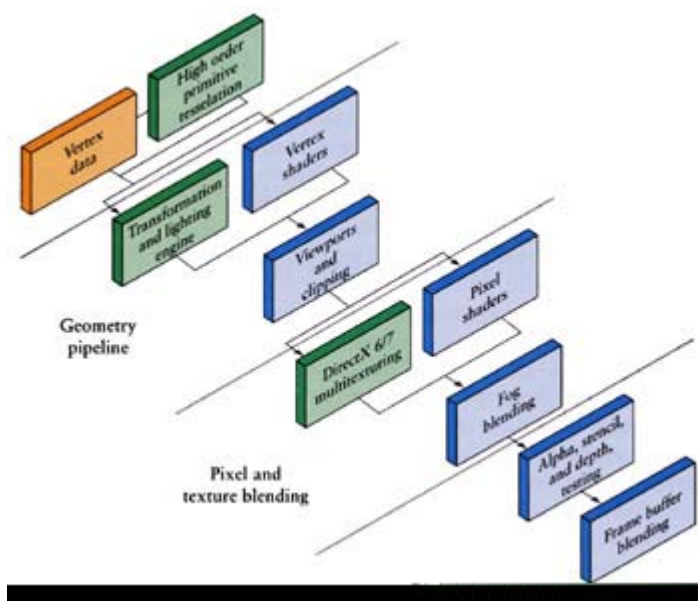


Figure 4.1: Vertex and pixel shaders replace sections of the fixed function pipeline.

VERTEX SHADERS: TECHNICAL OVERVIEW

Vertex shaders replace the TnL (texture and lighting) part of the Direct3D and OpenGL pipeline. This pipeline is still available and is now referred to (since DirectX8) as the fixed-function pipeline since it's not programmable as shaders are. Since vertex shaders are intended to replace the FFP TnL part of the pipeline, vertex shaders must produce essentially the same output as the FFP from the same input. Now among other things, the pipeline depends upon the current rendering state—thus it's possible to perform calculations on things like fog or textures. But if fog or texturing isn't turned on, the values that are generated will never be used. It is important to note that a vertex shader's *only* job is to take a set of input vertices and generate an output set of vertices. There is a one-to-one correspondence between an input vertex and an output vertex. A vertex shader *cannot* change the number of vertices since it's called once for each vertex in a primitive.

The input to a vertex shader is one or more vertex streams plus any other state information (like current textures) as shown in [Figure 4.2](#). The output is a new vertex position in clip coordinates (discussed in the next section), and any other information that the shader provides, like vertex color, texture coordinates, fog values, and the like.

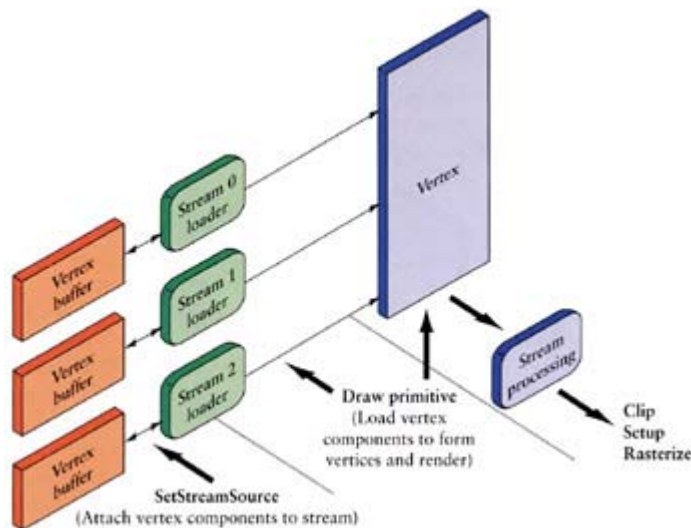


Figure 4.2: DirectX 8 introduced vertex streams.

PIXEL SHADERS: TECHNICAL OVERVIEW

Whereas vertex shaders replace the FFP in the traditional rendering pipeline, pixel shaders replace the pixel-blending section of the multitexture section of the pipeline. To understand how pixel shaders operate, you should be familiar with the dualistic nature of the texture pipeline. Traditionally, two paths are running simultaneously in the hardware—the color pipe (also called the vector pipe) and the alpha pipe (also called the scalar pipe). These two pathways handle the color and alpha operations of the texture processing unit. Frequently, you will have set up a mode so that the color operations are performed with one set of parameters, whereas the alpha operations are performed with a different set. At the end, the results are combined into a resulting rgba value. You can see this in [Figure 4.3](#).

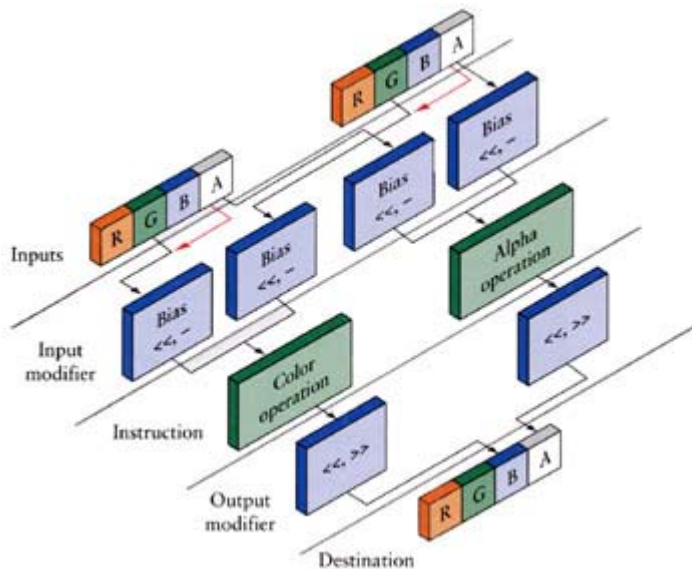


Figure 4.3: Pixel shader hardware can perform different operations simultaneously on a color vector and an alpha scalar.

Although the traditional texture pipeline allows you to specify a cascade of operations on sequential textures, pixel shaders allow you to specify these with more generality. However, the dual nature of the pipeline is still there. In fact, you'll probably be spending some time fine-tuning your pixel shaders so that you can get color and alpha operations to "pair," that is, to run simultaneously in the hardware.

Another dualistic nature of pixel shaders is they have two separate types of operations: arithmetic and texturing. Arithmetic operations are used to generate or modify color or texture coordinate values. The texture operations are operations to fetch texture coordinates or sample a texture using some texture coordinates. No matter the type of operations (coloring, texturing, or a blend), the output of the pixel shader is a single **rgba** value.

VERTEX SHADERS, PIXEL SHADERS, AND THE FIXED FUNCTION PIPELINE

So you might be wondering, how does all this stuff fit together? Well there are two cases to consider. If you're using a shader in conjunction with the FFP, then you'll have to consider what the FFP operations are. The FFP vertex operations are going to provide your pixel shader with two color registers: one diffuse, one specular. The FFP pixel shader is going to *expect* two color registers as input: one diffuse, one specular. (Note that using specular is a render state.) It gets interesting when you write your own vertex and pixel shader and ignore the FFP altogether.

Your vertex shader needs to provide a valid vertex position, so you'll need to perform the transformation step and provide a valid vertex position in the oPos register. Other than that, you've got the fog, point size, texture coordinates, and of course, the two color registers. (Note that the FFP pixel operations expect only one set of texture coordinates and the two color values.) Since your pixel shader operates on these values, you are free to stick any value into them you want (within the limits of the pixel shader precision). It's simply a way of passing data directly to the pixel shader from the vertex shader.

However, you should be aware that texture coordinates will always be perspective correct interpolated from the vertex positions. The fog value will always be interpolated as well. The two color values will be interpolated *only* if the shading mode is Gouraud. The color interpolation in this case will also be perspective correct. Setting the shading mode to flat and placing data in the color registers is the preferred method of getting values unchanged to the pixel shaders.

Specular color is added by the pixel shader. There is no rendering state for specular when using pixel shaders. Fog, however, is still part of the FFP, and the fog blend is performed after the pixel shader executes.

VERTEX SHADERS

To quote the Microsoft DirectX 8.0 documentation,

Vertex processing performed by vertex shaders encompasses only operations applied to single vertices. The output of the vertex processing step is defined as individual vertices, each of which consists of a clip-space position (x, y, z, and w) plus color, texture coordinate, fog intensity, and point size information. The projection and mapping of these positions to the viewport, the assembling of multiple vertices into primitives, and the clipping of primitives is done by a subsequent processing stage and is not under the control of the vertex shader.

What does this mean? Well, it means that whatever your input from the vertex streams (because you will have specified these values), the output from a vertex shader for a single pass will be

- One single vertex in clip-space coordinates
- Optional color values (specular and diffuse)
- Optional texture coordinates
- Optional fog intensity
- Optional point sizing

This is shown in [Figure 4.4](#).

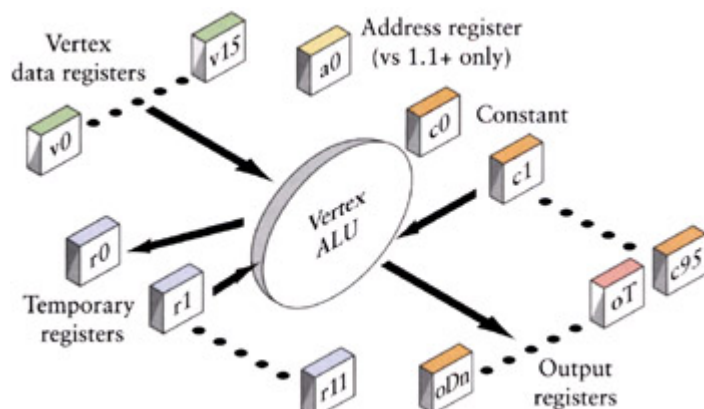


Figure 4.4: Vertex operations take input from vertex stream(s) and constants, and place the results

in output registers.

So, at the very least your minimal vertex shader needs to take the object's vertex positions and transform them into clip-space coordinates. The optional parts are determined by the rendering state. Since the object has to be rendered with *some* properties, you'll have texture coordinates and/or color specifications as output. But the constant and absolute requirement for every vertex shader is that you provide vertex positions in clip-space coordinates. Let's start with that.

Our first vertex shader will do just that, transform the untransformed input vertex to clip space. There are two assumptions in the following code. The first is that the input vertex position shows up in shader register v0. The actual register number depends on the vertex shader input declaration, which tells the shader the format of the input stream. The second is that we've stored the world-view-projection matrix in the vertex shader constants. Given those assumptions, we can write the following shader:

```
// v0      -- position
// c0-3    -- world/view/proj matrix

// the minimal vertex shader
//transform to clip space

dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3
```

This shader uses four of the dot product shader instructions to perform a matrix multiply using the rows of the world-view-projection matrix sequentially to compute the transformed x, y, z, and w values. Each line computes a value and stores the scalar result into the elements of the output position register. Note that this is usually how the first few lines of your shader will look. It's possible that you might not need to perform the matrix multiply (e.g., if you transform the vertex before the shader is run). In any case, the minimal valid vertex shader must set all four elements of the oPos register.

There are some tricky issues with performing transformations, so let's review what this section of the shader has to do and what pitfalls there are. Along the way, we'll discuss some DirectX C++ code to perform the setup the shader requires.

Transformations and Projections

Typically, you'll start out with your object in what are called "object" or "model" coordinates—that is, these are the vertex positions that the model was originally read in with. Most of the projects I've worked on have created them with the object centered about the origin. Static objects might get an additional step where their vertices are transformed to some location in world space and then never touched again—creating a group of trees from a single set of original tree vertex coordinates, for example. Each new tree would take the values for the original tree's position and then use a unique transformation matrix to move the tree to its unique position.

So, for every vertex in our tree example, we need to transform it from its local model space into the global world space coordinate system. That's done pretty simply with a matrix multiplication. For every point,

$$v_{\text{output}} = v_{\text{input}} T$$

So, this set of vertices in world coordinates is what we assume we are starting with. What we need to do is get from world coordinates to clip coordinates.

The Trip from World to Clip Coordinates

The trip from world space to clip space is conceptually done in three separate steps. The first step is to actually get the model into the global, world coordinate system. This is done by multiplying the object's vertices by the world transformation matrix. This is the global coordinate system that makes all objects share the same coordinate system. If you are used to OpenGL, this is called the *model* transformation.

The second step is to get the object's vertices into the *view* coordinate system. This is done by multiplying by the view transformation. The result of this step is to place the object's vertices in the same space as the viewer, with the viewpoint at the origin and the gaze direction down the z axis. (DirectX uses the positive z axis, whereas OpenGL uses the negative.) Once the vertices have been transformed, they are said to be in *eye* space or *camera* space.

It should be noted that typically an optimization step is to concatenate the world and view matrices into a single matrix since OpenGL doesn't have a separate world matrix and instead makes you premultiply the viewing parameters into its modelview matrix. The same effect can be had in DirectX by leaving the world matrix as the identity and using just the view matrix as the equivalent of OpenGL's modelview. Remember, the object is not only to get the results we want but also to do it in as few steps as possible.

Now you might remember that there was a *zNear* and a *zFar* value and a field-of-view parameter that are used in the setup of the viewing parameters. Well, here's where they get used. Those values determined the view frustum—the truncated pyramid (for perspective projection) in which only those things that are inside get rendered. What actually gets calculated from those values is the *projection* matrix. This matrix takes those values and transforms them into a unit cube. An object's coordinates are

said to be in NDC (normalized device coordinates) or, more practically, *clip* space. For a perspective projection, this has the effect of making objects farther away from the viewpoint (i.e., the origin in view coordinates) look smaller. This is the effect you want, that objects farther away get smaller. The part of this transformation that produces more problems is not that this is a linear transformation in the *z* direction, but that (depending upon how wide the field of view is) the actual resolution of objects in the *z* direction gets less the closer you get to the *zFar* value. In other words, most of the resolution of the depth value (the *z* value) of your objects in clip space is concentrated in the first half of the viewing frustum. Practically, this means that if you set your *zFar/zNear* ratio too high, you'll lose resolution in the back part of the viewing volume and the rendering engine will start rendering pixels for different objects that overlap, sometimes switching on a frame/frame basis, which can lead to *sparkling* or *z-fighting*.

The output of the view transformation is that everything now sits in relation to a unit cube centered about the origin. The cube in DirectX has one corner located at $(-1, -1, -1)$ and the other at $(1, 1, 1)$. Everything inside this cube will get rendered; everything outside the cube will get clipped. The nice thing about not writing your own rendering engine is what to do about those objects that cross the boundary. The rendering engine has to actually create new vertices where the object crosses the boundary and render only up to those locations. (These vertices are created from the interpolated values provided by the FFP or the vertex shader—that is, the vertex shader isn't run for these intermediate vertices.) This means that it has to also correctly interpolate vertex colors, normals, texture coordinates, etc. A job best left up to the rendering engine.

To summarize: We have three different matrix transformations to get from world coordinate space to clip space. Since, for a single object, you usually don't change the world, view, or projection matrices, we can concatenate these and get a single matrix that will take us from model space directly to clip space.

$$M^{\text{WorldViewProj}} = (M^{\text{World}})(M^{\text{View}})(M^{\text{Proj}})$$

We recalculate this matrix every time one of these original matrices changes—generally, every frame for most applications where the viewpoint can move around—and pass this to the vertex shader in some of the constant vertex shader registers.

Now let's look at some actual code to generate this matrix. In the generic case, you will have a world, view, and projection matrix, though if you're used to OpenGL, you will have a concatenated world-view matrix (called the modelview matrix in OpenGL). Before you load the concatenated world-view-projection matrix (or WVP matrix), you'll have to take the transpose of the matrix. This step is necessary because to transform a vertex inside a shader, the easiest way is to use the dot product instruction to do the multiplication. In order to get the correct order for the transformation multiplication, each vertex has to be multiplied by a column of the transformation matrix. Since the dot product operates on a single register vector, we need to transpose the matrix to swap the rows and columns in order to get the correct ordering for the dot product multiplication.

We do this by creating a temporary matrix that contains the WVP matrix, taking its transpose, and then passing that to the `SetVertexShaderConstant()` function for DirectX 8, or the `SetVertexShaderConstantF()` function for DirectX 9.

```
// DirectX 8 !
D3DXMATRIX      trans;

// create a temporary matrix holding WVP. Then
// transpose and store it
D3DXMatrixTranspose( &trans ,
                    &(m_matWorld * m_matView * m_matProj) );
// Take the address of the matrix (which is 4
// rows of 4 floats in memory. Place it starting at
// register r0 for a total of 4 registers.
m_pd3dDevice->SetVertexShaderConstant(
    0,          // what register # to start at
    &trans,     // address of the value(s)
    4 );       // # of 4-element values to load
```

Once that is done, we're almost ready to run our first vertex shader. There are still two items we have to set up—the vertex input to the shader and the output color. Remember that there are usually two things that the vertex shader has to output—transformed vertex positions and some kind of output for the vertex—be it a color, a texture coordinate, or some combination of things. The simplest is just setting the vertex to a flat color, and we can do that by passing in a color in a constant register, which is what the next lines of code do.

```
// DirectX 8!
// set up a color
float teal [4] = [0.0f,1.0f,0.7f,0.0f]; //rgba ordering

// specify register r12
m_pd3dDevice->SetVertexShaderConstant(
```

```
12,    // which constant register to set  
teal,  // the array of values  
1 );   // # of 4-element values
```

Finally, you need to specify where the input vertex stream will appear. This is done using the `SetStreamSource()` function, where you select which vertex register(s) the stream of vertices shows up in. There's a lot more to setting up a stream, but the part we're currently interested in is just knowing where the raw vertex (and later normal and texture coordinate) information will show up in our shader. For the following examples, we'll assume that we've set up vertex register 0 to be associated with the vertex stream. Most of the vertex shader code you'll see will have the expected constant declarations as comments at the top of the shader.

So with the vertex input in `v0`, the WVP matrix in `c0` through `c3`, and the output color in `c12`, our first self-contained vertex shader looks like this.

```
// v0      -- position  
// c0-3    -- world/view/proj matrix  
// c12     -- the color value  
  
// a minimal vertex shader  
// transform to clip space  
dp4 oPos.x, v0, c0  
dp4 oPos.y, v0, c1  
dp4 oPos.z, v0, c2  
dp4 oPos.w, v0, c3  
  
// write out color  
mov oD0, c12
```

Transforming Normal Vectors

In order to perform lighting calculations, you need the normal of the vertex or the surface. When the vertex is transformed, it's an obvious thing to understand that the normals (which I always visualize as these little vectors sticking out of the point) need to be transformed as well; after all, if the vertex rotates,

the normal must rotate as well! And generally you'll see applications and textbooks using the same transformation matrix on normals as well as vertices, and in *most* cases, this is ok. However, this is true only if the matrix is orthogonal—that is, made up of translations and rotations, but no scaling transformations. Let's take a shape and transform it and see what happens so that we can get an idea of what's happening. In [Figure 4.5](#), we show the normals on the surface.

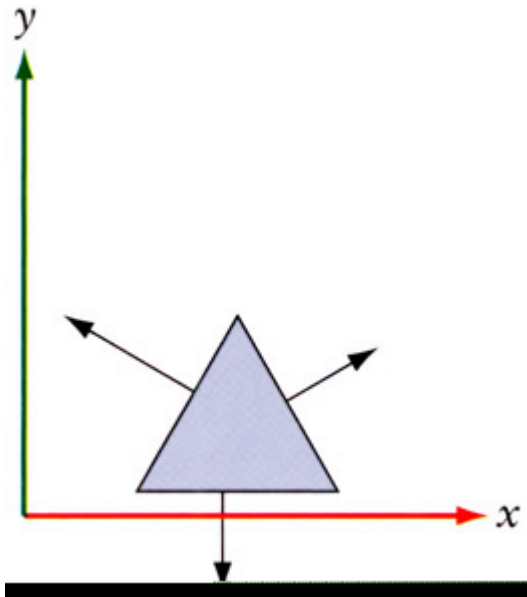


Figure 4.5: Surface normals prior to an affine transformation.

If we apply a general transformation matrix to this shape and the normals as well, we'll get the shape shown in [Figure 4.6](#).

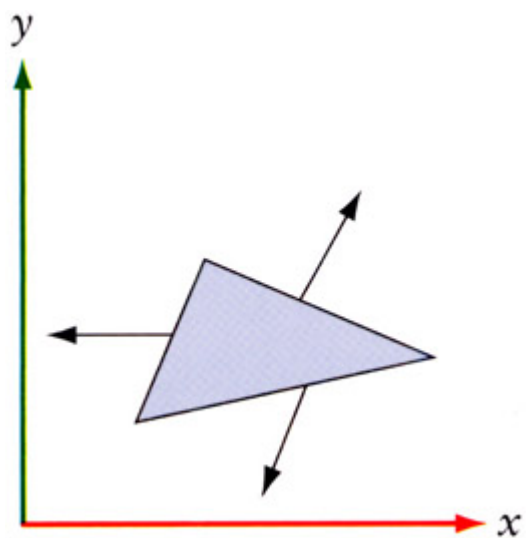


Figure 4.6: Surface normals following an affine transformation.

Although the shape may be what we desired, you can clearly see that the normals no longer represent what they are supposed to—they are no longer perpendicular to the surface and are no longer of unit length. You could recalculate the normals, but since we just applied a transformation matrix to our vertices, it seems reasonable that we should be able to perform a similar operation to our normals that correctly orients them with the surface while preserving their unit length.

If you're interested in the math, you can look it up [TURKOWSKI 1990]. But basically, it comes down to the following observations. When you transform an object, you'll be using one of these types of transformations.

1. Orthogonal transformation (rotations and translations): This tends to be the most general case since most objects aren't scaled. In this case, the normals can be transformed by the same matrix as used for vertices. Without any scaling in it, the transpose of a matrix is the same as its inverse, and the transpose is easier to calculate, so in this case, you'd generally use the transpose as a faster-to-calculate replacement for the inverse.
2. Isotropic transformation (scaling): In this case, the normals need to be transformed by the inverse scaling factor. If you scale your objects only at load time, then an optimization would be to scale the normals after the initial scaling.
3. Affine transformation (any other you'll probably create): In this case, you'll need to transform the normals by the transpose of the inverse of the transformation matrix. You'll need to calculate the inverse matrix for your vertices, so this is just an additional step of taking the transpose of this matrix.

In fact, you can get away with computing just the transpose of the adjoint of the upper 3×3 matrix of the transformation matrix [RTR 2002].

So, in summary,

- If the world/model transformations consist of only rotations and translations, then you can use the same matrix to transform the normals.
- If there are uniform scalings in the world/model matrix, then the normals will have to be renormalized after the transformation.
- If there are nonuniform scalings, then the normals will have to be transformed by the transpose of the inverse of the matrix used to transform the geometry.

If you know that your WVP matrix is orthogonal, then you can use that matrix on the normal, and you don't have to renormalize the normal.

```
// a vertex shader for orthogonal transformation matrices
// v0      -- position
// v3      -- normal
```

```

// c0-3  -- world/view/proj matrix

// transform vertex to clip space
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

// transform normal using same matrix
dp3 r0.x, v3, c0
dp3 r0.y, v3, c1
dp3 r0.z, v3, c2

```

On the other hand, if you have any other kind of matrix, you'll have to provide the inverse transpose of the world matrix in a set of constant registers in addition to the WVP matrix. After you transform the normal vector, you'll have to renormalize it.

```

// a vertex shader for non-orthogonal
// transformation matrices
// v0 -- position
// v3 -- normal
// c0-3 -- world/view/proj matrix
// c5-8 -- inverse/transpose world matrix

// transform vertex to clip space
dp4 oPos.x, v0, c0
dp4 oPos.y, v0, c1
dp4 oPos.z, v0, c2
dp4 oPos.w, v0, c3

// transform normal

```

```

dp3 r0.x, v3, c5
dp3 r0.y, v3, c6
dp3 r0.z, v3, c7

// renormalize normal
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0, r0, r0.w

```

There are a series of macroinstructions (such as $m4 \times 4$) that will expand into a series of dot product calls. These macros are there to make it easy for you to perform the matrix transformation into clip space. *Do not* make the mistake of using the same register for source and destination. If you do, the macro will happily expand into a series of dot product calls and modify the source register element by element for each dot product rather than preserving the original register.

Vertex Shader Registers and Variables

Shader registers are constructed as a vector of four IEEE 32-bit floating point numbers as illustrated in [Figure 4.7](#).

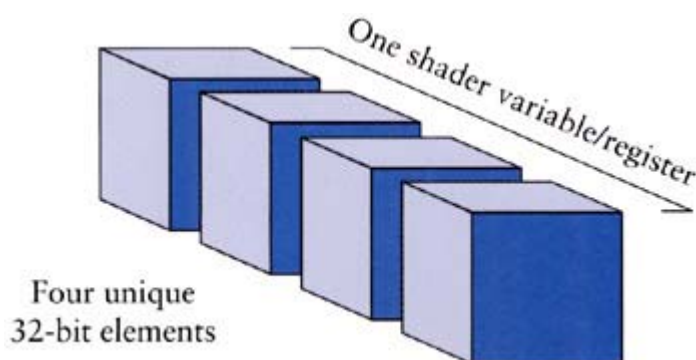


Figure 4.7: Vertex shader (and 2.0+ pixel shader) registers are made of four float vector elements.

While hardware manufacturers are free to implement their hardware as they see fit, there are some minimums that they have to meet. Since vertex shaders are going to be passed back into the pipeline, you can expect that the precision will match that of the input registers, namely, closely matching that of IEEE 32-bit float specification with the exceptions that some of the math error propagation rules (NaN, INF, etc.) are simplified. On those output registers that are clamped to a specific range, the clamping does not occur till the shader is finished. Note that you'll get very familiar behavior from vertex shader

math, which can lull you into a sense of security when you start dealing with the more limited math precision of pixel shaders, so be careful!

PIXEL SHADERS

A pixel shader takes color, texture coordinate(s), and selected texture(s) as input and produces a single color rgba value as its output. You can ignore any texture states that are set. You can create your own texture coordinates out of thin air. You can even ignore any of the inputs provided and set the pixel color directly if you like. In other words, you have near total control over the final pixel color that shows up. The one render state that will change your pixel color is fog. The fog blend is performed after the pixel shader has run.

Inside a pixel shader, you can look up texture coordinates, modify them, blend them, etc. A pixel shader has two color registers as input, some constant registers, and texture coordinates and textures set prior to the execution of the shader through the render states ([Figure 4.8](#)).

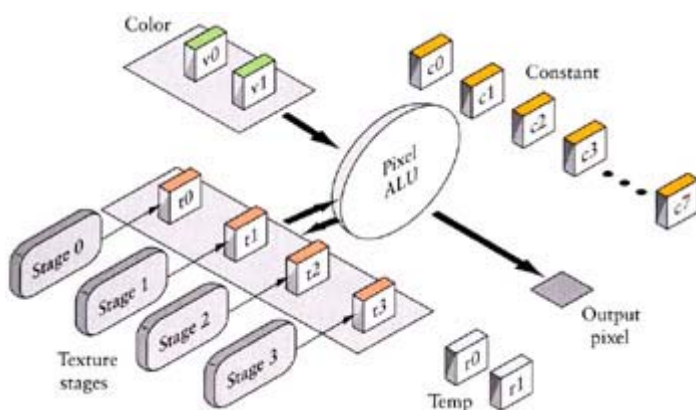


Figure 4.8: Pixel shaders take color inputs and texture coordinates to generate a single output color value.

Using pixel shaders, you are free to interpret the data however you like. Since you are pretty much limited to sampling textures and blending them with colors, the size of pixel shaders is generally smaller than vertex shaders. The variety of commands, however, is pretty great since there are commands that are subtle variations of each other.

In addition to the version and constant declaration instructions, which are similar to the vertex shader instructions, pixel shader instructions have texture addressing instructions and arithmetic instructions.

Arithmetic instructions include the common mathematical instructions that you'd expect. These instructions are used to perform operations on the color or texture address information.

The texture instructions operate on a texture or texture coordinates that have been bound to a texture stage. There are instructions that can sample a texture or you can assign a texture to a stage using the `SetTexture()` function. This will assign a texture to one of the texture stages that the device currently

supports. You control how the texture is sampled through a call to `SetTextureStageState()`. The simplest pixel shader we can write that samples the texture assigned to stage 0 would look like this.

```
// a pixel shader to use the texture of stage 0
ps.1.0

// sample the texture bound to stage 0
// using the texture coordinates from stage 0
// and place the resulting color in t0
tex t0

// now copy the color to the output register
mov r0, t0
```

There are a large variety of texture addressing and sampling operations that give you a wide variety of options for sampling, blending, and other operations on multiple textures.

Conversely, if you didn't want to sample a texture but were just interested in coloring the pixel using the iterated colors from the vertex shader output, you could ignore any active textures and just use the color input registers. Assuming that we were using either the FFP or our vertex shader to set both the diffuse and specular colors, a pixel shader to add the diffuse and specular colors would look like this.

```
// a pixel shader to just blend diffuse and specular ps.1.0

// since the add instruction can only access
// one color register at a time, we need to
// move one value into a temporary register and
// perform the add with that temp register
mov r0, v1
add r0, v0, r0
```

As you can see, pixel shaders are straightforward to use, though understanding the intricacies of the individual instructions is sometimes a challenge.

Unfortunately, since pixel shaders are so representative of the hardware, there's a good deal of unique behavior between shader versions. For example, almost all the texture operations that were available in pixel shader 1.0 through 1.3 were replaced with fewer but more generic texture operations available in pixel shader 1.4 and 2.0. Unlike vertex shaders (for which there was a good implementation in the software driver), there was no implementation of pixel shaders in software. Thus since pixel shaders essentially expose the hardware API to the shader writer, the features of the language are directly represented by the features of the hardware. This is getting better with pixel shaders 2.0, which are starting to show more uniformity about instructions.

DirectX 8 Pixel Shader Math Precision

In pixel shader versions 2.0 or better (i.e., DirectX 9 compliant), the change was made to make the registers full precision registers. However, in DirectX 8, that minimum wasn't in place. Registers in pixel shaders before version 2.0 are *not* full 32-bit floating point values ([Figure 4.9](#)). In fact, they are severely limited in their range. The minimum precision is 8 bits, which usually translates to an implementation of a fixed point number with a sign bit and 7-8 bits for the fraction. Since the complexity of pixel shaders will only grow over time, and hence the ability to do lengthy operations, you can expect that you'll rarely run into precision problems unless you're trying to do something like perform multiple lookup operations into large texture spaces or performing many rendering passes. Only on older cards (those manufactured in or before 2001) or inexpensive cards will you find the 8-bit minimum. As manufacturers figure out how to reduce the size of the silicon and increase the complexity, they'll be able to squeeze more precision into the pixel shaders. DirectX 9 compliant cards should have 16- or 32-bits of precision.

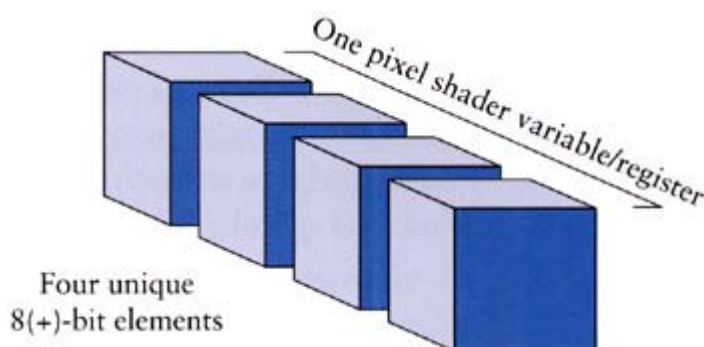


Figure 4.9: Pixel shader registers prior to version 2.0 consisted of a four-element vector made of (at least) 8-bit floating point elements; 2.0 pixel shaders are 32-bit floats.

As the number of bits increases in the pixel shader registers, so will the overall range. You'll need to examine the `D3DCAPS8.MaxPixelShaderValue` Or `D3DCAPS9.PixelShader1xMaxValue` capability value in order to see the range that pixel registers are clamped to. In DirectX 6 and DirectX 7,

this value was 0, indicating an absolute range of [0,1]. In later versions of DirectX, this value represented an absolute range, thus in DirectX 8 or 9, you might see a value of 1, which would indicate a range of $[-1,1]$, or 8, which would indicate a range of $[-8,8]$. Note that this value typically depends not only on the hardware, but sometimes on the driver version as well!

No, No, It's Not a Texture, It's just Data

One of the largest problems people have with using texture operations is getting over the fact that just because something is using a texture operation, it doesn't have to be texture data. In the early days of 3D graphics, you could compute lighting effects using hardware acceleration only at vertices. Thus if you had a wall consisting of one large quadrangle and you wanted to illuminate it, you had to make sure that the light fell on at least one vertex in order to get some lighting effect. If the light was near the center, it made no difference since the light was calculated only at the vertices, and then linearly interpolated from there—thus a light at the center of a surface was only as bright as at the vertices. The brute force method of correcting this (which is what some tools like RenderMan do) is to tessellate a surface till the individual triangles are smaller than a pixel will be, in effect turning a program into a pixel accurate renderer at the expense of generating a huge number of triangles.

It turns out that there already is a hardware accelerated method of manipulating pixels—it's the texture rendering section of the API. For years, people have been doing things like using texture to create pseudolighting effects and even to simulate perturbations in the surface lighting due to a bumpy surface by using texture maps. It took a fair amount of effort to get multiple texture supported in graphics hardware, and when it finally arrived at a fairly consistent level in consumer-level graphics cards, multitexturing effects took off. Not content with waiting for the API folks to get their act together, the graphics programmers and researchers thought of different ways to use layering on multiple texture to get the effects they wanted. It's this tradition that pixel shaders are built upon. In order to get really creative with pixel shaders, you have to forget the idea that a texture is an image. A texture is just a collection of 1D, 2D, or 3D matrix of data. We can use it to hold an image, a bump map, or a lookup table of any sort of data. We just need to associate a vertex with some coordinate data and out in the pixel shader will pop those coordinates, ready for us to pluck out our matrix of data.

Chapter 5: Shader Setup in DirectX

OVERVIEW

Dealing with network executives is like being nibbled to death by ducks.

--Eric Sevareid

This section assumes that you have had some experience with using DirectX, at least with getting a Direct3D device loaded. (A device you initialized to interface to the underlying hardware.) You should also understand the discussion in [Chapter 2](#) on exactly what we mean by a vertex. DirectX 8 introduced

the concept of a vertex element stream, so now all data is fed to the rendering engine via one or more vertex element streams. The basics of getting ready to use shaders (or the FFP rendering) is as follows:

1. Create a D3D device.
2. Check to make sure all features you need are supported in that device.
3. Create your vertex data structure(s) (vertex element).
4. Create the vertex shader interface declaration(s).
5. Create your vertex buffer(s) and fill it with data.
6. Create and assemble your vertex and pixel shaders.
7. Associate a vertex stream with the vertex buffers.
8. Select the vertex and pixel shaders into the device.
9. Set any shader constants.
10. Render your data.

Typically, you'll take steps 1–6 at initialization time, though if your program goes into different segments (e.g., loading different levels in a game), steps 3–6 might occur at this time. Steps 7–10 might occur in your rendering loop, possibly iterating over steps 8–10 if you have multiple passes on the same data. If you use effects files (supported by the DirectX utility library), you can store shaders as part of the .X files along with your data and get the shader selection automatically when you render. Effects files are discussed in the DirectX help files that come with the DirectX SDK. You can find information about the SDK at [MSSDK].

SELECTING YOUR D3D DEVICE

The interface through which you communicate with the hardware (or software emulation) is the D3D device interface. When you create a device using the interface's `CreateDevice()` function, you specify two important settings for shader usage. The first is the type of device—hardware, software, or reference rasterizer.

- `D3DDEVTYPE_HAL`: Hardware rasterization and shading with software, hardware, or mixed transform and lighting.
- `D3DDEVTYPE_REF`: This is the Microsoft Direct3D reference rasterizer. It's provided so that software and hardware developers can test their programs against a known implementation. Hardware manufacturers use this to make sure their hardware and drivers are performing properly. You can use this to check against driver bugs. If you get weird results using hardware and you think it's not your fault, test it with the reference rasterizer. If the problem goes away, you've probably found a hardware or driver bug.
- `D3DDEVTYPE_SW`: A pluggable software device previously registered with the `RegisterSoftwareDevice()` function. I know of none that are available yet.

If you don't have a videocard that can support hardware processing, then you can select the reference rasterizer (refrast, for short). The refrast is useful only for testing purposes. It supports almost every feature, so it's useful for testing code when you don't have a device handy that has the features you

need. But you do not want to use it in production code since it's very slow. In fact, the refract is not normally viewable to the retail installation of DirectX, but has to be switched on in the control panel. However, it does give you a way of testing things like shaders without having the hardware.

Once you've selected your device, you choose the other important setting—the behavior flag. This flag tells the device how to behave. The flags that concern shaders are as follows:

- `D3DCREATE_HARDWARE_VERTEXPROCESSING`: Specifies hardware vertex processing.
- `D3DCREATE_PUREDEVICE`: Specifies hardware rasterization, transform, lighting, and shading. Used as a modifier for the preceding flag.
- `D3DCREATE_MIXED_VERTEXPROCESSING`: Specifies mixed (both software and hardware) vertex processing.
- `D3DCREATE_SOFTWARE_VERTEXPROCESSING`: Specifies software vertex processing.

Unless you are programming exclusively for a chipset that supports hardware vertex processing,^[4] you should specify "mixed" vertex processing to get the maximum speed advantage out of the hardware while letting the vertex processing fall back to the software driver. Note that if you previously used the "pure" device (an optimized hardware interface), then you'll be forced to select "mixed" if you want to use vertex shaders on devices that don't have hardware support. See the section titled "Checking for Shader Support" to check to see if your current device supports shaders.

Note that there are two times you can call `GetDeviceCaps()`. The first is when you have a device specific interface, that is, an `IDirect3D` interface. After you call `CreateDevice()` on that interface, you'll get an `IDirect3DDevice` interface. Calling `GetDeviceCaps()` on the device-specific interface may tell you that it doesn't support vertex shaders. When you create the `IDirect3DDevice` interface (possibly using software or mixed vertex processing flags), you'd call `GetDeviceCaps()` on it and get the interface augmented with any software vertex pipeline capabilities that are available in software vertex processing mode.

^[4]As of July 2002, the cards that have some support for pixel and vertex shaders are ATI's Radeon 8500 or better, the NVIDIA GeForce3 or better (except the GeForce4MX and GeForce4Go), the Matrox Parhelia, the SiS Xabre 400, and the 3DLabs WildcatVP.

CHECKING FOR SHADER SUPPORT

One of the first things you'll need to do is to check for the supported versions of vertex and pixel shaders in your device. After you've loaded a device, you'll need to get the device capabilities bits and check the shader versions encoded there. The vertex and pixel shaders are encoded as major/minor numbers, and you can use DX macros to separate them.

```
// Querying the device capabilities bits
// Assumes m_d3dDevice is a valid device pointer
D3DCAPS8 d3dCaps;
```

```

m_d3dDevice->GetDeviceCaps( &d3dCaps );

// get vertex shader version
printf('This device supports %d.%d vertex shaders\n',
    D3DSHADER_VERSION_MAJOR(d3dCaps.VertexShaderVersion),
    D3DSHADER_VERSION_MINOR(d3dCaps.VertexShaderVersion));

// get pixel shader version
printf('This device supports %d.%d pixel shaders\n',
    D3DSHADER_VERSION_MAJOR(d3dCaps.Pixel ShaderVersion),
    D3DSHADER_VERSION_MINOR(d3dCaps.PixelShaderVersion));

```

If you are using the D3Dapp framework, then there's a function called `ConfirmDevice()` that you can use to test for the desired amount of support. For example, if your program requires vertex shaders version 1.1 or later and pixel shaders version 1.4 or later, then your `ConfirmDevice()` function would look something like this.

```

HRESULT CMyd3DApplication::ConfirmDevice(
    D3DCAPS8* pCaps,
    DWORD dwBehavior,
    D3DFORMAT Format )
{
    // A valid pointer to the capsbits is provided

    DWORD PxlVrsn, VtxVrsn;

    VtxVsn = pCaps->VertexShaderVersion;
    PxlVsn = pCaps->PixelShaderVersion;

    // check for vertex shader version 1.1

```

```

if ( D3DSVS_VERSION(1.1) > VtxVrsn )
{
    return E_FAIL; // bad vertex shader vsn.
}

// check for pixel shader version 1.4
if ( D3DSPS_VERSION(1.4) > PxlVrsn )
{
    return E_FAIL; // bad pixel shader vsn.
}

return S_OK;
}

```

The D3DCAPS that are passed in are fetched from the `IDirect3DDevice` interface.

Vertex elements are just the parts that make up the vertex data we're interested in rendering. This is typically going to be the position, normal, specular and diffuse color, one or more sets of texture coordinates, etc. Though this seems all rather mundane, the point to note is that you can specify your own data as a vertex element as well. It doesn't have to be one of the traditional position, normal, color items at all, but could be a set of alternative data that you'd frequently use—perhaps the object dynamically changes colors every frame and you need additional per vertex colors. Although this gives you a great deal of flexibility, remember that you don't want to be sending a bunch of additional data per object that you infrequently use—this will eat up memory bandwidth. If only some of the information changes (like the colors) while some of the information remains static (like the positions), it's better to separate the data into two streams. (You'll need to check the CAPS bit first. That said, be aware that some of the older videocards have a real problem with trying to use data from more than one stream simultaneously.)

There are two pieces of information that you'll need to create when deciding how to assemble vertex elements: the actual data structure and a vertex format flag that describes how the data is laid out. The first is just a regular C/C++ structure. This example shows a structure that holds vertex position, normal, and color data. You can also use the D3DTYPES nested in your own structure if you like.

```

// Define the vertex data structure

```

```

struct MYCUSTOMVERTEX
{
    // The untransformed position for the vertex.

    float x, y, z;

    // The vertex normals

    float nx, ny, nz;

    // The vertex colors.

    DWORD diffuse_color;

    DWORD specular_color;
};

```

The next step is to define a bit field that holds the DirectX3D flags that describe the format we just created.

```

// Then define the format constant

const unsigned int D3DFVF_MYCUSTOMVERTEX =

    ( D3DFVF_XYZ | D3DFVF_NORMAL |

      D3DFVF_DIFFUSE | D3DFVF_SPECULAR )

```

Once you have the data element layout, you can create a data buffer.

CREATING A VERTEX SHADER INTERFACE DECLARATION

In order to let the driver know the format of the data, you have to create a vertex shader interface declaration, which is somewhat confusingly called just the vertex shader declaration in the DirectX documentation. The vertex shader declaration declares the mapping between input data streams and vertex buffers. This should be the same layout as your vertex elements, along with any "extra" space you might want to allocate for your own purposes (indicated in the *stride* of the vertex stream). In DirectX 8, the vertex shader and the vertex stream declarations are related, which places some restriction on the vertex stream format when using the FFP. So you can't use extra space in the stream when using the FFP. These restrictions were removed with DirectX 9.

If you want to use the FFP in DirectX 8, then you are restricted to using the flexible vertex types. Your declaration would consist of a union of flexible vertex format (FVF) bit masks. If you don't need any

types other than those provided by an FVF format and you don't mind the input registers being mapped to default values, then you can use FVFs for your vertex shaders as well. The declaration for our example would look like the following code:

```
// use FVF to define the declaration
const unsigned int D3DFVF_MYCUSTOMVERTEX =
    ( D3DFVF_XYZ | D3DFVF_NORMAL |
      D3DFVF_DIFFUSE | D3DFVF_SPECULAR )
```

When you use FVF code, the vertex shader registers are mapped for you. [Table 5.1](#) lists the FVF codes and the corresponding vertex shader input registers.

Table 5.1: FFP Vertex Shader Register Mapping

FVF	NAME	VERTEX SHADER
D3DFVF_XYZ	D3DVSDE_POSITION	v0
D3DFVF_XYZRHW	D3DVSDE_BLENDWEIGHT	v1
D3DFVF_XYZBI-5	D3DVSDE_BLENDINDICES	v2
D3DFVF_NORMAL	D3DVSDE_NORMAL	v3
D3DFVF_PSIZE	D3DVSDE_PSIZE	v4
D3DFVF_DIFFUSE	D3DVSDE_DIFFUSE	v5
D3DFVF_SPECULAR	D3DVSDE_SPECULAR	v6
D3DFVF_TEX1-8	D3DVSDE_TEXCOORD0-7	v7-v14
	D3DVSDE_POSITION2	v15
	D3DVSDE_NORMAL2	v16

Thus for our example, the position, normal, diffuse and specular colors would show up in our vertex shader as register v1, v3, v5, and v6, respectively. Only stream 0 can be used when using an FVF format.

If you decide to use your own format, you specify the shader interface declaration without using a FVF, but rather through using your own custom register declaration. You'll need to create a token/element-size array and associate it with a vertex stream. In DirectX 8, tokens are `DWORD`s. In DirectX 9, tokens are `D3DVERTEXELEMENT9` structures.

You'll use the vertex shader declarator macros to help you create the token array. Typically, you'll give the stream number and the stream elements in order. You'll give a register number and a size for each element in the stream. This will then take data out of the stream and place it into the vertex shader register indicated in the macro. You don't have to declare the registers in any order. The following declarations, when used with the previous vertex element format, take the data from stream 0 and place the position in v1, the normal in v6, the two colors in v3 and v4, respectively.

```
// DirectX 8!
DWORD MyVertexDeclarator[] =
{
    D3DVSD_STREAM(0),
        D3DVSD_REG( 0, D3DVSDT_FLOAT3),
        D3DVSD_REG( 6, D3DVSDT_FLOAT3),
        D3DVSD_REG( 3, D3DVSDT_D3DCOLOR),
        D3DVSD_REG( 4, D3DVSDT_D3DCOLOR),
    D3DVSD_END()
};
```

It's possible to specify more than one stream in the declaration. The next example takes the format we used in the preceding example and adds two additional streams. In stream 1, we are providing a series of single floating point values (perhaps a blend weight) to show up in register v1, and stream 2 has two floating point values (perhaps texture coordinates) that'll show up in register v2.

```
// DirectX 8!
DWORD MyOtherVertexDeclarator[] =
{
    D3DVSD_STREAM(0),
        D3DVSD_REG( 0, D3DVSDT_FLOAT3),
        D3DVSD_REG( 6, D3DVSDT_FLOAT3),
```

```

        D3DVSD_REG( 3, D3DVSDT_D3DCOLOR),

        D3DVSD_REG( 4, D3DVSDT_D3DCOLOR),

D3DVSD_STREAM(1),

        D3DVSD_REG( 1, D3DVSDT_FLOAT1),

D3DVSD_STREAM(2),

        D3DVSD_REG( 2, D3DVSDT_FLOAT2),

D3DVSD_END()
};

```

In DirectX 9, things are a little different since the vertex pipeline is being prepared for tessellation processing. In addition to the stream and size of the data, you also specify the offset in the stream (it's no longer order dependent), a tessellating method, the usage, and the usage index. Once you have the structure, you create the declaration using `CreateVertexDeclaration()` and then set it using `SetVertexDeclaration()`.

```

// DirectX 9!

D3DVERTEXELEMENT9 MyDecl[] =
{
    // stream, offset, type
    { 0, 0, D3DDECLTYPE_FLOAT3,
        // method, usage, usage index
        D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
    { 0, 12, D3DDECLTYPE_FLOAT3,
        D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 0},
    { 0, 24, D3DDECLTYPE_FLOAT2,
        D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},
    // Second stream is second mesh
    { 1, 0, D3DDECLTYPE_FLOAT3,
        D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 1},
    { 1, 12, D3DDECLTYPE_FLOAT3,
        D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_NORMAL, 1},
D3DDECL_END()

```

```
};

// and to validate and create the tokenized version
IDirect3DVDec19* pDecl;

g_pD3DDevice->CreateVertexDeclaration( MyDecl, & pDecl );

// Set the declaration
g_pD3DDevice->SetVertexDeclaration( pDecl );
```

Once you've decided the format of your vertex data and the mapping to the stream, you're ready to fill your stream.

CREATING A VERTEX BUFFER

Before you can specify a vertex stream, you'll need to create a vertex buffer. There is a large selection of possible formats, so we'll just pick a fairly simple one to start with. One of the most common formats is one that mimics a typical format supported in previous OpenGL and Direct3D implementations, that of a simple untransformed vertex that contains color and normal data.

Once the structure is defined, you create an array of structures and fill them up with your data. Once you have your data in your array, completely filled out, you create a vertex buffer and copy the data from your array into the vertex buffer array. This *can* cause the video driver to optimize the data and store it in the graphics cards memory for faster processing (a lot depends on the hardware and the desires of the driver writers). Streams were designed to enable driver writers to perform a DMA (direct memory access) from multiple buffers in parallel while accessing the data in as linear a mode as possible.

A simple example of creating, filling, and loading a vertex buffer using the vertex elements types we defined can be seen in the following code:

```
// DirectX 8!

// First allocate memory for the data, 3000 elements

const int ArraySize = 3000;

MYCUSTOMVERTEX * pVertexArray =

    new MYCUSTOMVERTEX[ArraySize];
```



```

if ( 0 == pVertexBuffer)

    return E_FAIL;


// some code here to fill in the data


// now the data array is created and filled,
// we need to create a vertex buffer of
// the correct size
// Create a vertex buffer handle
IDirect3DVertexBuffer8* pVertexBuffer;


if( FAILED( g_pD3DDevice->CreateVertexBuffer(
    ArraySize * sizeof( MYCUSTOMVERTEX ). // memory size
    0. // Usage
    D3DFVF_MYCUSTOMVERTEX, // Our custom format define
    D3DPOOL_DEFAULT, // Let D3D decide where to allocate
    &pVertexBuffer // pointer to the memory
) )

    return E_FAIL;


// we now have a vertex buffer, so we can now fill it
// We need to tell D3D that we're touching the buffer
// and when you're done with changing it.


// lock it
BYTE** plocalVBPointer; // create local pointer
if( FAILED( g_pVertexBuffer ->Lock(
    0, // offset into the vertex data
    0, // size to lock (0 means entire buffer)
    &plocalVBPointer, // pass pointer to VB data pointer

```

```

    0
    )))

    return E_FAIL;

// modify local pointer with vertex data
::memcpy( plocalVBPointer, pVertexArray,
          ArraySize * sizeof( MYCUSTOMVERTEX ) );

// unlock it
pVertexBuffer->Unlock();

// can now delete pVertexArray if we don't need to
// modify the data anymore since we've made a
// copy into the video card's memory.

```

Though it may seem to be a large chunk of code, the basics are pretty simple. The complicated part is understanding that you need to copy the data into device memory so that the device can process the data as quickly as possible. Thus you may need to keep two copies of your data around if you are going to need to modify it. On the other hand, once you've created your data and passed it off to the device, you can free up that system memory for your own use if you don't need to change it anymore.

If we were using a custom vertex format (one not covered by FVF flags), then we would make the FVF flag in the `CreateVertexBuffer()` call zero to indicate that we were using a nonstandard vertex format. In that case, we would be unable to use the FFP to render our primitive and would have to use a vertex shader.

Vertex Buffer Strategies for Static Data

When you've got static data—static meaning data that isn't changing every frame—you should optimize the vertex buffer usage to make things easy for the hardware to optimize. Basically, you need to create, lock, and fill the vertex buffer as infrequently as possible. You should also get into the habit of creating one largish vertex buffer and filling it with as many vertices as you can—even with vertices from different objects. You can use index primitives to select which objects to draw, but the idea is to keep the size of the vertex buffers as large as is optimal for the hardware (currently, the size is around 2000 or so vertices). Note that if you add in a few degenerate triangles (a triangle that has a repeated vertex), you can stitch together different objects that share the same rendering properties so that you can render

them using the same rendering call. If the data is truly static, meaning once you enter it, it's never going to change, then you should use the `D3DUSAGE_WRITEONLY` flag. It's best to create one largish buffer, to hold all of your static objects if you can. If you need random access into the buffer, then try to use a vertex buffer size that's a multiple of 32 bits and use an index buffer to allow the driver to cache data.

Vertex Buffer Strategies for Dynamic Data

Dynamic data is data that you are creating or modifying just about every frame. In this case, you want to create as few vertex buffers as possible since it's generally faster to reuse vertex buffers than to create a bunch of them. Dynamic vertex buffers should be allocated with around 1000 vertices. When you render, you lock the buffer, fill it with your vertices, unlock, render, then start over. The first lock should use the `D3DLOCK_DISCARD` flag to tell the hardware that it can create a new vertex buffer memory location. If you are fetching the data piecewise and filling the vertex buffer with numerous locks, you should use the `D3DLOCK_NOOVERWRITE` until you are ready to start refilling the vertex buffer from the beginning. If you have any shared vertices, it's best to use an indexed vertex buffer, since this gives the driver the opportunity to cache the data.

Customized Vertex Buffer Formats

If you are creative, you can create your own complicated vertex formats that can optimize performance. For example, you can usually specify multiple texture coordinates (up to eight) in an FVF. Remember that texture coordinates can hold from one to four floating point values. Say you have an application where you need to create a vehicle that should be dented. You could encode an extra texture coordinate slot in the FVF to be an "indentation" value that you "dent" a vertex in by. You could have up to eight vehicles using the same vertex buffer rather than eight separate vertex buffers. On the other hand, you don't want to destroy the cache coherency by making the distance between the referenced parts of a vertex too large.

VERTEX DATA STREAMS

When you render a primitive, you have to specify the source of the vertex data. This can be something as simple as the address of an array of positional data to be rendered in order; or something more complicated such as a set of arrays for positional, color, and perhaps texture coordinates; or a custom vertex format (which can be truly custom or a combination of FVF flags) where you have the positional, color, texture coordinate, vertex normal, and maybe some blending weights all interleaved to make up each element of the vertex array. You also have the option of adding in space for your own data.

Although it may seem odd to force the programmer to interleave all the various positional, color, and texture coordinate data, the reason this is done is to greatly increase the cache coherency. Since all the data that's about to be used together is contiguous in memory, there's an excellent chance that it'll be in the cache as opposed to being in system or GPU memory somewhere where the processor will stall

waiting for the memory to be brought in. It's tedious to assemble the data in this way, but once you make the effort, you'll see a real improvement in your rendering speed. It's for your own good, really!

There can be more than one input stream—the exact number depending upon the `D3DCAPS8.MaxStreams` capabilities bit—that you can query as shown next.

```
// DirectX 8!
// Querying the maximum number of streams
// Assumes m_d3dDevice is a valid device pointer
D3DCAPS8 d3dCaps;

m_d3dDevice->GetDeviceCaps( &d3dCaps );

printf(''This device supports %d streams\n'',
       d3dCaps.MaxStreams );
```

The minimum number of streams supported in DirectX 8 and 9 software vertex processor is 16. Unless you're doing some pretty complicated vertex blending operations, you'll probably need no more than one to four simultaneous vertex streams. Using multiple streams can hurt performance. It's a tradeoff between sending extra data along the stream and missing data in the cache. If you're thinking about multiple streams, it's a good idea to set up some test cases.

DirectX 9 stream can also specify a stride value, for which you'd also query the `D3DCAPS9.MaxStreamStride` capabilities bit, that you can query as shown next.

```
// DirectX 9!
// Querying the maximum stream stride
// Assumes m_d3dDevice is a valid device pointer
D3DCAPS9 d3dCaps;

m_d3dDevice->GetDeviceCaps( &d3dCaps );

printf(''This largest stream stride is %d bytes\n'',
       d3dCaps.MaxStreamStride );
```

CREATING WELL-FORMED VERTEX DATA

Associating a Vertex Buffer with a Stream

The next step, after we've created a vertex buffer, is getting ready to render the buffer. You'll typically be setting up the stream sources that go with a particular vertex buffer and then rendering the object with a render primitive call.

```
// DirectX 8!

// Set vertex stream 0 with our vertex buffer
g_pD3DDevice->SetStreamSource(

    0,                // Stream number to set (start at 0)

    pVertexBuffer,    // the vertex buffer handle

    0                 // stride

);

// We can set a vertex shader or an FVF code here
// so (for now) we show it using our FVF code
g_pD3DDevice->SetVertexShader( D3DFVF_MYCUSTOMVERTEX );

// render it
g_pD3DDevice->DrawPrimitive(

    D3DPT_TRIANGLELIST, // our type of primitive

    0,                  // starting index into vertex buffer

    ArraySize/3,        // how many triangles = # primitives/3

);
```

So you can see there's the setup of the vertex buffer, which should happen only at initialization time. Then there's the preparation for the actual rendering where you'll be associating vertex buffers with streams, which will probably happen many times while you are rendering a scene. In the `SetVertexShader()` function call in the example, I used the FVF format flag. There really isn't a vertex shader in this case but simply a call to the rendering engine so that it knows the layout of the vertex information.

This was a change introduced in DirectX 8. A "real" vertex shader (consisting of code written in the shader language) would be written for a particular FVF and would know how to access the vertex stream for the correct data, even doing something like remapping the texture coordinate information in the example given earlier. This should give you an idea of the potential power of writing your own shaders. In DirectX 9, if you want to use the FFP, you'll have to set the vertex format using the `SetFVF()` function.

The actual pipeline from vertex data to rendering is illustrated in [Figure 5.1](#).

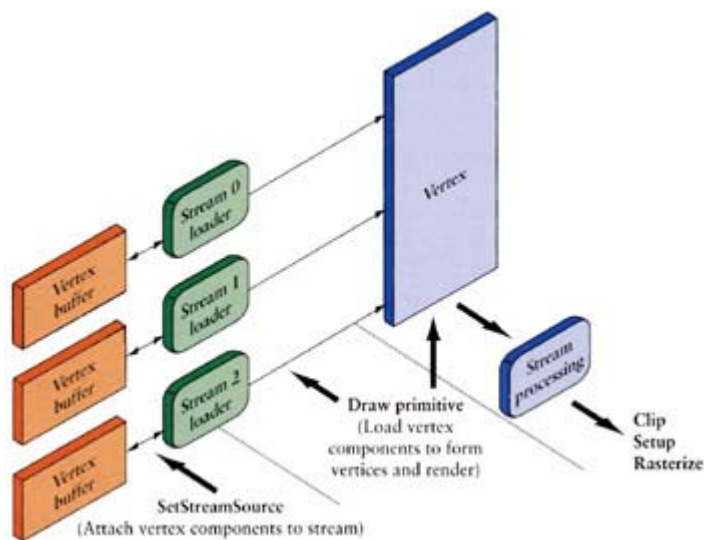


Figure 5.1: Vertex streams get loaded into vertex registers by a render call.

The input to a vertex shader is one or more vertex streams plus any other state information (like textures). The output is a new vertex position in clip coordinates and any other information that the shader provides, like vertex color, texture coordinates, fog values, etc.

CREATING AND USING PIXEL AND VERTEX SHADERS

In order to create a shader, you need to have the shader source in memory. You can do this in any number of ways. The easiest is to leave the shader in an ASCII file and to simply read the file and generate the binary shader tokens using an API function. If you are working on a commercial product, you could assemble the shader into a binary format and save the format either in a file or as part of a larger data set. Be aware that the tokens are just that, tokens, and that if someone was curious, he or she could easily parse the tokens back into assembly. Let's assume that you've got an ASCII shader program in memory, either read in from a file or typed in directly into a buffer. The steps you need to do to actually use it in your program are as follows:

1. Create the shader and read it into memory.
2. Translate (assemble) the shader into binary tokens.
3. Create a shader from the tokens and get a handle (DX8) or interface (DX9).
4. Select the shader.

5. (Optional) Set any constant registers for that shader.

You'll probably create all of your shader token arrays at load time or precompile them and store them as data with your program. At run time, you'd convert the token array into a shader handle. When you need to render a new object with a particular shader, you'd load the vertex and/or pixel shader from its handle and any constant registers that need to be updated, plus any textures associated with that object before you make your render call. Shaders should be treated just as any other state variable. In terms of expense, changing shaders comes after setting textures, so you should render your objects by sorting first by textures, then by shaders.

ASSEMBLING VERTEX AND PIXEL SHADERS

Assembling the shader simply means taking the ASCII string and converting it to binary tokens. There's nothing magical about this step. If you're a masochist, you can write the shader in binary yourself, but it is easier to use the `D3DXAssembleShader()` helper function, which will assemble a pixel or vertex shader. This function will take the input ASCII buffer and its length and, if it succeeds, return a pointer to the tokenized buffer. If it fails and you haven't set the flags so that the shader isn't validated, the error messages (in somewhat readable ASCII) will tell you where you went wrong.

```
// DirectX 9!

// Assume we have a valid D3D pointer

DWORD dwFlags = 0;

LPD3DBUFFER* pShaderTokens, ppErrMsgs;

HRESULT hRet = g_pD3DDevice->D3DXAssembleShader(
    pSrcBuff,          // pointer to the ASCII buffer
    ::strlen(pSrcBuff), // length of buffer
    dwFlags,           // assemble flags
    &pShaderTokens,     // pointer to the shader
    ppErrMsgs          // where error messages go
);
```

If the shader was assembled successfully, the return value will be `D3D_OK`, and you can then use the handle/interface to the assembled shader to load the shader into the device. At this point, you just have

a valid token array; nothing is actually loaded into the device. There are various flavors of the helper function that can load from a resource handle or from a file instead of from a memory buffer.

The next step is to take the assembled tokens and create the shader. This will store the shader in the device and give you a handle or interface so that you can select it later on. The DirectX 9 interface is a little cleaner than the DirectX 8 one so I've shown it here, but the basics are the same.

```
// DirectX 9!

// Assume we have a valid D3D pointer
// and a token array
IDirect3DVShader9 * pShader;

HRESULT hRet = g_pD3DDevice->CreateVertexShader(
    pShaderTokens, // pointer to token array
    &pShader       // shader interface
);
```

SETTING A SHADER

After all that setup, actually using a shader is pretty easy: you just call `SetVertexShader()` or `SetPixelShader()` using the handle (DirectX 8) or interface (DirectX 9) that you got from a create shader call. Since there has been a change when using a shader or the FFP in DirectX 9, we'll discuss loading a shader in two version-specific sections.

Loading a DirectX 8 Vertex Shader

If you don't want to use a vertex shader but use the FFP, then DirectX 8 will require you to make a call to `SetVertexShader()` but with a FVF code. This will activate the FFP. If you have a vertex shader, then you take the handle to the shader and call `SetVertexShader()`.

```
// DirectX 8!

// Assume we have a valid D3D pointer
// pShader is a handle to a vertex shader

g_pD3DDevice->SetVertexShader(pShader);
```


Loading a DirectX 9 Vertex Shader

Since vertex declarations are decoupled from the vertex shaders in DirectX 9, they can be used with a vertex shader or the FFP. If you want to use the FFP, you need to call `SetVertexShader()` with a null argument and then call `SetFVF()` with the FVF you are using.

For a vertex shader, you call `SetVertexShader()` with the shader interface from the `CreateVertexShader()` call. If you're using a vertex element format that can be described with FVF, you can call `SetFVF()`. If you have a customized format, you would call `SetVertexDeclaration()` instead.

Loading a Pixel Shader

Loading a pixel shader is just as easy; you make a call to `SetPixelShader()` and the shader is active. The only difference is that the DirectX 8 uses a handle, whereas DirectX 9 uses an interface.

SETTING SHADER CONSTANTS

One of the significant differences between DirectX 8 and DirectX 9 vertex shaders is that DirectX 8 has only a single floating point constant array, whereas DirectX 9 contains three different constant arrays: a floating point, an integer array, and a binary array. Shader constants are constant across set shader calls in DirectX 8. Since `def` statements in pixel shaders actually load the constants, loading a pixel shader with `defs` in it will override any constants set previously. In DirectX 9, only the `defs` become *local* and only persist for the life of the pixel shader. The previous value of the constant returns when the shader is unloaded. Constant definitions in DirectX 9 vertex shaders work just like they do for pixel shaders.

Setting DirectX 8 Vertex Shader Constants

DirectX 8 has only a single array of constants for vertex registers. If you want to set the constants in the shader you have loaded, then you'd make a call to `SetShaderConstant()` with the appropriate arguments, and the values you pass in will get loaded to the shader constant registers.

Note

`def` statements in vertex shaders don't actually do anything except pass back some tokens that you have to insert manually into the shader token stream. Nobody does it this way. Use `SetShaderConstant()` instead.

The first example is the DirectX 8 way to set vertex constant registers. Note that a "register" is a float[4] value, thus you'll need to have values for the .xyzw elements of the register. If you want to set more than one serial register, you can create an array of the required size and use the initial element value plus the number of float[4] elements to set the constants. The following example demonstrates how to set both a single register and an array of registers.

Note

It's a common mistake to forget to count each float[4] as a single register element.

```
// DirectX 8!

// Assume we have a valid D3D pointer

float singleRegister[4] = {0.0f, 0.5f, 1.0f, 2.0f};
float arrayValue[4*20] = {. . .};

//set a single register
g_pD3DDevice->SetVertexShaderConstant(
    5,           // the vertex constant register number
    singleRegister, // beginning address
    1 );        // the number of 4*float registers

//set 20 serial register values
g_pD3DDevice->SetVertexShaderConstant(
    6, // the vertex constant register number
    arrayValues, // beginning address
    20 ); // the number of 4*float registers
```

Setting DirectX 9 Vertex Shader Constants

In DirectX 9, there are three types of shader constants, and you have to use a unique call to set each type. Other than the new constant types, setting the constants in DirectX 9 is similar to setting them in a DirectX 8 shader—you provide the constant number to set, the initial array address, and the total number to set, and that's it. Instead of the DirectX 8 `SetVertexShaderConstant()` call, there are now three calls to set the integer, boolean, and float values, having added a letter I, B, and F suffix, respectively to indicate these new functions. Although the integer constants are composed of four-element registers (and thus are set to a count of four-elements), the boolean registers are single-element registers and are set to one element per count. Constant declarations made in a shader are only for the life of the shader.

The following example demonstrates how to set the integer, boolean, and float arrays for the DirectX 9 style vertex shader.

```
// DirectX 9!

// Assume we have a valid D3D pointer

// and we have some varying arrays in varying sizes
int integerArray[4*10] = {. . .};
BOOL boolArray[11] = {. . .}; //note - NOT AN ARRAY
float floatArray[4*12] = {. . .};

//set 10 integer arrays
g_pD3DDevice->SetVertexShaderConstantI(
    0,          // the vertex constant register number
    integerArray, // beginning address
    10 );       // the number of 4 element registers
//set 11 boolean values
g_pD3DDevice->SetVertexShaderConstantB(
    0,          // the vertex constant register number
    boolArray,  // beginning address
    11 );       // the number of boolean registers

//set 12 float arrays
g_pD3DDevice->SetVertexShaderConstantF(
    0,          // the vertex constant register number
    floatArray, // beginning address
    12 );       // the number of 4 element registers
```

Querying the Number of Vertex Shader Constant Registers

If you query the device capability bits member `MaxVertexShaderConst`, you can get the maximum number of floating point vertex shader registers that are available to you. This number is at least 96 and

for VS 2.0 can be 256. There are no integer or boolean registers prior to VS 2.0. For VS 2.0, there are at least 16 single-element boolean registers and 16 four-element integer registers.

Setting Pixel Shader Constants

Setting DirectX 8 and 9 pixel shader constants is almost exactly the same as the vertex shader constants, except that the name of the function that you call is `SetPixelShaderConstants()`.

```
// Assume we have a valid D3D pointer

float singleValue[4] = {0.0f,0.5f,1.0f,2.0f};

//set a single register

g_pD3DDevice->SetPixelShaderConstant(
    2,          // the pixel constant register number
    singleValue, // beginning address
    1 );        // the number of 4*float registers
```

Unlike vertex shaders, the `def` instruction in pixel shaders works without any other effort. However, in DirectX 9, using a `def` statement produces a change only for the life of that shader.

RENDERING

You've created your vertex data, set the vertex shader interface, set up the vertex streams, assembled and loaded your vertex and pixel shaders. You've set the textures and shader constants. You've set the necessary render states. You've selected the shaders you want. What's left? Only a call to one of the rendering functions to put everything in motion.

If you're using one of the higher order primitive calls, your vertex shader will be called on each vertex that is created by the primitive call. If you're using a `DrawPrimitive` call, then each vertex will get passed to your vertex shader. Once a triangle is passed through a vertex shader, the rasterization process will start generating pixels. Each of these pixels will get passed to your pixel shader routine. Some data (like texture coordinates) will get interpolated from the vertex data before it gets passed to the pixel shader depending upon the settings of render states. For the best speed possible, you should sort your objects by texture so that objects that share a texture get rendered together. After textures sort by vertex buffer, the next thing to sort by are objects that share shaders. You should also try to batch up setting shader constants and not set them singly. That's it! There's a lot to keep track of, but there are plenty of

samples in the DirectX SDK, and the DirectX programming documentation covers many of these functions in greater detail. Render on!

Chapter 6: Shader Tools and Resources

I wrote a tool called ShaderLab to go along with this book. It's the tool that I used to develop most of the samples. I was writing it to give you an interactive tool to experiment with shader code. I wanted it to be easy to use, able to run on many different hardware platforms, etc. Then along comes RenderMonkey from ATI, which does all these things and more. So I shelved ShaderLab and decided to use RenderMonkey as the experimentation application in this book. I think you'll find it's pretty useful.

All the tools here require Windows 2000 or Windows XP with DirectX 8 or 9 installed to run. Some of them also require specific hardware to be installed. They are also all free, so you should try to download them and give them a try. I typically find that I use more than one tool while I'm programming since they each have their strengths.

RENDERMONKEY

[Figure 6.1](#) shows the RenderMonkey shader tool. RenderMonkey is an extensible shader tool from ATI. It's got some pretty nifty features, including a plugin interface so you can write your own tools, and multipass capabilities. It supports almost all the shader languages that are out there. There's a copy of the RenderMonkey SDK on the disk. Once you install it, you should have no problems reproducing the samples in the book. RenderMonkey is also available on the Web at <http://www.ati.com> in the developer section.

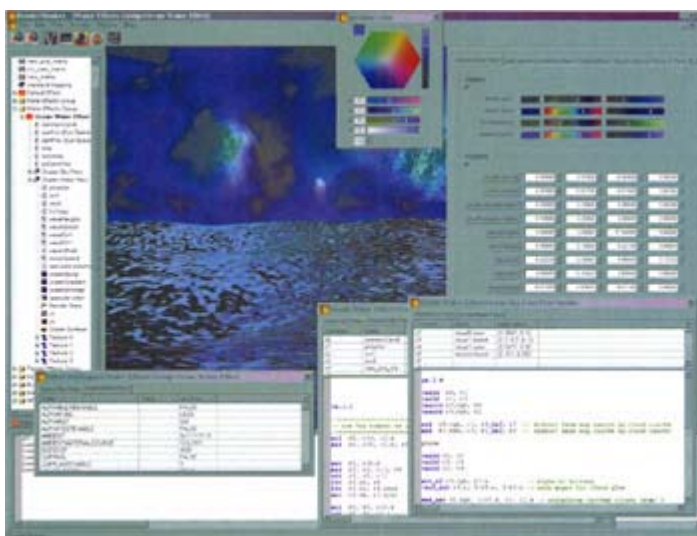


Figure 6.1: RenderMonkey in action.

NVIDIA EFFECTS BROWSER

The Effects Browser from NVIDIA, shown in [Figure 6.2](#), is another nifty tool that lets you explore with vertex and pixel shaders. It also lets you program the register combiners on NVIDIA hardware directly. It comes with a huge palette of various samples of shaders for both DirectX and OpenGL. As [Figure 6.2](#) shows, there is a collection of effects on the left side, any shaders or other setup is in the middle, and the effect itself is displayed on the right side. It's available from <http://developer.nvidia.com>.

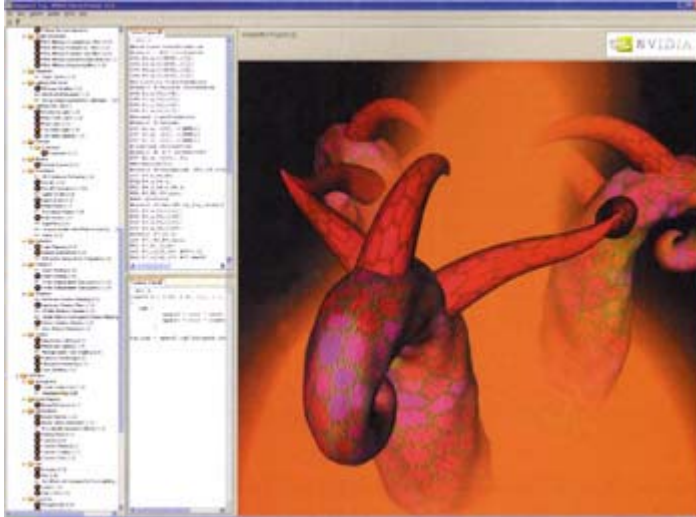


Figure 6.2: NVIDIA Effects Browser.

NVIDIA SHADER DEBUGGER

The Shader Debugger from NVIDIA is almost reason enough to buy an NVIDIA card. Having interactive debugging capability is something that you really need if you're serious about shader writing. The most interesting thing about the debugger is that it attaches itself to a process and intercepts calls to the hardware—this means that you can view any program that uses shaders and see how the effects are being done.^[5]

[Figure 6.3](#) shows that the Shader Debugger displays the register contents on the right, the input and output registers in the middle, and the shader code and temporaries registers on the left. Like any good debugger, you can set breakpoints and step through the code. It's available from <http://developer.nvidia.com>.

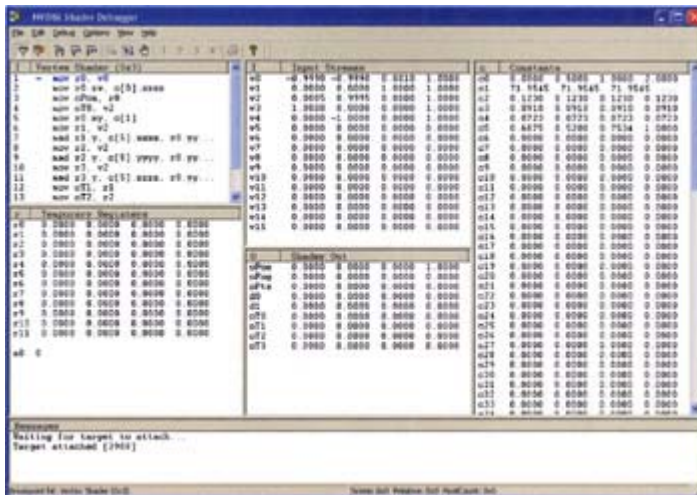


Figure 6.3: NVIDIA Shader Debugger.

D3D Shader Debugger

With DirectX 9, you get integration of a shader debugger that integrates with the Visual Studio IDE.

[Figure 6.4](#) shows the D3D Shader Debugger. It's available in the *extras* of the DirectX 9 SDK disk. Just follow the *readme* instructions and you'll be able to install it.

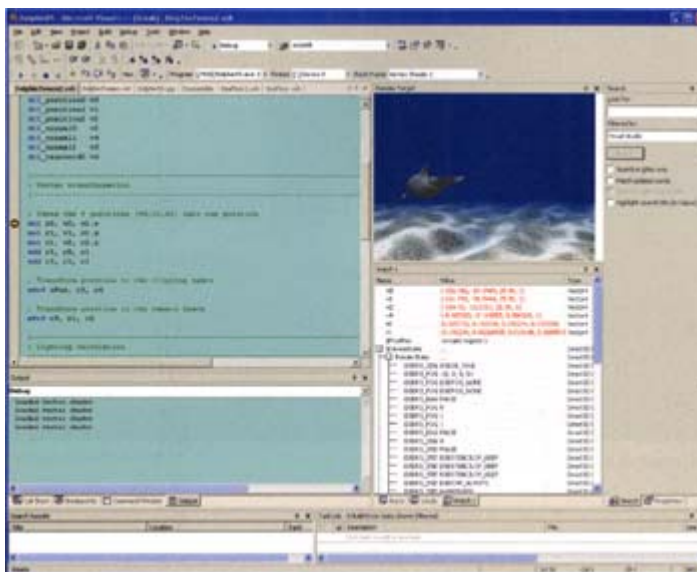


Figure 6.4: The D3D Shader Debugger that shipped with the DirectX 9 SDK.

The version that I got to test (beta 2) required you to start your D3D application and then attach to the process. Since the hardware drivers hadn't yet caught up to the release, I had to use *refrast* as the device. You'd select *Direct3D* as the program type you wanted to debug. Once that was done you could set break-points in the shader code, modify register values, etc.

^[5] I should point out that if you have Visual Studio.NET, then the DirectX 9 SDK comes with the D3D Shader Debugger, which will allow you to debug on any hardware—the caveat being that you'll probably have to use the reference rasterizer.

SHADER STUDIO

Shader Studio is a shader editing and exploration tool written by John Schwab. It has a nice multiwindow interface so that you can just bring up the section you need. [Figure 6.5](#) shows the shader register window. There's also a texture window, a materials windows, some lighting and object windows, and of course, a display window. You can download a copy from <http://www.shaderstudio.com>.

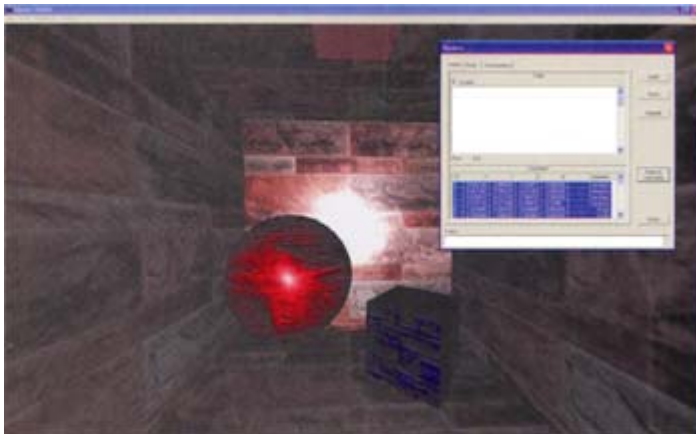


Figure 6.5: Shader Studio.

COLORSPACE TOOL

I mentioned this tool in [Chapter 2](#). I've included it so that you can play around with it and see how you can get some unexpected results when adding colors. ColorSpace lets you interactively add two colors together and see the effects of the various strategies for handling oversaturated colors. [Figure 6.6](#) shows a yellow and a bright red. Since the color that you get when you add these two colors together has a red value of 510, it's impossible to display this color either on a color monitor or in a shader.



Figure 6.6: ColorSpace lets you see the effects of clamping vs. clipping vs. scaling when two colors oversaturate.

A shader will automatically clamp the color for you, which can dramatically alter the color. Other alternatives are to multiply all the **rgb** values by a scale factor so that they all fit in the color range—this

has the effect of darkening the color. The other step is to clip the color—that is, maintain the brightness by proportionately taking away some of the oversaturated color and giving it to the unsaturated color. The three button colors show you the results of the different strategies for handling oversaturated colors.

RESOURCES

There are a wealth of resources available though it's sometimes a bit tough to find them. I've attempted to collect some of the most useful ones and include them in this section.

DirectX and Shader Programming

Of course, the best site is the Microsoft Web site. The Microsoft Developer Network (MSDN) Web site is the place to start. It'll have all the latest code, the retail and debug SDK release, as well as some articles designed to help you learn DirectX and Shader programming. Go to <http://msdn.microsoft.com/DirectX>.

The NVIDIA developer site contains lots of code samples and documentation. Go to <http://developer.nvidia.com>.

The ATI site isn't quite as full as the NVIDIA site, but they have some high-quality stuff there as well. Go to the Developer section at <http://www.ati.com>.

At this time, 3DLabs has been acquired by Creative Labs. This likely means that Creative will be selling the 3DLabs consumer market cards while 3DLabs sells to the workstation market. You can probably find out lots of interesting things at <http://www.3dlabs.com/support/developer>.

Matrox is the only other graphics card vender that supports shaders, but they are a bit lacking in the developer support area. They are on the Web at <http://www.matrox.com>.

Wolfgang Engel has a book *Direct3D ShaderX* and a Web site that's devoted to the book, <http://www.shaderx.com>.

The Game Programming Gems series from Charles River Media has some sections on shader programming at <http://www.charlesriver.com>.

And, of course, my own site is devoted to keeping up with shaders and shader technology: <http://www.directx.com>.

Graphics Programming in General

Every year you can attend Siggraph (<http://www.siggraph.org>) or GDC, the Game Developer's Conference (<http://www.gdconf.com>). Siggraph is a bit more polished (and cheaper), and it's starting to get lecturers from the games community. The GDC is the place to go if you're a game programmer—

you can live, eat, and breathe games for five days. You'll see more innovative stuff at Siggraph, but you'll get more practical information at the GDC. Stuff typically shows up at Siggraph, then someone programs it into a game and talks about it at the next GDC. If you've never been to one, you should really go—you'll be changed forever. Both conferences typically have hardware vendor-sponsored sessions. If you're an ACM (Association for Computing Machinery) member (<http://www.acm.org>), you can get a discount to Siggraph.

The proceedings of each Siggraph are published, and the course notes are available on CD as well. If you can't make it to Siggraph, it's well worth the investment to get a copy of the proceedings and the course notes. The ACM also has published a collection of papers titled *Seminal Graphics* [WOLFE 1998], which contains reprints of papers that have long been out of print, like Phong's original paper.

Microsoft also hosts a yearly event called Meltdown (typically in July or August) where you can get the inside technical stuff from the DirectX guys themselves. It's a very heavy-duty graphics and multimedia get together. A lot of hardware manufacturers will bring their latest boards so that you can test your code on it. Search <http://www.microsoft.com> for more information.

I should also mention that NVIDIA and ATI have also taken to holding their own one- to two-day intensive sessions ("The Gathering" and "Mojo Day," respectively) on programming their hardware. The benefit is they are reasonably cheap, and you get a videocard to boot.

Game Developer Magazine (<http://www.gdmag.com>) is a monthly magazine that frequently has articles on graphics programming, and if you fill out a questionnaire, you can get a free subscription.

The Journal of Graphics Tools (<http://www.acm.org/jgt/>) is a "quarterly journal whose primary mission is to provide the computer graphics research, development, and production community with practical ideas and techniques that solve real problems." It's a bit more theoretical than *Game Developer*, but they do have some articles that are useful to the everyday graphics programmer. If you are an ACM member (<http://www.acm.org>), you can get it at a discount. The first quarter issue of 2003 is on hardware accelerated rendering techniques.

The IEEE's *Computer Graphics and Applications* (<http://www.computer.org/cga/>) is another monthly magazine that's chock full of information. You'll get practical as well as academic articles, but it's also got "Jim Blinn's Corner," which is worth reading all by itself.

I've mentioned the Game Programming Gems series from Charles River Media already. That series was inspired by the *Graphics Gems* series of books from Academic Press (<http://www.academicpress.com>). There's an online repository for the source code from the books as well as links for the books at <http://www.graphicsgems.org>.

One of my favorite graphics books is *Real-Time Rendering* [RTR 2002] (get the 2nd edition) by Tomas Akenine-Möller and Eric Haines, published by AK Peters. They also have a Web site devoted to their

book (<http://www.realtimerendering.com>), which they update frequently. If you only get one graphics book, get this one.

Chapter 7: Shader Buffet

OVERVIEW

I'm not going to take the usual route that you see in these shader collections. Instead, I'm going to do it building-blocks style, that is, I'll give you, say, a variety of diffuse shaders and a variety of specular ones. You get to mix and match. I want to impart the fact that you can create and use these in any way you want; there's no right or wrong way. Once you understand how a Lambertian diffuse term behaves, you'll know those situations when you want that effect, and you can take the snippet of code that allows you to do Lambertian. If you want some specular, you can choose between Phong, Blinn–Phong, Schlick, Cook–Torrance, etc.

This way, you'll recognize the individual traits of each piece of code and what it takes to compute it. For example, if you're already computing specular, you've done all the work required to compute a diffuse color as well, except for the actual color modulation. The most important thing to do is to think of these techniques as a palette from which to artistically create your objects, rather than a cookbook from which you can choose only a single recipe.

There are many instances where you will reuse the same instructions; for example, local lights vs. infinite lights will make you go through the same calculations for both the specular term and the diffuse term. In the buffet, I'll present one shader that uses a local light, another that uses a directional one. Since the only difference will be in how the light direction vector is calculated, I assume that once you've decided what type of light you want for a shader, you'll be able to cut and paste to get the correct vector for that light type.

VERTEX SHADERS

Common Code for Vertex Shaders

Since all the vertex shaders require setup code that's one of a few variations on a theme, these components are collected and discussed a bit more in depth.

Component 1: Vertex Transform and Copy

Description This is the simplest of the setup codes; it is simply the matrix transformation of the incoming vertex position and it places a copy of the transformed vertex position into register `r0` for use later in the shader. This is necessary since `oPos` is a write-only register.

Prerequisites Assumes that `c[TRANS_MVP]` points to the first register in the transposed model-view-projection matrix. Assumes that the vertex position is in register `v0`.

Summary When completed, this fragment will have placed the transformed vertex position (in homogenous clip-space coordinates) into `oPos` and `r0`.

```
// code component 1

// emit projected position with a copy in r0
m4x4   r0, v0, c[TRANS_MVP]
mov     oPos, r0
```

Component 2: Vertex Transform and Copy, Normal Transform

Description This is the same as component 1 but also transforms the normal of the vertex by the inverse transpose model-view-projection matrix.

Prerequisites Assumes that `c[TRANS_MVP]` points to the first register in the transposed model-view-projection matrix, and that `c[TRANS_INV_MVP]` contains the transpose of the inverse of the model-view-projection matrix. This, of course, can change depending upon whether you use uniform scaling matrix transformations or not, in which case you can use the same matrix as `c[TRANS_MVP]` for the normal transformation. See the section titled "Transforming Normal Vectors" in [Chapter 4](#) for more information. Also assumes that the vertex position is in register `v0` and that the vertex normal is in `v3`.

Summary When completed, this fragment will have placed the transformed vertex position (in homogenous clip-space coordinates) into `oPos` and `r0`, and a transformed and normalized vertex normal value in `r1`. Note the use of the `dp3-rsq-mul` sequence. You'll frequently see this when it's necessary to normalize a vector.

```
// code component 2

// emit projected position with a copy in r0
m4x4   r0, v0, c[TRANS_MVP]
mov     oPos, r0

// calculate a unit vertex normal
// transform the normal
m3x3   r1,    v3,    c[TRANS_INV_MVP]
```

```

// calculate the length into r1.w
dp3    r1.w,  r1,    r1
// calculate 1/sqrt(length) into r1.w
rsq    r1.w,  r1.w
// normalize it
mul    r1,    r1, r1.w // r1=|n|

```

Component 3: Vertex Transform and Copy, Normal Transform, Local View (Light) Vector

Description This is the same as component 2 but also takes the viewer position (or eye position) and creates a view direction normalized vector.

Prerequisites The same as component 2, with the assumption that the eye position is input in register `c[EYE_POS]` in *clip-space* coordinates since generally you'll have a unique eye per frame (i.e., you do the transform outside the shader).

Note This is the code for a "local viewer"; that is, the view direction is calculated per vertex. This code is exactly the same for a "local light"; the only difference is that you'd place the light position in clip coordinates instead of the eye position.

Summary When completed, this fragment will have placed the transformed vertex position (in homogenous clip-space coordinates) into `oPos` and `r0`, a transformed and normalized vertex normal value in `r1`, and the normalized view vector in `r2`.

```

// code component 3

// emit projected position with a copy in r0
m4x4   r0, v0, c[TRANS_MVP]
mov     oPos, r0

// calculate a unit vertex normal
// transform the normal
m3x3   r1,    v3,    c[INV_TRANS_MVP]
// calculate the length into r1.w

```

```

dp3    r1.w,  r1,    r1
// calculate 1/sqrt(length) into r1.w
rsq    r1.w,  r1.w
// normalize it
mul    r1,    r1, r1.w // r1=|n|

// create view direction vector in r2
// (If this was a light position we
// could use it to create a light direction)
add    r2,    r0,    -c[EYE_POS]
// calculate the length into r2.w
dp3    r2.w,  r2, r2
// calculate 1/sqrt(length) into r2.w
rsq    r2.w,  r2.w
// normalize it
mul    r2, r2, r2.w // r2=|v|

```

Constant Color Shading Shaders

This is the simplest vertex shader. This is the shader that you'd use to place a constant color on an object. The color is set before the object is rendered.

Uses Use this to generate a solid color object with no shading effects.

Description If you want an object to have the same color irrespective of lighting conditions or viewpoint, this shader will do it.

Prerequisites Object color is passed in as a constant, `v0` is the vertex position.

Results The color is written to `oD0` so that it can be used in the fixed function pipeline. The result of the shader is shown in [Figure 7.1](#).

```

// Constant Color Shading.vsh

vs.1.0    // Shader version 1.0

```

```
// emit transformed position
m4x4 oPos, v0, c[TRANS_MVP]

// stick the color into the output diffuse register
mov oD0, c[COLOR]
```



Figure 7.1: Constant vertex color gives color to an object but no hint of depth.

Vertex Color Shading

This is the shader that you'd use to place a color on an object that can vary on a per vertex basis. The vertex color is passed in along with the vertex position.

Uses Use this to generate a multicolor object with no shading effects.

Description If you want an object to have a per vertex color that's irrespective of lighting conditions or viewpoint, this shader will do it.

Prerequisites Object color is passed as `v4`, `v0` is vertex position.

Results The color is written to `oD0` so that it can be used in the fixed function pipeline. The color might be interpolated or not depending upon the shading render state. The result of the shader is shown in [Figure 7.2](#).

>

```
// Vertex Color Shader.vsh

vs.1.0    // Shader version 1.0

// emit transformed position
m4x4  oPos, v0, c[TRANS_MVP]

// stick the color into the output diffuse register
mov   oD0, v4
```

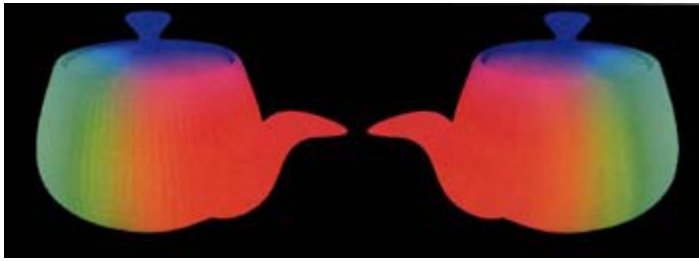


Figure 7.2: Assigning a per vertex color gives more control at the expense of more stream data.

Ambient Shading

When you want to add ambient light to a scene, the ambient light is modulated by the material color of the vertex to create the final color.

Uses Use this to generate the effect of ambient illumination on an object.

Description Ambient light shader provides a constant illumination on an object irrespective of the viewer or the object's orientation. The ambient object color is modulated with the ambient light color.

Prerequisites Object color and light color are passed in as constants, `v0` is vertex position.

Results The color is written to `oD0` so that it can be used in the fixed function pipeline. The result of the shader is shown in [Figure 7.3](#).

```
// Ambient Shading.vsh

vs.1.0    // Shader version 1.0
```



```
// emit transformed position
m4x4 oPos, v0, c[TRANS_MVP]

// multiply the ambient light color by the
// ambient color of the object
mov r0, c[AMBIENT_LIGHT_COLOR]
mul oD0, c[AMBIENT_COLOR], r0
```

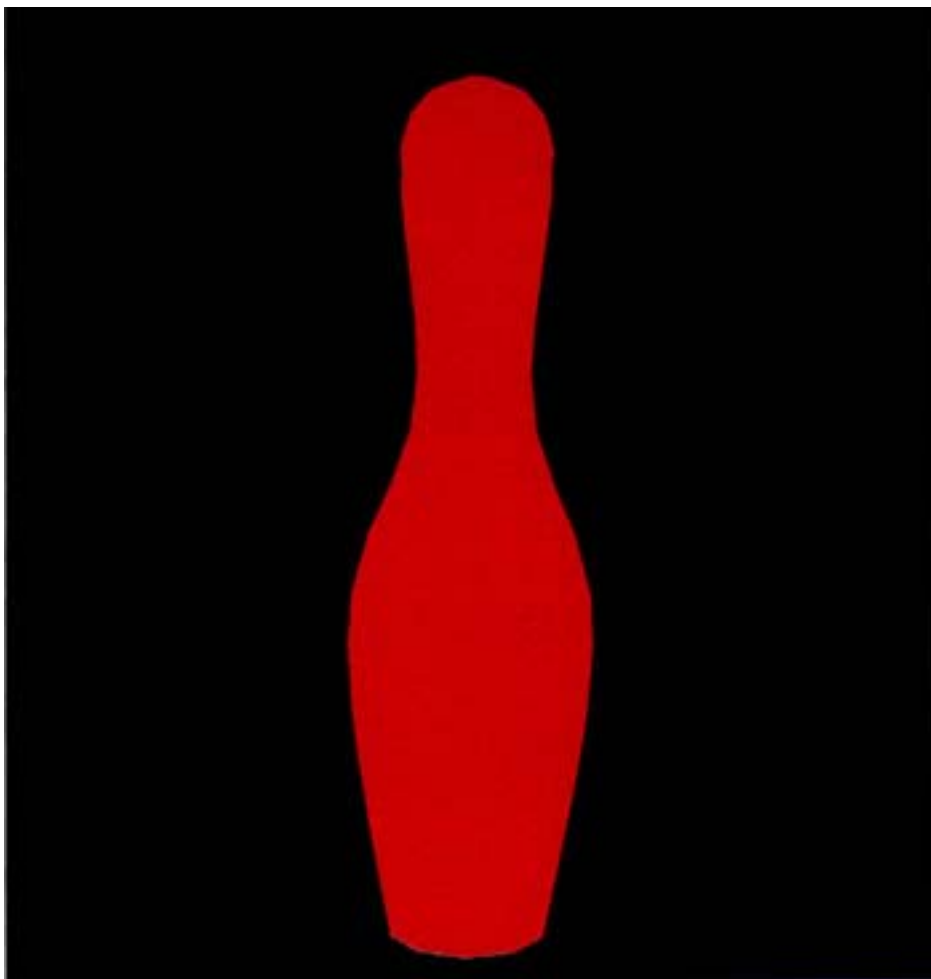


Figure 7.3: Ambient shading gives color plus shading, but the unilluminated areas are black.

Lambertian Diffuse Lighting with an Infinite Light

This shader uses the same math that the traditional fixed function pipeline uses to generate the diffuse lighting term and an infinite light. The diffuse light color and diffuse material color are specified. The diffuse colors are frequently the same as the ambient in many situations.

Uses Use this to generate the diffuse term for an infinitely distant light shining on a vertex. When combined with a Blinn-Phong specular term, you'll get the same results as the fixed function pipeline does.

Description This is the traditional expression for diffuse lighting. Since the light source is infinite, any objects using this shader with that light source will all be lit from the same side. Diffuse lighting doesn't depend upon the viewing angle, only upon the angle between the surface normal and the light direction. This shader will compute the light direction vector, normalize it, then compute the dot product of the light vector and the normal to get the intensity. The diffuse colors are then modulated by each other and the intensity.

Prerequisites All the assumptions for use of component 2. Material diffuse color is passed in as a constant. Light diffuse color is passed in as a constant. The light direction is passed in as a constant in clip-space coordinates—it should be normalized. v_0 is vertex position. v_3 is the normal.

Results The color is written to oD_0 so that it can be used in the fixed function pipeline. The result of the shader is shown in [Figure 7.4](#).

```
// Diffuse Shader with infinite light

vs.1.1

// transform the vertex
m4x4 oPos, v0, c[TRANS_MVP]

// store light vector (assume its normalized)
mov r0, c[LIGHT_VECTOR] // r0 = |l|

// compute |n| dot |l|
dp3 r5.x, r0, v1 // r5.x = n dot l

// no need to clamp in next step
// - it's done automatically
//max r5.x, r5.x, c12.x // max(n dot l, 0)
```

```
// compute diffuse
mov r10, c[MATERIAL_DIFFUSE]
mul r11, r10, c[LIGHT_DIFFUSE]
// r11 = diffuse material color times
// diffuse light color

// modulate by |n| dot |l|
mul oD0, r11, r5.xxxx
```

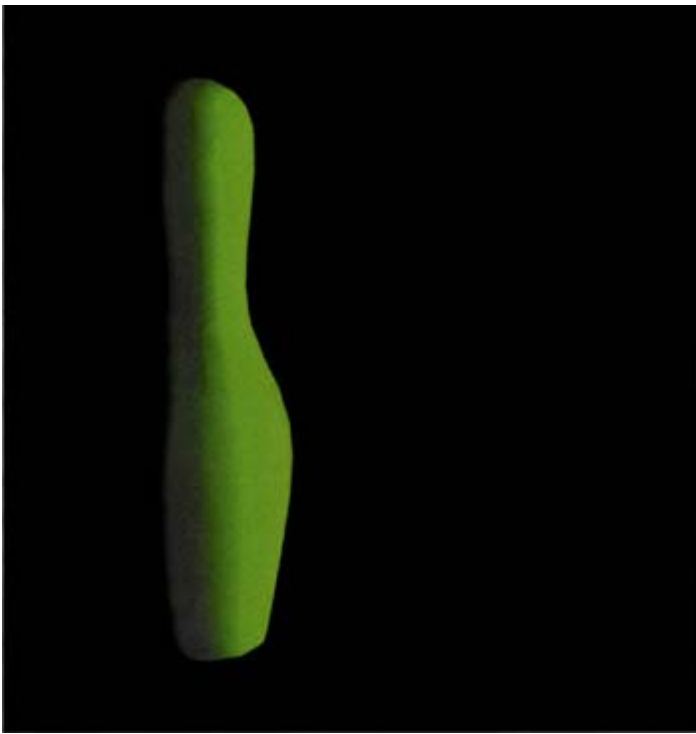


Figure 7.4: Infinite lights are the most common type of lights. The light vector for all vertices is the same.

Lambertian Diffuse Lighting with a Positional Light

This shader uses the same math that the traditional fixed function pipeline uses to generate the diffuse lighting term and a positional (or local) light. The diffuse light color and diffuse material color are specified.

Uses Use this to generate the diffuse term for a local light shining on a vertex. When combined with a Blinn-Phong specular term, you'll get the same results as the fixed function pipeline does.

Description The only difference between this shader and the previous one is that this one has a positional light. Diffuse lighting doesn't depend upon the viewing angle, only the angle between the surface normal and the light direction. In this case, we've got our light position in world coordinates—the same as our vertex position and normal. Thus we can skip transforming them since we can create a light directional vector in world coordinates, and dot that with our normal, which is also in world coordinates. The angle between the normal and light direction will be the same as long as they are in the same coordinate system when we take the dot product.

Prerequisites All the assumptions for use of component 2. The material diffuse color is passed in as a constant. The light diffuse color is passed in as a constant. The light position is passed in as a constant in world coordinates. `v0` is vertex position. `v3` is the normal.

Results The diffuse color is written to `oD0` so that it can be used in the fixed function pipeline. The result of the shader is shown in [Figure 7.5](#).

```
// Lambertian Diffuse Shader with local light

vs.1.1

// transform the vertex
m4x4 oPos, v0, c[TRANS_MVP]

// compute the light vector
// first light pos - vertex pos
sub r0, c[LOGHTPOS], v0
// then normalize it
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0, r0, r0.w // r0 = |l|

// compute |n| dot |l|
dp3 r5.x, r0, v1 // r5.x = n dot l
// no need to clamp in next step
```

```
// - it's done automatically

// compute diffuse
mov r10, c c[MATERIAL_DIFFUSE]
mul r11, r10, c7
// r11 = diffuse material color times
// diffuse light color

// modulate by |n| dot |l|
mul oD0, r11, r5.xxxx
```

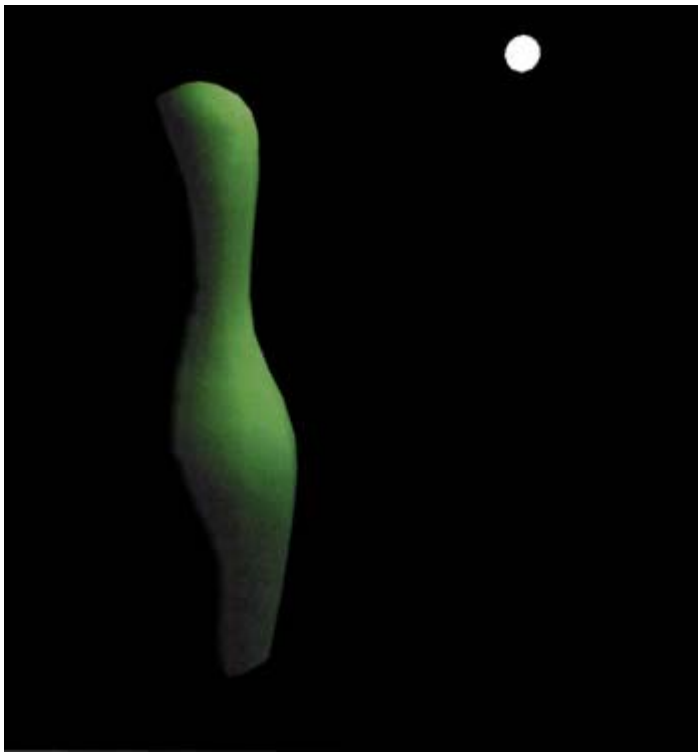


Figure 7.5: A positional light means calculating a unique light direction vector for each vertex. For light sources close to a surface, this overhead is sometimes necessary to get the light looking correct.

Blinn—Phong Specular Lighting

This shader uses the same math that the traditional fixed function pipeline uses to generate the specular lighting term. It uses the built-in `lerp` function to calculate the lighting term.

Uses Use this to generate the specular term for a light shining on a vertex. When combined with a Lambertian diffuse term, you'll get the same results as the fixed function pipeline does.

Description This is the implementation of Blinn's simplification to Phong's lighting equation. It requires computation of the half-angle vector. The specular term generated is

$$i_s = (m_s \otimes s_s)(\hat{n} \bullet \hat{h})^{m_s}$$

where m_s is the power value, usually referred to as the "shininess" parameter. This value can be in the range [0, 128]. And \hat{h} is the normalized half-angle vector, computed from the light direction.

$$\hat{h} = \frac{\hat{l} + \hat{v}}{|\hat{l} + \hat{v}|}$$

Prerequisites All the assumptions for use of component 2. The material specular color is passed in a constant. The light specular color is passed in a constant. The light position is passed in a constant, with the specular power value tagging along in the w component of the constant.

Results The specular color is written to `oD1` so that it can be used in the fixed function pipeline. This is a render state that's turned off by default. [Figure 7.6](#) shows the specular only term, and then the specular term plus ambient and diffuse from previous shaders.

```
// Phong-Blinn Specular

vs.1.1

// transform the vertex
m4x4 oPos, v0, c[TRANS_MVP]

// compute the light vector
// 1st light pos - vertex pos
sub r0, c[LIGHT_POS]. v0
// then normalize it
dp3 r0.w, r0, r0
```

```

rsq r0.w, r0.w
mul r0, r0, r0.w // r0 = |l|

// compute the view vector
sub r1, c[EYE_POS], v0
dp3 r1.w, r1, r1
rsq r1.w, r1.w
mul r1, r1, r1.w // r1 = |v|

// compute half vector
add r3, r1, r0
mul r3, r3, c12.yyyy // r3 = |h|

// compute n dot l
dp3 r5.x, r0, v1 // r5.x = n dot l
max r5.x, r5.x, c12.x // max(n dot l ,0)

// compute n dot v
dp3 r5.z, r1, v1 // r5.z = n dot v
max r5.z, r5.z, c12.x // max( n dot v, 0)

// compute n dot h
dp3 r5.y, r3, v1 // r5.y = n dot h
max r5.y, r5.y, c12.x // max (n dot h, 0)

// the lit instruction expects:
// r5.x = n dot l
// r5.y = n dot h
// r5.z = ignored
// r5.w = shininess value

mov r5.w, c13.x // move shininess in

```

```
lit r6.z, r5 // result is in r6.z

// compute specular
// multiply light color * material specular
mov r10, c[MATERIAL_SPECULAR]
mul r9, r10, c[LIGHT_SPECULAR]
// attenuate by specular intensity
mul oD0, r9, r6.zzzz
```

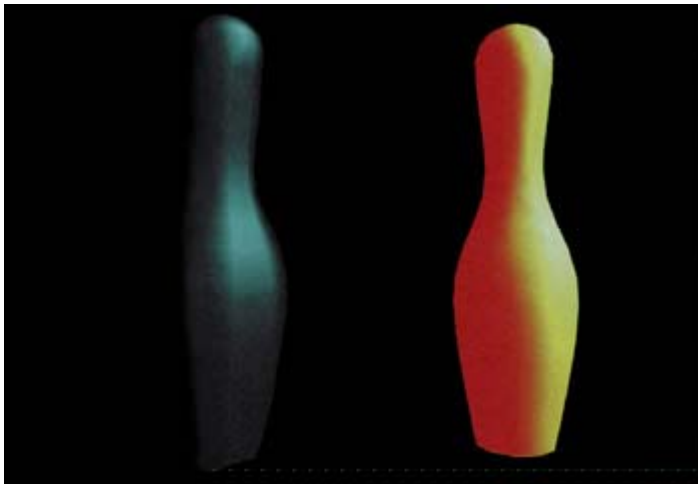


Figure 7.6: Blinn-Phong specular on the left and then the ambient, diffuse, and specular terms combined on the right.

Decal Texture Shader

A decal texture is a texture that's not affected by the color values of a vertex, but the texture itself is used to calculate the color of the vertex through use of the vertex's texture coordinates. Any color values from the vertex shader are ignored. For this example, we need to assume that when the shader is run there is a single texture active and that there is no computation using the vertex's color. This means that we're using a rendering setting of `D3DTEXTURE_STAGE_SELECTARG1` for DirectX. Thus our first decal shader will simply transform the position vector (as all vertex shaders must do) and then pass along the texture coordinates from the vertex stream.

Uses Use this to apply a texture as a decal (i.e., no shading).

Description This shader will result in rendering an object where the intensity of the color is unaffected by any lighting conditions. You'd use this kind of effect when you were rendering textured objects in an

environment where the lighting was constant and uniform, or when you always want the object to be seen, such as with a HUD or a user interface.

Prerequisites Texture 0 is valid and loaded. v_0 is the vertex position and v_7 is the texture coordinate. The texture used for the bowling pin is shown in [Figure 7.7](#).

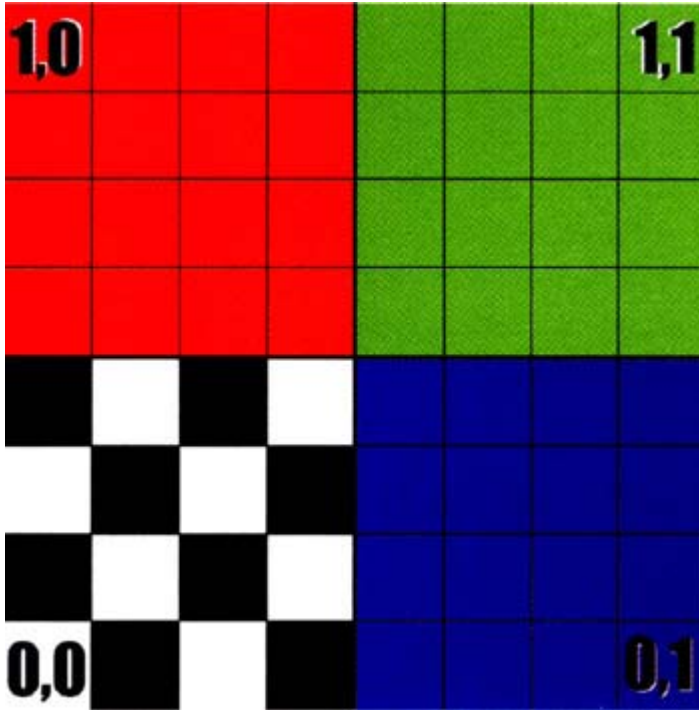


Figure 7.7: The decal we're going to wrap the object with.

Results No color is computed for this vertex; the texture coordinates are passed to the FFP or pixel shader, and the texture is sampled. The result of the shader is shown in [Figure 7.8](#).

```
// Decal texture.vsh

vs. 1.0    // Shader version 1.0
m4x4  oPos, v0, c4 // emit projected position

// it's assumed that the texture blending is set
// to D3DTOP_SELECTARG1 so we just decal the texture
// and ignore any color calculations

mov    oT0.xy , v7    //copy texcoords
```

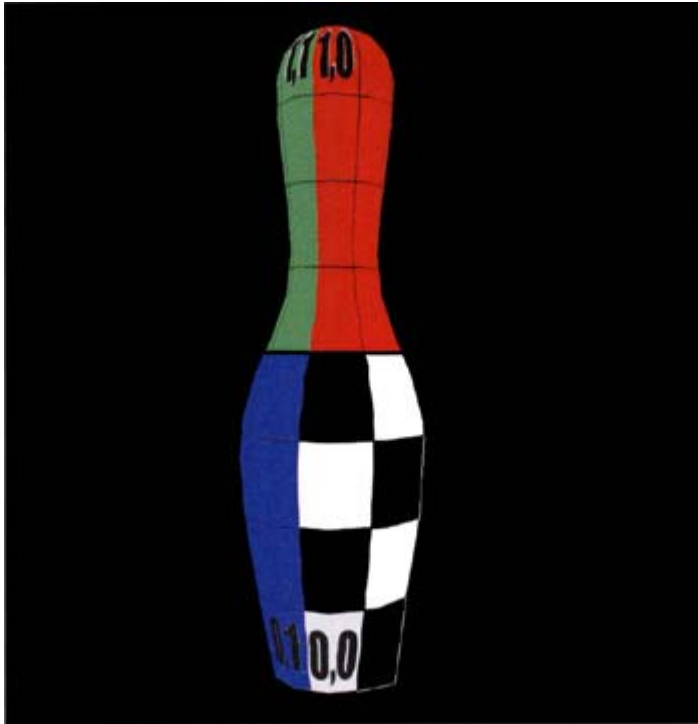


Figure 7.8: Decal texturing slaps a texture onto a surface with no other coloring effects.

Color Modulated Decal Shader

The advantage of the shader listed earlier is that we don't need to perform any lighting calculations on it—it simply shows up on top of the object. Unfortunately, most interesting environments aren't like that. We want our textures to blend into an object that's affected by the environment. The simplest effect on an object is taking into account the object's color—that is, the color of an object is blended or *modulated* (i.e., multiplied) by the texture.

Uses Use this to generate a multicolor object blended with a texture.

Description This shader performs a blend between a texture and a vertex color. In order for the graphics pipeline to know what we want it to do, we need to use the `D3DTOP_MODULATE` for DirectX. The only real difference in this shader is that we're calculating a color and sticking it in the output color register. If you don't tell the pipeline to use the color register, then calculating a color value will have no effect on the output (except to slow down the shader). The FFP will observe any texture blending states you may set. If a pixel shader were used, it would have to be written to do the blending.

We have a texture that is multiplied with our color, and this might or might not be what you want. Most of the time, if this is your first attempt to blend colors with textures, you are disappointed because the texture appears much darker than you'd expect. This is an effect of the color math since all of our colors

are floating point values in the range [0,1]. If you take a texel intensity of 0.5 and a vertex intensity of 0.5 and multiply them, you get a color intensity that's 0.25—typically, *not* what you want.

There are various ways to get around this. If you're applying a texture over an entire object, you can increase the brightness of the base object color. If you're applying a texture to a small area, then you can make the texture overbright. DirectX has some blending operations that will double (D3DTEXTURECOMBINE_MODULATE2X) or quadruple (D3DTEXTURECOMBINE_MODULATE4X) the resulting colors while still clamping them to a maximum of 1.0. You can also fix this in the pixel shader.

Prerequisites Object color is passed in register `v4`. Texture 0 is valid and loaded. `v0` is the vertex position, and `v7` contains the texture coordinates.

Results The color is written to `oD0` so that it can be used in the fixed function pipeline. The result of the shader is shown in [Figure 7.9](#).

```
// Vertex Color Shading.vsh

vs.1.0    // Shader version 1.0

// emit transformed position
m4x4  oPos, v0, c[TRANS_MVP]

// stick the color into the output diffuse register
mov  oD0, v4

mov  oT0.xy , v7    //copy texcoords

// How these get blended will depend upon
// render states and/or pixel shader code
```



Figure 7.9: To blend a texture with a surface color, you'll need to modulate the texture color.

Fog Shader

The value is the fog factor to be interpolated and then routed to the fog table. Only the scalar x-component of the fog is used. You cannot use vertex shader fog with a pixel shader unless you perform the calculations yourself. The oFog register bypasses the pixel shader code and is used after the pixel shader when fog calculations are processed by the pipeline.

Uses Use this to set per vertex fog values that will be processed by the pipeline after the pixel shader has run.

Description When applying vertex fog, fog calculations are applied at each vertex in a polygon, and then interpolated across the face of the polygon during rasterization. The fog value is clamped to the [0, 1] range when the shader exits.

In order to use fog, it must be enabled. In DirectX, this is a call to set the render state with the `D3DRS_FOGENABLE` parameter. Vertex fog must be used when setting the fog intensity using the vertex shader. You also should set the fog parameters and fog color as well.

```
// Enable fog mode blending and color.

m_pd3dDevice-> SetRenderState(D3DRS_FOGENABLE, TRUE);

// select linear fog and parameters

m_pd3dDevice-> SetRenderState(D3DRS_FOGVERTEXMODE,

    D3DFOG_LINEAR);

Float FogStart = 0.2f, FogEnd = 0.9f;

m_pd3dDevice-> SetRenderState(

    D3DRS_FOGSTART, (DWORD*)&FogStart);

m_pd3dDevice-> SetRenderState(

    D3DRS_FOGEND, (DWORD*)&FogEnd );

// Set the fog color to red.

m_pd3dDevice->SetRenderState(D3DRS_FOGCOLOR,

    D3DCOLOR_XRGB(255,0,0) );
```

Prerequisites The fog render states and parameters must be set.

Results You see that the object is rendered in a fog depth in [Figure 7.10](#).

```
// Fog Shader.vsh

// This fog shader places the vertex into the fog depth
// based upon the vertical height of the vertex. The
// closer the vertex to y=0, the thicker the fog. The
// object sticks out of the fog.

vs.1.1

// transform the vertex
m4x4 oPos, vO, c[TRANS_MVP]
```

```

// compute the light vector
// 1st light pos - vertex pos
sub r0, c[LIGHT_POS], v0
// then normalize it
dp3 r0.w, r0, r0
rsq r0.w, r0.w
mul r0, r0, r0.w // r0 = |1|

// compute n dot 1
dp3 r5.x, r0, v1 // r5.x = n dot 1
max r5.x, r5.x, c12.x // max(n dot 1, 0)

// output n dot 1 as the vertex color
// this gives a gray color scheme
mov oD0, r5.x

// now compute z value (since RenderMonkey orients
// with z = vertical)

// shift vertices up by c6.z
mov r0.z, v0.z
add r0.z, r0.z, c6.z

// c6.x is fog max position
// compute fog max minus shifted vertex position
// divided by fog max [0, 1] range
// (max-vertex.z)/max
rcp r3.x, c6.x
mad r3.x, r0.z, -r3.x, c5.z

// Fog.x is the fog value mov oFog.x, r3.x

```



Figure 7.10: Fog is done after the texture stage. In vertex shaders, you can just set the vertex fog intensity. Here, the fog is red and the pin is standing out of the fog.

Point Size Shader

Point primitives may sound fairly uninteresting, but they show their worth by the fact that they require very little in the way of generation or rendering. This allows you to generate a great deal of point primitives and thus add a great deal of "detail" to a scene. For this reason, point primitives are usually used in particle systems. Since the "point" is always oriented toward the viewpoint, you can always be sure that your point will be seen, and you can thus judge how dense to make the particles to get the effect you want.

Uses Use this shader to generate small point sprites. Since the maximum value of the point is fairly small, this limits using the point size as a replacement for a billboard.

Description Unlike the other registers, the point size register opts can be written only into the x channel (since it's really just a single scalar register). Also unlike the other registers, there's no clamping to a $[-1, 1]$ or $[0, 1]$ range. The range maximum depends upon the current device, but the range will be $[0, \text{MaxPointSize}]$. Any values that are less than 1 that do not cover a pixel center will not get rendered.

Point sprites ignore any texture coordinate you might specify and allow you to use a complete texture to cover a point. Our first point size pixel shader will show how the point size can vary. We'll let the point size vary between 0 and 64. In your program, you'd probably want to vary the point size so that it'd be dependent upon the distance from the viewpoint. In this shader, we're going to generate the maximum value for our points by using some of the commands in the shader language. Since 64 is a power of 2,

we can use part of the `lit` instruction to do this for us. We can take our constant register value of 2 (held in the `c0.w` channel) and use the `mad` instruction to generate the value 6. We can then pass the 2 and 6 constants to `lit` and let it generate 2^6 or 64 for us. We can then multiply by our fractional clock time to vary this value over the [0,64] range and store the result into `oPts.x`.

Prerequisites The diffuse color and inverse matrix are stored in constants. There's a constant that's set per frame, with the `y` element being the fractional seconds in the range[0.1], plus a series of constants set in a register, with the `y` component being equal to 2.

Results You'll get a point sprite that ranges from 0 to 64 pixels across with a frequency of one second. The results of the shader are shown in [Figure 7.11](#).

```
// Point Size Shader.vsh

vs.1.1

// This shader hooks RemderMokey's time-varying cos
// value to c5 and multiplies that abs (value)
// by MaxPointSize. This value is typically 64 pixels
// clamped by the hardware. I set the color to a row of
// the transformation matrix.

m4x4 oPos, v0, c[TRANS_MVP]
// semi-random output color

mov oDO, c[TRANS_MVP+1] // output color

max r0, c5, -c5 // abs(cos)

// only the x component can be set
// c4.x is MaxPointSize
mul oPts.x, c4.x, r0.x
```

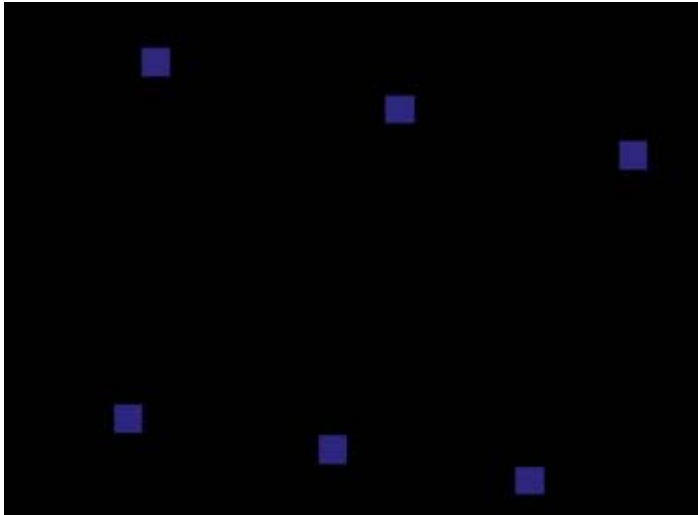



Figure 7.11: Point shaders are somewhat limited since point size has only a fixed maximum value.

Multitexture Shader

A multitexture shader is just a shader that applies two or more textures. Depending upon the blending, you can decide exactly how the two textures interact.

Uses Use this to apply a texture as a decal, with other textures modifying the initial texture, either as a *dirt* or *scuff* map, a specular map, or as a bump map.

Description This shader shows how to use multiple textures. The vertex shader sets up the usual things, but we've allowed it to hold a random value in the $[0, 1]$ range that can be used to morph the texture coordinates of one of the textures. We reuse the original texture coordinates for the scuff map and make it randomly oriented around the y axis by moving the u coordinate. The offset is set outside the shader.

Prerequisites Texture 0 is valid and loaded. v_0 is the vertex position and v_7 are the texture coordinates. The texture used for the decal is shown in [Figure 7.12](#).



Figure 7.12: The bowling pin texture map.

With that texture used as the decal texture, we're going to apply a scuff texture. We're going to need texture wrapping enabled to allow for texture coordinates to go outside the [0,1] range. The texture we're going to use as the scuff map is shown in [Figure 7.13](#).

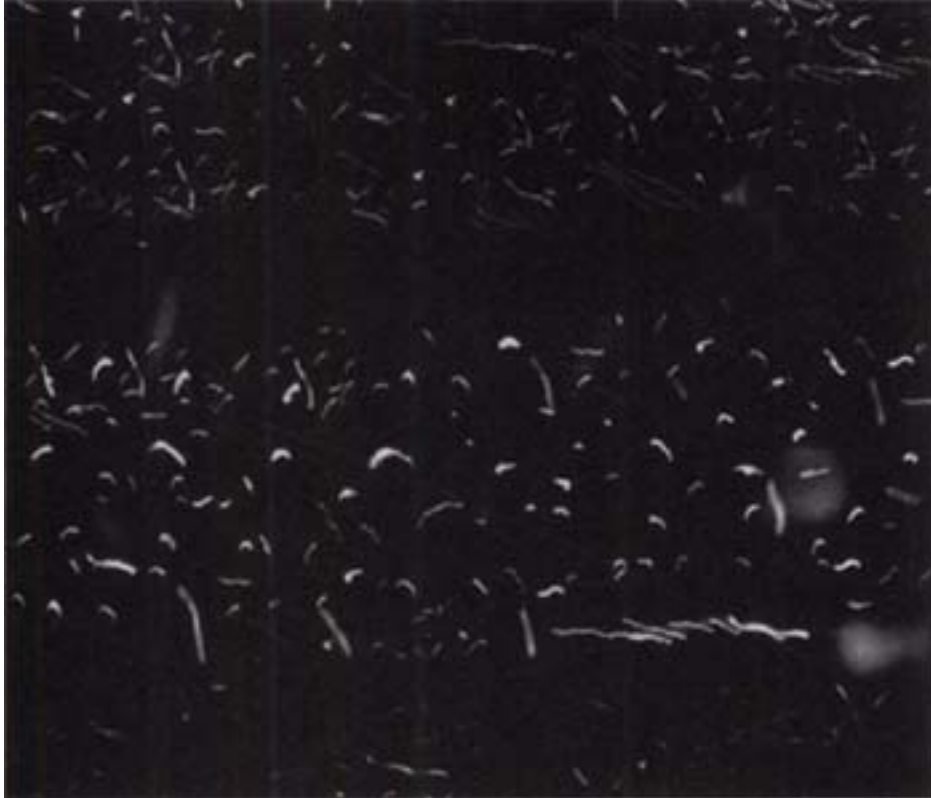


Figure 7.13: The scuff map.

Results The results are shown in [Figure 7.14](#). You can see that in addition to our decal texture, we've added a scuff map to give it a beat-up appearance.

```
// multitexture.vsh

vs.1.0    // Shader version 1.0
m4x4 oPos, v0, c4 // emit projected position

// copy decal texcoords
mov  oT0.xy , v7    //copy texcoords

// we want the scuff map to be randomly
```

```
// oriented around pin, but only in the u direction
// so we modify the u coordinate

mov r0, v7

add r0.x, r0.x, c[OFFSET].x

mov oT1.xy , r0 // copy texcoords
```



Figure 7.14: The pin with the texture map and the scuff map added on top of it.

If you don't need any blending modes other than those provided by the FFP, then there's no need to use a pixel shader. One advantage of using a pixel shader is that the shader determines how the textures get blended so you generally don't have to worry about render states for the blending modes.

```
// multitexture.psh

ps.1.1 // Shader version 1.1

// sample the textures
texld r0, t0.xy // decal map
texld r1, t1.xy // scuff map
add r0, r1 // subtract scuff colors
```

Silhouette Edge Shader

This shader is a simple one for rendering silhouette edges. It's designed to be rendered on your object as a separate pass since all it does is render the silhouette edges. The actual shading of everything else needs to be done as another pass over the vertices. It requires alpha blending to be enabled.

Uses Use this to generate a silhouette edge for well-tessellated objects since this shader depends upon detecting when a vertex's normal is perpendicular to the view vector.

Description This shader assumes that when a vertex's normal is perpendicular (or nearly so) to the view direction, that it's part of a triangle that's on the edge of the object; hence you need a reasonably smooth object for this approach to work. Curved objects work better than objects with flat edges. The best objects contain curves that are all a similar degree of curvature since quickly curving surfaces provide fewer edge pixels.

The dot product of the view vector to the normal vector is calculated for a vertex. It's then rescaled from the $[-1,1]$ range to the $[0,1]$ range and placed in the diffuse color's alpha channel. The degree of thickness is controlled by the alpha reference value set using the `D3DRS_ALPHAREF` state. The vertex's alpha value (range $[0,1]$) is then compared to the alpha reference value (range $[0x00,0xFF]$) to determine if the vertex should be rendered. Since this is done on the interpolated values in the alpha-blending stage this is a per pixel test.

Prerequisites All the assumptions for use of component 2. A constant is set up that contains the opaque (outline) color and some required range conversion values. The ModelView matrix is passed in along with the MVP matrix.

Since you'll render the internal shading in another pass, the vertex position for this pass must be performed in exactly the same manner as any other passes or else z fighting may occur. You need to enable alpha blending, specify the blending test, and set the silhouette edge threshold as `dwSilhouetteAmount` render state as in this code.

```
m_pd3dDevice-> SetRenderState(D3DRS_ALPHATESTENABLE,
                               TRUE) ;

m_pd3dDevice-> SetRenderState(D3DRS_ALPHAFUNC,
                               D3DCMP_GREATEREQUAL) ;

m_pd3dDevice-> SetRenderState(D3DRS_ALPHAREF,
                               dwSilhouetteAmount ) ;
```

Results The diffuse color is written whenever the alpha threshold is crossed. Rendering all edges of the object, the output of the silhouette shader is shown in [Figure 7.15](#).

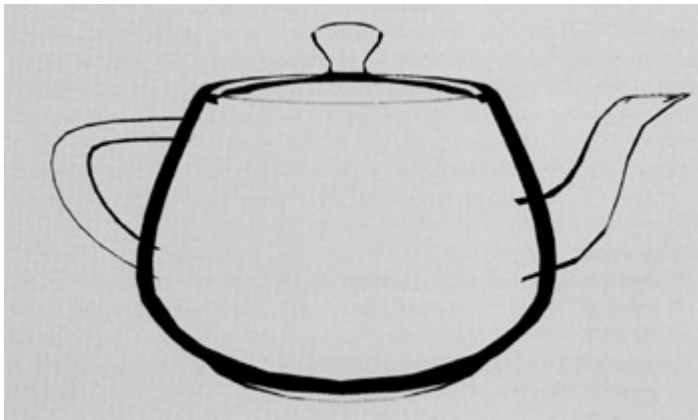


Figure 7.15: A simple silhouette shader.

```
// Silhouette Edge Shader

vs.1.0 // Shader version 1.0

// emit projected position

// It's assumed that this is how you calculated
// the position for any other pass of the object
m4x4 oPos, v0, c[TRANS_MVP]

// Now ModelView matrix

// calculate normalized position vector
m4x4 r0, v0, c[TRANS_MV]

dp3 r0.w, r0, r0
```

```

rsq    r0.w,    r0.w

mul    r0, r0, r0.w // r1=|p|

// and normalized normal vector

m3x3   r1, v3. c[TRANS_MV]

dp3    r1.w,    r1, r1

rsq    r1.w,    r1.w

mul    r1, r1, r1.w // r1=|n|

// calculate normalized tangent vector

dp3    r2.w,    r0, r1

// convert it from the [-1,1] range to [0.1] range

// and place it in the output alpha

// c[CONST].x == 0.5

mad    oDO.w, r2.w. c[CONST].x, c[CONST].x

// place silhouette color in the rgb output

mov    oDO.xyz, c[SILHOUETTE_COLOR]

```

VERTEX AND PIXEL SHADERS

There are a variety of techniques that require processing by both vertex and pixel shaders. This is the most common technique for shader usage.

Cartoon Shading

Cartoon shading is just trying to get an object shaded as though it were drawn in the comic book style. It's typified by a simple two or three color shading scheme, occasionally with an emphasized object outline.

Uses This technique allows you to specify the color of an object based upon lighting calculations, allowing you to specify the number of shading contours, where the contours occur according to the shading level and the color of the contours.

Description This technique uses a vertex shader to calculate the $\mathbf{n} \cdot \mathbf{l}$ value and to pass that value into the pixel shader as a texture coordinate. In this example, we do not use the texture coordinate to sample a texture—no textures are loaded, we just use the texture coordinates as an indication of how illuminated the pixel is and then choose the color based upon that value.

Prerequisites Object contour colors and thresholds are set in the pixel shader as constants. $\mathbf{v0}$ is the vertex position; the normal is passed in as $\mathbf{v3}$.

Results This shader uses a Lambertian calculation to compute the diffuse lighting intensity and passes that value to the pixel shader through a texture register. The output is shown in [Figure 7.16](#).

```
// Real - Time Shader Programming

// toon shader.vsh

// calculation in the pixel shader.
// Pass out the normal, light and half angle vectors in eye
// space coordinates in three texture coordinates.

vs.1.1

// transform the vertex
m4x4 oPos, v0, c[TRANS_MVP]

// compute the normal in eye space
m3x3 r0, v1, c[MV_MATRIX]
mov oTO, r0 // write the normal to tex coord 0

// compute the light vector
sub r0, c[LIGHT_POS], v0
dp3 r1, r0, r0
```

```

rsq r1, r1
mul r0, r0, r1 // r0 = |l|

// transform the light vector into eye space
m3x3 r1, r0, c[MV_MATRIX]
mov oT1, r1 // write the light vector to tex coord 1

// compute half vector
// transform the vertex position into eye space
m4x4 r0, v0, c[MV_MATRIX]
dp3 r3, r0, r0 // normalize to get the view vector
rsq r3, r3
mul r0, r0, r3 // r0 = |v|

add r0, r1, -r0 // add the light vector and the view vector =
    half angle
dp3 r3, r0, r0 // normalize the half angle vector
rsq r3, r3
mul r0, r0, r3 // r0 = |h|

// write the half angle vector to tex coord 2
mov oT2, r0

```



Figure 7.16: Cartoon shading using traditional lighting equations and a three-stage color graduation.

This shader computes the light intensity as $\mathbf{n} \cdot \mathbf{l}$ and places it in texture 0's u component. The next step can be done in a number of ways. You could load the shading gradient you want in a 1D texture as texture 0 and use that without a pixel shader. Or you can use the opportunity to code a pixel shader and let the shader code determine the colors. You can then load the colors and contour control values or do what we've done here, program them directly into the pixel shader.

```
// toon shader.psh

ps.1.4

// get the normal in r0
texcrd r0.xyz, t0

// get the light vector r1
texcrd r1.xyz, t1

// get the half angle vector in r2
texcrd r2.xyz, t2

// r3 = n dot l
dp3 r3, r0, r1

// subtract cutoff1 from n dot l
sub r3, r3, c0

// if greater than zero use lit color (c3) otherwise
// use the unlit (c2)
// store result in r5
cmp_sat r5, r3, c3, c2

// r4 = n dot h
```

```

dp3 r4, r0, r2

// subtract cutoff2 from n dot h
sub r4, r4, c1

// if greater than zero use specular color (c4)
// otherwise previous color
// store result in output register r0
cmp_sat r0, r4, c4, r5

```

Dot-3 Bump Mapping

Dot-3 bump mapping is the most commonly used style of bump mapping. It uses a texture called the *normal map* to store the bump textures. Instead of rgb values, these are the actual xyz normal perturbations that are going to be applied on the surface. In previous versions of DirectX, this was supported directly by the D3DTOP_DOTPRODUCT3 blending operation.

Uses This shader demonstrates how to create a dot-3 bump mapping.

Description Takes a normal map texture and modulates the diffuse color texture map by the dot product of the light vector and the normal vector.

Prerequisites `v0` is the vertex position; the normal is passed in as `v3`. The textures are the same ones used in the multitexture shader.

Results The result of the shader is shown in [Figure 7.17](#).

```

// Dot - 3 Bump.vsh

vs.1.0 // Shader version 1.0

// emit transformed position
m4x4 oPos, v0, c[TRANS_MVP]

// Transform normal and tangent

```

```

m3x3  r7,v8,c0
m3x3  r8,v3,c0

// Cross product
mul  r0,-r7.zxyw,r8.yzxw;
mad  r5, -r7.yzxw,r8.zxyw,-r0;

// Transform the light vector
dp3  r6.x,r7,-c16
dp3  r6.y,r5,-c16
dp3  r6.z,r8,-c16

// Multiply by a half to bias, then add half
mad  r6.xyz,r6.xyz,c20,c20

mov  oT0.xy,v7.xy
mov  oT1.xy,v7.xy
mov  oDO.xyz,r6.xyz

```



Figure 7.17: Taking a bump map to generate normal perturbations that look like real geometry

except at the edges of the object.

The pixel shader reads in the texture map and the normal map. The normal is passed in as the v0 register, and then it's used with the light vector to generate a dot product. The dot product is then used to modulate the base texture.

```
// Dot - 3 Bump.psh

ps.1.1    // Shader version 1.1

tex t0     // Sample texture
tex t1     // Sample normal
mov r0,t1

dp3 r0,t1_bx2,v0_bx2; // Dot(light,normal)
mul r0,t0,r0          // Modulate against base color
```

Fresnel Shader

This technique comes from the DirectX Shader Workshop from Meltdown 2002. It approximates the Fresnel term by computing the term as

$$F = (1 - e \cdot n)^4$$

which is done in the shader rather than through a texture lookup. This shader gives a simulation of the Fresnel edge effect.

Uses Use this shader to add a simulated Fresnel term with minimal effort.

Description This shader computes an edge effect. It can be added to an existing shader to generate something that looks like a Fresnel effect. It uses the dot product of the view direction with the vertex normal to add in a specified color.

Prerequisites v0 is the vertex position; the normal is passed in as v3. In addition to the MVP matrix, we'll need the MV matrix and an eye position input as constants.

Results The Fresnel shader effect is shown in [Figure 7.18](#). [Figure 7.19](#) shows the Fresnel shader used with a diffuse and specular shader.

```
// Fresnel . vsh
```

```

vs.1.0 // Shader version 1.0

// emit transformed position
m4x4 oPos, v0, c[TRANS_MVP]

// Transform position and normal to world space
m4x4 r0, v0, c[TRANS_M]
m3x3 r1, v3, c[TRANS_M]
// Compute normalized view vector
add r0, -r0, c[EYE_POS] // vector
dp3 r0.w, r0.xyz, r0.xyz
rsq r0.w, r0.w
mul r0, r0, r0.w // r0 = |v|

// Compute e dot n
dp3 r0, r0, r1;

// compute the color complement
add r0, c[COLOR], -r0

// cube the value
mul r1, r0, r0
mul r0, r1, r1

// multiply by Fresnel and place in output
mul OD0, r0, c[FRESNEL]

```

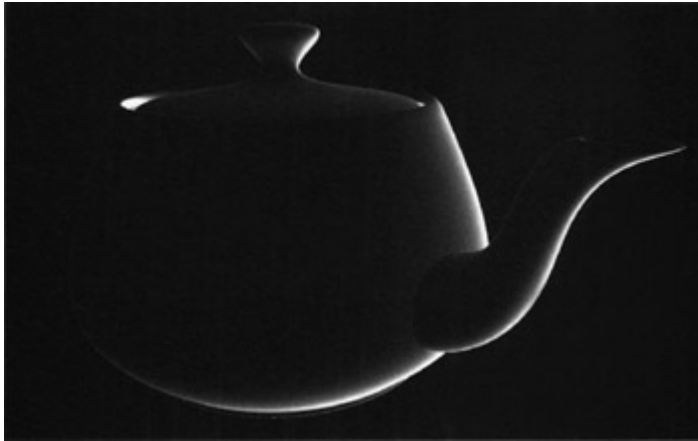


Figure 7.18: A simplified Fresnel shader used to generate a shiny specular when an object is illuminated from behind.



Figure 7.19: The Fresnel shader output added to the ambient, diffuse, and specular output.

Chapter 8: Shader Reference

OVERVIEW

Results! Why, man, I have gotten a lot of results. I know several thousand things that won't work.
--Thomas A. Edison

Chapter 8 contains detailed explanations about the vertex and pixel shader registers, instructions, macros, and other operations that can be programmed into shaders. There's a general discussion on coding style and other issues that are common to both pixel shaders and vertex shaders. Note that there's a section in this chapter that discusses new and updated shader features by DirectX release, which compliments the features and changes by shader version that can be found in the introduction to each of the vertex and pixel reference collections.

In addition, each vertex instruction contains C-like pseudocode that should give you a concrete example of how the instruction actually works. Since some of the instructions have a low precision and a high precision output simultaneously, there's a bit mask that simulates this precision restriction on the output as well. Each of these pseudocode sections look like this.

```
SetSourceRegisters();

// Simulate some shader instruction

WriteDestinationRegisters();
```

The `SetSourceRegisters()` section is some "magic" code that will take the input arguments and any modifiers (swizzles negates, etc.) and write them into some temporary source and destination register placeholders. The pseudocode then operates on that entire register (when appropriate). Then there's the `SetDestinationRegisters()` section that will take into consideration any destination write masks on the instruction argument and only write out those register elements.

In an effort to keep things up to date, I have provided an online version of this reference at <http://www.directx.com/shader/reference/>.

SHADER STYLE AND COMMENTS

You can place C-style comments in your code. Just place two forward slashes (`//`), and anything after that till the end of the line will be ignored. The tokenizer will remove comments and ends of lines from your code as it works, so this means that it will read past ends of lines until it completes an instruction. That is, it treats all white space the same, so you can be indiscriminate in using any kind of white space. This means that things like the following example are legal:

```
// a single line with a comment at the end
mov r1, c[5+a0.x] // comment

// can be written as...

mov r1    , c
[5        // five
+         // plus the
```

```
a0          // address register's (can't break up)
.
x           // x element
]
```

Any errors that are reported when you assemble lines like this might be reported on the line that begins the actual instruction, not the line that contains the error.

All instructions and register names are case sensitive. This means that all instructions to be used in a DirectX shader have to be lowercase and that register names have to be mixed case exactly as they are described.

Note

You'll occasionally see in early examples the "asm" style comments—a semicolon followed by a comment. This is currently still accepted by the assembler but is not a supported comment type.

SHADER REVISION HISTORY BY DIRECTX VERSION

Here is the revision history by DirectX version. Each section contains the shader version changes from the previous version. For what actually changed in each shader version, you'll need to check the revision history in the vertex (Part I) or pixel (Part II) shader sections of this chapter.

DirectX 9.0

Vertex and pixel shader versions 2 and 3 added. Added requirement that some registers be declared before use for all shader versions.

DirectX 8.1

No change in vertex shaders. Pixel shader versions 1.2, 1.3, and 1.4 added.

DirectX 8.0

Introduction of pixel and vertex shaders. Pixel shader versions 1.0 and 1.1 and vertex shader versions 1.0 and 1.1 supported.

Part I: Vertex Shader Reference

A vertex shader is made up of a set of instructions that operate on data for a single vertex. The shader is executed once for each vertex in a render primitive. A vertex shader will have one vertex input provided to it and will be expected to have one vertex output.

QUICK OVERVIEW

A vertex shader must do at least one thing: place the vertex's position in clip-space coordinates in the output position register. That is, the vertex position must be transformed by the current model/world/view transformation matrix. Now if your world, model, and projection matrix were identity, then you could just pass in the value that is fed to the shader, but usually you'd be getting the vertex position in world coordinates, and you'll need to transform the point and store that in the output register.

And though this will make the shader assembler happy, it doesn't do anything about describing how the vertex *looks*. So the second thing that is necessary to get it to be a useful shader is to specify how the vertex looks. There's more than one way to do this. If you are using a pixel shader, then the pixel shader decides what the pixel color will be, and you can get away with not setting it in the vertex shader. If you're using the fixed function pipeline, you should set the color either by setting the color register or by sampling a texture.

Failing to set the color (at least on hardware drivers) leaves these values undefined. On most debugging drivers and on the reference rasterizer, the color will default to black.

In addition, you'll need to specify the vertex shader version you are using as your first shader instruction to get the shader to assemble at all.

VERTEX SHADER REVISION HISTORY

Here is the revision history of vertex shaders. Each section contains the changes from the previous version.

DirectX 9.0

Version 3.0

Vertex Shader 3.0 specification was added.

Version 2.0

Vertex Shader 2.0.

The `dc1` statement must be included in all vertex shaders to decouple the vertex stream from the vertex shader. Registers (except temp and constant) must be declared before use.

Static flow control added.

Minimum instruction count increased to 256.

`al` register added.

The following macros were added: `abs`, `crs`, `lrp`, `nrp`, `pow`, `sincos`, `sgn`.

The following instructions were added: `call`, `callnz`, `else`, `endif`, `endloop`, `endrep`, `if`, `label`, `loop`, `mov`, `rep`, `ret`.

The `mov`, `expp`, `def`, `frc`, `logp` and `vs` instructions were modified.

DirectX 8.1

DirectX 8.0a

Version 1.1

Support for the `a0.x` address register was added.

Version 1.0

Initial release of vertex shaders.

INPUT REGISTERS

The input registers are how data gets passed to the vertex shader. Direct3D provides input vertex registers, which will contain input data to the vertex shader. What this data is depends upon the format of the vertex stream(s) that are currently active.

The other set of input registers is the constant registers, which are set up prior to running the vertex shader. It's up to you to correctly set up these registers. For example, you probably want to pass in the current modelview matrix, light positions, etc., prior to calling the shader.

c[n]: The Constant Registers

Aside from the actual instructions that you enter into your shader, you will most probably need to get values into those instructions, and you do it through the constant floating-point registers. They are called the float constant registers simply because they are floating point values that are read only once they are passed to the shader and the shader is running. However, you'll frequently find that you'll be setting values of the constant registers prior to the beginning of each frame of rendering (e.g., setting the world/model/projection matrix values), or even prior to each render primitive call (to set a primitive or shader-specific value).

One optimization made in the assembly of vertex shaders is that there can be only one constant register referenced from any single instruction. However, you may use that constant more than once, and each reference to the constant can independently swizzle or negate the constant's vector elements.

```
add    r5,    v0    v0           // fine
add    r5,    c1    c2           // Error! 3 different registers
add    r5,    c5    c5           // fine - doubles r5
add    r5,    c5    -c5          // fine - zero in r5
```

```
add    r5    c5.xxxx,    c5.yyyy    // OK fills r5 with x+y
```

The syntax that allowed for constant registers is also a little more complex. You can reference them like any other, or you can place the numeric part inside square brackets. DirectX 8.1 introduced the address register, `a0.x`, which allows you to have a programmable index into the constant registers. Some examples are shown next.

```
mov    oD1, c5            // fine - regular way
mov    oD1, c[5]          // using brackets
mov    oD1, c[5+a0.x]     // fine for DX8.1+
```

There are two ways to get the constant registers loaded. One way is to use the `def` statement and then (in DirectX 8) postprocess this to add the generated shader code fragment to the shader. The preferred (and easier) way is to just set the values using `SetVertexShaderConstant()` to take a vector of four floats and store them in the specified register. You can set an array of constants using this call.

There are *at least* 96 float constant registers in DirectX 8 shaders, while there are *at least* 256 float constants in DirectX 9 vertex shaders. You could check the number specified in the `MaxVertexShaderConst` member of the `D3DCAPS` structure.

DirectX 9.0 Constant update



DirectX 9 also introduced 16 integer constants and 16 bool constants as well as modifying the way that constants can be set. In addition to the `SetVertexShaderConstantx()` calls, the `def` instruction in vertex shaders 2.0 now actually does something. When the `def` instruction is encountered in a shader, the current value of the temporary register specified is pushed then set immediately to that value for the life of the shader. These are called *immediate* constants. When the shader exits the constant register is popped back to its original value. Any call to get a constant's value will necessarily be done after the shader. The post processing step required in DirectX 8 shaders is no longer supported.

vn: The Input Vertex Registers

The input vertex registers contain the vertex information that the shader is going to work on. There are a total of sixteen input registers designated as `v0` through `v15`. Each vertex register is read only and consists of a four-element floating point vector. A vertex shader instruction may access only one input register at a time, but may use that register more than once in that instruction.

The vertex information is taken from the current vertex stream(s) and loaded into the vertex registers, and then the shader is run. What is stored in them and their order are completely up to you, and will consist of things like the vertex position, normal, color values (diffuse, specular), texture coordinates, and blending values. You specify what's in each input vertex register when you declare a shader register and use it in `SetStreamSource()`. This is where the mapping between your input vertex stream and the input vertex shader registers occurs. See [Chapter 5](#) for more details on setting up a mapping between a stream and the input registers.

If you are using the FVF formats in DirectX 8, then the input vertex registers are given to the shader in the following order:

Table 8.1: FVF Vertex Shader Register Mapping

VECTOR COMPONENT	FVF TAG	SHADER DECLARATION NAME	REGISTER
Position	D3DFVF_XYZ	D3DVSDE_POSITION	v0
Blend weight	D3DFVF_XYZRHW	D3DVSDE_BLENDWEIGHT	v1
Blend indices 1 through 5	D3DFVF_XYZB1 through D3DFVF_XYZB5	D3DVSDE_BLENDINDICES	v2
Normal	D3DFVF_NORMA	D3DVSDE_NORMAL	v3
Point size	D3DFVF_PSIZE	D3DVSDE_PSIZE	v4
Diffuse	D3DFVF_DIFFUSE	D3DVSDE_DIFFUSE	v5
Specular	D3DFVF_SPECULAR	D3DVSDE_SPECULAR	v6
Texture coordinates	D3DFVF_TEX0 through D3DFVF_TEX7	D3DVSDE_TEXCOORD0 through D3DVSDE_TEXCOORD7	v through v14
Position 2		D3DVSDE_POSITION2	v15
Normal2		D3DVSDE_NORMAL2	v16

INTERNAL TEMPORARY REGISTERS

The DirectX documentation places these in the input register section, but that is misleading since these registers must be initialized before they can be used, and since the results are lost between each shader invocation, they can't be used to pass any information between shader calls.

rn: The Temporary Registers

Twelve four-element vectors are available as temporary registers. Temporary registers are unique in that you can, unlike constant registers, use as many as you like as arguments for shader instructions. Temporary registers are considered uninitialized when a vertex shader starts, and an attempt to read from a temporary register before it is written to will cause the creation of a vertex shader to fail. This also means the temporary register memory is volatile—once the shader returns, whatever values were in the temporary registers are lost.

an: The Address Registers

Address registers are designed to make it easy to index into the array of constant registers. The address registers allow you to provide a signed integer offset into the constant registers. These registers may be written to *only* by the mov instruction (*mov* in DirectX 9) and are *write only*; that is, they can be used only for indexing into the constant register array, and you can't use them any other way.



No address registers were available in VS 1.0 (DX8.0) vertex shaders, and only one address register element, a0.x, was made available in later versions.

If you use the address register and the calculated offset is outside the legal range for a valid constant register, then the value returned will be a register of zeros. The address register can contain a signed integer offset. The calculated value in the register is stored as the largest floating point integer value that is not greater than the original value. This means that for positive values the fractional part is truncated, whereas for negative values the value is modified to the next larger integer value; that is, it rounds toward negative infinity.



The address register is initialized to 0, 0, 0, 0 when a shader is entered, but DirectX 8.1 shader assembler requires you to set the value in a0.x before using it. DirectX 9 does not force you to initialize the register before you use it.

```
mov a0.x, c1.x    // fails if vs.1.0, ok for vs.1.1
mov a0, c1        // error, only a0.x is OK
mov a0.x, c1      // OK only a0.x is written
mov a0.x, v0.x    // OK
mov a0.x, r0.x    // OK
mov a0.y, r0.x    // fails if vs.1.0 or vs.1.1
```

You can use the address register by itself as an index or in conjunction with an offset. You *cannot* use it more than once or with another register. You can use it with a positive integer constant but only if they are being added; any negative sign will cause the compiler to give you a syntax error. However, the value that you mov into the address register *can* be negative.

```
mov r1 , c[a0.x]      // as index
mov r1 , c[5+a0.x]    // as offset
mov r1 , c[a0.x-5]    // Error! Can't subtract
mov r1 , c[-a0.x+5]   // Error! Can't negate
m4x4 oPos, v0, c[5+a0.x] // Can use even in macros
```

Finally, although not an instruction per se, it's useful to understand the pseudocode that you would use in the emulation of the address register assignment. Here's an example of how the mov instruction might be written in a simulator.

```
SetSourceRegisters();

// Simulate the mov a0.x, Source0.x instruction
a0.x = (int) ::floor( Source0.x );

WriteDestinationRegisters( );
```

aL: The Loop Counter

9.0

The loop counter is a scalar register that was added for control of 1 oop blocks. It's initialized from the parameters passed to the loop instruction and is a valid reference only inside a loop block—this includes inside a subroutine called from inside a loop block. The loop counter can be used for relative addressing of constants inside a loop block.

The largest change with vertex shader 2.0 is the addition of flow control. This gives the ability for iterative vertex shaders, but also the potential to create slow shaders as well. If possible you should unroll any loops inside shaders if possible.

OUTPUT REGISTERS

The output registers are where your vertex shader will place its results for further processing down the pipeline. The output registers are write only, and have a letter o prefix for "output." Only the output position registers have to be written to by the vertex shader; the others are optional.

oPos: The Output Position Registers

The oPos register consists of one four-element floating point vector that is expected to contain the vertex position in homogenous clip-space coordinates. The vertex shader must write to the output position register. These are homogenous coordinates, so the *w* element must be written to as well.

oD0, oD1: The Output Data Registers

The output data registers are two four-element floating point vectors, and are typically used to hold the diffuse (oD0) and specular (oD1) color values generated from the shader. Both of these registers are (possibly) interpolated between the vertices of the triangle before being passed to the pixel shader as color registers *v0* and *v1*.

If you are also using a pixel shader, then there is absolutely no reason that these values have to be color values. If you want to pass one or two four-element vectors (possibly interpolated) to your pixel shader, the output destination registers are the way you'd do it. As long as the pixel shader generates some color value as output, it doesn't have to come from the vertex data. But keep in mind that those registers are usually lower precision and clamped to the [0,1] range.

oTn: The Output Texture Coordinate Registers

There are eight texture coordinate registers represented as eight four-element floating point vectors. They should contain the texture coordinates to be used by the pipeline for this vertex.

Write only as many register elements as there are texture coordinates in the input texture map. For example, for a 2D texture you would write only to the *.xy* elements using a mask. When texture projection is enabled, you'll need to write to all elements of the texture register.

For PS 1.4 or later you should not select the `D3DTTFF_PROJECTED` flag when using the programmable pipeline. This flag instructs the pipeline to divide all the elements by the last element (a perspective divide in the hardware) before rasterization takes place, and is intended to be used as part of the FFP for projected textures.

Texture coordinates for a singly tiled texture will usually be in the range [0,1]. If you are tiling a large area, you might run into a situation where your texture coordinates run into large multiples of this range. This limit can be found for a particular device by examining the `MaxTextureRepeat` member of the `D3DCAPS` structure. Note that the `MaxTextureRepeat` value's interpretation is affected by the `D3DPTURECAPS_TEXREPEATNOTSCALEDDBYSIZE` capability bit.

However, if the texture coordinates are to be read by a pixel shader using the `texcoord` or `texcrd` instructions, then the texture coordinate range might depend on the instruction and the pixel shader version since the pixel shader registers have less precision than the vertex shader registers. Pixel shaders version 2.0 have higher precision.

oFog: The Output Fog Register

The output fog register is a single floating point value indicating the vertex fog intensity. Note that this is used by *vertex* fog (the render state `D3DRS_FOGVERTEXMODE` is set and `D3DRS_FOGENABLE` is *true*), not *table* fog. The value in the register is the fog factor to be interpolated and then routed to the fog blend. Use the `oFog.x` element to set this value. On output from the shader, the fog register is clamped to the [0,1] range. The vertex fog value is used after the pixel shader. The value is inaccessible to the pixel shader.

oPts: The Output Point Size Register

The output point size register. Only the `oPts.x` element is used. The point size maximum can be determined by examining the `MaxPointSize` member of the `D3DCAPS8` structure. If this value is 1.0, then you can't modify the point size. A typical value is 64, which translates to 64 pixels—*independent of screen resolution*.

VERTEX SHADER INSTRUCTIONS

Vertex shaders version 1.0 and 1.1 can be up to 128 instructions long, while DirectX 9 shaders can be up to 256 instructions long. These instructions are a collection of general instructions, macroinstructions, and version and constant definition instructions. They are instructions that operate on input registers or constants and can selectively choose which of the register's four float elements to use. These elements in turn can be swizzled, negated, or masked.

The philosophy of shaders is to keep them simple and straightforward. Thus you'll see a very limited set of instructions, which, with a bit of cleverness, can actually be expanded to a wider variety of uses, as we'll see later. Generally (with a few notable exceptions), each shader instruction corresponds to one clock cycle of execution time. This means that the longer your shader program, the longer it'll take to execute with almost a direct linear correspondence. Macroinstructions are just that, and they can expand up to 12 general instructions, so remember that when you're calculating the shader size and

number of clock cycles. On the other hand, the register masks and swizzles add no execution time and should be used freely.

abs(macro)

vs 2.0

This macro computes the absolute value of the input register.

One slot

```
abs Dest0, Source0
```

This macro is equivalent to

```
max Dest0, Source0, -Source0
```

which you can use if you're using a prevertex shader 2.0 shader. In any case, you'll end up with the absolute value of the Source0 in Dest0.

Setup One source register, Source0.

Results Dest0 is filled with the absolute value of Source0.

```
abs r0 , r0
abs r0.z, r0.z
```

```
SetSourceRegisters();
```

```
// Simulate the add instruction
```

```
TempReg.x = abs( Source0.x );  
TempReg.y = abs( Source0.y );  
TempReg.z = abs( Source0.z );  
TempReg.w = abs( Source0.w );  
  
WriteDestinationRegisters();
```

add

vs 1.0, 1.1, 2.0

Adds two sources into the destination register.

One slot

```
add Dest0, Source0, Source1
```

Adds the Source0 and Source0 registers and places the result in Dest0 register.

Setup Two source registers, Source0 and Source1.

Results Each element of Dest0 is filled with the element-by-element addition of the elements of Source0 and Source1.

```
add r0 , r0 , c2  
add r0.z, r0.z, -r0.z
```

```
SetSourceRegisters();
```

```
// Simulate the add instruction
```

```
TempReg.x = Source0.x + Source1.x;
```

```
TempReg.y = Source0.y + Source1.y;
```

```
TempReg.z = Source0.z + Source1.z;

TempReg.w = Source0.w + Source1.w;

WriteDestinationRegisters();
```

call

vs 2.0

Makes an unconditional function call to the instruction label.

One slot

call_InstructionLabelID

Pushes the address of the following instruction onto the internal shader stack and then sets the current instruction address to the address of the instruction that follows the label instruction with the name *InstructionLabelID*. The instruction label ID will be an integer in the range [1, 16]. Calls cannot be nested.

Typically, you'd create a shader subroutine that terminates with the ret instruction.

Setup Requires a valid, existing instruction label.

Results The shader execution is transferred to the instruction following the instruction label.

```
call_1
call_16
call_Fred // Error! Invalid label
call_0    // Error! Invalid label (out of range)

// Simulate the call instruction
```

```
// make a cast to a bare function pointer
typedef (void (*fp)(void));

// take address of the label
fp pFP = (fp)InstructionLabelID;
pFP(); // call the function
// returns here only when ret is executed
```

callnz

vs 2.0

Call if Not Zero. Makes a function call to the instruction label.

One slot

```
callnz InstructionLabelID BoolSource0
```

If the boolean register Source0 is not zero, then the address of the following instruction is pushed onto the internal shader stack, and then the current instruction address is set to the address of the instruction that follows the label instruction with the name *InstructionLabelID*. The instruction label ID will be an integer in the range [1, 16]. Calls cannot be nested.

Typically, you'd create a shader subroutine that terminates with the ret instruction.

Setup Source0 is a boolean register. Requires a valid, existing instruction label.

Results If the source register is not zero, the shader execution is transferred to the instruction following the instruction label.

```
callnz 1 b0 // transfer execution to label1 if != b0
callnz 2 r0 // Error! Not a boolean register
```

```

// Simulate the callnz instruction

// make a cast to a bare function pointer
typedef (void (*fp)(void));

if ( 0 != Boolean argument )
{
    fp pFP = (fp)IntructionLabelID;
    pFP(); // call the function
}

```

crs(macro)

vs 2.0

The three-component cross product computed.

Two slots

```
crs Dest0, Source0, Source1
```

Computes the three-component cross product using the right-hand rule. There are fairly severe restrictions on the use of swizzles. The *w* elements of all registers are ignored.

This macro is equivalent to the following sequence of instructions:

```
mul Dest0.xyz, Source0.yzxw, Source1.zxyw
mad Dest0.xyz, -Source1.yzxw, Source0.zxyw, Dest0
```

Setup Two source registers, Source0 and Source1. These registers *must not* be the same as the destination register. The source registers *must not* have any swizzles

The destination register *must* have a destination mask, and that mask *must not* contain a reference to the *w* element of the destination register.

Results The cross product of the two input registers is stored in the specified elements of the destination register.

```
crs r0.xyz, r1.. r2 // fill r0 with dp3
```

```
SetSourceRegisters();
```

```
// Simulate the crs macro
```

```
TempReg.x = Source0.x * Source1.z - Source0.z * Source1.y;
```

```
TempReg.y = Source0.z * Source1.x - Source0.x * Source1.z;
```

```
TempReg.z = Source0.x * Source1.y - Source0.y * Source1.x;
```

```
// note w component ignored
```

```
WriteDestinationRegisters();
```

dcl

vs 2.0

Declare. Map a vertex element to an input register.

Takes no slots

```
dcl Dest0
```

In order to make it easier to optimize and verify shaders VS 2.0 now requires a declaration statement on all input registers. Thus all texture or vertex input registers must be declared before use in the shader.

Dest0 will be a specific input register. The partial precision modifier (`_pp`) can be applied to the declaration statement to indicate a lower precision is acceptable when using this register. You must supply a component mask on Dest0 to indicate which elements are in use and valid. `dcl` statements must appear before the first executable instruction.

```
dcl    t1.rg // using a 2D texture
dcl    t2     // using a 4D texture (default mask)
dcl_pp t3     // indicate partial precision is OK
```

def

vs 1.0, 1.1, 2.0

Sets the value of vertex shader floating point constants. In DirectX 8 it is up to the programmer to insert these into the shader code.

No slot

```
def    Dest0,    value0,    value1,    value2,    value3
```

Stores four floating point values in the elements of the Dest() register. If these instructions are used in a shader, the instructions must follow the `vs` instruction and precede any other instructions.

Setup Four floating point values separated by commas.

Results In DirectX 8 this instruction has no effect upon the shader code to follow. You must manually insert the returned code fragment into your shader. If you use the `def` in a shader, then when the shader is compiled, you will have to use the fourth parameter returned from `D3DXAssembleShader()`. This parameter will contain an `ID3DXBuffer` interface, which will contain a compiled shader code fragment. You will have to manually insert this fragment into your shader declaration.



In DirectX 9, this instruction causes the register to immediately assume the values specified. Previous values are restored when the shader exits.

```
def    r0, 0.0f, 0.5f, 0.25f, -1.0f
```

```
def    r1, 1.0f, 2.0f, 5.0f, 10.0f
```

defi

vs 2.0

Sets the value of vertex shader integer constants.

No slot

```
defi IntDest0, value0, value1, value2, value3
```

Stores four integer values in the elements of IntDest() register for use in this shader.

Setup Four integer values separated by commas.

Results Locally sets these values into the register. A local call takes precedence over an external `SetVertexShaderConstantI()` call to set a shader constant. The previous values of the register are restored upon exit from the shader.

```
defi i0, 0, 2, 4, 8
defi i1, -2, -1, 1, 2
```

defb

vs 2.0

Sets the value of vertex shader boolean constants.

No slot

```
defb BoolDest0, value0, value1, value2, value3
```

Stores four boolean values in the elements of BoolDest0 register for use in this shader. Zero indicates false. Nonzero indicates true.

Setup Four booleans separated by commas.

Results Locally sets these values into the register. A local call takes precedence over an external `SetVertexShaderConstantB()` call to set a shader constant. The previous values of the register are restored upon exit from the shader.

```
defb b0, 0, 1, 0, 2 // false, true, false, true
```

dp3

vs 1.0, 1.1, 2.0

Three-component dot product (dot product 3) is computed and the result replicated in all specified channels of the destination register.

One slot

```
dp3 Dest0, Source0, Source1
```

Computes the dot product of the Source0 and Source1 registers, and places the result in the Dest0 register. Only the x, y, and z values are used to compute the dot product; the w component is ignored.

Setup Two source registers, Source0 and Source1.

Results Unless otherwise masked, each element of Dest0 is filled with the dot product of the first three elements of registers Source0 and Source1.

```
dp3    r0 , v3, c2 // fill r0 with dp3
dp3    r1.x, v3, c2 // just fill r1.x
```

```
SetSourceRegisters();
```

```
// Simulate the dp3 instruction

TempReg.x = TempReg.y = TempReg.z = TempReg.w =

    Source0.x * Source1.x +

    Source0.y * Source1.y +

    Source0.z * Source1.z;

// note w component ignored

WriteDestinationRegisters();
```

dp4

vs 1.0, 1.1, 2.0

Four-component dot product (dot product 4) is computed and the result stored in all specified channels of the destination register.

One slot

```
dp4 Dest0, Source0, Source1
```

Computes the dot product of the Source0 and Source1 registers, and places the result in the Dest0 register. If no mask is specified on the destination, then the entire register is filled with the dot product.

Setup Two source registers, Source0 and Source1.

Results Unless otherwise masked, each element of Dest0 is filled with the dot product of the four elements of registers Source0 and Source1.

```
dp4    r0,    v3,  c2
dp4    r1.x,  v3,  c2 // just fill r1.x
```

```
SetSourceRegisters();
```

```
// Simulate the dp4 instruction

TempReg.x = TempReg.y = TempReg.z = TempReg.w =

    Source0.x * Source1.x +

    Source0.y * Source1.y +

    Source0.z * Source1.z +

    Source0.w * Source1.w;

WriteDestinationRegisters();
```

dst

vs 1.0, 1.1

Computes a distance vector in the format typically used for attenuated lighting calculations.

One slot

dst Dest0. Source0, Source1

Creates a distance vector from a set of distance squared and reciprocal distance values, and puts them in a format that can be used for attenuated lighting calculations.

Setup Two source registers are required to be set up. Source0 should be set up as $[n/a, d^2, d^2, n/a]$. Source1 should be set up as $[n/a, 1/d, n/a, 1/d]$. Elements noted as n/a are not used, and their values are ignored.

Results Dest0 will be filled with elements that correspond to $[1, d, d^2, 1/d]$. Dest0.y is computed from the product of Source0.y and Source1.y.

dst r2, r0, r1

```
SetSourceRegisters();
```

```
// Simulate the dst instruction

TempReg.x = 1;

TempReg.y = Source0.y * Source1.y;

TempReg.z = Source0.z;

TempReg.w = Source1.w;

WriteDestinationRegisters();
```

else

vs 2.0

Provided an alternate path of execution for an `if- else-endif` block.

One slot

`else`

Must be inside of an `if-endif` block. If the boolean argument of the `if` statement is false, then the execution will skip to the `else` instruction and continue to the terminating `endif` statement. If the boolean was true then execution will skip over the code enclosed by the `else-endif` block. There can be only one `else` statement in an `if-endif` block.

Setup The `else` statement must be between an `if` and `endif` statement.

Results If the argument provided to the `if` statement was false, then the code inside the `else-endif` block will be executed.

```
else
```

endif

vs 2.0

The termination point for an `if-endif` or `ifc-endif` block.

Zero slots

```
endif
```

When used with the `if` or `ifc` instruction, creates a block of instruction over which execution can be specified a number of times.

Setup You must have an `if` or `ifc` instruction in your shader prior to this instruction.

Results Execution is controlled by the `if` or `ifc` instruction that proceeds this instruction. When the argument of that statement is false then execution will jump to the statement following the `endif`.

```
if b1
    // if b1 != 0, this section gets executed
else // optional else statement
    // if b1 = 0, this section gets executed
endif
```

endloop

vs 2.0

The termination point for a `loop-endloop` block.

One slot

```
endloop
```

When used with the `loop` instruction, creates a block of instructions over which execution can be specified a variable number of times.

Setup You must have a `loop` instruction in your shader prior to this instruction.

Results When the loop reached the `endloop` instruction, the loop counter (specified in the `loop` instruction) is incremented by the increment value (also specified in the loop instruction). See the loop instruction to see the pseudocode of a loop-endloop block.

```
endloop
```

```
// simulate the endloop instruction
// assume that LoopCounter, LoopStep, LoopIterator
// were defined in the loop instruction and
// StartLoopOffset is the instruction following
// the loop instruction

LoopCounter += LoopStep;

--LoopIterator;

if ( LoopIterator > 0 )
    goto StartLoopOffset

// fall through
```

endrep

vs 2.0

The termination point for a `rep-endrep` block.

Zero slots

endrep

When used with the `rep` instruction, creates a block of instruction over which execution can be specified a number of times.

Setup You must have a `rep` instruction in your shader prior to this instruction.

Results Execution is controlled by the `rep` instruction that precedes this instruction. When the iteration count of that statement is zero then execution will jump to the statement following the `endrep`. See the `rep` instruction for simulation code.

```
defi i0, 20, 0, 0, 0
rep    i0 // i0.x is used = 20
    // this section gets executed 20 times
endrep
```

exp(macro)

vs 1.0, 1.1, 2.0

This macro computes power of two to at least 20 bits of precision. By default, only the source register's *w* element is used. The results are replicated in the entire destination register. Note that the `expp instruction` sets the destination's *w* element to 1.

At least 12 instruction slots

```
exp    Dest0,    Source0
```

Calculates for $2^{\text{Source0.w}}$, and writes the result in Dest0. Unless otherwise specified, Source0.w is the input value, and all elements of Dest0 are written with the exponentiated value. This is somewhat different from the `expp instruction`, which always sets Dest0.w to 1. A replicate swizzle is required on the source register.

```
exp    r0,    c1.w    // fill all of r0 with exp2 (c1.w)
exp    r0.x, c1.y    // store exp2 (c1.y) in r0.x
```

```

SetSourceRegisters();

// Simulate the exp macro
TempReg.x =
    TempReg.y =
    TempReg.z =
    TempReg.w = ::pow (2, Source0.w);

WriteDestinationRegisters();

```

expp

vs 1.0, 1.1, 2.0

Computes partial precision power of two. For DirectX 8 the results are broken into a partial precision part and a higher precision integer and fractional parts. This allows you to use the lower precision single element or a more complicated integer/fraction calculation when you need higher precision. The destination's *w* element is set to 1. Only the integer part of the source register's *w* element is used. If $\text{Source0.w} < 0$ then the results are undefined.



For DirectX 9, the partial precision result fills the destination register.

Note

Don't confuse this with the exp macro!

One slot

```
expp    Dest0, Source0
```

The DirectX 8 version computes low- and higher precision values for $2^{\text{Source0.w}}$, where *Dest0.z* contains the low-precision single-element approximation, *Dest0.x* and *Dest0.y* contain the integer and fractional parts, and *Dest0.w* is set to 1.

You have a choice in which part of the results to use. The low-precision part will contain the exponent of the input value to 10 bits of precision. The two-part higher precision part will contain the exponent of the integer part of the input value, and the fractional part of the input value, which you will have to provide a function to compute the value of 2^n for $0 \leq n \leq 1$ to your desired precision, and then add that to the integer's exponent value.

The DirectX 9 version just computes the low precision part.

Setup Stores the value you want the exponent of in Source0.w. The value should be positive. The other register elements are ignored. A replicate swizzle is required on the Source Register.

Results Dest0.z will contain a low-precision exponential value. Dest0.x will contain the exponential of the integer part of the input. Dest0.y will contain the fractional part of the input, *not* the exponential of the fractional part. You have to do the conversion yourself. Dest0.w is set to 1.0.

```
expp      r0, r1.w
```

```
// DirectX 8 version
```

```
SetSourceRegisters();
```

```
// Simulate the expp instruction
```

```
float wWhole = Source0.w; // take all
```

```
float wInt = (int)Source0.w; // take integer part
```

```
// compute the higher precision parts
```

```
TempReg.x = pow(2,wInt);
```

```
TempReg.y = Source0.w - wInt; // fractional part of w
```

```
// calculate the 2^(Source0.w) then chop
```

```
// to 10 bits precision
```

```
TempReg.z = pow(2,wWhole) & 0xffffffff00;
```

```
// set w to 1
```

```
TempReg.w = 1;
```

```

WriteDestinationRegisters();

// DirectX 9 version
SetSourceRegisters();
// Simulate the expp instruction
float wWhole = Source0.w; // take all
TempReg.x = TempReg.y =
    TempReg.z = TempReg.w =
    pow(2,wWhole) & 0xffffffff00;

WriteDestinationRegisters();

```

frs(macro)

vs 1.0, 1.1, 2.0

This macro removes the integer part of the input register's elements, and places the fractional remainder into the destination register's elements. The sign of the results is always positive.

Three instruction slots

```

frc Dest0, Source0

```

Takes the fractional parts of Source0's elements and places them in Dest0's elements. The sign of the input arguments is ignored. Version 1.x requires an. xy write mask.

```

frc    r0.xy, r1 // use r1.xy and store fractions in r0.xy

// use r1.x and store fraction in r0.y, r0.x (and z & w)
// remain unchanged

```

```

frc    r0.y , r1.x

// this has no effect on the results, since the
// sign is ignored
frc    r0.y , -r1.x

frc    r0, r1 // Error! No write mask.

```

```

SetSourceRegisters();

// Simulate the frc macro
TempReg.x = abs( Source0.x );
TempReg.y = abs( Source0.y );
TempReg.z = abs( Source0.z );
TempReg.w = abs( Source0.w );

TempReg.x = TempReg.x - (int)TempReg.x;
TempReg.y = TempReg.y - (int)TempReg.y;
TempReg.z = TempReg.z - (int)TempReg.z;
TempReg.w = TempReg.w - (int)TempReg.w;

WriteDestinationRegisters ();

```

if

vs 2.0

The start of an if-else-endif block. Conditionally execute a block of code.

One slot

```
if BoolReg0
```

The argument *must* be a boolean constant register. There *must* be a terminating `endif` that follows the `if` instruction. The `else` instruction is optional and must be between the `if` and `endif` statements. If the boolean argument is true, then execution will continue immediately after the `if` statement, until either the `else` or `endif` statement are reached.

`if` blocks can be placed inside an `if-endif` or a `loop-endloop` block, but they must be entirely inside them.

Setup The argument must be a boolean register. You must have an `endif` instruction in your shader following to this instruction.

Results Execution is controlled by the `if` instruction. When the boolean of that statement is false then execution will jump to the statement following the `if`, which must be either an `else` or an `endif` statement.

```
if b1
    // if b1 != 0, this section gets executed
else // optional else statement
    // if b1 = 0, this section gets executed
endif
```

label

vs 2.0

Defines a label for use with a `call` or `callnz` instruction.

Zero slots

```
label <n>
```

The `label` instruction marks the next instruction as having the specified label, thus making it a target for a subroutine call. The argument `<n>` *must* be integer label in the range [0,16]—that is, there can be a total of sixteen labels.

Setup The argument must be an integer.

Results When a `call` or `callnz` instruction calls the integer label, execution immediately (and conditionally for the `callnz` instruction) goes to the instruction following the `label` statement. Execution will return when a `ret` instruction is encountered

```
// VS 2.0
call 12 // somewhere in shader call subroutine

label 12 // label 12

// the subroutine instructions go here

ret      // execution returns after the call
```

lit

vs 1.0, 1.1, 2.0

Computes the traditional diffuse and specular lighting coefficients when passed on the resulting dot products from $\mathbf{n} \cdot \mathbf{1}$ and $\mathbf{n} \cdot \mathbf{h}$ and a power coefficient.

One slot

```
lit    Dest0, Source0
```

You'll need to calculate normalized $\mathbf{n} \cdot \mathbf{1}$ and $\mathbf{n} \cdot \mathbf{h}$ dot products, and specify a specular power value prior to using this instruction. The results will be the traditional diffuse component in `Dest0.y`,

$$\text{Dest0.y} = i_d = (n \bullet l)$$

and the traditional specular component (i.e., Blinn's equation) in Dest0.z,

$$\text{Dest0.z} = i_s = (n \bullet h)^{m_s}$$

Note that there are no k parameters in the equations. If you need them, you'll have to do the multiplication in your shader.

Setup Source0.x should contain the normalized dot product between the normal and the direction from the vertex to the light. Source0.y should contain the normalized dot product between the normal and the half-angle vector. Source0.w should contain the power value in the range -128 to $+128$. Source0.z is ignored.

Results Dest0.x and Dest0.w are set to 1. If Source0.x ($n \bullet l$) is positive, it's stored in Dest0.y; else Dest0.y is set to 0. If both Source0.x and Source0.y ($n \bullet h$) are positive, then Dest0.z is set to Source0.y raised to the Source0.w power; else it is set to zero. The power value (Source0.w) is clamped to the range $[-128, 128]$.

Note

Early versions of the SDK documentation incorrectly stated that negative exponential values would cause undefined results.

```
lit      r0      r1
```

```
SetSourceRegisters();
// Simulate the lit instruction

// these are constants
TempReg.x = TempReg.w = 1;

// if n dot l is positive...
if ( Source0.x > 0 )
{
    Dest0.y = Source0.x;
```

```

// if n dot h is positive
if ( Source0.y > 0 )
{

    // clamp the power value to an 8.8
    // fixed point representation of the
    // maximum allowable value

    const float kPowerMax = 127.9961f;
    float ClampedPower = Source0.w;
    if      (ClampedPower < -kPowerMax )
        ClampedPower = -kPowerMax;
    else if (ClampedPower > kPowerMax )
        ClampedPower = kPowerMax;

    // actual value in shader math is only
    // good to seven fractional bits of
    // precision
    Dest0.z = pow( Source0.y, ClampedPower );
}
else
{
    Dest0.z = 0; // if n dot h was negative/zero
}
}
else
{
    Dest0.y = 0; // if n dot 1 was negative/zero
}

WriteDestinationRegisters();

```

log(macro)

vs 1.0, 1.1, 2.0

This macro computes \log_2 of the input argument in at least 20-bit precision. The absolute value the source register's *w* element is used. Unlike the `logp instruction`, the destination's *w* element is not set to 1.

Note

Don't confuse this with the `logp instruction`.

At least 12 instruction slots

```
log    Dest0, Source0
```

Computes \log_2 of the absolute value of the Source0.*w* element (unless otherwise specified) and places the result into all elements of Dest0. If the argument is equal to zero, all result registers are set to minus infinity. This is somewhat different from the `logp instruction`, which always sets Dest0.*w* to 1.

```
log    r0,    r1
```

```
SetSourceRegisters();
```

```
// Simulate the log macro
```

```
if ( 0 == Source0.w )
```

```
{
```

```
    TempReg.x = TempReg.y = TempReg.y = TempReg.w =
```

```
        MINUS_INFINITY;
```

```
}
```

```
else
```

```
{
```

```
    // note we use absolute value
```

```
    TempReg.x = TempReg.y = TempReg.y = TempReg.w =
```



```

    log(abs(Source0.w))/log(2);
}

WriteDestinationRegisters();

```

logp

vs 1.0, 1.1, 2.0

Computes partial precision \log_2 . In DirectX 8 the results are broken into a single 10-bit precision part and a higher precision dual-element part. This allows you to use the lower precision single element or a more complicated integer/ fractional calculation if you need higher precision.



For DirectX9 the partial result fills the entire destination register.

One slot

```
logp Dest0, Source0
```

The DirectX 8 version computes low- and higher precision values for $\log_2^{\text{Source0.w}}$. The destination's *w* element is set to 1. Only the source register's *w* element is used. If Source0.w is negative, then its absolute value is used. If Source0.w is zero, then the results are negative infinity in Dest0.x and Dest0.z, and 1.0 in Dest0.y.

You have a choice in which part of the results to use. The low-precision part will contain the \log_2 of the input value to 10 bits of precision.

The two-part higher precision part represents the exponent and mantissa. This allows you to use the lower precision single element or a more complicated exponent/mantissa calculation when you need higher precision.

To use the high-precision results, you'll need to provide a function that computes \log_2 in the range [1,2] with your desired precision. You'd then add this result to the value returned in Dest0.x to get the \log_2 of your input value.

The DirectX 9 version just computes the low-precision part.

Note

Don't confuse this with the log macro!

Setup For DirectX 8, store the value you want the \log_2 of in Source0.w. The value should be positive. The other register elements are ignored. For DirectX 9, indicate the element by using a (required) replicate swizzle.

Results DirectX 8: Dest0.z contains the low precision (10-bit) single-element approximation. Dest0.x contains the most significant part of the dual-element result. This value can be negative. Dest0.y contains the mantissa of the dual-element result in the form of an exponentiated value in the range [1,2). You have to do the conversion yourself.

DirectX 9: All destination elements will have the low-precision result.

```
logp    r0,    r1
```

```
// DirectX 8 version:
```

```
SetSourceRegisters();
```

```
// Simulate the logp instruction
```

```
float v = abs(Source0.w); // only positive values
```

```
TempReg.y = TempReg.w = 1.0f;
```

```
if ( 0 == v )
```

```
{
```

```
    TempReg.x = TempReg.z = MINUS_INFINITY;
```

```
}
```

```
else
```

```
{
```

```
    float logValue = (float)(log(v)/log(2));
```

```
    // store exponent
```

```
    Dest0.x = (int)::floor( logValue );
```

```
    // store mantissa, lop off anything more than
```

```
    // 8 bits of significance
```

```
    int p = (*(unsigned long*)&v
```

```

        & 0x7FFFFFFF | 0x3F800000;

Dest0.y = *(float*)&p;

// store low-precision part to 10 bits
unsigned long temp = *(unsigned long*)&logValue;
Dest0.z = *(float*)& temp & 0xFFFFFFFF00;
}

WriteDestinationRegisters();

// DirectX 9 version
SetSourceRegisters();

// Simulate the logp instruction
float v = abs(Source0.w); // only positive values
float logValue;

if ( 0 == v )
{
    logValue = MINUS_INFINITY;
}
else
{
    logValue = (float) (log(v)/log(2));
    logValue = (int)::floor( logValue );
    // store low-precision part to 10-bits
    unsigned long temp = *(unsigned long*)&logValue;
    logValue = *(float*)& temp & 0xFFFFFFFF00;
}

TempReg.x = TempReg.y =

```

```
TempReg.z = TempReg.w =  
LogValue;  
WriteDestinationRegisters();
```

loop

vs 2.0

The starting point for a loop-endloop block. Iterate over a block of code a number of times.

One slot

```
loop IntSource0
```

When used with the endloop instruction, creates a block of instruction over which execution can be specified a variable number of times. Each time through the loop the loop counter is incremented by the specified amount. Compare this to the rep instruction, which does not increment the loop counter independently.

Setup The argument must be integer register. IntSource0.x holds the number of times the loop is to execute. The loop counter register gets incremented at the endloop.

The counter can be used to index into the constant register array. IntSource0.y is the initial value for the loop counter register. IntSource0.z specifies the step for the loop counter register. Execution will go to the statement following the matching end loop instruction when IntSource0.x <= 0. The loop counter register, aL, is available inside the loop.

You *must not* nest loops, nor jump either out of or into a loop-endloop block. The endloop instruction must precede the loop instruction.

Results The instructions in the loop-endloop block are executed IntSource0.x times with the loop counter getting incremented each time through.

```
loop i0 // assume i0.x, i0.y, and i0.z are set up  
  
// aL is available and incremented each time
```

```
// through the loop
```

```
endloop
```

```
// simulate the loop instruction
```

```
// -- loop statement begin
```

```
aL = IntReg0.y; // loop counter
```

```
StartLoopOffset: // <- label
```

```
if ( IntSource0.x <= 0 )
```

```
goto EndLoopOffset;
```

```
// -- loop statement end
```

```
// section of code between the loop/endloop
```

```
// -- endloop statement begin
```

```
aL += IntSource0.z; // increment loop counter
```

```
IntSource0.x--; // decrement
```

```
goto StartLoopOffset;
```

```
EndLoopOffset: // <- label
```

```
// -- endloop statement end
```

lrp(macro)

vs 2.0

Linear interpolation between two registers (a.k.a. "lerp") using a fraction specified in a third register. This is done on an element-by-element basis.

Two slots

```
lrp Dest0, Source0, Source1, Source2
```

This macro instruction interpolates between two floating point numbers, Source0 and Source1, based upon a third, Source2. When Source2 is zero, Source0 is placed in the destination. When Source2 is one, Source1 is placed in the destination. Values in the [0,1] range interpolate between Source1 and Source2. If the value is outside the range [0,1] the result is indeterminate. Dest0 must be a temporary register and cannot be the same as Source0 or Source2.

The macro expands to the following code:

```
add Dest0, Source1, -Source2
mad Dest0, Dest0, Source0, Source2
```

Setup Dest0 must be a temporary register, and Source2 should be in the range [0,1].

Results The value between Source0 and Source1 is interpolated from the value in Source2. The result is written in Dest0.

```
lrp r0, r1, r2, c14
lrp r0, r1, r2, r14.x // lrp using a single value
```

```
SetSourceRegisters();
```

```
// simulate lrp instruction
```

```
TempReg.x = Source3.x * (Source1.x - Source2.x) + Source2.x;
```

```
TempReg.y = Source3.y * (Source1.y - Source2.y) + Source2.y;
```

```
TempReg.z = Source3.z * (Source1.z - Source2.z) + Source2.z;
```

```
TempReg.w = Source3.w * (Source1.w - Source2.w) + Source2.w;
```

```
WriteDestinationRegisters();
```

m3x2(macro)

vs 1.0, 1.1, 2.0

Matrix 3 by 2. Performs a matrix multiply on the input vector and input matrix and stores the result. This macro is typically used for 2D transformation calculations.

Two instruction slots

```
m3x2 Dest0, Source0, Source1
```

Does a matrix multiply assuming that Source0 is the input vector and the matrix starts at element Source1 and there are the correct number of registers available after Source1. Only those elements actually used in the calculations are read; only those that are calculated are written to the destination registers. The *w* elements in the source matrix and vector are unused, and only the *x* and *y* elements of the destination are written.

This macro

```
m3x2 Dest0, Source0, Source1
```

expands to the following:

```
dp3 Dest0.x, Source0, Source1
```

```
dp3 Dest0.y, Source0, Source2
```

Note

You can use the *swizzle* or *negate* modifier if you expand this macro yourself.

Warning

Make sure that your Dest0 and Source0 registers aren't the same registers. It will compile and run, but your results will be incorrect.

Note

You are not allowed to use the *swizzle* or *negate* modifier on Source1.

```
m3x2 r0, v0. c6 ; // will use c7 as well
m3x2 r0, v0, c6.yzxw // Error! Can't use swizzle
```

```
SetSourceRegisters();

// Simulate the m3x2 macro
TempReg.x = Source0.x * Source1.x +
            Source0.y * Source1.y +
            Source0.z * Source1.z;
TempReg.y = Source0.x * Source2.x +
            Source0.y * Source2.y +
            Source0.z * Source2.z;

WriteDestinationRegisters();
```

m3x3(macro)**vs 1.0, 1.1, 2.0**

Matrix 3 by 3. Performs a matrix multiply on the input vector and input matrix and stores the result. This macro typically is used for normal transformations during lighting calculations.

Three instruction slots

```
m3x3 Dest0, Source0, Source1
```

Does a matrix multiply assuming that Source0 is the input vector and the matrix starts at element Source1 and there are the correct number of registers available after Source1. Only those elements actually used in the calculations are read; only those that are calculated are written to the destination registers. The *w* elements in the source matrix and vector are unused, and only the *x*, *y*, and *z* elements of the destination register are written.

This macro

```
m3x3 Dest0, Source0, Source1
```

expands to the following:

```
dp3      Dest0.x, Source0, Source1
dp3      Dest0.y, Source0, Source2
dp3      Dest0.z, Source0, Source3
```

Note

You can use the *swizzle* or *negate* modifier if you expand this macro yourself.

Warning

Make sure that your Dest0 and Source0 registers are different. It will compile, but your results will be incorrect.

Note

You are not allowed to use the *swizzle* or *negate* modifier on Source1.

```
m3x3 r0, v0, c6 ; // will use c7 & c8 as well
m3x3 r0, v0, c6.yzxw // Error! Can't use swizzle
```

```

SetSourceRegisters();

// Simulate the m3x3 macro

TempReg.x = Source0.x * Source1.x +
           Source0.y * Source1.y +
           Source0.z * Source1.z;

TempReg.y = Source0.x * Source2.x +
           Source0.y * Source2.y +
           Source0.z * Source2.z;

TempReg.z = Source0.x * Source3.x +
           Source0.y * Source3.y +
           Source0.z * Source3.z;

WriteDestinationRegisters();

```

m3x3(macro)

vs 1.0, 1.1, 2.0

Matrix 3 by 4. Performs a matrix multiply on the input vector and input matrix and stores the result.

Four instruction slots

```
m3x4 Dest0. Source0. Source1
```

Does a matrix multiply assuming that Source0 is the input vector and the matrix starts at element Source 1 and there are the correct number of registers available after Source1. Only those elements actually used in the calculations are read; only those that are calculated are written to the destination registers. The *w* elements in the source matrix and rector are unused.

This macro

```
m3x4    Dest0, Source0, Source1
```

expands to the following:

```
dp3      Dest0.x, Source0, Source1
dp3      Dest0.y, Source0, Source2
dp3      Dest0.y, Source0, Source3
dp3      Dest0.y, Source0, Source4
```

Note

You can use the *swizzle* or *negate* modifier if you expand this macro yourself.

Warning

Make sure that your Dest0 and Source0 registers are different. It will compile, but your results will be incorrect.

Note

You are not allowed to use the *swizzle* or *negate* modifier on Source1.

```
m3x4    r0, v0, c6 ; // will use c7, c8, & c9 as well
m3x4    r0, v0, c6.yzxw // Error! Can't use swizzle
```

```
SetSourceRegisters();
```

```
// Simulate the m3x4 macro
```

```
TempReg.x = Source0.x * Source1.x +
```

```

        Source0.y * Source1.y +
        Source0.z * Source1.z;
TempReg.y = Source0.x * Source2.x +
        Source0.y * Source2.y +
        Source0.z * Source2.z;
TempReg.z = Source0.x * Source3.x +
        Source0.y * Source3.y +
        Source0.z * Source3.z;
TempReg.w = Source0.x * Source4.x +
        Source0.y * Source4.y +
        Source0.z * Source4.z;

WriteDestinationRegisters();

```

m4x3(macro)

vs 1.0, 1.1, 2.0

Matrix 4 by 3. Performs a matrix multiply on the input vector and input matrix and stores the result.

Three instruction slots

m4x3 Dest0, Source0, Source1

Does a matrix multiply assuming that Source0 is the input vector and the matrix starts at element Source1 and there are the correct number of registers available after Source1. All source register elements are used, but Dest0.w will be unmodified.

This macro

m4x3 Dest0, Source0, Source1

expands to the following:

```
dp4      Dest0.x, Source0, Source1
dp4      Dest0.y, Source0, Source2
dp4      Dest0.z, Source0, Source3
```

Note

You can use the *swizzle* or *negate* modifier if you expand this macro yourself.

Warning

Make sure that your Dest0 and Source0 registers are different. It will compile, but your results will be incorrect.

Note

You are not allowed to use the *swizzle* or *negate* modifier on Source1.

```
m4x3 r0, v0, c6 ; // will use c7 & c8 as well
m4x3 r0, v0, c6.yzxw // Error! Can't use swizzle
```

```
SetSourceRegisters();
```

```
// Simulate the m4x3 macro
```

```
TempReg.x = Source0.x * Source1.x +
           Source0.y * Source1.y +
           Source0.z * Source1.z +
           Source0.w * Source1.w;
```

```
TempReg.y = Source0.x * Source2.x +
           Source0.y * Source2.y +
           Source0.z * Source2.z +
```

```

        Source0. w * Source2.w;

TempReg.z = Source0. x * Source3.x +

        Source0. y * Source3.y +

        Source0. z * Source3.z +

        Source0. w * Source3.w;

WriteDestinationRegisters();

```

m4x4(macro)

vs 1.0, 1.1, 2.0

Matrix 4 by 4. Performs a matrix multiply on the input vector and input matrix and stores the result.

Four instruction slots

```
m4x4 Dest0, Source0, Source1
```

Does a matrix multiply assuming that Source0 is the input vector and the matrix starts at element Source1 and there are the correct number of registers available after Source1. All source register elements are used, and all destination registers will be written.

This macro

```
m4x4 Dest0, Source0, Source1
```

expands to the following:

dp4	Dest0.x, Source0, Source1
dp4	Dest0.y, Source0, Source2
dp4	Dest0.z, Source0, Source3
dp4	Dest0.w, Source0, Source4

Note

You can use the *swizzle* or *negate* modifier if you expand this macro yourself.

Warning

Make sure that your Dest0 and Source0 registers are different. It will compile, but your results will be incorrect.

Note

You are not allowed to use the *swizzle* or *negate* modifier on Source1.

```
m4x4  r0, v0, c6 ; // will use c7, c8, & c9 as well
m4x4  r0, v0, c6.yzxw // Error! Can't use swizzle
```

```
SetSourceRegisters();
```

```
// Simulate the m4x4 macro
```

```
TempReg.x = Source0.x * Source1.x +
            Source0.y * Source1.y +
            Source0.z * Source1.z +
            Source0.w * Source1.w;
```

```
TempReg.y = Source0.x * Source2.x +
            Source0.y * Source2.y +
            Source0.z * Source2.z +
            Source0.w * Source2.w;
```

```
TempReg.z = Source0.x * Source3.x +
            Source0.y * Source3.y +
            Source0.z * Source3.z +
            Source0.w * Source3.w;
```

```
TempReg.w = Source0.x * Source4.x +  
           Source0.y * Source4.y +  
           Source0.z * Source4.z +  
           Source0.w * Source4.w;
```

```
WriteDestinationRegisters();
```

mad

vs 1.0, 1.1, 2.0

Multiply and add. Multiplies two registers, then adds a third to the result, and then stores the result.

One slot

```
mad  Dest0, Source0, Source1, Source2
```

Multiplies Source0 by Source1, then adds Source2 to the result. The final result is stored in Dest0.

Setup Source0 and Source1 are the registers to be multiplied. Source2 is the register to be added to the result of the multiplication of Source0 and Source1.

Results Dest0 contains (Source0* Source1) + Source2.

```
mad    r0, r0, r1, r2
```

```
Set SourceRegisters();
```

```
//Simulate the mad instruction
```

```
TempReg.x = Source0.x * Source0.x + Source1.x;
```

```
TempReg.y = Source0.y * Source0.y + Source1.y;
```



```
TempReg.z = Source0.z * Source0.z + Source1.z;

TempReg.w = Source0.w * Source0.w + Source1.w;

WriteDestinationRegisters();
```

max

vs 1.0, 1.1, 2.0

Stores the maximum value from comparing two source registers element by element into the destination register.

One slot

max Dest0, Source0, Source1

Finds the maximum between elements of Source0 and Source1, then stores the results in elements Dest0. The resulting register may not be equal to either input register since the comparison is per element.

Setup Source0 and Source1 are the registers to be compared.

Result Dest0 contains the maximum elements of the two input registers done on an element-by-element basis.

```
max    r0, r1, r2
```

```
SetSourceRegisters();

// Simulate the max instruction
TempReg.x = Source0.x > Source1.x ?
    Source0.x : Source1.x;
TempReg.y = Source0.y > Source1.y ?
```

```

    Source0.y : Source0.y;

    TempReg.z = Source0.z > Source0.z ?

    Source0.z : Source.z;

    TempReg.w = Source0.w > Source0.w ?

    Source0.w : Source0.w;

    WriteDestinationRegisters();

```

min

vs 1.0, 1.1, 2.0

Stores the minimum value from comparing two source registers element by element into the destination register.

One slot

```
min    Dst0, Source0. Source1
```

Finds the minimum between elements of Source0 Source1, then stores the results in elements Dest0. The resulting register may not be equal to either input register since the comparison is per element.

Setup Source0 and Source1 are the registers to be compared.

Results Dest0 contains the minimum of the two input registers done on an element-by-element basis.

```
min    r0, r1, r2
```

```

    SetSourceRegisters();

    // Simulate the min instruction

    TempReg.x = Source0.x < Source0.x ?

    Source0.x : Source0.x;

```

```

TempReg.y = Source0.y < Source0.y ?
    Source0.y : Source0.y;
TempReg.z = Source0.z < Source0.z ?
    Source0.z : Source0.z;
TempReg.w = Source0.w < Source0.w ?
    Source0.w : Source0.w;
WriteDestinationRegisters();

```

mov

vs 1.0, 1.1, 2.0

Stores the source registers into the destination register. Useful for moving from a temporary register into an output register or for swizzling. The source and destination registers can be the same.

For DirectX 8, the mov instruction is the only instruction that can use the address register as a destination and only in vertex shaders version 1.1 or later. If the address register is the destination, then the value is rounded to the integer value that is less than or equal to the initial value. In VS 2.0 you must use the mova instruction to set the address register.

One slot

```
mov    Dest0, Source0
```

Move Source0 into Dest0. A special case is in DirectX8 when the Dest0 is an address register. In this case, the value stored is the closest integer value that is less than the initial value. This means that it rounds the number toward negative infinity. Thus 1.5 would get stored as 1, while -1.5 would get stored as -2. In both cases, the value stored is the integer value that's closest and *less* than the initial value. In DirectX9, only floating point data can be moved.

Setup Source0 is the register to be copied.

Results Dest0 contains a copy of Source0, unless it's the address register in a DirectX8 shader, in which case it is the nearest integer value that's less than or equal to the initial value in the register. If the destination is the address register, then, unless otherwise specified, only the Source0.x register is used.

```
mov r0 , r1
mov a0.x , clw. // initializing address register, DirectX 8
```

```
SetSourceRegisters();

// Simulate the mov instruction
// it's the address register in a DirectX 8 shader
if ( Source0 == a0 )
{
    // use only integer part
    TempReg.x = (int) :: floor ( Source0.x );
}
else // DirectX 9 shader or Source0 is a float
{
    TempReg.x = Source0.x;
    TempReg.y = Source0.y;
    TempReg.z = Source0.z;
    TempReg.w = Source0.W;
}

WriteDestinationRegisters();
```

mov

vs 2.0

Mova data from a floating point register into the address register.

One slot

```
movb Dest0, Source0
```

This instruction rounds Source0 to the nearest integer and places the result in Dest0. Dest0 must be the address register. Rounding is to nearest even, though this is not exactly specified and applications should not rely on this behavior. That is, for values equidistant between two integers, some implementations may round up, down, or randomly pick a direction. The `_sat` modifier is not supported.

Setup Source0 is the floating point register to be rounded, then placed in the address register.

Results The rounded value from Source0 is placed in the address register.

```
movb a0.x, cl.w // move one element
movb a0, cl // move all
```

```
SetSourceRegisters();

// use only integer part
// Note: RoundToNearestInteger () is
// Implementation dependent
a0.x = RoundToNearestInteger( Source0.x );
a0.y = RoundToNearestInteger( Source0.y );
a0.z = RoundToNearestInteger( Source0.z );
a0.w = RoundToNearestInteger( Source0.w );

WriteDestinationRegisters();
```

mul

vs 1.0, 1.1, 2.0

Multiplies the two source registers element by element and stores them in the destination register.

One slot

```
mul Dest0, Source0, Source1
```

Multiplies Source0 by Source1 and stores the result in Dest0.

Setup Source0 and Source1 are the two registers to be multiplied.

Results Dest0 contains the result of the multiplication of Source0 and Source 1.

```
mul    r0, r1, r2
```

```
SetSourceRegisters ();
```

```
// Simulate the mul instruction
```

```
TempReg.x = Source0.x * Source0.x;
```

```
TempReg.y = Source0.y * Source0.y;
```

```
TempReg.z = Source0.z * Source0.z;
```

```
TempReg.w = Source0.w * Source0.w;
```

```
WriteDestinationRegisters ();
```

nop

vs 1.0–2.0

Defines the null instruction (No-Operation).

One slot, possibly optimized out.

```
nop
```

You can use it to create a shader that does nothing but take up slots and/or time as it executes to see how a shader of that length would affect your rendering. It's possible that a driver might optimize away this instruction.

Setup None.

Results Nothing.

```
nop
```

nrm(macro)

vs 2.0

This macro will normalize all elements of a register.

Three slots

```
nrm Dest0, Source0
```

This macro will take all elements of Source0 and normalize them so that the square root of the sum of squares of all elements in Dest0 is one. Dest0 cannot be the same register as Source0.

This macro

```
nrm Dest0, Source0
```

is equivalent to the following:

```
dp4    Dest0.x, Source0, Source0
rsq    Dest0.x, Dest.x
mul    Dest0, Source0, Dest0.x
```

```
nrm    r0, v0
```

pow(macro)

vs 2.0

Computes the power function for a scalar value.

Three slots

```
pow    Dest0, Source0, Source1
```

Only the *W* element of the source registers are used. Only the absolute value of the Source0 is used. Dest0 is filled with $\text{abs}(\text{Source0.x})$ raised to the Source1.x power. The result is replicated in all elements of the destination.

This macro

```
pow    Dest0, Source0, Source1
```

is equivalent to the following:

```
log  Dest0.w, Source0 // takes absolute value
mul  Dest0.w, Dest0.w, Source1.w
exp  Dest0, Dest0.w
```

```
pow r0, r3, c6 // assume r3.x and c6.x are set
```

rcp

vs 1.0, 1.1, 2.0

Computes the reciprocal of an element of the source register and stores it in the destination register.

One slot

```
rcp  Dest0, Source0
```

Computes the reciprocal of a single element of the source register and stores it in all elements of the destination register. Only one element of the source is used. If no element is specified, then Source0.w is used. A value of exactly 1 on input returns 1 on output (no round-off error), whereas a value of 0 on input returns positive infinity.

Setup Source0 contains the element of which to take the reciprocal. If unspecified, Source0.w is used.

Results Dest0 contains the reciprocal of the specified element copied in all elements.

Note

This is one of the few instructions that will take more than one clock to execute. Use it sparingly, and when you do use it, try to arrange your code so that you don't need the results immediately.

```
rcp  r0, r1
```

```

SetSourceRegisters ();

// Simulate the rcp instruction
if ( 0.0f == Source0.w ) // if 0
{
    TempReg.w = PLUS_INFINITY;
}
else if ( 1.0f == Source0.w ) // if 1
{
    TempReg.w ==1.0f;
}
else
{
    TempReg.w = 1.0f/Source0.w;
}

TempReg.x = TempReg.y = TempReg.z = TempReg.w;

WriteDestinationRegisters ();

```

rep

vs 2.0

Repeat. Indicates the start of a rep-endrep block.

One slot

rep IntSource0

IntSource0 must be an integer register. Only the .x element is used. The maximum initial value can be 255. Execution over the block will continue for IntSource0.x times, as long as the number is positive. Compare this to the loop instruction, which additionally increments over the loop counter independently.

Setup IntSource0 must be an integer register with the .x element initialized to the number of times to iterate through the block.

Results The instructions in the rep - endrep blocks are executed IntSource0.x times.

```
defi i0, 10, 0, 0, 0 // i0.x is set to the count

rep i0

    //the instructions here will get executed i0.x times

endrep
```

```
// Simulate the rep instruction
int LoopCounter = IntReg0.x;
if (LoopCounter <= 0) goto EndLoop
// the instructions following the loop
// instruction would go here

// Simulate endloop instruction
aL += IntReg0.z;
LoopCounter--;
goto TopLoop;
EndLoop:
```

ret

vs 2.0

Indicates the end of a subroutine.

One slot

`ret`

This instruction will return to the calling instruction (a `call` or `callnz` instruction) or return from the main function.

Setup Returns to the address following the most recent `call` or `callnz` instruction, or returns from the main function.

Results The path of execution is changed to the next instruction on the instruction stack.

```
ret
```

rsq

vs 1.0, 1.1, 2.0

Computes the reciprocal square root of specified element of the source register and stores it in all elements of the destination register.

One slot

`rsq Dest0, Source0`

Computes the reciprocal square root of the specified element of the source register and stores it in all elements of the destination register. If no element is specified, then `Source0.w` is used. The absolute value of the input is used. A value of exactly 1 on input returns 1 on output (no round-off), whereas a value of 0 on input returns positive infinity.

Setup `Source0` contains the element of which to take the reciprocal square root. If unspecified, `Source0.w` is used.

Results Dest0 contains the reciprocal square root of the absolute value of the specified element copied in all elements.

Note

This is one of the few instructions that will take more than one clock to execute. Use it sparingly, and when you do use it, try to arrange your code so that you don't need the results immediately.

```
rsq    r0, r1
```

```
SetSourceRegisters ();
```

```
// Simulate the rsq instruction
```

```
float v = abs (Source0.w);
```

```
if ( 0.0f == v) // if 0
```

```
{
```

```
    TempReg.w = PLUS_INFINITY;
```

```
}
```

```
else if ( 1.0f == v) // if 1
```

```
{
```

```
    TempReg.w = 1.0f;
```

```
}
```

```
else
```

```
{
```

```
    TempReg.w = 1.0f/sqrt (v);
```

```
}
```

```
TempReg.x = TempReg.y = TempReg.z = TempReg.w;
```

```
WriteDestinationRegisters ();
```

sge

vs 1.0, 1.1, 2.0

Sets Greater-Than or Equal-To. Stores 1 in the destination register if the first source register is greater than or equal to the second source register. If not it stores 0 in the destination register. Does an element-by-element comparison and assignment.

One slot

```
sge Dest0, Source0, Source1
```

Compares the two source registers element by element. If the first source register's element is greater than or equal to the second source register's element, the value 1 is placed in the destination register's element. If not, it stores 0 in the destination register's element. The resulting register can be a mix of 0s and 1s.

Setup Source0 and Source1 are the registers to be compared.

Results The element Dest0.*n* contains 1.0 if the Source0.*n* is greater than or equal to Source1.*n*; otherwise, it contains 0.0. This is done for all elements of Dest0.

```
sge      r0, r1, r2
```

```
SetSourceRegisters ();
```

```
// Simulate the sge instruction
```

```
TempReg.x = Source0.x >= Source0.x ?
```

```
    1.0f : 0.0f;
```

```
TempReg.y = Source0.y >= Source 0.y ?
```

```
    1.0f : 0.0f;
```

```
TempReg.z = Source0.z >= Source0.z ?
```

```
    1.0f : 0.0f;
```

```
TempReg.W = Source0.w >= Source0.w ?  
    1.0f : 0.0f;  
  
WriteDestinationRegisters();
```

sgn (macro)

vs 2.0

Computes the sign of each element in a register.

Three slots

```
sgn Dest0, Source0, Source1, Source2
```

Computes the sign of the elements of Source0, using two temporary scratch registers. All elements of the source registers are compared. The comparison is done element by element. Source1 and Source2 should be temporary registers and should not be the same. If an element in Source0 was > 0 , then the corresponding element in Dest0 will be 1. If it was < 0 , then the result will be -1 . If it was 0 the result will be 0.

This macro

```
sgn Dest0, Source0, Source1, Source2
```

is equivalent to the following:

```
slt Source1, Source0, -Source0
```

```
slt Source2, -Source0, Source0
add Dest0, Source2, -Source1
```

Note

Source1 and Source2 will be modified after this macro!

```
sgn    r3, r1, r2
```

sincos (macro)

vs 2.0

Computes the sine and cosine values for a scalar argument.

Eight slots

```
sincos  Dest0, Source0, Source1, Source2
```

Estimates the sine and cosine value inside a shader with a maximum error of 0.002 through the use of a Taylor series expansion. Source0 must have a replicate swizzle to indicate which element to use. This should be a value in radians between $\pm\pi$. Dest0 should be a temporary register. The destination must have .x, .y, or .xy as a write mask.

Setup One element of Source0 has to have the value in radians. Source1 and Source2 have to be set up with the following values to perform the expansion.

```
Source1 = [1/(7!*128),1/(6!*64),1/(4!*16),1/(5!*16)]
Source2 = [1/(3!*8),1/(2!*8),1,0.5]
```

Results The resulting sine and cosine values are written in Dest0.x and Dest0.y respectively.

```
// setup values
def c1, 1.0f/(7!*128),1.0f/(6!*64),
      1.0f/(4!*16), 1.0f/(5!*16)
```



```
def c2, 1.0f/(3!*8), 1.0f/(2!*8), 1.0f, 0.5f

// assume value to take sin/cos of is in r0.x
sincos r0.xy, r0.x, c1, c2
```

slt

vs 1.0, 1.1, 2.0

Set Less-Than. Stores 1 in the destination register if the first source register is less than the second source register. If not, it stores 0 in the destination register. Does an element-by-element comparison and assignment.

One slot

```
slt Dest0, Source0, Source1
```

Compares the two source register element by element. If the first source register's element is less than the second source register's element, the value 1 is placed in the destination register's element. If not, it stores 0 in the destination register's element. The resulting register may consist of a mix of 0s and 1s.

Setup Source0 and Source1 are the registers to be compared.

Results The element Dest0.*n* contains 1.0 if the Source0.*n* is less than Source1.*n* otherwise, it contains 0.0. This is done for all elements of Dest0.

```
slt    r0. r1, r2
```

```
SetSourceRegisters();
```

```
// Simulate the slt instruction
```

```
TempReg.x = Source0.x < Source1.x ?
```

```
1.0f : 0.0f;
```

```

TempReg.y = Source0.y < Source0.y ?
    1.0f : 0.0f;

TempReg.z = Source.z < Source0.z ?
    1.0f : 0.0f;

TempReg.w = Source0.w < Source0.w ?
    1.0f : 0.0f;

WriteDestinationRegisters();

```

sub

vs 1.0, 1.1

Subtracts one register from another and places the result into the destination register.

One slot

```
sub  Dest0, Source0, Source1
```

Subtracts Source1 from Source0 and places the result in the Dest0 register.

Setup Two source registers, Source0 and Source1.

Result Each element of Dest0 is filled with the element-by-element subtraction of the element of Source1 from Source0.

```
sub  r0, r0, c2
```

```
SetSourceRegisters();
```

```
// simulate the sub instruction
```

```
TempReg.x = Source0.x - Source1.x;
```

```
TempReg.y = Source0.y - Source1.x;

TempReg.z = Source0.z - Source1.z;

TempReg.w = Source0.w - Source1.w;

WriteDestinationRegisters();
```

vs

vs 1.0, 1.1, 2.0

Defines the version of the vertex shader code you are using.

No slots

```
vs.integer1.integer2 // DirectX 8
vs_interger1_integ2 // DirectX 9
```

The argu for DirectX 8 shaders, and `vs_s_y` for DirectX 9 shaders, where x is the main version number, and y is the minor version number. Both values the integers.

Setup Two integers that instruct the assembler on the major and minor version numbers of the shader version you want to use. This must be the first instruction in your shader.

Results Tells the assembler what features to allow in the shader instruction to follow.

```
//DirectX 8
vs.1.0 // not using the address register in this one
vs.1.1 // uses address register

//DirectX 9
vs_2_0
```

Register Masks, Swizzles, Negates, and Indexing

By default, a register reference in a shader expands to a reference to all the elements of the register, in x, y, z, w order. This means that whenever you write R, where R is the name of some register, it automatically gets expanded (at least conceptually) to `R.xyzw`. What I want to get across is that the specific element-by-element reference to the register is made without your having to do anything. This is merely semantic convenience and efficiency.

What this means is that it costs nothing extra to reorder or even ignore specific elements of a register if it makes sense to. I'll say it again: it costs nothing to use these masks in your shaders. They are there so you can take advantage of the single-instruction multiple-data nature of the shader language to possibly merge similar computations or save on register usage. If, in fact, you don't need to have a value stored in every element of a register, then by all means use a destination mask and have the value you need written only to the destination register.

destination mask/write mask

Note the word *destination*. Masks can be used only to select which element of a register is to be written to. (Hence they are often referred to as *write* masks.) Even if an instruction usually writes to all four elements of the destination, the mask can be used to select which element(s) of the destination are written. If you leave an element out of a mask, it doesn't get written. The element masks must be in order; x comes before y which comes before z which comes before w.

```
mov    r1, c1 // use all
mov    r1.xyzw, c1 // use all explicitly (default)
mov    r1.xw, c1 // just move c.x and c.w
mov    r1.wx, c1 // Error! - invalid order
mov    r1.wzyx, c1 // Error! - invalid order
```

Source swizzle

Note the word *source*. Swizzles can be used only to select the order and the elements of a register to use as a source. A swizzle *must* specify four elements though there is no restriction on the order. A swizzle consists of four letters, `xyzw`. If there are not four elements specified, the last element specification is replicated.

```
mov    r1, c1 // use all - same as below
mov    r1, c1.xyzw // use all in order
mov    r1, c1.wzyx // reverse the order
```

```

mov    r1,  c1.wwww // just use the w element
mov    r1,  c1.xyzy // replace w with y
mov    r1.W, c1.zxyx // move c.x into r1.w
mov    r1.Z, c1.xzwy // move c.w into r1.z
mov    r1,  c1.x // same as c1.xxxx
mov    r1,  c1.xy // same as c1.xyyy

```

source absolute value

Available only in vs 2.0. The absolute modifier takes the absolute value of the source register. If the `_abs` is used with the negate modifier, the `_abs` is done first.

```

vs_2_0 // vertex shader 2.0 or better

mov r5,  r5_abs // absolute value of c5 in c5
mov r5, -r5_abs // all values in c5 will be negative

```

saturation instruction modifier

Available only in vs 2.0. The saturation instruction modifier clamps the results to the [0,1] range.

```

vs_2_0 // vertex shader 2.0 or better

add_sat r5, r5, r5 // absolute value of c5 in c5

```

source negation

Negation can be used to negate an entire source register. They can be used with swizzles.

```

mov    r1, c1          // move c1 into r1
mov    r1, -c1         // move negative c1 into r1
mov    r1.w, -r1       // negate just r1's w and store it
mov    r1.w, -c1.zzzy // move -c1.y into r1.w
mov    -r1, c1         // Error! Can't negate destination

```

address registers

Not available in shader version 1.0. Only register `a0.x` can be used for version 1.1. The address register can be used as a signed offset into the constant register file, and it can be used only in the `mov` or `movb` instruction. The values in the address register when used must compute to the legal range for the constant registers (i.e., 0 to 95 for most 1.1 vertex shaders).

```
vs.1.1
mov    a0.x, c5.w    // load a0.x

mov    r1, c1        // regular move, format 1
mov    r1, c[1]      // same thing alternative format
mov    r1, c[1+a0.x] // relative move
```

Part II: Pixel Shader Reference

QUICK OVERVIEW

Just as vertex shaders operate on a single vertex, pixel shaders operate on a single pixel. Pixel shaders' instructions break down into two categories: those that work on the color and alpha blending and those that work on texture addressing. Generally you can perform a color and/or alpha blending operation or a texture addressing operation with a single pixel shader instruction.

While vertex shaders are straightforward, pixel shaders are a bit more idiosyncratic since they more closely mimic the underlying hardware, and the hardware is evolving. Pixel shader versions 1.0 through 1.3 tend to show the influence of hardware manufactured by NVIDIA, while version 1.4 exposes the way the ATI hardware works. Version 2.0 shaders introduce yet another level of features while removing or subsuming some of the capabilities of the previous versions and provide a better experience.

Like vertex shaders, you can reference individual pieces of a register; in the case of pixel shaders, these are specified by *rgb* and *a* instead of *xzy* and *w*. Unlike vertex shaders, component swizzles are not supported by the pixel shader specification until PS 2.0.

In PS 1.0–1.3, the *r*, *g*, and *b* components are treated together as a single RGB color component, while the *a* component is treated as alpha, which *can* be routed separately. Whereas with PS 1.4 and 2.0, you can mask out or extract the components independently.

DirectX 9 update



DirectX 9 introduced some significant changes with the pixel shader 2.0 specification. Flow control, higher-precision registers, separate texture coordinate and texture sampling registers, and a new output target are some of the significant changes introduced. DirectX 9 also introduced a high-level-shading-language (HLSL) that at the time of this writing is still being specified.

While I've made every effort to try to get the basics of the changes up to date, I'm still awaiting the third beta release for DirectX 9, so I suggest that you get the latest version of the DirectX 9 SDK and check there for the latest changes. The <http://www.directx.com> Web site will also post any updates or errata for the documentation published in this book.

INSTRUCTION ORDER AND INSTRUCTION COUNTS

The order in which instructions appear in a pixel shader is determined by the shader version. There are five different types of instructions that can be in a pixel shader. The very first instruction must be the shader version declaration, followed by the constant definitions. The phase instruction is found only in PS 1.4 shaders. None of these instructions count toward the total number of instructions; only the two remaining types, the texture addressing and the arithmetic instructions, count toward the total instruction count.

DirectX 9 update



Significant changes were made to pixel shaders between versions 1.3 and 1.4 due to the addition of the phase instruction in PS 1.4. The `phase` instruction was removed for PS 2.0 shaders and arithmetic and texture addressing instructions could be mixed.

Note

Destination registers are independent of the read port count restrictions.

For Pixel Shaders 1.0 Through 1.3

Texture addressing instructions must precede the color-blending instructions. There can be a maximum of eight arithmetic instructions and a maximum of four texture instructions. For PS 1.0, there's an additional limitation in that the total number of instructions is limited to eight ([Figure 8.1](#)).

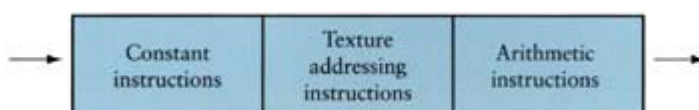


Figure 8.1: Constant, texture addressing, and arithmetic instructions for pixel shaders 1.0 through 1.3.

The syntax for the texture address computation operation is included within the shader program itself in the form of texture declarations. These control how address perturbation operations are applied to the color values sampled from the textures during their preloading into the temporary register file for use in the body of the pixel shader.

For Pixel Shaders 1.4

Order is much more significant for PS 1.4 with the introduction of the `phase` instruction, and the number of instructions that can be in each phase is limited as well. If there is no phase instruction, then everything after the `vs` and `def` instructions is treated as being in phase two. There can be only one `phase` instruction in a shader.

Phase one and two instructions must be in the order shown in [Figure 8.2](#), and each can consist of up to six texture addressing instructions and up to eight arithmetic instructions.

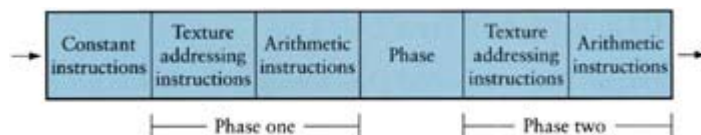


Figure 8.2: Phase instruction in pixel shaders 1.4.

Thus you can have up to a total of 28 instructions in a PS 1.4 shader, not including the version, define, or phase instructions. There are also more restrictions on which registers can be the source or destination registers. This can vary between the phase of the pixel shader and the individual instruction. See the reference section for an instruction about which registers can be used in each phase. After the `phase` instruction, the alpha component of the temporary registers is lost, and the alpha channels are uninitialized.

For Pixel Shaders 2.0

There can be a maximum of 32 texture instructions and 64 arithmetic instructions, and they can be in any order in the shader.

TEXTURE ADDRESSING

Texture coordinates are either passed directly to the pixel shader, or looked up from a texture map, or computed in the shader.

For Pixel Shaders 2.0

The texture coordinates can only be used as texture coordinate data and not as sampled texture data. PS 2.0 shaders now provide separate sampling registers that are used to sample texture data.

TEXTURE STAGE AND TEXTURE SAMPLE STATES

With the advent of pixel shaders, some of the texture stage states and texture sampling states are ignored when a pixel shader is active. States that are active can be changed without having to reload the pixel shader.

[Table 8.1](#) shows the texture stage states and how they are active in the pixel shader versions; whereas, [Table 8.2](#) shows the texture sampling states and how they are active in the pixel shader versions.

Table 8.1: Directx 9 Texture Stage State

TEXTURE STAGE STATE	ACTIVE IN THESE PIXEL SHADER VERSIONS
D3DTSS_ADDRESSU/V/W	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_BORDERCOLOR	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MAGFILTER	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MINFILTER	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MIPFILTER	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MIPMAPLODBIAS	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MAXMIPLEVEL	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MAXANISOTROPY	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_BUMPENVLSCALE	1.0, 1.1, 1.2, 1.3, 1.4
D3DTSS_BUMPENVMATnn	1.0, 1.1, 1.2, 1.3, 1.4
D3DTSS_BUMPENVLOFFSET	1.0, 1.1, 1.2, 1.3, 1.4
D3DTSS_TEXCOORDINDEX	Ignored
D3DTSS_TEXTURETRANSFORMFLAGS	Ignored
D3DTSS_COLOROP	Ignored
D3DTSS_ALPHAOP	Ignored
D3DTSS_COLORARGn	Ignored
D3DTSS_ALPHAARGn	Ignored
D3DTSS_RESULTARG	Ignored

Table 8.2: Directx 9 Texture Sampling State

TEXTURE SAMPLING STATE	ACTIVE IN THESE PIXEL SHADER VERSIONS
D3DTSS_ADDRESSU/V/W	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_BORDERCOLOR	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MAGFILTER	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MINFILTER	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MIPFILTER	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MIPMAPLODBIAS	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MAXMIPLEVEL	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_MAXANISOTROPY	1.0, 1.1, 1.2, 1.3, 1.4, 2.0
D3DTSS_SRGBTEXTURE	Ignored
D3DTSS_ELEMENTINDEX	Ignored
D3DTSS_DMAPOFFSET	Ignored

PIXEL SHADER MATH PRECISION IN DIRECTX 8 CLASS HARDWARE

Registers in pixel shaders are *not* full 32-bit floating point values. In fact, they are severely limited in their range. The minimum precision is 8 bits, which usually translates to an implementation of a fixed point number with a sign bit and 7–8 bits for the fraction. Since the complexity of pixel shaders will only grow over time, and hence the ability to do lengthy operations, you can expect that you'll rarely run into precision problems unless you're trying to do something like perform multiple lookup operations into large texture spaces. Only on older cards (those designed in 2001) will you find the 8-bit minimum. As manufacturers figure out how to reduce the size of the silicon and increase the complexity, they'll be able to squeeze more precision into the pixel shaders.

As the number of bits increase in the pixel shader registers, so will the overall range. You'll need to examine the `D3DCAPS8` value `MaxPixelShaderValue` in order to see the range to which pixel registers are clamped. For PS 1.4 and 2.0, the texture registers have a valid range of $\pm \text{MaxTextureRepeat}$. [Figure 8.3](#) shows DirectX 8's 8-bit pixel register construction.

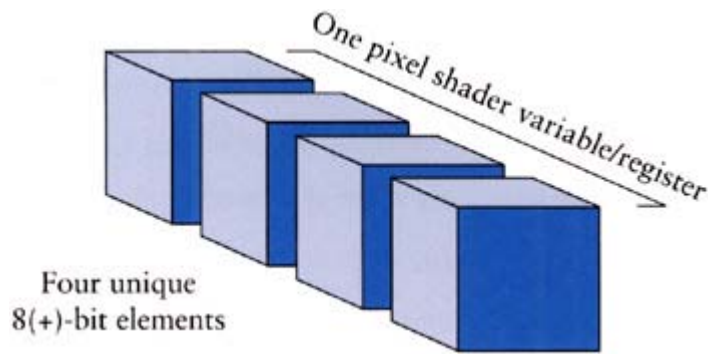


Figure 8.3: Pixel shader register construction.

In DirectX 6 and DirectX 7, this value was 0, indicating an absolute range of $[0, 1]$. In later versions of DirectX, this value represented an absolute range; thus in DirectX 8, you might see a value of 1, which would indicate a range of $[-1, 1]$, or 8, which would indicate a range of $[-8, 8]$.

PIXEL SHADER MATH PRECISION DIRECTX 9 CLASS HARDWARE

One of the most significant though subtle changes that was introduced in DirectX 9 pixel shaders is higher precision registers. The pixel shader 2.0 temporary registers and constants must have a minimum of 16-bit signed floating point. This allows for much more complicated and subtle shaders and reduces the chances of banding due to loss of precision. Some hardware implementations will support a full 32-bit IEEE float precision. Input texture coordinate registers are a minimum of 24-bit signed floating point numbers. The input color registers are unchanged from the DirectX 8 shader implementation and are still limited precision numbers in the $[0, 1]$ range.

PIXEL SHADER REVISION HISTORY

Here is the revision history of pixel shaders. Each section contains the changes from the previous version. Significant changes were made to pixel shaders in versions 1.4 and 2.0.

DirectX 9.0

Version 3.0

Pixel Shader 3.0 specification was added.

Version 2.0

Texture and arithmetic instructions can be mixed.

Floating point register support (16- and 32-bit) added.

The `cnd`, `bem`, and `phase` instructions were removed.

The `exp`, `frc`, `log`, `rcp`, `texlb`, `texlp`, `dp2add`, and `rsq` instructions were added.

The `abs`, `crs`, `lrp`, `max`, `min`, `sincos`, `m4x4`, `m4x3`, `m3x4`, `m3x3`, `m3x2`, `nrm`, and `pow` macros were added.

The `lrp` instruction was changed to a macro.

All source modifiers except `negate` removed.

All instruction modifiers except `saturate` removed.

Partial precision hint added.

Co-issue was removed.

Multiple registers are possible as output.

Constants now default to a 0,0,0,0 value.

The number of constant registers was increased to 32.

Sixteen sample registers were added.

DirectX 8.1

Version 1.4

The `phase`, `bem`, `texcrd`, `texdepth`, and `texld` instructions were added.

Dropped support for all texture address instructions except `texkill`.

Texture registers are read only and can be used only in texture addressing instructions.

The read port limit for texture registers was decreased to one because they can't be used in arithmetic instructions.

The number of temporary registers was increased to six.

The read port limit for temporary registers increased to three.

The `caps` bit for `MaxTextureRepeat` must be at least eight (from one).

`_x8`, `_d4`, `_d8` instruction modifiers were added.

`_x2` source register modifier was added.

`. r`, `. g` source register selectors were added.

`. r`, `. g`, and `. b` destination masks were added.

Version 1.3

The `texm3x2depth` instruction was added.

Version 1.2

The `cmp`, `dp4`, `texdp3`, `texdp3tex`, `texm3x3`, and `texreq2rgb` instructions were added.

The read port limit for texture registers was increased from two to three. The `.b` argument modifier was added.

Version 1.1

Texture registers are no longer read only.

The read port limit for texture register and vertex color registers was increased from one to two.

`.b` register selectors were added.

Arithmetic operations can use more register types as source registers.

Version 1.0

Initial release of pixel shaders.

OUTPUT REGISTERS

Unlike the vertex shaders, DirectX 8 pixel shaders do not have any specialized output, but use the temporary `r0` register as the single color output. See the Temporary Registers section later in this chapter for more information. DirectX 9 shaders use the multi-element texture register as output, holding four colors and a depth value.

For Pixel Shaders 1.3

A texture register can hold an output depth value.

For Pixel Shaders 1.4

The `r5` register can hold a depth value.

For Pixel Shaders 2.0

PS 2.0 now supports new types of render destinations. The output register for regular shader destinations is `oc0`. If you have multiple render targets you'll use the MRT or MET as render targets.

INPUT REGISTERS

The input registers are how data gets passed to the pixel shader. All registers consist of a four-element vector, with at least 8 bits of precision per component in some of the earlier hardware implementations,

which may cause round-off to be a problem in long, complicated shaders. In PS 1.4 class hardware or later, if you need more precision, then you can use texture coordinate registers to pass in higher precision values. See the Texture Registers subsection later in this section for more information.

Input to the pixel shader consists of the vertex color registers, the texture registers, and the constant registers. The vertex color registers are the direct output of the vertex shader, the values in `oD0` and `oD1`, or if no vertex shader was used, the diffuse and specular colors are generated by the fixed function pipeline. The float constant registers are constants that are set up prior to the shader being called or are set inside the shader code itself. The texture registers are interpolated values taken from the vertex data.

v[n]: The Vertex Color Registers

Pixel shader vertex color registers consist of two read only registers that are generated as output from the vertex shader or the fixed function pipeline. If the current render state is flat shading `D3DSHADE_FLAT`, then the vertex color registers use only the vertex shader color data from the first vertex in the triangle. If Gouraud shading is enabled (i.e., `D3DSHADE_GOURAUD`), then the pixel values will be a linear interpolation of the three vertex color values used in the current triangle's vertices. The vertex color registers (potentially) have the lowest precision of all the registers, but a minimum of 8 unsigned bits of precision ([Table 8.3](#)).

Table 8.3: Vertex Color Register Properties

PS VERSION	1.0	1.1	1.2	1.3	1.4	2.0
NUMBER	2	2	2	2	2 [□]	2
PORT LIMITS	1	2	2	2	2	1
READ/WRITE?	RO	RO	RO	RO	RO	RO
RANGE	0,1	0,1	0,1	0,1	0,1	0,1

[□]In PS 1.4, you can use the vertex color register only in phase two.

c[n]: The Constant Registers

The pixel shader float constant registers are similar to the ones we used for vertex shaders. Just like those vertex shader constant registers, the pixel shader constant registers are read only. There are at least eight constant registers in DirectX 8 and 32 in DirectX 9, but generally only one can be used at a time in a single pixel shader instruction ([Table 8.4](#)).

Table 8.4: Constant Register Properties

PS VERSION	1.0	1.1	1.2	1.3	1.4	2.0
NUMBER	8	8	8	8	8	32
PORT LIMITS	2	2	2	2	2	2
READ/WRITE?	RO	RO	RO	RO	RO	RO
RANGE	−1,1	−1,1	−1,1	−1,1	−1,1	16- or 32- bit float precision

Because of the limited precision of DirectX 8 pixel shader math (the register values might be represented internally by a fixed point representation), pixel shader constant registers can only have a range of $[-1, 1]$.



DirectX 9 pixel shader constants are at least 16-bit floating point values. You set pixel shader constant registers either by using the `def` instruction in the shader or by using `SetPixelShaderConstant()`. Unlike vertex shaders, the `def` instruction requires no other effort from the programmer; the constant values are set when the shader is loaded. You can override values set in the shader using the `def` instruction after the shader is loaded by a subsequent call to `SetPixelShaderConstant()`.



In DirectX 9 pixel shader the `def` instruction will push the previous value of the constant and pop it when the shader exits.

You generally cannot use constant registers with the texture instructions—the exception being the `texm3x3spec` instruction.


t[n]: the Texture Registers


Unlike vertex shaders (which have texture coordinate registers), pixel shaders up to PS 1.4 have texture color registers. For PS 2.0 they are texture coordinate registers. The number of texture registers is shown in [Table 8.5](#).

Table 8.5: Texture Register Properties

PS VERSION	1.0	1.1	1.2	1.3	1.4	2.0
------------	-----	-----	-----	-----	-----	-----

Table 8.5: Texture Register Properties

PS VERSION	1.0	1.1	1.2	1.3	1.4	2.0
NUMBER	4	4	4	4	6	8
PORT LIMITS	1	2	3	3	1	1
READ/WRITE?		RW	RW	RW	RO	RO
RANGE	±Max	±Max	±Max	±Max	±Max	±Max
	Pixel-	Pixel-	Pixel-	Pixel-	Texture-	Texture-
	Shader	Shader	Shader	Shader	Repeat	Repeat
	Value	Value	Value	Value		

 For PS 1.0, the texture registers are read/write for texture addressing instructions, but read only for arithmetic instructions.

There is a significant difference between the way that the texture registers are used between pixel shaders 1.0 through 1.3 and pixel shaders 1.4, and again at pixel shaders 2.0.

For Pixel Shaders 1.1 Through 1.3

The value in the texture register is a sample of the texture at the interpolated point—that is, it's a color value. The interpolated value consists of (*u*, *v*, *w*, *q*) data, as interpolated from the current texture stage attributes. There is a one-to-one association between the texture stage and the texture coordinate declaration order. This means that the texture selected in stage *n* will provide the color values passed into the pixel shader in the texture registers. Texture coordinates are read/write, which means that you can use them as temporary registers if necessary.

For Pixel Shaders 1.4

The texture registers for 1.4 pixel shaders should be thought of as texture coordinate registers since they can contain only read-only texture coordinates. This means that the data in the texture register is *not* color data as in previous pixel shader versions. The `texcrd` instruction is used to pass texture coordinates, and the `texld` instruction is used to sample a texture. Texture samples will end up in temporary registers.

The texture stage that is used to sample from is no longer associated with the texture coordinate declaration order, but is determined by the destination register number. Thus when you actually load a texture value using `texld`, the texture register used as the source register will contain the texture coordinate information, while the destination register number will correspond to the texture to sample.

Initialized texture registers contain the output from the texture-sampling units after any filtering modes set for that texture stage. Unused, uninitialized texture registers have a setting of opaque black (0,0,0,1). It also means that you can no longer use them as temporary registers.

Loading a texture coordinate will place only three components into the register, even though a texture coordinate starts out as a four element register. Thus you have to use a destination register write mask when using the `texld` or `texcrd` instruction.

If you want to perform a perspective divide in a PS 1.4, the `D3DTEXTUREF_PROJECTED` flag is ignored. You can perform a perspective divide on the `.x` and `.y` texture coordinates using the `_dw` modifier, and the results require a `.xy` write mask.

For Pixel Shaders 2.0

In order to end confusion about texture coordinates `vs. texture` sampling using the texture registers, the PS 2.0 model breaks these into two separate sets of registers.

TEMPORARY REGISTERS

r[n]: The Temporary Registers

The temporary registers can be used to hold anything you want in a pixel shader.

Temporary registers are considered uninitialized when a vertex shader is first loaded, and an attempt to assemble a shader that fails to initialize a temporary register before using it will fail validation ([Table 8.6](#)).

PS VERSION	1.0	1.1	1.2	1.3	1.4	2.0
NUMBER	2	2	2	2	6	12
PORT LIMITS	2	2	3	3	3	3
READ/WRITE?	RW	RW	RW	RW	RW	RW
RANGE	±Max	±Max	±Max	±Max	±Max	16- or 32-
	Pixel-	Pixel-	Pixel-	Pixel-	Pixel-	bit float
	Shader	Shader	Shader	Shader	Shader	precision
	Value	Value	Value	Value	Value	

The `r0` register is a special case, and all pixel shaders except PS 2.0 must write `r0` as the final color of the pixel that is being rendered. PS 2.0 has added specific output registers.

The `r5`, `.r` and `.g` register elements are used as output if you used the `texdepth` instruction.

Note

Since texture registers are read/write (except in PS 1.0, 1.4, and PS 2.0), you are free to use unused (or no longer needed) texture coordinate registers as extra temporary registers.

s[n]: The Texture Sampling Stage Registers

9.0

The DirectX 9 pixel shader 2.0 model separated the texture registers into texture coordinate registers and texture sample stage registers. A sampling stage register identifies a sampling unit that can be used in texture load statements. A sampling unit corresponds to a texture sampling stage, encapsulating the sampling-specific state provided in the `TextureStageState` construct (called `SamplerState` in DirectX 9). Each sampler uniquely identifies a single texture surface which is set to the corresponding sampler using `SetTexture()` method. The same texture surface can be set at multiple samplers. A texture cannot be simultaneously set as a render target and a texture. A sampler register may only appear as an argument in texture load statements

Two new render states were added that effect texture sampling stage registers:

`D3DSAMP_SRGBTEXTURE` which indicates that this sampler should assume gamma of 2.2 on the content and convert it to gamma 1.0 (linear gamma) before sending it to the pixel shader.

`D3DSAMP_ELEMENTINDEX` describes which element index to use when a MET surface is set to the sampler. The MET surface must be supported or this render state is ignored.

Declaring Register Usage

9.0

DirectX 9's PS 2.0 model requires that all registers (except temp and constant) used in a shader be declared via the `dc1` statement down to the component (rgba) level.

This is to simplify things for optimization and verification.

OUTPUT REGISTERS—THE MULTI-ELEMENT TEXTURE (MET) AND THE MULTIPLE RENDER TARGET (MRT)

9.0

?/td>

In DirectX 8 pixel shaders, only the `r0` register was used to hold the output color. In DirectX 9 there is better support for specifying a texture as a render target: the Multiple Render Target (MRT) or a subset called the Multi-Element

Texture (MET). This allows one to specify up to a four-dimensional texture (with an optional depth value) as output from a pixel shader.

If you're not using a MRT/MET, the syntax is to use the `oC0` register as the output color register. If your device supports multiple render targets, the output colors are sent to the appropriate render target. To use MRT/MET you'd use the four output color registers and output depth register. These registers are `oC0` through `oC3`. The output registers must be written to in ascending order, and all elements of each register must be written to. Each register can only be written to once in a shader. If there's a depth buffer attached to the MRT surface, you can specify a depth value for the pixel in the `oDepth` register. This is a scalar register.

The restrictions on using an MRT as a destination are

- The color elements can only be written with an `.rgba` mask.
- You can write to an `oCn` or `oDepth` register only once per shader.
- You must write to the `oC0` register.
- `No_sat` or source modifiers.
- You cannot use an output mask when writing to `oCn`.
- The range of `oCn` registers have to be updated sequentially. You're not allowed to skip outputs (i.e., you can't write `oC0` and `oC2`, and not `oC1`).

The restrictions on using a MET as a destination are

- All surfaces are allocated atomically.
- Only 32-bit formats are supported.
- MET surfaces have to be off-screen.
- No fragment processing instructions.
- MET surfaces can't be filtered when sampled from.

The restrictions on using `oDepth` are

- You *cannot* specify a write mask when writing to `oDepth`.
- You *must* specify a replicate swizzle when writing to `oDepth`.

abs(macro)

ps 2.0

This macro computes the absolute value of the input register on an element-by-element basis.

One arithmetic instruction slot.

```
abs    Dest0,    Source0
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.2 phase 2									
2.0				x	x	x	x	x	

Computes the absolute value of each element of Source0 and places the result in Dest0.

```
abs    r4,    r4
```

This macro is equivalent to the following code:

```
cmp    Dest0,    Source0,    Source0,    -Source0
```

add ps 1.0–2.0

Adds two source colors and places the sum into a third register.

One arithmetic instruction slot.

add Dest0, Source0, Source1

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0				x	x	x	x	x
1.1			x	x	x	x	x	x
1.2			x	x	x	x	x	x
1.3			x	x	x	x	x	x
1.4 phase 1				x		x	x	x
1.4 phase 2				x	x	x		x
2.0				x	x	x	x	x

Adds the Source0 and Source1 registers and places the result in the Dest0 register.

```
add    r0,    r0,    c2
add    r0,    r0,    1-r1
add    r0,    r0,    t0_bias
```

bem ps 1.4

Bump-Environment-Map. Apply a fake bump environment map transform.

Two arithmetic instruction slots.

bem Dest0.rg, Source0, Source1

- Only one `bem` instruction can appear in a shader.
- The `bem` instruction must be in phase one.
- The source registers can be temporary or constant registers.
- The destination must be a temporary register.
- The destination mask must be `.rg` (or `.xy`).
- The instruction cannot be co-issued.

Note?

This instruction works only in PS 1.4. In PS 2.0 you can use two `dp2add` instructions to emulate the `bem` instruction.

```
bem r3.rg, r0, r0 // bump texture 3 * c2, add in r0
bem r4.xy c2, r0 // bump texture 4 * r0, add in c2
```

LEGAL REGISTER ARGUMENTS

[illegible]

cmp

ps 1.2–1.4

Compare. Performs a conditional assignment based on a comparison of the values in the source registers.

Two arithmetic slots for PS 1.2 and 1.3; one for PS 1.4.

```
cmp    Dest0,    Source0,    Source1,    Source2
```

If the value in Source 0 is greater than or equal to zero, then Source1 is placed in the destination; otherwise, Source2 is placed in the destination. This comparison is performed on an element-by-element basis. Dest0 must be a different register from any of the source registers.

Note

For PS 1.2 through 1.3, the source register can be any register, while the destination can be either a texture or a temporary register. There can be a maximum of three cmp instructions in a pixel shader.

Bug

For PS 1.2 and 1.3, the destination register cannot be the same as a source register, but validation fails to catch this. The cmp instruction takes up two slots, but validation counts this as only one slot.

PS 1.4

For PS 1.4, the source register can be temporary registers, constant registers, or vertex color registers for phase two, whereas the source register can be temporary registers or constant registers for phase one. The destination register must be a temporary register.

```
cmp    r0,    r1,    c1,    c2
```

LEGAL REGISTER ARGUMENTS

DESTINATION

SOURCE

PS version

v

c

t

r

v

c

t

r

1.0

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.1								
1.2			x	x	x	x	x	x
1.3			x	x	x	x	x	x
1.4 phase 1				x		x		x
1.4 phase 2				x	x	x		x

cnd **ps 1.1–1.4**

Conditional. Performs a conditional assignment based on a comparison of the value in r0.a (PS 1.1–1.3) or another register (PS1.4) with 0.5.

One arithmetic instruction slot.

```
cnd Dest0, Source0, Source1, Source2
```

Compares the value in r0.a to 0.5. Source0 *must* be r0.a for PS 1.1–1.3. If Source0 is greater than 0.5, then Dest0 is set to the value in Source1. If not, then Dest0 is set to the value in Source3.



PS 1.4

For PS 1.4, the cnd instruction operates on an element-by-element basis.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				SOURCE1/2			
	v	c	t	r	v	c	t	r	v	c	t	r

1.0												
1.1			x	x				r0.a	x	x	x	x
1.2			x	x					x	x	x	x
1.3			x	x				r0.a	x	x	x	x
1.4 phase 1				x		x		x		x		x
1.4 phase 2				x	x	x		x	x	x		x

You can also use this command to compare two values with an added step as shown.

```
cnd    r0,    r0.a,    c1,    c2 // PS 1.1-PS    1.3

// to compare two values v0 and v1
sub r0, v0, v1_bias; r0 = v0 - (v1 - 0.5) = v0 - v1 + 0.5
cnd r0, r0.a, c0, c1; if r0.a > 0.5 use c0, else c1
; thus v0-v1+0.5 > 0.5 = v0-v1 > 0 = v0 > v1
```

crs (macro)

ps 2.0

The three component cross product.

Two arithmetic instruction slots.

```
crs Dest0, Source0, Source1
```

Computes the three component cross product using the right-hand rule. There cannot be ands swizzles on the source registers. Dest0 must have a write mask that is one of the following: `.x`, `.y`, `.z`, `.xy`, `.xz`, `.yz`, or `.xyz`. The destination register should be different from the source registers.

```
crs r1, v0, r0
```



This macro is equivalent to the following code sans write mask on Dest0.

```
mul Dest0, Source0.zxyw, Source1.yzxw
mad Dest0, Source0.yzxw, Source1.zxyw, -Dest0
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	
dcl									ps 2.0

Declare how a texture sampler register is to be used.

Takes no slots.

```
dcl_<type> Dest0
```

In order to make it easier to optimize and verify shaders, PS 2.0 now requires a declaration statement on all sampler, color input, and texture coordinate registers. All sampler registers must be declared before use in the shader. Dest0 will be a specific input register. The *<type>* tag is a texture register indicator of the type of texture this register is to be used with. The allowable types are; 2d, cube, and volume.

Color input and texture coordinate register must specify a mask of which components are used. Texture coordinate register may specify the partial precision (*_pp*) modifier.

Must appear before any arithmetic or texture instructions.

```
// sampler registers
dcl_2d    s1 // s1 will be using a 2D texture
dcl_cube  s2 // s2 will be using a cubemap
dcl_volume s3 // s3 will be using a 3D texture

// color and texture coordinate registers
dcl t1.rg
dcl_pp t2.rg
dcl v0.rgba
```

def

ps 1.0–2.0

Defines the constants to be used in the shader.

Takes no slot or time.

```
def Const1, float1, float2, float3, float4
```

The first argument is the name of the constant—`c0`, `c1`, etc. The remaining arguments are the four floating point values to be placed in the constant register in *a*, *r*, *g*, *b* order. The constants are available once the shader is loaded, but a subsequent call to `SetPixelShaderConstant()` will overwrite any constant values previously set except on PS 2.0, where the constants defined in the shader take precedence.

```
def c0, 0.0f, 0.0f, 0.0f, 0.0f
def c1, 1.0, 0.5, 0.25, 0.125
```

dp2add

ps 2.0

Two-component dot product plus a scalar add. Can be used to emulate the bem instruction.

One arithmetic instruction slot.

```
dp2add Dest0, Source0, Source1, Source2
```

Computes the two component dot product of the Source0 and Source1 (elements `.x` and `.y`) and add in a scalar value from Source2 (which must have a replicate swizzle). The results are replicated to all elements of the destination.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2								
1.3								

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.4 phase 1								
1.4 phase 2								
2.0				x	x	x	x	x

```
dp2add r0, v0, r1, c4.z
```

dp3

ps 1.0–2.0

Three-component dot product (dot product three) of the color arguments, replicated to all channels of the output register.

One arithmetic instruction slot.

```
dp3 Dest0, Source0, Source1
```

Computes the dot product of the Source0 and Source1 color registers and places the result in all channels of the Dest0 register. Only the `.r`, `.g`, and `.b` values are used to compute the dot product; the alpha component is ignored. The `dp3` instruction *does not* clamp the results to the range [0,1]. If you want to clamp the results, use the `_sat` modifier. If a write mask is used, only the selected channels are written.

Note

Since the `dp3` instruction is a vector operation, it is always scheduled for the vector pipeline. Thus when used with instruction pairing, it's always got to be the vector

operation. It can be co-issued if the dp3 is writing color channels while the other instruction is writing the alpha channel.

```
dp3 r0,    r3, r4 ; rgba is set
dp3 r0.rgb, r3, r4 ; only rgb is set
dp3 r0.a,   r3, r4 ; only a is set
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0				x	x	x	x	x
1.1			x	x	x	x	x	x
1.2			x	x	x	x	x	x
1.3			x	x	x	x	x	x
1.4 phase 1				x		x		x
1.4 phase 2				x	x	x		x
2.0				x	x	x	x	x

dp4

ps 1.2–2.0

Four-component dot product (a.k.a. Dot-product four) of the color arguments, replicated to all channels of the output register.

Two arithmetic instruction slots in PS 1.2 and 1.3; one slot in PS 1.4 and 2.0.

```
dp4 Dest0, Source0, Source1
```

Computes the dot product of the Source0 and Source1 color registers and places the result in all channels of the Dest0 register. The .r, .g, .b, and a values are used to compute the dot product. The dp4 instruction *does not* clamp the results to the range [0,1]. If you want to clamp the results, use the _sat modifier. If a write mask is used, only the selected channels are written.

Note This instruction cannot be co-issued with another instruction.

For PS 1.2 and 1.3: A maximum of four dp4s are allowed in a single pixel shader.

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE			
PS version	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2				x	x	x	x	x
1.3				x	x	x	x	x
1.4 phase 1				x		x		x
1.4 phase 2				x	x	x		x
2.0				x	x	x	x	x

Bugs PS 1.2 and 1.3 The destination register should not be the same as the source registers, and validation does not catch this. Though this instruction takes two slots, it's actually counted as taking one so that it's possible to overflow the instruction buffer.

```
dp4    r0,    r3, r4; rgba is set
dp4    r0.rgb, r3, r4; only rgb is set
dp4    r0.a,   r3, r4; only a is set
```

exp ps 2.0

Computes the base two exponent of a scalar value.

One arithmetic instruction slot.

```
exp Dest0, Source0
```

Computes 2^{Source0} , where Source0 must have a replicate swizzle.

Setup Store the value you want the exponent of in an element of Source0. Use the replicate swizzle to select this element.

Results All elements of Dest0 will contain the exponential value.

```
exp r1, r2.y // replicate exp(r2.y) in r1
```

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE			
PS version	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2								
1.3								
1.4 phase 1								
1.4 phase 2								
2.0				x	x	x	x	x
frc	ps 2.0							

This instruction removes the integer part of the input register on an element-by-element basis.

One arithmetic instruction slot.


```
frc Dest0, Source0
```

Takes the fractional parts of Source0's elements and places them in Dest0's elements on a per-element basis. The truncation used is Source0-floor(Source0), so the results are always positive; that is, any negative values in Source0 will result in the fraction necessary to subtract to reach the next integer < Source0. For example, if Source0.x = -5.4, Dest0.x will equal 0.6. Shouldn't use with `_sat` modifier.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2								
1.3								
1.4 phase 1								
1.4 phase 2								
2.0				x	x	x	x	x

Setup Store the value you want the fractional part of in Source0.

Results Dest0 will contain the fractional values of Source0.

```
frc r6, r2
```

Compute the base two logarithm of a scalar.

One arithmetic instruction slot.

log Dest0, Source0

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2								
1.3								
1.4 phase 1								
1.4 phase 2								
2.0				x	x	x	x	x

Computes the base two log of a Source0 element. You must use a replicate swizzle to select the element of Source0. Dest0 will be filled with the result. If the source value is zero, then the destination result will be MINUS_INFINITY (or at least a really big negative number).

```
log r0, r5.w // place log2(r5.w) in r0
```

lrp (macro 2.0)

ps 1.0–2.0

Linear interpolation between two registers (lerp) using a fraction specified in a third register. It's a macro in PS 2.0.

One arithmetic instruction slot in PS 1.0–1.4; two slots in PS 2.0.

```
add Dest0, Source0, Source1, Source2
```

Source0 contains the fractional interpolant value. When Source0 is zero, Source2 is placed in the destination. When Source0 is one, Source1 is placed in the destination. Values in the [0,1] range interpolate between Source1 and Source2. If the value is outside the range [0,1], the result is indeterminate.

Note You need to be careful that the value of the interpolant doesn't exceed the [0, 1] range. When computing the interpolant, use the `_sat` modifier to clamp the value. In PS 2.0 the destination should be different from all source registers.

```
lrp r0, s0, s1, s2
```

This PS 2.0 macro is equivalent to the following code:

```
add Dest0, Source1, -Source2
mad Dest0, Dest0, Source0, Source2
```

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE			
PS version	v	c	t	r	v	c	t	r
1.0				x	x	x	x	x
1.1			x	x	x	x	x	x

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.2			x	x	x	x	x	x
1.3			x	x	x	x	x	x
1.4 phase 1				x		x		x
1.4 phase 2				x	x	x		x
2.0				x	x	x	x	x
M3?(macro)								ps 2.0

Perform a 3x2 matrix multiply on a 3 element vector.

Two arithmetic instruction slots.

```
m3x2 Dest0, Source0, Source1
```

Multiplies a 3 element vector by a 2x3 matrix. The result is stored in Dest0.xy. The next register from Source1 is implicitly used. Dest0 must not be the same as any source registers, both explicit and implicit.

You cannot use swizzle or negate modifiers on Source1.

```
m3x2 r1, r3, c4 // c5 use implicit
```

This macro is equivalent to the following code:

```
dp3 Dest0.x, Source0, Source1
dp3 Dest0.y, Source0, Source1+1
```

m3? (macro)

ps 2.0

Perform a 3 ?3 matrix multiply on a 3 element vector.

Three arithmetic instruction slots.

```
m3x3 Dest0, Source0, Source1
```

Multiplies a 3 element vector by a 3 ?3 matrix. The result is stored in Dest0.xyz. The next two registers from Source1 are implicitly used. Dest0 must not be the same as any source registers, both explicit and implicit.

LEGAL REGISTER ARGUMENTS

?	DESTINATION				SOURCE				
PS version	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	

You cannot use swizzle or negate modifiers on Source1.

m3x3 r1, r3, c4 // c5 and c6 use implicit

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	

This macro is equivalent to the following code:

```
dp3 Dest0.x, Source0, Source1
dp3 Dest0.y, Source0, Source1+1
dp3 Dest0.z, Source0, Source1+2
```

m3? (macro)

ps 2.0

Perform a 3 ?4 matrix multiply on a 3 element vector.

Four arithmetic instruction slots.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	
m4x3 (macro)									ps 2.0

Perform a 3x4 matrix multiply on a 4 element vector.

Three arithmetic instruction slots.

```
m4x3 Dest0, Source0, Source1
```

Multiplies a 4 element vector by a 3x4 matrix. The result is stored in Dest0.xyz. The next two registers from Source1 are implicitly used. Dest0 must not be the same as any source registers, both explicit and implicit.

You cannot use swizzle or negate modifiers on Source1.

```
m4x3 r1, r3, c4 // c5 and c6 use implicit
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	

This macro is equivalent to the following code:

```
dp4 Dest0.x, Source0, Source1
dp4 Dest0.y, Source0, Source1+1
dp4 Dest0.z, Source0, Source1+2
```

m4x4(macro)

ps 2.0

Perform a 4x4 matrix multiply on a 4 element vector.

Four arithmetic instruction slots.

```
m4x4 Dest0, Source0, Source1
```

Multiplies a 4 element vector by a 4x4 matrix. The result is stored in Dest0.xyzw. The next three registers from Source1 are implicitly used. Dest0 must not be the same as any source registers, both explicit and implicit.

You cannot use swizzle or negate modifiers on Source1.

```
m4x3 r1, r3, c4 // c5, c6 and c7 use implicit
```

This macro is equivalent to the following code:

```
dp4 Dest0.x, Source0, Source1
dp4 Dest0.y, Source0, Source1+1
dp4 Dest0.z, Source0, Source1+2
dp4 Dest0.w, Source0, Source1+3
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	
mad	ps 1.0–2.0								

Multiply and add. Multiplies two registers, adds a third to the result, and then stores the result in the destination.

One arithmetic instruction slot.

```
mad Dest0, Source0, Source1, Source2
```

Multiplies Source0 by Source1, then adds Source2 to the result. The result is then stored in Dest0.

```
mad r0, r0, r1, r2
```

max(macro) ps 2.0

Stores the maximum value from comparing two source registers into the destination register on an element-by-element basis.

Two arithmetic instruction slots.

```
max Dest0, Source0, Source1
```

Places the maximum element from each source register in the corresponding element of the destination register.

Dest0 should not be the same as the source registers.

```
max r1, r2, c0
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0				x	x	x	x	x
1.1			x	x	x	x	x	x

LEGAL REGISTER ARGUMENTS

?	th>	DESTINATION				SOURCE			
PS version		v	c	t	r	v	c	t	r
1.2		?	?	x	x	x	x	x	x
1.3		?	?	x	x	x	x	x	x
1.4 phase 1		?	?	?	x	?	x	?	x
1.4 phase 2		?	?	?	x	x	x	?	x
2.0		?	?	?	x	x	x	x	x

LEGAL REGISTER ARGUMENTS

?	th>	DESTINATION				SOURCE				
PS version		v	c	t	r	v	c	t	r	s
1.0		?	?	?	?	?	?	?	?	?
1.1		?	?	?	?	?	?	?	?	?
1.2		?	?	?	?	?	?	?	?	?
1.3		?	?	?	?	?	?	?	?	?
1.4 phase 1		?	?	?	?	?	?	?	?	?
1.4 phase 2		?	?	?	?	?	?	?	?	?
2.0		?	?	?	x	x	x	x	x	?

This macro is equivalent to the following code:

```

add Dest0, Source0, -Source1
cmp Dest0, Dest0, Source0, Source1

```

min(macro)

ps 2.0

Stores the minimum value from comparing two source registers into the destination register on an element-by-element basis.

Two arithmetic instruction slots.

```
min Dest0, Source0, Source1
```

Places the minimum element from each source register in the corresponding element of the destination register.

Dest0 should not be the same as the source registers.

```
min r1, r2, c0
```

LEGAL REGISTER ARGUMENTS

?

DESTINATION

SOURCE

PS version

v

c

t

r

v

c

t

r

s

1.0

?

?

?

?

?

?

?

?

?

1.1

?

?

?

?

?

?

?

?

?

1.2

?

?

?

?

?

?

?

?

?

1.3

?

?

?

?

?

?

?

?

?

1.4 phase 1

?

?

?

?

?

?

?

?

?

1.4 phase 2

?

?

?

?

?

?

?

?

?

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
2.0				x	x	x	x	x	

This macro is equivalent to the following code:

```
add Dest0, Source0, -Source1
cmp Dest0, Dest0, Source1, Source0
```

mov

ps 1.0–2.0

Stores the source register into the destination register. The source and destination registers can be the same but then it might be considered `nop`.



In PS 2.0 the `mov` instruction is the only instruction that can be used to set the MRT or MET color elements or the `oDepth` register. The restrictions on using the MRT or MET elements as a destination are

- The color elements can only be written with an `.rgba` mask.
- You can write to an `oCn` or `oDepth` register only once per shader.
- You *must* write to the `oC0` register.
- No `_sat` or source modifiers, but `oDepth` must specify a replicate swizzle.
- You *cannot* specify a write mask when writing to `oCn`.

One arithmetic instruction slot.

```
mov Dest0, Source0
```

Moves Source0 into Dest0.

```
mov r0, r1
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0				x	x	x	x	x
1.1			x	x	x	x	x	x
1.2			x	x	x	x	x	x
1.3			x	x	x	x	x	x
1.4 phase 1				x		x		x
1.4 phase 2				x	x	x		x
2.0				x	x	x	x	x

mul ps 1.0–2.0

Multiplies the two source registers and stores them in the destination register.

One arithmetic instruction slot.

```
mul Dest0, Source0, Source1
```

Multiplies Source0 by Source1 component by component and stores the result in Dest0.

```
mul    r0, r1, r2
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0				x	x	x	x	x
1.1			x	x	x	x	x	x
1.2			x	x	x	x	x	x
1.3			x	x	x	x	x	x
1.4 phase 1				x		x		x
1.4 phase 2				x	x	x		x
2.0				x	x	x	x	x

nop ps 1.0–2.0

No operation. Just takes up a slot.

Takes no space or time.

```
_____

nop

_____
```

The nop instruction performs no operation. It's possible that it may get optimized to nothing.

```
nop
```

nrm(macro) ps 2.0

This macro will normalize all elements of a register.

Three arithmetic instruction slots.

```
nrm Dest0, Source0
```

This macro will take the `.xyz` elements of `Source0` and normalize them so that the square root of the sum of squares of all elements in `Dest0` is one. `Dest0` must have a `.xyz` or `.xyzw` write mask. If the full write mask is used `Source0.w` is scaled as well.

`Dest0` should not be the same register as `Source0`.

```
nrm r1, r2
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	

This macro is equivalent to the following code sans `Dest0` write mask.

```
dp3 Dest0.x, Source0, Source0
rsq Dest0.x, Dest0.x
mul Dest0, Source0, Dest0.x
```

phase

ps 1.4

A PS 1.4 pixel shader breaks a pixel shader up into two sections. This allows the pixel shader to increase the number of instructions.

Takes no space or time.

phase

There are no arguments for the `phase` instruction. Instructions that come in the shader before the `phase` instruction are considered phase one instructions. Any instructions that come after the `phase` instruction are considered phase two instructions. There can be only one `phase` instruction in a shader. Each phase can use up to six texture operations followed by up to eight arithmetic operations. The shader must still place the `ps` and `def` instructions at the top of the shader.

The `phase` instruction is a marker used to indicate the intention to perform a dependent texture read, more texture fetches, or more arithmetic instructions.

Note?

The alpha component of the temporary registers must be reinitialized after the phase command.

```
ps.1.4
def c1, 0.0, 0.5, 1.0, 2.0
```

```
// phase 1 instructions will be here

// texture then arithmetic

phase

// phase 2 instructions will be here

// texture then arithmetic
```

pow(macro)

ps 2.0

Computes the power function for a scalar value.

Three arithmetic instruction slots.

```
pow Dest0, Source0, Source1
```

Computes a scalar values raised to a power. Both source registers require a replicate swizzle to specify the element of each register to use. The Source0 element will be raised to the power of the Source1 element. The result is copied to all element of Dest0 unless a write mask is specified.

```
pow r1.x, r1.x, c1.y // r1.x = r1.x ^^ c1.y
```

This macro is equivalent to the following code sans Dest0 write mask.

mad

ps 1.0–2.0

```
log Dest0, Source0
```

```
mul Dest0, Dest0, Source1
```

```
exp Dest0, Dest0
```

The math used is $b^x = 2^{(x \cdot \log_2(b))}$.

ps

ps 1.0–2.0

Defines the version of the pixel shader code you are using.

Takes no space or time.

```
ps.integer1.integer2 // DirectX 8
```

```
ps_integer1_integer2 // DirectX 9
```

The argument is of the form ps.X.Y (for DirectX 8) or ps_X_Y (for DirectX 9), where ? is the main version number, and Y is the minor version number. Both values are integers.

```
ps.1.4 // DirectX 8
```

```
ps_2_0 // DirectX 9
```

rcp

ps 2.0

Computes the reciprocal of an element of the source register and stores it in the destination register.

One arithmetic instruction slot.

```
rcp Dest0, Source0
```

Computes the reciprocal of a single element of the source register and stores it in all elements of the destination register. Only one element of the source is used. A replicate swizzle must be used to select the element. A value of exactly 1 on input returns 1 on output (no round-off error) while a value of 0 on input returns INFINITY (or a really big positive number). Dest0 must be a temporary register.

Setup Source0 contains the element to take the reciprocal of, specified by a replicate swizzle.

Results All elements of Dest0 will contain the reciprocal of the specified element.

```
rcp r0, r5.x // r0 = 1/(r5.x)
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE				
	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	
rsq									ps 2.0

Computes the reciprocal square root of an element of the source register and stores it in the destination register.

One arithmetic instruction slot.

```
rsq Dest0, Source0
```

Computes the reciprocal square root of a single element of the source register and stores it in all elements of the destination register. Only one element of the source is used. A replicate swizzle must be used to select the element. The absolute value of the source is used. A value of exactly 1 on input returns 1 on output (no roundoff error) while a value of 0 on input returns `INFINITY` (or a really big positive number). Dest0 must be a temporary register.

Setup Source0 contains the element to take the reciprocal square root of, specified by a replicate swizzle.

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE				
PS version	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	

Results All elements of Dest0 will contain the reciprocal square root of the specified element.

```
rsq r0, r5.x // r0 = 1/sqrt (r5.x)
```

Computes the sine and cosine for a scalar value.

Eight arithmetic instruction slots.

sincos Dest0, Source0, Source1, Source2

Estimates the sine and cosine value inside a shader with a maximum error of 0.002 through the use of a Taylor series expansion. Source0 must have a replicate swizzle to indicate which element to use. This should be a value in radians between π . The sine and cosine values will be stored in Dest0.xy respectively. The destination must have .x, .y or .xy as a write mask. The destination shouldn't be the same as any source. You cannot use the _sat modifier. The Source1 and Source2 registers must be constant registers.

Setup One element of Source0 has to have the value in radians. Source1 and Source2 have to be set up with the following values to perform the expansion.

```
Source1 = [1/(7!*128),1/(6!*64),1/(4!*16),1/(5!*16)]
Source2 = [1/(3!*8),1/(2!*8),1,0.5]
```

```
sincos r0.xy, r1.x, c4, c5
```

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE				
PS version	v	c	t	r	v	c	t	r	s
1.0									
1.1									
1.2									
1.3									
1.4 phase 1									
1.4 phase 2									
2.0				x	x	x	x	x	

sub**ps 1.0–1.4**

Subtracts the two source registers and stores them in the destination register.

One-arithmetic instruction slot.

```
sub    Dest0,    Source0,    Source1
```

Subtracts Source1 from Source0 component by component and stores the result in Dest0.

```
sub    r0,    c0,    v2
```

tex**ps 1.0–1.3**

Sample the color from the current texture stage and place it in the designation register.

No texture slots for PS 1.0; one for PS 1.1–1.3.

```
tex    Dest0
```

Take the color from the current texture stage and place it in the destination register. The texture used is the one that's assigned to the current texture stage using `SetTexture()`. The texture coordinates are the output of the vertex shader `oTn` registers or the FFP.

The destination register must be a texture register. The texture sampled is associated with the number of the texture register used as the destination. For FFP, you can select the texture with the `D3DTSS_TEXCOORDINDEX` texture stage state flag.

LEGAL REGISTER ARGUMENTS

?	th>	DESTINATION				SOURCE			
PS version		v	c	t	r	v	c	t	r
1.0		?	?	?	x	x	x	x	x
1.1		?	?	x	x	x	x	x	x
1.2		?	?	x	x	x	x	x	x
1.3		?	?	x	x	x	x	x	x
1.4 phase 1		?	?	?	x	?	x	?	x
1.4 phase 2		?	?	?	x	x	x	?	x
2.0		?	?	x	x	x	x	x	?

LEGAL REGISTER ARGUMENTS

?	th>	DESTINATION			
PS version		v	c	t	r
1.0		?	?	x	?
1.1		?	?	x	?
1.2		?	?	x	?
1.3		?	?	x	?
1.4 phase 1		?	?	?	?
1.4 phase 2		?	?	?	?
2.0		?	?	?	?

Note? This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

tex r0

2D Bump/Environment mapping.

One texture instruction slot.

```
texbem    Dest0,    Source0
```

The texbem instruction uses the `.r` and `.g` channels of the source texture as for du and dv perturbation values. The perturbation data is transformed by the 2×2 bump environment-mapping matrix, and then added to the current stage's texture coordinates. Then the current stage's texture is sampled and the results put into the destination register. Note that du and dv are treated as signed values.

The destination register must be a texture register. The texture sampled is associated with the number of the texture register used as the destination. The destination register number must be greater than the source register number.

For PS 1.0 and 1.1 The `_bx2` input modifier cannot be used.

Note

This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE			
PS version	v	c	t	r	v	c	t	r
1.0			x				x	
1.1			x				x	
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								

LEGAL REGISTER ARGUMENTS

?	th>	DESTINATION				SOURCE			
PS version		v	c	t	r	v	c	t	r
2.0		?	?	?	?	?	?	?	?

Bug After using `texbem` or `texbeml`, the source register should be assumed to be corrupted and should not be read again without being reinitialized. Validation does not catch this.

```
texbem t1, t0
```

texbeml

ps 1.0–1.3

2D Bump/Environment mapping with luminance correction.

One texture instruction slot.

```
texbeml Dest0, Source0
```

The `texbem` instruction uses the `.r` and `.g` channels of the source texture as for du and dv perturbation values. The perturbation data is transformed by the 2×2 bump environment-mapping matrix, and then added to the current stage's texture coordinates. Then the current stage's texture is sampled, a luminance correction is added, and the results put into the destination register. Note that du and dv are treated as signed values.

LEGAL REGISTER ARGUMENTS

?	th>	DESTINATION				SOURCE			
PS version		v	c	t	r	v	c	t	r

1.0			x				x	
1.1			x				x	
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

The luminance value and the bias texture stage are used to apply a luminance correction, and then these corrected values are used to sample the current stage's texture.

The destination register must be a texture register. The texture sampled is associated with the number of the texture register used as the destination. The destination register number must be greater than the source register number.

For PS 1.0 and 1.1 The `_bx2` input modifier cannot be used.

Note

This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

Bug After using `texbem` or `texbeml`, the source register should be assumed to be corrupted and should not be read again without being reinitialized. The validator does not catch this.

```
texbeml t0, t1
```

texcoord

ps 1.0–1.3

Pass the texture coordinate values (u, v, w, 1) as color (r, g, b, 1).

One texture instruction slot.

```
texcoord Dest0
```

The `texcoord` instruction allows the programmer to pass texture coordinates into the shader as color values. No texture is sampled by this instruction. This means that, for example, if the vertex shader passed color values (or normals, etc.) in the texture coordinates registers (instead of texture coordinates), the values the `texcoord` instruction will pass in are the interpolated and perspective corrected values, just as if the values were texture coordinates. You might want to do this if you need higher precision values than the color registers provide. This allows any type of value (colors, normals, positions, texture coordinates) to get passed through the texture pipeline and be transformed by the same calculations that one would expect of texture coordinates.

If you are using the FFP, you'll need to use the `D3DTSS_TEXCOORDINDEX` texture stage state flag.

The values in the texture register contain `xyzw` values, but only the `xyz` values will be used. (Any missing texture coordinates will be set to 0. The `w` component will be set to 1.) The texture coordinate data is copied. The values are clamped to the range `[0,1]`.

Note

This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
texcoord t0
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION			
	v	c	t	r
1.0			x	
1.1			x	
1.2			x	
1.3			x	
1.4 phase 1				
1.4 phase 2				
2.0				
Texcrd	ps 1.4			

Passes the texture coordinate data from the source into the destination register as color data. Similar to the `texcoord` instruction.

One texture instruction slot.

```
texcrd Dest0, Source0
```

Dest0 must be a temporary register, and Source0 must be a texture register. Only texture coordinates are passed by this instruction; no texture is sampled. The texture coordinate data is interpreted as color data. Unlike the `texcoord` instruction, the values are not clamped to the [0,1] range. The source register can hold data in the `?MaxTextureRepeat` range, while the destination register can hold data in the `?MaxPixelShaderValue` range (which is probably smaller), so you should be careful about the size of the source data.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2								
1.3								
1.4 phase 1				x			x	
1.4 phase 2				x			x	
2.0								

Note You can pass only three coordinates with this instruction, as defined by the modifier, either `.xyz` or `.xyw`. If no mask is specified, the `.xyz` mask is assumed. The unused fourth channel (either `z` or `w`) will be undefined.

You can perform a perspective divide using the `_dw` modifier on the fetched texture coordinates, in which case only `.xy` will be valid.

Note The `D3DTTFF_PROJECTED` flag is ignored.

```
texcrd r2, t2.xyz
texcrd r2, t2.xyw
texcrd r2, t2_dw.xyw
```

texdepth

ps 1.4

Calculates the depth value to be used in the depth test for this pixel. Uses `r5` register.

One texture instruction slot.

```
texdepth Dest0
```

The `Dest0` register can only be the `r5` register, and this instruction can be used only in phase two. The instruction uses the value of (`r5.r` or `r5.g`) and uses that value as the pixel's depth value. The `.r` element is used as the `z` value, and the `.g` element as the `w` value. If `r5.g` is zero, the value of 1.0 will be used. Compare to the `texm3x2depth` instruction.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION			
	v	c	t	r
1.0				
1.1				
1.2				
1.3				

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION			
	v	c	t	r
1.4 phase 1				x
1.4 phase 2				
2.0				

Note

After using this instruction, you can't use the `r5` register in the remainder of the pixel shader.

```
texdepth r5
```

texdp3

ps 1.0, 1.3

Calculates a dot product using the (`r`, `g`, `b`) texture data in the source and destination texture coordinate data (`u`, `v`, `w`), and then stores the scalar result in the destination register.

One texture instruction slot.

```
texdp3 Dest0, Source0
```

The `texdp3` takes the color data found in the texture specified by `Source0`'s texture and performs a dot product on the texture coordinate data specified by the texture associated with `Dest0`'s register number. The result is stored in all elements of `Dest0`. The register number of the `Dest0` register must be greater than the `Source0` texture register number. Assumes that a texture has already been loaded into `Source0`. The output is clamped to the range `[0,1]`.

This instruction could be used to get a higher precision `dp3` since it's performed by a texture-addressing instruction, then a color instruction.

Note This instruction *cannot* be used in PS 1.0, 1.1, 1.4, or 2.0 shaders.

```
texdp3 t1, t0
```

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE			
PS version	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

texdp3tex ps 1.2, 1.3

Calculates a dot product using the (r, g, b) texture data in the source and destination texture coordinate data (u, v, w), and then uses the scalar result to sample the texture associated with the destination register. Useful as a texture lookup instruction when you store nontexture data in a texture.

One texture instruction slot.

```
texdp3tex Dest0, Source0
```

The `texdp3tex` takes the color data found in the texture specified by Source0's texture and performs a dot product on the texture coordinate data specified by the texture associated with Dest0's register number. The scalar result is then used to sample the texture data in the texture associated with Dest0. The result is stored in Dest0. The register number of the Dest0 register must be greater than the Source0 texture register number. Assumes that a texture has already been loaded into Source0.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

You can use this instruction so that you can use a 1D texture as a lookup table. If you are using `texm3x3*`, you might consider using this instruction instead since it consumes fewer clocks.

Note This instruction *cannot* be used in PS 1.0, 1.1, 1.4, or 2.0 shaders.

```
texdp3tex t1, t0
```

texkill ps 1.0–2.0

Terminates processing of the pixel if any of the values of the `.xyz` (for PS 1.0–1.4) or `.xyzw` (for PS 2.0) texture coordinates or registers are less than 0. Used to simulate clip planes, cheesy transparency, etc.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION			
	v	c	t	r
1.0			x	
1.1			x	
1.2			x	
1.3			x	
1.4 phase 1				
1.4 phase 2			x	x
2.0			x	x

One texture instruction slot.

```
texkill Source0
```

This instruction is used to instantly terminate processing of a pixel. No texture is sampled, but it uses the texture coordinates of the source register number (PS 1.0–1.3) or the data in the texture source register or the temporary register (PS 1.4). If in PS 1.4, it must be in phase two. If there are fewer than three texture coordinates, only those provided are used in the test. The `_sat` and `_pp` modifiers are not allowed.

Note If multisampling is enabled, you won't get an antialiasing effect along any `texkill`-generated edge.

```
texkill t0
```

texld ps 1.4–2.0

Sample the texture and load the destination register with color data using the texture coordinates provided.

One texture instruction slot.

```
texld Dest0, Source0           // PS 1.4
texld Dest0, Source0, Source1 // PS 2.0
```

For the PS 1.4 version This instruction samples the texture associated with the destination register using texture coordinates provided in the source register. The destination register must be a temporary register. If used in phase one, the source register must be a texture register; if in phase two, it can be a texture or temporary register. Texture registers produce a nondependent read, whereas temporary registers produce a dependent read. If you use a temporary as a source register, the *x*, *y*, and *z* elements must have been initialized in phase one.

For the PS 2.0 version Removed dependency on destination register. The Source 1 register must be a sampling register, where the register number identifies the texture to sample. Source0 provides the texture coordinates. The number of texture coordinates required to sample depends upon the declaration of the sampling register. For example, a cube map will require three texture coordinates to be provided. Signed textures will provide values in the $[-1, 1]$ range, while unsigned textures will provide them in the $[0, 1]$ range. Sampling a texture with lower dimensions than provided is allowed, but it's a run-time error to attempt to sample from a texture that requires more texture coordinates than are provided. Also see the `texldb` and `texldp` instructions. You can't use the `_sat` modifier or a write mask on Dest0. You can't use negate or swizzles on the source registers.

Note?

The `D3DTEXTUREF_PROJECTED` flag is ignored. For PS 1.4 you can use `.xyz` or `.zyw` modifiers on the texture registers as long as the same modifier is used in both phases. You can mix the `.xyw` modifier with the `_dw` modifier. You can use the `_dz` modifier only on a temporary register, but not more than twice per shader.

LEGAL REGISTER ARGUMENTS

[illegible]

LEGAL REGISTER ARGUMENTS

?	t	>	DESTINATION				SOURCE0				SOURCE1			
PS version	v	c	t	r	v	c	t	r	s	v	c	t	r	s
1.2														
1.3														
1.4 phase 1				x			x					x		
1.4 phase 2				x			x	x				x	x	
2.0				x			x	x						x

```
// PS 1.4
texld r4, t0
texld r4, t0.xyw
texld r4, t0_dw.xyw

// PS 2.0
texld r4, t0.xy, s3
```

texldb

ps 2.0

Sample the texture and load the destination register with color data using the texture coordinates provided, biasing the mipmap level before sampling the texture.

LEGAL REGISTER ARGUMENTS

?	t	>	DESTINATION				SOURCE0				SOURCE1			
PS version	v	c	t	r	v	c	t	r	s	v	c	t	r	s

1.0	?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td>
1.1	?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td>
1.2	?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td>
1.3	?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td>
1.4 phase 1	?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td>
1.4 phase 2	?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td> ?/td>
2.0	?/td> ?/td> ?/td> x ?/td> ?/td> x x ?/td> ?/td> ?/td> ?/td> ?/td> x

One texture instruction slot.

```
texldb Dest0, Source0, Source1
```

This instruction is a modification of the `texld` instruction, and all behavior and restrictions that apply to `texld` apply to this instruction as well. Also see the `texld` and `texldp` instructions.

In addition to sampling a texture as `texld` does, the `texldb` instruction uses the `Source0.w` element to bias the mipmap level. A positive value will bias towards smaller mipmaps, while a negative value will bias towards larger mipmaps. The acceptable range for the bias is $[-3, +3]$. Values outside this range will produce undefined results.

The `D3DSAMP_MAXMIPLEVEL` flag is valid and will be added to the bias specified here before sampling occurs.

```
texldb r4, t0, s2
```

texldb

ps 2.0

Sample the texture and load the destination register with color data using the texture coordinates provided, performing a perspective projection on the texture coordinates before sampling the texture.

One texture instruction slot.

```
texldp Dest0, Source0, Source1
```

This instruction is a modification of the `texld` instruction, and all behavior and restrictions that apply to `texld` apply to this instruction as well. Also see the `texld` and `texldb` instructions.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE0					SOURCE1				
	v	c	t	r	v	c	t	r	s	v	c	t	r	s
1.0														
1.1														
1.2														
1.3														
1.4 phase 1														
1.4 phase 2														
2.0				x			x	x						x

A perspective projection is performed on the texture coordinates prior to sampling. The texture coordinates are divided by `Source0.w`, then these projected coordinates are used for the sampling. `Source0.w` should never be zero.

The `D3DTTFF_PROJECTED` flag is not valid for PS 2.0

```
texldp r4, t0, s2
```

You can perform your own projection in the following code:

```
// scale texture coordinates
rcp Dest0.w, Source0.w
```

```
mul Dest0, Source0, Dest0.w

// you could warp coordinates here if desired

// sample texture
texld Dest0, Dest0, Source1
```

texm3x2depth

ps 1.3

Calculates the *z* and *w* depth values of the pixel. Used after setup by `texm3x2pad`. You might use this instruction for generating *z* value for sprites.

One texture instruction slot.

```
texm3x2depth Dest0, Source0
```

Assumes that `texm3x2pad` instruction was used to load a texture and set up a vector. This will be used as the depth *z* value. This instruction is used to calculate the depth *w* value. If the *w* value is zero, then the destination has a value of one stored in it. If the *w* value is not zero, then the destination has the value *z/w* stored in it. The result is not automatically clamped to the [0,1] range. The result is used as the depth value for the pixel, ignoring the existing pixel depth value.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2								
1.3			x				x	

LEGAL REGISTER ARGUMENTS

?	th>	DESTINATION				SOURCE			
PS version		v	c	t	r	v	c	t	r
1.4 phase 1		?	?	?	?	?	?	?	?
1.4 phase 2		?	?	?	?	?	?	?	?
2.0		?	?	?	?	?	?	?	?

Note?

This instruction *cannot* be used in PS 1.0, 1.1, 1.2, 1.4, or 2.0 shaders.

```
texm3x2depth t1, t0
```

texm3x2pad

ps 1.0–1.3

Used with other pixel shader texture operations to perform 3 ?2 matrix multiplies.

One texture instruction slot.

```
texm3x2pad Dest0, Source0
```

This instruction is used to represent stages where only the texture coordinate is used. This instruction cannot be used by itself but is the setup instruction for a variety of other texture coordinate manipulating instructions. These corresponding stages have no textures bound, and no sampling will occur. The input argument, Source0, should still be specified.

Note?

This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
texm3x2pad t1, t0
```

LEGAL REGISTER ARGUMENTS

?		DESTINATION				SOURCE			
PS version		v	c	t	r	v	c	t	r
1.0				x				x	
1.1				x				x	
1.2				x				x	
1.3				x				x	
1.4 phase 1									
1.4 phase 2									
2.0									
texm3x2tex		ps 1.0–1.3							

Used with other pixel shader texture operations to perform 3 ?2 matrix multiplies. And does a texture lookup.

One texture instruction slot.

```
texm3x2tex Dest0, Source0
```

The `texm3x2tex` instruction is used with the `texm3x2pad` instruction. It performs a second matrix multiply and then samples the texture associated with `Dest0`.

Note?

This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
texm3x2tex t1, t0
```

?		DESTINATION				SOURCE			
PS version		v	c	t	r	v	c	t	r

PS version	v	c	t	r	v	c	t	r
1.0			x				x	
1.1			x				x	
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

texm3x3

ps 1.2, 1.3

Used with other pixel shader instructions. Used to perform the final stage of a 3x3 matrix multiply.

One texture instruction slot.

```
texm3x3 Dest0, Source0
```

This instruction must be used with the `texm3x3pad` instruction, which is used to set the first and second row of a 3x3 matrix multiply. The `texm3x3` instruction will perform the third row multiply of the matrix multiply and store the three-element result in Dest0. The `ra` value of the destination is set to 1. Any textures associated with the registers are ignored. The register number associated with Dest0 must be higher than the register number associated with Source0. It assumes the source register has been loaded in some way. You can use this as a higher precision per pixel matrix transform.

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE			
PS version	v	c	t	r	v	c	t	r
1.0								
1.1								

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

Note

This instruction *cannot* be used in PS 1.0, 1.1, 1.4 or 2.0 shaders.

```
texm3x3 t1, t0
```

texm3x3pad

ps 1.0–1.3

Used with other pixel shader texture operations to perform 3x3 matrix multiplies. This instruction is used as a set-up for a 3x3 multiply.

One texture instruction slot.

```
texm3x3pad Dest0, Source0
```

This instruction is used as the first and second part of a 3x3 matrix multiply operation performed in a pixel shader's texture declaration stage. It's used to perform the first and second row multiplies (with two instructions). Texture coordinates corresponding to the declared texture are used as a row of the matrix. No texture should be bound at this stage.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0			x				x	
1.1			x				x	
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

This instruction is used as a setup for the `texm3x3`, `texm3x3spec`, `texm3x3tex`, or `texm3x3vspec` instruction.

Note This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
texm3x3pad t1, t0
```

texm3x3spec

ps 1.0–1.3

Performs specular reflection and environment mapping assuming a constant view direction. Used after setup by `texm3x3pad`.

One texture instruction slot.

```
texm3x3spec Dest0, Source0, Source1
```

This operation performs the final row multiplication of a 3 × 3 matrix multiply. It then takes the resulting vector and uses it as a normal vector. The value in Source1 is assumed to be a vector representing the eye direction. This vector is used with the normal vector to create a reflection vector, which is then used as the index for a texture lookup from the texture associated with the destination register. The result is stored in the destination register.

Both Dest0 and Source0 must be texture registers, while Source1 must be a constant register. No texture colors are sampled from the preceding two stages—that is, from the texture associated with the register numbers used by the two preceding `texm3x3pad` instructions that are required as setup for this instruction. The register number associated with Dest0 must be higher than the register number associated with Source0. The register number associated with Source0 must be higher than the register number associated with Source1. Compare with the `texm3x3vspec` instruction.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE0				SOURCE1			
	v	c	t	r	v	c	t	r	v	c	t	r
1.0			x				x			x		
1.1			x				x			x		
1.2			x				x			x		
1.3			x				x			x		
1.4 phase 1												
1.4 phase 2												
2.0												

Note This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
texm3x3spec t1, t0, c3
```

texm3x3tex

ps 1.0–1.3

Used with other pixel shader texture operations to perform 3x3 matrix multiply followed by a texture lookup. Used after setup by `texm3x3pad`.

One texture instruction slot.

texm3x3tex Dest0, Source0

The `texm3x3tex` instruction is used as the final of three instructions representing a 3x3 matrix multiply operation performed in a pixel shader's texture declaration. After the matrix multiplication, the resulting values are used as the (u, v, w) in a texture lookup into the texture associated with Dest0.

Both Dest0 and Source0 must be texture registers. No texture colors are sampled from the preceding two stages, and any texture associated with them is not sampled. The register number associated with Dest0 must be higher than the register number associated with Source0.

This instruction is typically used for transforming a normal vector into the correct tangent space and using it for look up.

Note This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
tex3x3tex t1, t0
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0			x				x	
1.1			x				x	
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

texm3x3vspec

ps 1.0–1.3

Performs specular reflection and environment mapping assuming a nonconstant view direction (i.e., a local viewer). Used after setup by `texm3x3pad`.

One texture instruction slot.

```
texm3x3vspec Dest0, Source0
```

This operation performs the final row multiplication of a 3x3 matrix multiply. It then takes the resulting vector and uses it as a normal vector. The view direction is taken from the w components of the three sets of texture coordinates used as rows of the matrix. This vector is used with the normal vector to create a reflection vector, which is then used as the index for a texture lookup from the texture associated with the destination register. The result is stored in the destination register.

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE			
PS version	v	c	t	r	v	c	t	r
1.0			x				x	
1.1			x				x	
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

You would use this instruction for a surface where you need specular highlights to vary with the eye direction. Both Dest0 and Source0 must be texture registers. No texture colors are sampled from the preceding two stages, and any texture associated with them is not sampled. The register number associated with Dest0 must be higher than the register number associated with Source0. Compare to the texm3x3spec instruction.

Note This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
texm3x3vspec t1, t0
```


Samples the texture using the (a, r) texture coordinates of the source register as texture address data (u, v) at the stage corresponding to the destination register number, with the result being stored in the destination register.

One texture instruction slot.

```
texreg2ar Dest0, Source0
```

This instruction is used when you want to remap texture coordinates. The input color vector's alpha value is used as a u , and the red value is used as a v ; the selected texture is sampled at these coordinates, and the value placed in the destination register. Both Source0 and Dest0 must be texture registers, and the actual register number of the Dest0 register must be greater than the Source0 texture register number. This assumes that you have already loaded a texture into Source0 and that values in Source0 are positive data.

Note This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
texreg2ar t1, t0
texreg2ar t0, t1 // Error! Source0 > Dest0
```

LEGAL REGISTER ARGUMENTS

	DESTINATION				SOURCE			
PS version	v	c	t	r	v	c	t	r
1.0			x				x	
1.1			x				x	
1.2			x				x	
1.3			x				x	

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.4 phase 1								
1.4 phase 2								
2.0								

texreg2gb

ps 1.0–1.3

Samples the texture using the (*g*, *b*) texture coordinates of the source register as texture address data (*u*, *v*) at the stage corresponding to the destination register number, with the result being stored in the destination register.

One texture instruction slot.

```
texreg2gb Dest0, Source0
```

This instruction is used when you want to remap texture coordinates. The input color vector's green value is used as a *u*, and the blue value is used as a *v*; the selected texture is sampled at these coordinates, and the value placed in the destination register. Both Source0 and Dest0 must be texture registers, and the actual register number of the Dest0 register must be greater than the Source0 texture register number. This assumes that you have already loaded a texture into Source0 and that values in Source0 are positive data.

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0			x				x	
1.1			x				x	
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

Note

This instruction *cannot* be used in PS 1.4 or 2.0 shaders.

```
texreg2gb t1, t0
texreg2gb t0, t1 // Error! Source0 > Dest0
```

texreg2rgb

ps 1.2, 1.3

Samples the texture using the (r, g, b) texture coordinates of the source register as texture address data at the stage corresponding to the destination register number, with the result being stored in the destination register.

One texture instruction slot.

```
texreg2rgb Dest0, Source0
```

LEGAL REGISTER ARGUMENTS

PS version	DESTINATION				SOURCE			
	v	c	t	r	v	c	t	r
1.0								
1.1								
1.2			x				x	
1.3			x				x	
1.4 phase 1								
1.4 phase 2								
2.0								

This instruction is used when you want to remap texture coordinates. It works the same as `texreg2ar` and `texreg2gb` but adds support for 3D volumetric textures. The texture can be a 3D volumetric texture or a cube map. Both Source0 and Dest0 must be texture registers, and the actual register number of the Dest0 register must be greater than the Source0 texture register number. This assumes that you have already loaded a texture into Source0 and that values in Source0 are positive data.

Note

This instruction *cannot* be used in PS 1.0, 1.1, 1.4, or 2.0 shaders.

```
texreg2rgb t1, t0
```

OUTPUT REGISTER MASKS, ARGUMENT, AND INSTRUCTION MODIFIERS

In order to give you more control over how an individual register or instruction is used, you have an array of masks, selectors, and modifiers to manipulate exactly how an instruction works and what register channels are used or written to.

source negation

ps 1.0–2.0

Negation can be used to negate an entire source register before it is used. Source negation is indicated by placing a minus sign in front of the source register to be negated. The source register values are unchanged.

Rules for using source negation:

- For use only with arithmetic instructions.

- Cannot be used with the invert modifier.
- Performed after other modifiers.

```
mov    t0, -v0          // t0 = -1.0 * v0
mul    t0, -v0, -c3
add    r0,  v0, -v1
mul    r1, 1-v0, -v1    // with invert modifier
```

source invert

ps 1.0–1.4

Subtracts all elements of a register from 1 and uses that as its output. Source negation is indicated by placing a 1- (number 1 followed by a minus sign) in front of the source register to be inverted. The source register values are unchanged.

Rules for using source invert:

- For use only with arithmetic instructions.
- Cannot be combined with any other register modifier.
- It can be combined only with the alpha-replicate modifier.

```
mov    r0, 1-v0        // swaps colors
mul    r1, 1-v0, -v1
```

source bias

ps 1.0–1.4

The bias modifier is used for shifting the range of the input register from the [0,1] range to the [−0.5, +0.5] range. The bias modifier is indicated by adding a `_bias` suffix to a register. Essentially, the modifier subtracts 0.5 from the register's values before they are used. Be careful when using this modifier with the color registers, since the range of the color registers is [0,1], and you'll get an implicit clamping. The source register values are unchanged.

If you use it with a `mov_2X` instruction or modifier, you can convert a register range from [0,1] to [−1,1], the same as a source signed scaling modifier.

Note

If you used the `D3DTOP_ADDSIGNED` texture operation in one of your DirectX texture stages, the bias modifier performs the same operation.

Rules for using source bias:

- For use only with arithmetic instructions.

- Cannot be combined with the invert modifier.
- Initial data outside the [0,1] range may produce undefined results.

```
// Shift range from [0,1] to [-0.5, 0.5]
mov      r0, r0_bias    // r0 = r0 - 0.5

// Shift range from [0,1] to [-0.5, 0.5]
// Then shift sign
mov      r0, -r0_bias    // r0 = 0.5 - r0

// shift range from [0,1] to [-1,1]
mov_x2   r0,    r0_bias
```

source signed scaling

ps 1.0–1.4

The signed scaling modifier (also called bias times two) is used for shifting the range of the input register from the [0,1] range to the [-1,+1] range, typically when you want to use the full signed range of which registers are capable. The bias modifier is indicated by adding a `_bx2` suffix to a register. Essentially, the modifier subtracts 0.5 from the register's values and then multiplies that result by 2 before they are used. The source register values are unchanged.

For PS 1.0 and 1.1, arguments for the `texm3x2*` and `texm3x3*` instructions can use the `bx2` modifier. For PS 1.2 and 1.3, arguments for any `tex*` instruction can use the `_bx2` modifier.

Note

If you used the `D3DTOP_ADDSIGNED2X` texture operation in one of your DirectX texture stages, the signed scaling modifier performs the same operation.

Rules for using signed source scaling:

- For use only with arithmetic instructions.
- Cannot be combined with the invert modifier.
- Initial data outside the [0,1] range may produce undefined results.

```
mov      t0, t0_bx2      // t0 = 2.0* (t0 - 0.5)
mov      r0, r0_bx2      // darken dull colors
```



The scale by two modifier is used for shifting the range of the input register from the $[0,1]$ range to the $[-1,+1]$ range, typically when you want to use the full signed range of which registers are capable. The scale by two modifier is indicated by adding a `_x2` suffix to a register. Essentially, the modifier multiplies the register values by 2 before they are used. The source register values are unchanged.

Rules for using scale by two:

- For use only with arithmetic instructions.
- Cannot be combined with the invert modifier.
- Available for PS 1.4 shaders only.

```
mov    r0, r0_x2 // 2x r0
```

source replication/selection

Just as vertex shaders let you select the particular elements of a source register to use, so pixel shaders do, with some differences. You can select only a single element, and that element will be replicated to all channels. You specify a channel to replicate by adding `.n` suffix to the register, where *n* is *r*, *g*, *b*, or *a* (or *x*, *y*, *z*, or *w*).

```
mov    r0,    v0.a    // ps.1.0
mov    r0.a,   v0.b    // ps.1.1 ps.1.2 ps.1.3

// these commands are an error if not ps.1.4
mov    r0,    v0.b    // same as v0.bbbb
mov    r0,    v0.g    // same as v0.gggg
mov    r0,    v0.brgn  // ps 2.0
```

SOURCE REGISTER SELECTORS

REGISTER SWIZZLE

PS version	.rrrr	.gggg	.bbbb	.aaaa	.gbra	.brga	.abgr
1.0				x			
1.1			x	x			
1.2			x	x			
1.3			x	x			
1.4 phase 1	x	x	x	x			
1.4 phase 2	x	x	x	x			
2.0	x	x	x	x	x	x	x

texture register modifiers

ps 1.4 only



PS 1.4 has its own set of modifiers for texture instructions. Since only the `texcrd` and `texld` instructions are used to load or sample textures with PS 1.4, these modifiers are unique to those instructions. Note that you can interchange `rgba` syntax with `xyzw` syntax, thus `_dz` is the same as `_db`.

Source Texture Register Selectors

These allow you to swizzle the source register to a limited extent. The syntax is added as a suffix on the register. They can be used anytime `texcrd` or `texld` can be used. Since the instructions will read only three components, these selectors allow you to fill the register's last two channels with either the `.z` value or the `.w` value instead of leaving it uninitialized. You can mix the `.xyw` selector with the `_dw` modifier. You can use the `_dz` modifier only on a temporary register, but not more than twice per shader. This allows you to map a 4D texture into 3D texture space so it can be manipulated in the shader.

```
texld    r0, t0.xyz      // r0.xyzw - t0.xyzz
texld    r0, t0.rgb      // alternate syntax
texld    r0, r0_dz.xyz   // with a register modifier
```

PS 1.4 SOURCE REGISTER SELECTORS

DESCRIPTION

SYNTAX

PS 1.4 SOURCE REGISTER SELECTORS

DESCRIPTION

Source register looks like `.xyzz`

Source register looks like `.xyww`

SYNTAX

`.xyz`

`.xyw`

Once you use a particular selector on a texture register, you cannot use a different one on the same source register in the same shader. For example, the following is a legal set of instructions; register `t2` is used with the `.xyz` selector twice:

```
texld    r0,    t2.xyz
texld    r1,    t2.xyz
```

However, the following, which uses register `t2` with the `.xyz` selector and then the `.xyw` selector is in error:

```
texld    r0,    t2.xyz
texld    r1,    t2.xyw    // Error register t2
// used again but with different selector.
```

Source Texture Register Modifiers

These modifiers allow you to do a perspective divide (either by the `.z` or the `.w` element) in the pixel shader. The syntax is added as a suffix on the register. They can be used anytime `texcrd` or `texld` can be used. Only the `.xy` channel of the destination will be modified. If the divisor is zero, then the destination is set to 1. The `_dw` modifier is for Phase 1; the `_dz` modifier is for Phase 2.

PS 1.4 SOURCE REGISTER MODIFIERS

DESCRIPTION

Divide x,y by z

Divide x,y by w

SYNTAX

`_dz`

`_dw`

```

texld r0, t0_dz

// these are the same as above

texld r0, t0_dz.xyz
texld r0, t0_db.xyz
texld r0, t0&_db.rgb

```

You can mix the `.xyzw` selector with the `_dw` modifier. The `_dw` modifier can be used as many times as necessary in Phase 1. After Phase 1, the `.w` channel is invalid, thus you can't use the modifier. You can use the `_dz` modifier only on a temporary register (thus, only in Phase 2), and not more than twice per shader. The following shows what phase an instruction would be valid or invalid for (I've ignored usage restrictions on texture register, etc.):

```

// Phase 1
texld r0, r0_dz // Invalid - dz Phase 2 only
texld r0, r0_dw // Valid phase

// Phase 2
texld r0, r1_dz.xyz // Invalid - text register
texld r0, r1_db.xyz // Invalid - _db == _dz
texld r0, r0_dz.xyz // Valid - temp register
texld r0, r0_dw.xyz // Invalid - w is undefined

```

Destination Write Masks

These write masks control which channel(s) are written to. They can be used anytime `texcrd` or `texld` can be used. No mask is the same as specifying all. Only the combinations shown in the table can be used.

PS 1.4 DESTINATION WRITE MASKS

DESCRIPTION

Writes to the xyzw channels

Writes to the xyz channels

Writes to the xy channels

SYNTAX

xyzw

xyz

xy

```
mov r0.xy,    t0

mov r0.rg,    t0 // same as previous

mov r0.xyzw,  t0

mov r0,       t0 // same as previous
```

destination write mask

ps 1.0–2.0

Note the word *destination*. Masks can be used only to select which elements of a register are to be written to. Unlike vertex shaders, however, all you can do is select all channels (`.rgba`), color channels only (`.rgb`), or the alpha channel (`.a`)—though later pixel shaders allow more control. This mimics the traditional lighting pipeline in which you can have color and alpha channels processed separately. Omitting a mask is the same as specifying the full mask. The alpha mask is also referred to as the scalar mask since it uses a scalar value. The color write mask is sometimes referred to as the vector mask. An alternative syntax is to use `.xyzw` instead of `.rgba`.

Destination write masks are supported only for arithmetic instructions, with the exception of the `texcrd` and `texld` instructions. The `dp3` instruction can use only `.rgb` or `.rgba` masks for PS 1.0–1.3.

Destination masks are particularly important when you start getting set up for instruction pairing.

Note that with PS 1.4 shaders you have the ability to operate on individual channels, giving you a lot more flexibility.

DESTINATION WRITE MASK DESCRIPTIONS

MASK

OPERATION

DESTINATION WRITE MASK DESCRIPTIONS

MASK	OPERATION
.rgb	The operation works on the color channel (rgb) and is scheduled for execution in the vector pipeline.
.a	The operation works on the alpha channel and is scheduled for execution in the scalar pipeline.
.r, .g, .b	Let's you select the destination channel to write to.
.rgba	The operation works on the color and alpha channel, and is scheduled for parallel execution in the vector and scalar pipelines. This is the default if a mask is not specified.
.(r)(g)(b)(a)	Arbitrary mask. Must be listed in .rgba order but can use any of the masks.

DESTINATION WRITE MASK SELECTORS

PS version	SELECTOR						
	r	g	b	a	rgb	rgba	(r)(g)(b)(a)
1.0				x	x	x	
1.1				x	x	x	
1.2				x	x	x	
1.3				x	x	x	
1.4 phase 1	x	x	x	x	x	x	x
1.4 phase 2	x	x	x	x	x	x	x
2.0	x	x		x	x	x	x

Here are some examples of using the write mask.

```
// color channel is modulated
mul r0.rgb, t0, v0

// alpha is added using a different source register
add r0.a,    t1, v1
```

```
//
mul r0.rgb, t0, v0
+add r0.a, t0, v0 // note instruction pairing

// variations that have the same effect
// no masks is equivalent to
mul r0, t0, v0
mul r0.rgba, t0, v0 // full specification
```

Note that specifying exactly the same operation on the color and alpha channel (including registers) will automatically cause pairing to occur. The following code fragments cause the same code to be assembled in the pixel shader:

```
// no masks, a single operation
mul r0, t0, v0
```

This is the same as writing

```
// full mask with a single operation
mul r0.rgba, t0, v0
```

This is the same as writing

```
// color and alpha mask with the same operation
mul r0.rgba, t0, v0 // on color
mul r0.a, t0, v0 // on alpha, same arguments
```

except it takes up an extra slot and will run slower. However, you can rewrite it as

```
// color and alpha mask with the same operation
```

```
// with pairing (DirectX 8 only!)
mul r0.rgb, t0, v0 // on color
+mul r0.a, t0, v0 // on alpha, same arguments
```

And now you've paired the instructions since you've freed one slot and reduced the run time. The point being that now you can change the alpha manipulations and perform something different in the scalar (alpha) pipe.

instruction modifiers

Note that these are placed on the actual *instructions*, not the arguments. The pixel shader assembler support shift and scale modifier flags, as well as a saturation modifier flag that affects the generated output result. The modifiers can be thought of as shift left (power-of-two multiply), shift right (power-of-two divide), and saturate (clamp output range to [0,1]).

Rules for using instruction modifiers:

- For use only with arithmetic instructions.
- The `_sat` can suffix any other instruction modifier.

INSTRUCTION MODIFIERS DESCRIPTION

MODIFIER	OPERATION
<code>_2x</code>	2× modifier. Multiply the results by 2 before storing in the register.
<code>_4x</code>	4× modifier. Multiply the results by 4 before storing in the register.
<code>_8x</code>	8× modifier. Multiply the results by 8 before storing in the register.
<code>_d2</code>	Half modifier. Divide the results by 2 before storing in the register.
<code>_d4</code>	Quarter modifier. Divide the results by 4 before storing in the register.
<code>_d8</code>	Eighth modifier. Divide the results by 8 before storing in the register.
<code>_sat</code>	Saturation modifier. Clamps the results to the range [0,1] before storing.
<code>_pp</code>	Partial precision hint.

INSTRUCTION MODIFIERS USAGE

	MODIFIER							
PS version	<code>_x2</code>	<code>_x4</code>	<code>_x8</code>	<code>_d2</code>	<code>_d4</code>	<code>_d8</code>	<code>_sat</code>	<code>_pp</code>

1.0	x	x		x			x
1.1	x	x		x			x
1.2	x	x		x			x
1.3	x	x		x			x
1.4 phase 1	x	x	x	x	x	x	x
1.4 phase 2	x	x	x	x	x	x	x
2.0							x
							x

Here are some examples of using instruction modifiers.

```
add_x2      r0, v1, v1
add_d2      r0, v1, v0
add_sat     r0, v1, v0
add_x2_sat  r0, v1, v1
add_d2_sat  r0, v1, v1
add_sat_d2  r0, v1, v1 // Error! _sat must be last
```

partial precision declaration modifier

ps 2.0



DirectX 9 introduced the partial precision declaration modifier for texture coordinate register usage. This modifier allows the shader writer to provide a *hint* that the operations on this texture coordinate register can be performed and stored at a lower precision (at least 16 bits). The implementation may ignore this hint. If applied the implementation could possibly propagate this lower precision through the shader.

Here is an example of using the partial precision modifier.

```
dcl_pp t2 // use t2 in lower precision
```

instruction pairing

DirectX 8 only

PS 2.0 removed the need for instruction pairing, but it's valid in PS 1.0–1.4.

You might see documentation talking about the scalar pipeline or the vector pipeline. This refers to the pipeline that corresponds to the alpha (scalar) or color (vector) hardware path. Since you typically want to process the alpha channel in a different manner from the color channels, there are (supposedly) separate, parallel, vector, and scalar hardware paths on the graphics processor. One is for vector processing (color) and one is for scalar processing (alpha). Since the pixel shader assembler can't be assumed to be that smart, the assembler needs help in being told when to pair color operations with alpha operations.

In order to use instruction pairing, you first break up the operations to be performed on the color and alpha channels (assuming of course that they *are* different) using the output masks. You then place a plus sign in front of the second instruction of the pair. Generally, you want to take a look at your pixel shader code and see which instruction that operates only on the color channels can be paired with a potentially independent instruction that operates on the alpha channel.

Note

For PS 1.0, the destination register for paired instructions must be the same. For all other versions, the destination register can be different for the coissued instruction. The `dp4` and `bem` instructions can't be co-issued.

The following example demonstrates instruction pairing. We have an operation on the color channels of texture zero. At the same time, we want to add in the alpha channel from texture one. We can get these operations to happen at the same time by instruction pairing.

```
// an example of instruction pairing

// an RGB (vector) operation
mul r0.rgb, t0, v0 // note write mask

// now co-issue an alpha operation
+add r0.a, t1, v0 // note alpha mask and plus sign
```

The reason that it's worthwhile to do this is that pairing instructions allows the pixel shader to execute the operations in parallel, thus reducing the number of clocks required and achieving better graphics processor utilization. Pairing also increases the number of slots that is available since each pair of instructions takes only one slot.

Note that you can pair the vector with the scalar or the scalar with the vector. As long as one operates on the color channels, while the other works on the alpha channel and one immediately follows the other, they can be in either order.

```
// the colors are modulated
mul r0.rgb, t0, v0
// alpha is added
+ add r0.a, t0, v0
```

The `dp3` instruction is a special case. When used, it can be paired with an instruction that is operating on the alpha component of its destination register. The `dp3` uses the `.rgb` elements so that you can pair it with an instruction that uses just the alpha pipe.

```
// ps.1.0
// two unrelated operations can get paired
dp3 r0.rgb, t0, v0
+add r0.a, t0, v0 // note same register is Dest0

// another example dp3 r0.rgb, t0, v0
mov r1.a, v1.a // note different destination for ps1.1+
```

The output masks can affect how the two pipelines are allocated, but there can be ambiguities in the order of operations unless explicit pairing syntax is used.

References

- [ABRASH 1992] Abrash, Michael. *Dr. Dobbs's Journal* 17 (8): 149, 1992. Also in Abrash, Michael, *Zen of Graphics Programming*. Coriolis Group, 1996, p. 649. Also see <http://www.whisqu.se/per/docs/graphics77.htm>.
- [ASHIKHMIN 2000] Ashikhmin, Michael, and Peter Shirley. *An Anisotropic Phong Model*. Technical Report UUCS-00-014. Dept. Computer Science, University of Utah, August 13, 2000.
- [BLINN 1977] Blinn, J. F. "Models of Light Reflection for Computer Synthesized Pictures," *Computer Graphics (Proceedings of Siggraph 77, Annual Conference Series)*, 1977, pp. 192–

198. Also see Wolfe, Rosalee (ed.). *Seminal Graphics: Pioneering Efforts that Shaped the Field*. New York: ACM Press, 1998 (ASIN: 158113052X).
- [BRDF] Some online BRDF databases: <http://www.cs.columbia.edu/CAVE/coret>, <http://www.graphics.cornell.edu/online/measurements>, and <http://math.nist.gov/~Fhunt/appearance>.
- [BUI 1998] Bui-Tong, Phong. "Illumination for Computer Generated Pictures," *Comm. ACM* 18 (6): 311–317, 1975. Also see Wolfe, Rosalee (ed.). *Seminal Graphics: Pioneering Efforts that Shaped the Field*. New York: ACM Press, 1998 (ASIN: 158113052X).
- [COOK 1982] Cook, Robert L., and Kenneth E. Torrance. "A Reflectance Model for Computer Graphics," *ACM Transaction on Graphics* 1(2): 7–24, 1982.
- [DEMPSKI 2001] Dempksi, Kelly. [*Real-Time Rendering Tricks and Techniques in DirectX*](#). Cincinnati, OH: Premier Press, 2001 (ISBN 1-931841-27-6).
- [ENGEL 2002] Engel, Wolfgang F. (ed.). *Direct3D ShaderX*. Plano, TX: Wordware Publishing, 2002 (ISBN 1-55622-041-3).
- [FAIRN 1998] Fairn, G., and D. Hansford. *The Geometry Toolbox for Graphics and Modeling*. Natick, MA: A. K. Peters, 1998 (ISBN: 1568810741).
- [FISHER 1994] Fisher, F., and A. Woo. "R.E versus N.H Specular Highlights," in Paul Heckbert (ed.), *Graphics Gems IV*. San Diego, CA: Academic Press, 1994, pp. 388–400.
- [FOSNER 2000] Fosner, Ron. "All Aboard Hardware T&L." *Game Developer*, April 2000.
- [GLASSNER 1992] Glassner, Andrew. "Darklights," in David Kirk (ed.), *Graphics Gems III*. San Diego, CA: Academic Press, 1992, pp. 366–368.
- [GOOCH 1998] Gooch, Amy Ashurst, Bruce Gooch, Peter Shirley, and Elaine Cohen. "A Non-Photorealistic Lighting Model for Automatic Technical Illustration," *Computer Graphics (Proceedings of Siggraph 98, Annual Conference Series)*, 1988, pp. 447–452. Also see <http://www.cs.utah.edu/~gooch/SIG98/abstract.html>.
- [GOOCH 2001] Gooch, Bruce, and Amy Ashurst Gooch. *Non-Photorealistic Rendering*. Natick, MA: A. K. Peters Ltd., 2001 (ISBN: 1-568811-33-0).
- [HALL 1989] Hall, R. *Illumination and Color in Computer Generated Imagery* (Monographs in Visual Computing). New York: Springer-Verlag, 1989 (ISBN: 0387967745).
- [HAEBERLI 1990] Haeberli, Paul E. "Paint By Numbers: Abstract Image Representations," *Computer Graphics (Proceedings of Siggraph 90, Annual Conference Series)* 24(4): 207–214, August 1990.
- [HE 1991] He, XiaoD., Kenneth E. Torrance, F. Sillion, and Donald P. Greenberg. "A Comprehensive Physical Model for Light Reflection." *Computer Graphics (Proceedings of Siggraph 91, Annual Conference Series)* 25(4), July 1991.
- [HE 1992] He, Xiao D., Patrick O. Heynen, Richard L. Phillips, Kenneth E. Torrance, David H. Salesin, and Donald P. Greenberg. "A Fast and Accurate Light Reflection Model," *Computer Graphics (Proceedings of Siggraph 92, Annual Conference Series)* 26(2): 253–254, July 1992.

- [HECHT 2001] Hecht, U. *Optics* (4th ed.). Boston, MA: Addison-Wesley Publishing, 2001 (ISBN: 0805385665).
- [HILL 2000] Hill, F. S. *Computer Graphics Using OpenGL* (2nd ed.). Upper Saddle River, N.J.: Prentice Hall, 2000 (ISBN: 0023548568).
- [KALNINS 2002] Kalnins, Robert D., Lee Markosian, Barbara J. Meier, Michael A. Kowalski, Joseph C. Lee, Philip L. Davidson, Matthew Webb, John F. Hughes, and Adam Finkelstein. "WYSIWYG NPR: Drawing Strokes Directly on 3D Models." To appear in *Computer Graphics (Proceedings of Siggraph 02, Annual Conference Proceedings)*, July 2002. Also see <http://www.cs.brown.edu/people/bjm/publications.htm>.
- [KOWALSKI 1999] Kowalski, Michael A., Lee Markosian, J. D. Northrup, Lubomir Bourdev, Ronen Barzel, Loring S. Holden, and John F. Hughes. "Art-Based Rendering of Fur, Grass, and Trees," *Computer Graphics (Proceedings of Siggraph 99, Annual Conference Series)*, 1999, pp. 433–438. Also see <http://www.cs.brown.edu/people/lem/research/kowalski-s99-preprint-300dpi.pdf>.
- [LASSETER 1987] Lasseter, John. "Principles of Traditional Animation Applied to 3D Computer Animation," *Computer Graphics (Proceedings of Siggraph 87, Annual Conference Series)* 21(4): 35–44, July 1987.
- [LENGYEL] Lengyel, Eric. *Mathematics for 3D Game Programming and Computer Graphics*. Rockland, MA: Charles River Media, 2002 (ISBN 1-58450-037-9).
- [MEIER 1996] Meier, Barbara J. "Painterly Rendering for Animation," *Computer Graphics (Proceedings of Siggraph 96, Annual Conference Series)*, August 1996. Also see <http://www.cs.brown.edu/people/bjm/publications.htm>.
- [MSSDK] Microsoft's DirectX SDK is available online by selecting the DirectX menu item found at <http://msdn.microsoft.com/library>.
- [OREN 1994] Oren, M., and S. Nayar. "Generalization of Lambert's Reflectance Model," *Com. Assn. Comp. Machinery*, 1994.
- [OREN 1992] Oren, M., and S. Nayar. "Generalization of the Lambertian Model and Implications for Machine Vision," *Int. J. of Comp. Vision* 14(3), 1992.
- [OSTROMOUKHOV 1999] Ostromoukhov, Victor. "Digital Facial Engraving," *Computer Graphics (Proceedings of Siggraph 99, Annual Conference Series)*, 1999, pp. 417–424.
- [OSTROMOUKHOV 1995] Ostromoukhov, Victor, and Roger D. Hersch, "Artistic Screening," *Computer Graphics (Proceedings of SIGGRAPH 95, Annual Conference Series)*, 1995, pp. 219–228.
- [POULIN 1990] Poulin, Pierre, and Alain Fournier. "A Model for Anisotropic Reflection," *Computer Graphics (Proceedings of Siggraph 99, Annual Conference Proceedings)*, 1999, pp. 273–282.
- [PRAUN 2001] Praun, Emil, Hughes Hoppe, Matthew Webb, and Adam Finkelstein. "RealTime Hatching," *Computer Graphics (Proceedings of Siggraph 01, Annual Conference Series)*, 2001, pp. 579–584.

- [ROMNEY 1969] Romney, G. W., G. S. Watkins, and D. C. Evans. *"Real Time Display of Computer Generated Half-Tone Perspective Pictures,"* *Proceedings 1968 IFIPS Congress*. Amsterdam: North Holland Publishing Co., 1969, pp. 973–978.
- [RTR 2002] Akenine-Möller, Tomas, and Eric Haines. *Real Time Rendering* (2nd ed.). Natick, MA: A. K. Peters Ltd., July 2002 (ISBN: 1568811829).
- [SCHLICK 1994] Schlick, Christophe. *"A Fast Alternative to Phong's Specular Shading Model,"* in Paul Heckbert (ed.), *Graphics Gems IV*. San Diego, CA: Academic Press, 1994, pp. 363–366. Also see <http://dept-info.labri.u-bordeaux.fr/~schlick/DOC/gem3.html>.
- [STREIT 1998] Streit, Lisa, and John Buchanan. *"Importance Driven Halftoning,"* *Eurographics* 98 (3)17, 1998. Also see <http://citeseer.nj.nec.com/streit98importance.html>.
- [STREIT 1999] Streit, Lisa, O. Veryovka, and John Buchanan. *"Non-photorealistic Rendering Using an Adaptive Halftoning Technique,"* *Computer Graphics (Proceedings of Siggraph 99, Annual Conference Series)*, 1999. Also see <http://pages.cpsc.ucalgary.ca/~jungle/skigraph99/papers/streit.pdf>.
- [STROTHOTTE 2002] Strothotte, Thomas, and Schlechtweg, Stefan. *Non-Photorealistic Computer Graphics: Modeling, Rendering, and Animation*. San Francisco: Morgan Kaufmann, April 2002 (ISBN 1-55860-787-0).
- [TROWBRIDGE 1975] Trowbridge, Y.S., and K.P. Reitz. *"Average Irregularity Representation of a Rough Surface for Ray Reflection,"* *J. Opt. Soc. of America* 65: 531–536, 1975.
- [TURKOWSKI 1990] Turkowski, Ken. *"Properties of Surface-Normal Transformations,"* in Andrew Glassner (ed.), *Graphics Gems I*. Diego, CA: Academic Press, 1990, pp. 539–547. Also see <http://www.worldserver.com/turk/computergraphics/papers.html#Image%20Registration>.
- [VERTH 2000] Van Verth, Jim. *"Math for Programmers." Course Notes. Game Developer's Conference*, San Jose, CA, 2001.
- [WARD 1992] Ward, Gregory. *"Measuring and Modeling Anisotropic Reflection,"* *Computer Graphics (Proceedings of Siggraph 92, Annual Conference Series)* 26 2): 265–272, July 1992.
- [WARNOCK 1969] Warnock, J. *"A Hidden-Surface Algorithm for Computer Generated Half-tone Pictures."* Computer Science Department TR 4–15, NTIS AD-753 671, University of Utah, 1969.
- [WEBB 2002] Webb, Matthew, Emil Praun, Hughes Hoppe, and Adam Finkelstein. *"Fine Tone Control in Hardware Hatching,"* *Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*, 2002, pp. 53–58.
- [WINKENBACH 1994] Winkenbach, Georges, and David H. Salesin. *"Computer-Generated Pen-and-Ink Illustration,"* *Computer Graphics (Proceedings of Siggraph 94, Annual Conference Series)*, 1994, pp. 91–100. Also see <http://inklination.com/Documents/Siggraph94.pdf>.
- [WOLFE 1998] Wolfe, Rosalee. (ed.). *Seminal Graphics: Pioneering Efforts that Shaped the Field*. New York: ACM Press, 1998 (ASIN: 158113052X).

About the CD-Rom

The CD-ROM included with this book contains all of the files you'll need to follow along. It contains ATI's RenderMonkey and a collection of shaders for the book written with RenderMonkey. Also included is the DirectX 8.1 SDK from Microsoft. If you don't have a graphics card capable of running shaders in hardware, RenderMonkey will let you run the shaders using the reference rasterizer from the SDK (but they will run very slowly!).

If you have a Windows 2000 or Windows XP machine, then the CD should automatically run your browser and bring up the index.html file on the CD. If nothing happens after you insert the CD, open the CD and click on the index.html file.

COMPONENT LIST

ATI RenderMonkey v0.52Beta.exe

The installation program for ATI's RenderMonkey (0.52b) is an easy to use tool for exploring vertex and pixel shader programming.

ATI RenderMonkey v0.5Beta.pdf

PDF documentation for RenderMonkey

ATI RenderMonkey Subdirectory

The subdirectory for all the additional shaders, models, and textures used in the book that go along with RenderMonkey

ColorSpace.exe

The ColorSpace tool described in the book

DirectX 8.1 SDK

The Microsoft DirectX 8.1 SDK will install the development tools for writing DirectX applications. If you have a fast Internet connection, you might try to download the DirectX 9 SDK instead. However, all of the samples on this CD-ROM will work with the DirectX 8.1 SDK also included on this CD-ROM.

SYSTEM REQUIREMENTS

RenderMonkey System Requirements

- Windows 2000 (Service Pack 2), Windows XP, Windows 98, or Windows ME

- DirectX 8.1 (or better) and the DirectX 8.1 (or better) compliant graphics card
- 128 MB RAM and 100 MB free hard drive space

DirectX SDK System Requirements

The Microsoft DirectX 8.1 Software Development Kit (SDK) contains the tools needed to build cutting-edge, media-rich, interactive applications. It includes the runtime release of DirectX 8.1, headers and libs, sample executables, sample source, documentation, DirectX utilities, and support for both C++ and Visual Basic development. Please see DirectX 8.1 Release Notes for the latest updated information.

IMPORTANT NOTES

This product requires Microsoft Windows 98, Windows ME, or Windows 2000. For Windows XP, download the latest Windows XP Service Pack. Please see Knowledge Base article Q322389, "How to Obtain the Latest Windows XP Service Pack," for additional information.

This product does not support either Windows 95 or Windows NT 4.0. If you have the DirectX 8.1 SDK installed, all you need to update are the "DirectX 8.1 Developer Runtimes" and "DirectX 8.1 Redist Install." Go to DirectX 8.1 Software Development Kit Componentized Download to download these packages separately.

UPDATES

I included as much DirectX 9 low-level shader languages material as possible prior to publication. DirectX 9 will be released very near the time you see this book in bookstores. If you have access to the Internet, you should check for updates to RenderMonkey (<http://www.ati.com>), updates to the DirectX SDK (<http://www.microsoft.com>), and updates to the shaders in this book (<http://www.directx.com>).

Index

Symbols and Numbers

^ (circumflex), [10](#)

// (forward slashes), placing C-style comments in shader code, [192-193](#)

–(minus sign). See [negate modifier](#)

1D texture, using as a lookup table, [337](#)

2 × 2 bump map matrix, applying, [290](#)

2D bump/environment mapping, [328](#), [329-330](#)

2D transformation calculations, m3x2 macro used for, [237](#)

_2x modifier, [368](#)

3 element vector

 multiplying by a 2×3 matrix, [306](#)

 multiplying by a 3×3 matrix, [306](#)

 multiplying by a 4×3 matrix, [308](#)

3-space, defining a plane in, [17](#)

3×2 matrix multiplies

 performing, [344](#)

 performing on a 3 element vector, [306](#)

 performing with a texture lookup, [345](#)

3×3 matrix multiplies

 performing, [347-348](#)

 performing followed by a texture lookup, [350](#)

 performing on a 3 element vector, [306-308](#)

 performing the final row multiplication of, [349](#), [351](#)

 performing the final stage of, [346](#)

3×4 matrix multiplies

 performing on a 3 element vector, [308-309](#)

 performing on a 4 element vector, [309](#)

3D graphics, math required for, [9-30](#)

3D scenes, rendered with graftals, [86-87](#)

3D volumetric textures, supporting, [356](#)

3Dlabs Web site, [145](#)

4 element vector

 multiplying by a 3×4 matrix, [309](#)

 multiplying by a 4×4 matrix, [311](#)

4×4 matrix multiply, performing on a [4](#)

 element vector, [310-311](#)

4D texture, mapping into 3D texture space, [361](#)

 _4x modifier, [368](#)

 _8x modifier, [368](#)

A

Abrash, Mike, [23](#)

abs macro, [205](#), [288-289](#)

`_abs` modifier (absolute modifier), [270](#)

absolute value

computing for the input register, [205](#)

of the input register, [288](#)

notation for, [11](#)

ACM (Association for Computing Machinery), [146](#)

add instruction, [206](#), [288-289](#)

addition, notation for, [11](#)

address register (a0.x), allowing a programmable index into constant registers, [197-198](#)

address register assignment, pseudocode used in the emulation of, [201](#)

address registers (an), [200-202](#), [271](#)

moving data from a floating point register into, [251-252](#)

affine transformation, [103](#), [104](#)

air-glass interface

averaged reflectance and transmittance values for, [69](#)

values for the Fresnel equation for, [67-70](#)

alpha component, representing the opacity of color, [22](#)

alpha mask, [364](#)

alpha operations

pairing color operations with, [370](#)

performing simultaneously with color operations, [93-94](#)

alpha pipe, [93](#)

alpha reference value, [178](#)

ambient color, [34](#)

ambient illumination

generating the effect of, [155-157](#)

scene with, [40](#), [41](#)

ambient light, [39-41](#), [43](#), [44](#)

ambient light equation, compared to the diffuse lighting equation, [42](#)

ambient light shader, [155-157](#)

ambient lighting, [34](#)

ambient shading, [155-157](#)

an. See [address registers](#)

angle, between two vectors, [15](#)

angle of incidence

- Fresnel value as a function of, [71](#)
- of incoming light, [59](#), [60](#)

anisotropic BRDFs, [80](#)

anisotropic reflection, [78-80](#)

antisymmetric, cross product as, [18](#)

Arago, Francois, [67](#)

arbitrary mask, [365](#)

arithmetic instructions, for pixel shaders, [108](#), [274](#)

arithmetic operations, of pixel shaders, [94](#)

array of structures, [123](#)

artistic screening, [82](#)

ASCII shader program, [130](#)

"asm" style comments, [193](#)

ATI, RenderMonkey shader tool, [139-140](#)

ATI session on hardware programming, [146](#)

ATI Web site, [145](#)

attenuated lighting calculations, [216](#)

attenuation factor, equation for, [51](#)

attenuation of lights, [33](#)

B

backscattering, [54](#)

base two exponent, computing for a scalar value, [301-302](#)

base two logarithm, of a scalar, [303-304](#)

Beckmann distribution function, [75-76](#)

behavior flags, for vertex processing, [115](#)

bem (bump environment map) instruction, [290-291](#)

`_bias` modifier, [357-358](#)

bias times two modifier. See [_bx2 modifier](#)

bidirectional reflectance distribution function. See [BRDF](#)

binary tokens

- converting ASCII strings to, [130-131](#)

- generating, [130](#)

blend indices vector component, [199](#)

blend weight vector component, [199](#)

blending operations, in DirectX, [167-168](#)

Blinn, model for specular reflection, [77-78](#)

Blinn-Phong specular lighting, [162-164](#)

Blinn's simplification

- implementing to Phong's lighting equation, [162](#)

- of Phong's equation, [48](#)

boolean array, setting for the DirectX 9 style vertex shader, [134-135](#)

boolean constants, setting the value of vertex shader, [213](#)

BRDF (bidirectional reflectance distribution function), [76-80](#)

BRDF parameters, implementing, [77-78](#)

BRDF specular function, [78](#)

BRDF texture, [78](#), [79](#)

Brewster's angle, [68-69](#)

bright areas, eliminating in a scene, [30](#)

brightness, increasing on one particular light or texture, [26](#)

brush-applied media, simulating the results from, [81](#)

bump environment map transform, applying a fake, [290-291](#)

bump mapping, most commonly used style of, [184-186](#)

`_bx2` modifier

- adding to a register, [358-359](#)

- cannot be used with `texbem`, [328](#)

- cannot be used with `texbeml`, [330](#)

C

C/C++ structure, for vertex elements, [118](#)

C-like pseudocode with vertex instructions, [191-192](#)

C-style comments, placing in shader code, [192-193](#)

cache coherency, increasing, [126](#)

call instruction, [207-208](#), [226-227](#)

callnz instruction, [208-209](#), [226-227](#)

camera space. See [eye space](#)

cartoon shading technique, [181-184](#)

cartoonlike feel, getting with cel shading, [84](#)

cel shading, [84](#), [85](#)

Cg higher level language, [90](#)

Charles River Media Web site, [145](#)

CIE diagrams, displaying perceived color space, [24](#)

circumflex (^), [10](#)

clamping

color values, [27](#), [28](#)

diffuse values to only positive, [42](#)

clip coordinates, getting world coordinates to, [98](#)

clip space, [99](#), [100](#)

clip-space coordinates

inputting eye position as, [152](#)

placing the vertex's position in, [195](#)

transforming vertex positions into, [96](#)

clipping, [27](#), [28-29](#)

cmp (compare) instruction, [292-293](#)

c[n]. See [constant registers](#)

cnd (conditional) instruction, [293-294](#)

color and alpha blending instructions, for pixel shaders, [273](#)

color data, passing texture coordinate data from the source into the destination register as, [332-333](#)

color images, for printing, [23](#)

color inputs, for pixel shaders, [107](#)

color modulated decal shader, [166-168](#)

color operations

pairing with alpha operations, [370](#)

performing simultaneously with alpha operations, [93-94](#)

color pipe, [93](#)

color registers, provided to FFP pixel shaders, [95](#)

color rgba value, generated as pixel shader output, [107](#)

color spaces, [23](#)

color values

- clamping, [27](#), [28](#)

- multiplying, [24-29](#)

- negative, [30](#)

- passing texture coordinates into the shader as, [330](#)

- piecewise multiplication of, [25](#)

- scaling by intensity, [28](#)

- shifting to maintain saturation, [28-29](#)

- in a texture register, [285](#)

color write mask, [364](#)

colors

- adding together interactively, [143-144](#)

- blending with textures, [167](#)

- exceeding the displayable range, [27](#)

- mathematics of, in color graphics, [22-30](#)

- physical range displayed by a device, [23](#)

ColorSpace tool, [29](#), [143-144](#), [145](#)

comic book style. See [cartoon shading technique](#)

commutative, dot product as, [16](#)

components, of vertex shaders, [150-153](#)

computer graphics, mathematics of color in, [22-30](#)

conditional assignment, performing, [292](#), [293-294](#)

conductive media, Fresnel equations for, [64-65](#)

conductivity, of materials, [36](#)

conductors, [36](#)

`ConfirmDevice()` function, [116-117](#)

constant arrays, in DirectX 9, [133](#)

constant color shading shaders, [153-154](#)

constant definitions, [274](#)

constant registers (`c[n]`), [197-198](#)

- for pixel shaders, [283-284](#)

constants

- defining for pixel shader use, [297](#)

- swizzling references to, [197](#)

Cook-Torrance model, [74](#), [76](#)

coordinates. See [points](#)

cosine

- of the angle between vectors, [16](#)

- computing for a scalar argument, [264](#)

cosine and sine, computing for a scalar value, [324-325](#)

crease angle cutoff, [22](#)

CreateDevice() function, specifying settings for shader usage, [114](#)

CreateVertexDeclaration() function, [122](#)

critical angle, for total internal reflection, [60-61](#)

cross product, [17-20](#)

- notation for, [11](#)

- relating vectors across, [20](#)

- right-hand rule followed by, [18-19](#)

cross product term, relating to a dot product term, [20](#)

crs macro, [209-210](#), [294-295](#)

cube map, sampling, [339](#)

current model/world/view transformation matrix, transforming the vertex position, [195](#)

current texture stage, sampling the color from, [326](#)

custom register declaration, without using a FVF, [120](#)

custom vertex formats, [125](#), [126](#)

customized vertex buffer formats, [126](#)

D

_d2 (half) modifier, [368](#)

D3D device, selecting, [114-115](#)

D3D Shader Debugger, [141-142](#), [143](#)

D3Dapp framework, ConfirmDevice() function, [116-117](#)

D3DCAPS8.MaxPixelShaderValue, [110](#)

D3DCAPS9.PixelShader1xMaxValue, [110](#)

D3DCAPS.MaxStreams capabilities bit, [127](#)

D3DCAPS.MaxStreamStride capabilities bit, [127](#)

D3DCREATE_HARDWARE_VERTEXPROCESSING flag, [115](#)

D3DCREATE_MIXED_VERTEXPROCESSING flag, [115](#)

D3DCREATE_PUREDEVICE flag, [115](#)

D3DCREATE_SOFTWARE_VERTEXPROCESSING flag, [115](#)

D3DDEVTYPE_HAL setting, with CreateDevice(), [114](#)

D3DDEVTYPE_REF setting, with CreateDevice(), [114](#)

D3DDEVTYPE_SW setting, with CreateDevice(), [114](#)

D3DLOCK_DISCARD flag, [126](#)

D3DLOCK_NOOVERWRITE flag, [126](#)

D3DRS_ALPHAREF state, [178](#)

D3DRS_FOGENABLE parameter, [169](#)

D3DSAMP_ELEMENTINDEX render state, [287](#)

D3DSAMP_MAXMIPLEVEL flag, valid with texldb, [341](#)

D3DSAMP_SRGBTEXTURE render state, [287](#)

D3DTEXTURECAPS_TEXREPEATNOTSCALEDDBYSIZE capability bit, [203](#)

D3DTOP_ADDSIGNED texture operation, bias modifier and, [358](#)

D3DTOP_ADDSIGNED2X texture operation, signed scaling modifier and, [359](#)

D3DTOP_DOTPRODUCT3 blending operation, [184](#)

D3DTOP_MODULATE, for DirectX, [166](#)

D3DTOP_SELECTARG1 rendering setting, for DirectX, [165](#)

D3DTSS_TEXCOORDINDEX texture stage state flag, [326](#), [331](#)

D3DTTFF_PROJECTED flag

- ignoring, [339](#)
- not selecting when using the programmable pipeline, [203](#)
- not valid for PS 2.0 using texldp, [342](#)

D3DUSAGE_WRITEONLY flag, using for static data, [125](#)

D3DVERTEXELEMENT9 structures, token as, in DirectX 9, [120](#)

D3DXAssembleShader() helper function, [130-131](#), [212](#)

_d4 (quarter) modifier, [280](#), [368](#)

_d8 (eighth) modifier, [280](#), [368](#)

darklights, [30](#)

data, static, [125](#)

dcl (declare) instruction, [210-211](#), [296](#)

dcl statement

- declaring register usage, [287](#)

- including in all vertex shaders, [196](#)

debugging capability, interactive, [140-141](#), [142](#)

decals, applying a texture as, [174](#)

decals, texture, [165](#)

decals, texture shader, [165-166](#)

declaration statement

- on all sampler, color input, and texture coordinate registers, [296](#)

- required on all input registers, [211](#)

def (define) instruction, [211-212](#), [297](#)

- in pixel shaders, [136](#)

- setting pixel shader constant registers, [284](#)

- in vertex shaders 2.0, [198](#)

def statements

- locking constant registers, [198](#)

- in pixel shaders, [133](#)

defb instruction, [213](#)

defi instruction, [212-213](#)

degenerate triangles, adding in, [125](#)

density, as a function of temperature, [70](#)

depth buffer, attached to the MRT surface, [288](#)

depth value

- calculating for a depth test, [333-334](#)

- resolution of, [99](#)

destination masks, [268-269](#)

destination register

- adding two sources into, [206](#)

- independent of the read port count restrictions, [274](#)

- storing source registers into, [250-251](#)

destination write masks, [363-367](#)

determinant notation, for the cross product, [18](#)

device capabilities bits, querying, [116](#)

device memory, copying data into, [125](#)

diamond-air interface, [61](#)

dielectrics, [36](#)

 Fresnel equations for, [65-70](#)

diffuse and specular colors, setting both, [109](#)

diffuse and specular lighting coefficients, computing, [227-229](#)

diffuse color, [34](#)

diffuse light, [41-44](#)

diffuse lighting, [34](#)

 changing for an object, [43](#)

 equation for calculating, [42](#)

 near uniform loss of, [54](#)

 traditional expression for, [157](#)

diffuse reflection, Oren-Nayar, [54-58](#)

diffuse term

 generating for a local light shining on a vertex, [158](#)

 generating for an infinitely distant light shining on a vertex, [157](#)

diffuse values, clamping to only positive, [42](#)

diffuse vector component, FVF vertex shader register mapping order, [199](#)

digital facial engraving, creating from 2D images, [82](#)

digital halftoning, using to get range and material, [82](#)

Direct3D and OpenGL pipeline. See [FFP \(fixed-function pipeline\)](#)

Direct3D flags, defining the format, [118-119](#)

direction vectors, [13](#), [20-21](#)

DirectX

 SDK, [114](#)

 shader setup in, [113-137](#)

 unit cube in, [99](#)

 vertex shader declaration documentation, [119](#)

 view matrix as the equivalent of OpenGL's modelview, [99](#)

DirectX 8

 input vertex registers order using FVF format, [199](#)

- loading the vertex shader, [132](#)
- pixel shader math precision, [109-110](#)
- setting vertex shader constants, [133-134](#)
- vertex streams introduced by, [93](#)

DirectX 8.0, version history, [193](#)

DirectX 8.0a, version history, [196](#)

DirectX 8.1, version history, [193](#)

DirectX 9, [90](#)

- compliant pixel shader versions, [109](#)
- creating a shader, [131-132](#)
- D3D Shader Debugger shipped with, [141-142](#), [143](#)
- higher precision registers for pixel shaders, [279](#)
- integer and bool constants introduced in, [198](#)
- loading the vertex shader, [132-133](#)
- preparing the vertical pipeline for tessellation processing, [122](#)
- setting vertex shader constants, [134-135](#)
- significant changes introduced with the pixel shader 2.0, [273-274](#)

DirectX 9.0

- revision history of, [196](#)
- version history, [193](#)

DirectX FFP, use of Blinn's equation for specular light, [49](#)

dirt map. See [scuff map](#)

disk with this book, RenderMonkey SDK, [139](#)

displayable range, colors exceeding, [27](#)

distance vector, computing, [216](#)

dithering, [82](#)

division, notation for, [11](#)

dot-3 bump mapping, [184-186](#)

dot product, [15-17](#)

- calculating using the (r, g, b) texture data in the source and destination texture coordinate data (u, v, w), [334](#), [336](#)
- information available from, [16-17](#)
- macroinstructions expanding into calls, [106](#)
- notation for, [11](#)

dot product 3. See [three-component dot product](#)

dot product instructions

- performing a matrix multiply, [97](#)

- taking the transpose of the matrix, [100](#)

dot product term, relating a cross product term to, [20](#)

dp2add instruction, [291](#), [297-298](#)

dp3 (dot product 3) instruction, [214](#), [298-299](#), [371](#)

dp3-rsq-mul sequence, [151](#)

dp4 (dot product 4) instruction, [215](#), [300-301](#)

DrawPrimitive call, [137](#)

driver bugs, checking for, [114](#)

dst instruction, [216](#)

_dw modifier, [362-363](#)

- mixing the .xyw modifier with, [339](#), [361](#)

- performing a perspective divide, [285](#), [333](#)

DWORDS, tokens as in DirectX 8, [120](#)

dwSilhouetteAmount render state, setting the silhouette edge threshold, [178](#)

dynamic data, [125-126](#)

dynamic vertex buffers, allocating, [125](#)

_dz modifier, [340](#), [362-363](#)

E

edge effect, computing, [187](#)

Effects Browser, from NVIDIA, [140](#), [141](#)

electric field, of a wave of energy, [64](#), [65](#)

element-by-element addition. See [piecewise addition](#)

element-by-element multiplication. See [piecewise multiplication](#)

element index, using when a MET surface is set to the sampler, [287](#)

element masks, [269](#)

else instruction, [217](#)

emissive color, [34](#)

emissive light, perceiving detail from, [23](#)

emissive lighting, [35](#)

`endif` instruction, [217-218](#)

`endloop` instruction, [218-219](#), [234](#)

`endrep` instruction, [219-220](#)

energy, calculating the percent transmitted or reflected, [66](#)

Engel, Wolfgang, [145](#)

engraving style, [81-82](#), [83](#)

environment, light reflected from, [39](#)

environment illumination, [34](#)

environment mapping, [348-350](#)

execution time

- of instructions and macroinstructions, [204](#)
- of `rcp`, [257](#)
- of `rsq`, [260](#)

`exp` instruction, [301-302](#)

`exp macro`, [220-221](#)

`expp` instruction, [220](#), [221-223](#)

eye

- gamuts of, [24](#)
- perceptiveness of, [23](#)

eye direction, specular highlights varying with, [352](#)

eye position, [151](#). See also [viewer position](#)

eye space, [98](#)

F

face, of a triangle, [21](#)

face normals, [21](#)

FFP (fixed-function pipeline)

- replacing sections of, [90-91](#)
- setting the color, [195](#)
- using in DirectX 9, [129](#)
- vertex declarations used with in DirectX 9, [132](#)
- vertex operations, [95](#)

field-of-view parameter, [99](#)

final color value, calculating, [35](#)

flat shading, render state as, [282](#)

flexible vertex format. See [FVF](#)

float array, setting for the DirectX 9 style vertex shader, [134-135](#)

float constant registers, [197](#)

- in DirectX 8 shaders, [198](#)
- for pixel shaders, [282](#), [283-284](#)

float constants, in DirectX 9 vertex shaders, [198](#)

floating point constants, setting the value of vertex shader, [211-212](#)

floating point register, moving data from into the address register, [251-252](#)

floating point vertex shader registers, getting the maximum number of, [136](#)

flow control, added with vertex shader 2.0, [202](#)

fog blend, performance of, [107](#)

fog calculations, applying, [169-172](#)

fog shader, [168-172](#)

forward scattering, [54](#)

forward slashes (//), placing C-style comments in shader code, [192-193](#)

four-component dot product, computing, [215](#), [300-301](#)

four-element vectors, passing to your pixel shader, [203](#)

frame of rendering, setting values of the constant registers prior to, [197](#)

`frc` (fractional) instruction, [302-303](#)

`frc` macro, [224-225](#)

Fresnel, Augustin, [67](#)

Fresnel equations, [64-74](#)

for dielectrics, [65-70](#)

Fresnel shader, [187-189](#)

Fresnel term

- approximating, [187](#)
- getting a reasonable estimate on, [71](#)
- for real materials, [70-74](#)

Fresnel value, at normal angles of incidence, [71](#)

full moon, non-Lambertian distribution of, [54](#), [55](#)

full precision registers, in pixel shaders version 2.0 or better, [109](#)

function call, making to the instruction label, [208-209](#)

functions

`ConfirmDevice()` function, [116-117](#)

`CreateDevice()` function, [114](#)

`CreateVertexDeclaration()` function, [122](#)

`D3DXAssembleShader()` helper function, [130-131](#), [212](#)

Gaussian distribution function, [75](#)

`GetDeviceCaps()` function, [115](#)

`RegisterSoftwareDevice()` function, [114](#)

`SetFVF()` function, [129](#), [132](#), [133](#)

`SetPixelShader()` function, [132](#), [133](#)

`SetPixelShaderConstants()` function, [136](#), [284](#), [297](#)

`SetShaderConstant()` function, [133](#)

`SetStreamSource()` function, [101](#)

`SetTexture()` function, [108](#), [287](#)

`SetTextureStage()` function, [291](#)

`SetTextureStageState()` function, [108](#)

`SetVertexDeclaration()` function, [122](#), [133](#)

`SetVertexShader()` function, [128](#), [132](#), [133](#)

`SetVertexShaderConstant()` function, [100](#), [198](#)

`SetVertexShaderConstantF()` function, [100](#)

FVF (flexible vertex format), [119](#)

FVF bit masks, [119](#)

FVF codes, vertex shader registers mapped in, [119](#)

FVF flag, making zero in the `CreateVertexBuffer()` call, [125](#)

FVF formats, in DirectX 8, [199](#)

G

Game Programming Gems series, [145](#)

gamuts, [23-24](#)

"The Gathering" session on hardware programming, [146](#)

Gaussian distribution function, modeling the slope distribution, [75](#)

GDC (Game Developer's Conference), [146](#)

geometric attenuation function, [76](#)

geometry processing units (GPUs), [5](#)

GetDeviceCaps () function, [115](#)

glancing angle, [69](#)

glass-air interface, [70](#), [71](#)

global coordinate system, [98](#)

global illumination, in the environment, [34](#)

global world space coordinate system, transforming from local space into, [98](#)

Gouraud shading, [282](#)

GPUs (geometry processing units), [5](#)

graftals, [86-87](#)

graphics programming, resources on, [146-147](#)

gray-scale vector, creating to run along the black-white axis of the color cube, [28](#)

Greater-Than or Equal-To, setting, [261](#)

H

half-angle vector, in Blinn's simplification, [48](#)

"half" vector, between the light and view vectors, [48](#)

halftoning, [82](#)

hard shading, [84](#)

hardware-accelerated lighting models, traditional 3D, [38-58](#)

hardware API, exposed by pixel shaders, [109](#)

hardware bugs, checking for, [114](#)

hardware FFP, number of light sources supported by, [40](#)

hardware pipeline, fitting vertex and pixel shaders into, [91-92](#)

hardware rasterization, [115](#)

hardware rasterization and shading, specifying, [114](#)

hardware vertex processing, specifying, [115](#)

hatching style, [82](#), [83](#)

high-level shading-language. See [HLSL](#)

higher level language shaders, [90](#)

higher order primitive section, of the FFP, [92](#)

HLSL (high-level-shading-language), introduced in DirectX 9, [273](#)

HSV (hue, saturation, and value) color space, [23](#)

human eye. See [eye](#)

I

ID3DVertexBuffer interface, manually inserting into a shader declaration, [212](#)

ideal diffuse reflection, [41](#)

IDirect3D interface, calling

GetDeviceCaps(), [115](#)

if blocks, placing inside if-endif or loop-endloop blocks, [225](#)

if-else-endif block

 providing an alternate path of execution for, [217](#)

 start of, [225-226](#)

if-endif block

 else instruction must be inside of, [217](#)

 termination point for, [217-218](#)

if instruction, [225-226](#)

ifc-endif block, termination point for, [217-218](#)

illuminated surface, [13](#)

illumination, physically based, [58-74](#)

illumination source. See [lights](#)

immediate constants, [198](#)

incident light, hitting a surface, [58](#)

incident light energy, relating with outgoing light energy, [76-77](#)

index of refraction, [60](#)

 data for, [70](#)

 as a function of the wavelength of the light, [62](#)

infinite light, Lambertian diffuse lighting with, [157-158](#), [159](#)

inner product. See [dot product](#)

input registers, [197-199](#)

 computing the absolute value of, [205](#)

 declaration statement required in, [211](#)

 mapping vertex elements to, [210-211](#)

for pixel shaders, [282-285](#)

removing the integer part of, [224-225](#), [302-303](#)

shifting the range of, [357](#)

input streams, determining the maximum number, [127](#)

input vertex

one-to-one correspondence with an output vertex, [92](#)

transforming to clip space, [97](#)

input vertex registers (vn), [197](#), [198-199](#)

input vertex streams

mapping with input vertex shader registers, [199](#)

specifying the location of, [101](#)

instruction label

making a function call to, [208-209](#)

making an unconditional function call to, [207-208](#)

instruction label ID, [207](#)

instruction modifiers, [367-369](#)

instruction pairing, [369-371](#)

instructions

add instruction, [206](#), [288-289](#)

bem (bump environment map) instruction, [290-291](#)

call instruction, [207-208](#), [226-227](#)

callnz instruction, [208-209](#), [226-227](#)

case sensitivity of, [193](#)

cmp (compare) instruction, [292-293](#)

cnd (conditional) instruction, [293-294](#)

dcl (declare) instruction, [210-211](#), [296](#)

def (define) instruction, [136](#), [198](#), [211-212](#), [284](#), [297](#)

defb instruction, [213](#)

defi instruction, [212-213](#)

dp2add instruction, [291](#), [297-298](#)

dp3 (dot product 3) instruction, [214](#), [298-299](#), [371](#)

dp4 (dot product 4) instruction, [215](#), [300-301](#)

dst instruction, [216](#)

else instruction, [217](#)

endif instruction, [217-218](#)

endloop instruction, [218-219](#), [234](#)

endrep instruction, [219-220](#)

exp instruction, [301-302](#)
expp instruction, [220](#), [221-223](#)
frc (fractional) instruction, [302-303](#)
if instruction, [225-226](#)
label instruction, [226-227](#)
lit instruction, [173](#), [227-229](#)
log instruction, [303-304](#)
logp instruction, [230](#), [231-234](#)
loop instruction, [234-236](#), [258](#)
mad (multiply and add) instruction, [173](#), [246-247](#), [312](#), [313](#)
max instruction, [247-248](#)
maximum allowed in vertex shaders, [204](#)
min instruction, [248-249](#)
mov instruction, [250-251](#), [315-316](#)
mova instruction, [251-252](#)
mul instruction, [253](#), [316-317](#)
nop (no operation) instruction, [254](#), [317](#)
order of appearance in a pixel shader, [274](#)
phase instruction, [274](#), [275](#), [319-320](#)
for pixel shaders, [273](#)
ps instruction, [321-322](#)
rcp (reciprocal) instruction, [256-257](#), [322-323](#)
rep instruction, [234](#), [258-259](#)
ret instruction, [207](#), [259](#)
rsq (reciprocal square root) instruction, [260-261](#), [323-324](#)
sge instruction, [261-262](#)
slt instruction, [265-266](#)
sub instruction, [266-267](#), [326](#), [327](#)
tex instruction, [326-327](#)
tex3x3pad instruction, using texm3x3 with, [346](#)
texbem instruction, [328-329](#)
texbeml instruction, [329-330](#)
texcoord instruction, [203-204](#), [330-331](#), [332](#)
texcrd instruction, [203-204](#), [285](#), [332-333](#)
texdepth instruction, [333-334](#)
texdp3 instruction, [334-335](#)
texdp3tex instruction, [336-337](#)
texkill instruction, [337-338](#)
texld instruction, [285](#), [338-340](#)

- texldb instruction, [340-341](#)
- texldp instruction, [341-342](#)
- texm3x2depth instruction, [343-344](#)
- texm3x2pad instruction, [343](#), [344-345](#), [345](#)
- texm3x2tex instruction, [345-346](#)
- texm3x3 instruction, [346-347](#), [348](#)
- texm3x3pad instruction, [347-348](#)
- texm3x3spec instruction, [284-285](#), [348](#), [348-350](#)
- texm3x3tex instruction, [350-351](#)
- texm3x3vspec instruction, [351-352](#)
- texreg2ar instruction, [352-353](#)
- texreg2gb instruction, [354-355](#)
- texreg2rgb instruction, [355-356](#)
- texdepth instruction, [286](#)
- types of in a pixel shader, [274](#)
- vs instruction, [267-268](#)

integer array, setting for the DirectX 9 style vertex shader, [134-135](#)

integer label, calling, [226](#)

integer part of the input register, removing, [224-225](#), [302-303](#)

intensity

- of a particular color element, [23](#)

- scaling color values by, [28](#)

inter-object reflectance term, simple lighting model's lack of, [40](#)

interactive debugging capability, [140-141](#), [142](#)

interactive tool, for experimenting with shader code, [139](#)

internal temporary registers, [200-202](#)

interpolated color value, in a texture register, [285](#)

inverse scaling factor, transforming normals by, [104](#)

inverse transpose model-view-projection matrix, transforming the normal of the vertex by, [150-151](#)

invert modifier, [357](#)

isotropic reflection, perfect, [39](#)

isotropic transformation, [104](#)

iterative vertex shaders, ability for, [202](#)

J-K

Jacobi's identity, relating vectors using cross products, [20](#)

L

label instruction, [226-227](#)

Lagrange's identity, [20](#)

Lambertian diffuse lighting

with an infinite light, [157-158](#), [159](#)

with a positional light, [158-161](#)

Lambertian shading, [41](#)

Lambert's cosine law, [41](#)

length, of a vector, [14](#)

Less-Than, setting, [265](#)

light attenuation, [51-53](#)

light diffuse color, passing as a constant, [157](#), [160](#)

light-material surface interactions, [34](#)

light sources. See *a/so* [lights](#)

interaction with a surface, [25](#)

light from a, [44](#)

light specular color, passing as a constant, [162](#)

light-surface boundary interactions, [34](#)

light wave

perpendicular nature of the magnetic and electrical fields of, [64](#), [65](#)

reflection of, [59](#)

refraction of, [59](#)

lighting, calculating at a vertex, [38-39](#)

lighting calculations, using `m3x3` macro for normal transformations, [239](#)

lighting equation, [50-51](#)

with the attenuation factor, [51](#)

refinements and alternative ways of calculating coefficients of, [51-58](#)

lighting models, traditional 3D hardware-accelerated, [38-58](#)

lights, [33](#). See *a/so* [light sources](#)

angle of incidence of incoming, [59](#)

attenuation of, [33](#)

interaction with surfaces, [36-38](#)

line art style, [81-82](#)

linear interpolation, between two registers, [236-237](#), [304-305](#)

lit function, calculating the lighting term, [162](#)

lit instruction, [173](#), [227-229](#)

local light. See also [positional light compared to "local viewer"](#), [152](#)

local lighting model, [39](#)

local model space, transforming into the global world space coordinate system, [98](#)

"local viewer", code for, [152](#)

log instruction, [303-304](#)

log macro, [230-231](#)

log₂ of the input argument, computing, [230](#)

logp instruction, [231-234](#)

 compared to the log macro, [230](#)

loop blocks, scalar register added for control, [202](#)

loop counter, [202](#), [219](#)

loop-endloop block

 starting point for, [234-236](#)

 termination point for, [218-219](#)

loop instruction, [234-236](#), [258](#)

loops, nesting not allowed, [235](#)

lrp macro, [236-237](#), [304-305](#)

luminance correction, 2D bump/environment mapping with, [329-330](#)

M

m3x2 macro, [237-239](#), [306](#)

m3x3 macro, [239-240](#), [306-308](#)

m3x4 macro, [241-242](#), [308-309](#)

m4x3 macro, [242-244](#), [309-310](#)

m4x4 macro, [244-246](#), [310-311](#)

macroinstructions, [106](#), [204](#)

macros

 abs macro, [205](#), [288-289](#)

- crs macro, [209-210](#), [294-295](#)
- exp macro, [220-221](#)
- frc macro, [224-225](#)
- log macro, [230-231](#)
- lrp macro, [236-237](#), [304-305](#)
- m3x2 macro, [237-239](#), [306](#)
- m3x3 macro, [239-240](#), [306-308](#)
- m3x4 macro, [241-242](#), [308-309](#)
- m4x3 macro, [242-244](#), [309-310](#)
- m4x4 macro, [244-246](#), [310-311](#)
- max macro, [312-313](#)
- min macro, [314-315](#)
- nrm (normalize) macro, [254-255](#), [318-319](#)
- pow macro, [255-256](#), [320-321](#)
- sgn macro, [263](#)
- sincos macro, [264](#), [324-325](#)

mad (multiply and add) instruction, [173](#), [246-247](#), [312](#), [313](#)

magnetic field, of a wave of energy, [64](#), [65](#)

magnitude, of a direction vector, [14](#)

mapping, declaring between input data streams and vertex buffers, [119](#)

marker, allowing for increased instructions in pixel shaders, [319](#)

masked rays, of the Cook-Torrance model, [75](#)

masks, [268-269](#), [363-367](#)

material boundary, physics of light interacting at, [59-74](#)

material diffuse color, passing in as a constant, [157](#), [160](#)

material internals, interaction of light with, [34](#)

material specular color, passing as a constant, [162](#)

materials, describing using colors, [34](#)

mathematical operations, notation for, [10-11](#)

mathematics

- of color in computer graphics, [22-30](#)
- required for 3D graphics, [9-30](#)

matrices, [10](#)

matrix, orthogonal, [102](#)

matrix multiply

- performing, [97](#)
- performing on the input vector and input matrix, [237](#), [239](#), [241-246](#)
- matrix transformations, getting from world coordinate space to clip space, [100](#)
- Matrox Web site, [145](#)
- max instruction, [247-248](#)
- max macro, [312-313](#)
- maximum value, storing from comparing two source registers, [247](#), [312-313](#)
- MaxPixelShaderValue
 - examining, [110](#)
 - seeing the range to which pixel registers are clamped, [278](#)
- MaxPointSize member, of the D3DCAPS structure, [204](#)
- MaxStreams capabilities bit, [127](#)
- MaxStreamStride capabilities bit, [127](#)
- MaxTextureRepeat member, of the D3DCAPS structure, [203](#)
- MaxVertexShaderConst device capability bits member, [136](#)
- MaxVertexShaderConst member, of the D3DCAPS structure, [198](#)
- Meltdown event, [146](#)
- MET (Multi-Element Texture), [287](#), [288](#)
- MET color element, setting, [315](#)
- MET surface, describing the element index to use, [287](#)
- metallic paint, shader simulating, [38](#)
- microfacets, [74](#)
- Microsoft Developer Network (MSDN) Web site, [144](#)
- Microsoft Direct 3D reference rasterizer, specifying, [114](#)
- min instruction, [248-249](#)
- min macro, [314-315](#)
- minimum value, storing from comparing two source registers, [248](#), [314-315](#)
- minus sign (–). See [negate modifier mipmap level, biasing, 340, 341](#)
- mixed (both software and hardware) vertex processing, [115](#)
- "model" coordinates, [98](#)
- model space, talking directly to clip space, [100](#)
- model transformation, in OpenGL, [98](#)

modelview matrix, premultiplying viewing parameters into, [99](#)

modifiers

`_2x` modifier, [368](#)

`_4x` modifier, [368](#)

`_8x` modifier, [368](#)

`_abs` modifier (absolute modifier), [270](#)

`_bias` modifier, [357-358](#)

`_bx2` modifier, [328](#), [330](#), [358-359](#)

`_d2` (half) modifier, [368](#)

`_d4` (quarter) modifier, [280](#), [368](#)

`_d8` (eighth) modifier, [280](#), [368](#)

`_dw` modifier, [285](#), [333](#), [339](#), [361](#)

`_dw` modifier, [362-363](#)

`_dz` modifier, [340](#), [362-363](#)

invert modifier, [357](#)

negate modifier, [270-271](#), [356-357](#)

`_pp` (partial precision hint) modifier, [296](#), [368](#), [369](#)

`_sat` modifier, [252](#), [270](#), [299](#), [300](#), [325](#), [367](#), [368](#)

`_x2` modifier, [281](#), [359](#)

`_x8` modifier, [280](#)

modulation, [35](#). See also [multiplication](#),

monitors, gamuts of, [24](#)

`mov` instruction, [250-251](#), [315-316](#)

`movb` instruction, [251-252](#)

MRT (Multiple Render Target), [287-288](#)

MRT color element, setting, [315](#)

`mul` instruction, [253](#), [316-317](#)

Multi-Element Texture. See [MET](#)

multicolor objects, generating with no shading effects, [154-155](#)

Multiple Render Target. See [>MRT \(Multiple Render Target\)](#)

multiple streams, using, [127](#)

multiplication

of color values, [25](#)

notation for, [11](#)

multitexture shader, [174-177](#)

multitexturing effects, [111](#)

N

NDC (normalized device coordinates), [99](#)

negate modifier

for pixel shaders, [356-357](#)

for vertex shaders, [270-271](#)

negation, [270-271](#), [356](#)

negative color values, [30](#)

nesting, not allowed for loops, [235](#)

night vision effect, [30](#)

No-Operation. See [nop instruction](#)

non-orthogonal transformation matrices, vertex shader for, [105-106](#)

Non-Photorealistic Animation and Rendering Conference (NPAR), [81](#)

nonconductive (dielectric) medium, Fresnel equations for, [64](#)

nonphotorealistic rendering (NPR), [81-87](#)

nonstandard vertex format, indicating, [125](#)

nonuniform scalings, in the world/model matrix, [104](#)

nop (no operation) instruction, [254](#), [317](#)

normal 2 vector component, FVF vertex shader register mapping order, [199](#)

normal map texture, [184](#)

normal transform, [150-151](#)

normal transformations, using $m_{3 \times 3}$ during lighting calculations, [239](#)

normal vector component, FVF vertex shader register mapping order, [199](#)

normal vectors

calculating, [20-21](#)

transforming, [102-106](#), [350](#)

normalized vectors, [14-15](#)

normalized view vector, placing in r_2 , [152-153](#)

normalizing, all elements of a register, [254-255](#)

normals

calculating for individual vertices, [21-22](#)

using cross products to create, [20-22](#)

notation, for this book, [10](#)

NPR (nonphotorealistic rendering), [81-87](#)

`norm` (normalize) macro, [254-255](#), [318-319](#)

null instruction. See [nop \(no operation\) instruction](#)

NVIDIA

developer site, [144](#)

Effects Browser, [140](#), [141](#)

session on hardware programming, [146](#)

Shader Debugger, [140-141](#), [142](#)

O

"object" coordinates, [98](#)

objects

calculating the shading of, [50](#)

generating multicolor with no shading effects, [154](#)

placing constant color on, [153](#)

rendering first by textures, then by shaders, [130](#)

sorting by texture, [137](#)

`oC0` register, as the output color register, [287](#)

`oD0` (diffuse output data register), [203](#)

`oD1` (specular output data register), [203](#)

`oDepth` register, [288](#), [315](#)

`oFog` register, [168](#), [204](#)

opaque, conductors as, [36](#)

opaque black, getting for uninitialized texture registers, [285](#)

OpenGL, [4](#)

premultiplying viewing parameters into its modelview matrix, [98-99](#)

use of Blinn's equation for specular light, [49](#)

operands, notation for, [10](#)

operator notation, [11](#)

`oPos` register, [97](#), [202](#)

`oPts` register, [172](#), [204](#)

- order dependence, of the cross product, [18](#)
- order independence, of the dot product, [16](#)
- Oren-Nayar diffuse reflection, [54-58](#)
- Oren-Nayar diffuse shading model, [56-58](#)
- orthogonal matrix, [102](#)
- orthogonal transformation, [103-104](#)
- orthogonal transformation matrices, vertex shader for, [104-105](#)
- `oTn`. See [output texture coordinate registers](#)
- outlining, [84](#)
- output color, setting for a shader, [101](#)
- output data registers, [203](#)
- output fog register, [168](#), [204](#)
- output point size register, [172](#), [204](#)
- output position registers, [195](#), [202](#)
- output registers
 - in DirectX 9, [287-288](#)
 - for pixel shaders, [281](#)
 - placing the results of vertex operations in, [96](#)
 - for vertex shaders, [202-204](#)
- output texture coordinate registers, [203-204](#)
- oversaturated colors
 - handling, [143-144](#)
 - strategies for dealing with, [27](#)

P

- painterly rendering style, [81](#)
- parallelepiped, defined by vectors, [20](#), [21](#)
- parallelogram, formed by two vectors, [19](#)
- partial precision, specifying (`_pp`) modifier, [296](#)
- partial precision declaration modifier, [369](#)
- partial precision \log_2 , computing, [231-234](#)
- partial precision modifier (`_pp`), applying to the declaration statement, [211](#)

partial precision power of two, computing, [221](#)

pen and ink style, [81-82](#)

per-polygon normals, [42](#)

per vertex fog values, setting, [168](#)

per-vertex normals, [42](#)

perfect isotropic reflection, [39](#)

perfect reflectors, all surfaces becoming, [69](#)

perpendicular phase, of the electric and magnetic fields in a wave of energy, [64](#), [65](#)

perspective divide

- performing in the pixel shader, [362](#)

- performing on fetched texture coordinates, [333](#)

perspective projection, performing on texture coordinates before texture sampling, [341](#)

perturbation data, transformed by a 2×2 bump environment-mapping matrix, [328](#), [329-330](#)

phase instruction, [274](#), [275](#), [319-320](#)

phase one instructions, [319](#)

phase two instructions, [319](#)

Phong's lighting equation, implementing Blinn's simplification to, [162](#)

Phong's specular equation, compared to Blinn's, [49](#)

Phong's specular light equation, [45-48](#)

physically based illumination, [58-74](#)

physically based surface models, [74-76](#)

piecewise addition, notation for, [11](#)

piecewise multiplication

- of color values, [25](#)

- notation for, [11](#)

pigment-saturated translucent coatings, subsurface scattering of, [36](#), [37](#)

pipeline, from vertex data to rendering, [129](#)

Pixar's RenderMan, [3](#)

pixel-blending section of the multitexture section, of the FFP, [93](#)

pixel registers, range clamped to, [110](#)

pixel shader code, defining the version being used, [321-322](#)

pixel shader constants, setting, [136](#)

pixel shader reference, [273-371](#)

pixel shaders, [90](#), [107-111](#), [273](#)

- adding diffuse and specular colors, [109](#)

- assembling, [130-132](#)

- breaking up into two sections, [319-320](#)

- checking for supported versions of, [116](#)

- creating and using, [130](#)

- `def` statements in, [133](#)

- determining how textures get blended, [177](#)

- higher precision registers in DirectX 9, [279](#)

- instructions for, [273](#)

- loading, [133](#)

- output of, [94](#)

- replacing sections of the fixed function

- pipeline (FFP), [91-92](#)

- revision history of, [279-281](#)

- technical overview, [93-94](#)

- texture addressing in version 2.0, [276](#)

- unique behavior between shader versions, [109](#)

- version 1.0 through 1.3, [274](#)

- version 1.4, [275](#)

- version 2.0, [276](#)

- versions 2.0 or better, [109](#)

- versions of, [273](#)

- vertex fog value inaccessible to, [204](#)

pixels

- calculating the color of particular, [24](#)

- calculating the z and w depth values of, [343](#)

- hardware accelerated method of manipulating, [111](#)

- terminating processing if texture coordinates or registers are less than 0, [337](#)

- tracing through scenes, [5](#)

plane, defining in 3-space, [17](#)

point primitives, adding "detail" to a scene, [172](#)

point size, varying, [173](#)

point size register (`oPts`), [172](#), [204](#)

point size shader, [172-174](#)

point size vector component, FVF vertex shader register mapping order, [199](#)

point sprites, [173](#)

points, [10](#), [12](#), [13](#)

Poisson, Simeon Denis, [67](#)

Poisson's bright spot, [67](#)

polarization angle, [68-69](#)

polarized lenses, [68](#)

polarized light, fraction transmitted and reflected, [66](#)

position 2 vector component, FVF vertex shader register mapping order, [199](#)

position vector component, FVF vertex shader register mapping order, [199](#)

positional light. See *also* [local light](#)

 Lambertian diffuse lighting with, [158-161](#)

Poulin-Fourier model, replacing randomly oriented v-shaped grooves with aligned cylindrical shapes, [78-79](#)

`pow` macro, [255-256](#), [320-321](#)

power function, computing for a scalar value, [255](#), [320-321](#)

`_pp` (partial precision hint) modifier, [296](#), [368](#), [369](#)

precision, indicating a lower for a register, [211](#)

primitive calls, [137](#)

printed images, manipulating to look like screen image, [23](#)

printers, gamuts of, [24](#)

prism, passing white light triangle, [62](#), [63](#)

projection matrix, calculating, [99](#)

PS 1.0 through 1.3, [274](#), [281](#)

PS 1.1 through 1.3, [285](#)

PS 1.4, [275](#), [280-281](#), [285](#)

PS 2.0, [276](#), [280](#)

 texture addressing, [276](#)

 texture registers in, [285](#)

`ps` instruction, [321-322](#)

"pure" device, specifying, [115](#)

Q

quartz, index of refraction as a function of wavelength for, [62](#)

R-S

r0 register, for single color output, [281](#)

ray tracing, [5](#)

r_{cp} (reciprocal) instruction, [256-257](#), [322-323](#)

real-time computer graphics, traditional approach in, [38](#)

"real" vertex shader, writing for a particular FVF, [129](#)

reciprocal, computing for an element of the source register, [256-257](#), [322-323](#)

reciprocal square root, computing, [260-261](#), [323-324](#)

reference rasterizer, [114-115](#)

reflectance, simulating from light traveling through a dielectric, [76](#)

reflected angle, of light, [70](#)

reflected colors, [25](#), [26-27](#)

reflected light, perceiving detail from, [23](#)

reflected ray, of light, [58](#)

reflection

- anisotropic, [78-80](#)

- of a light wave, [59](#)

reflection coefficients, [39](#)

reflection direction, of specular light, [46](#)

reflection vector

- computing for Phong's specular light equation, [46-47](#), [48](#)

- creating, [349](#), [351](#)

- specular light's intensity follows, [44](#), [45](#)

refracted ray, of light, [58](#)

refraction, [59-64](#)

- indices of, [60](#)

- spectrum spreading effect of, [62-63](#)

refractive index

- of a diamond, [61](#)

- temperature correction for, [64](#)

refract, [114-115](#)

register combiners, programming on NVIDIA hardware, [140](#)

register masks, [268-269](#)

register names, case sensitivity of, [193](#)

registers

automatic element-by-element reference to, [268](#)

declaring the usage of, in PS 2.0, [287](#)

linear interpolation between two, [236-237](#), [304-305](#)

multiplying two and adding a third, [246-247](#), [312](#), [313](#)

normalizing all elements of, [254-255](#), [318-319](#)

precision of, in pixel shaders, [276](#), [278-279](#)

selecting all elements from 1, [357](#)

setting both a single register and an array, [134](#)

setting vertex constant, [133](#)

subtracting one from another, [266-267](#)

`RegisterSoftwareDevice()` function, devices previously registered with, [114](#)

render primitive call, [128](#)

loading vertex streams into vertex registers, [129](#)

setting values of the constant registers prior to, [197](#)

render states, effecting texture sampling stage registers, [287](#)

rendering, [136-137](#)

nonphotorealistic, [81-87](#)

rendering functions, calling, [136](#)

RenderMan, [3](#)

RenderMonkey shader tool, [139-140](#)

included with this book, [6](#)

`rep-endrep` block

indicating the start of, [258-259](#)

termination point for, [219](#)

`rep` instruction, [258-259](#)

compared to `loop`, [234](#)

`ret` instruction, [259](#)

creating a shader subroutine terminating with, [207](#)

retroreflection, [54](#)

revision history

by DirectX version, [193](#)

of pixel shaders, [279-281](#)

of vertex shaders, [196](#)

Reynolds, Craig, [81](#)

rgb colors, [22-23](#)

filtering out specific, [30](#)

rgb values

assigning to a range of wavelengths, [62](#)

of lights, [33](#)

rgb vectors, color calculations on, [24-25](#)

rgba colors, [22-23](#)

rgba vectors, color calculations on, [24-25](#)

right-hand rule

computing the three-component cross product, [209-210](#)

followed by the cross product, [18-19](#)

rn. See [temporary registers](#)

rough diffuse reflecting surfaces, [54](#), [56](#), [57](#)

rough surfaces

backscattering effect exhibited by, [54](#)

efforts to better model, [55-58](#)

roughness distribution function, [75-76](#)

roughness, of a surface, [36](#)

rsq (reciprocal square root) instruction, [260-261](#), [323-324](#)

s-polarized component, [65](#)

sampler registers, declaring, [296](#)

SamplerState, in DirectX 9, [287](#)

sampling stage register, [286-287](#)

sampling unit, identified by a sampling stage register, [286-287](#)

sapphire, index of refraction as a function of wavelength for, [62](#), [63](#)

_sat modifier, [270](#), [367](#), [368](#)

cannot be used with the sincos macro, [325](#)

with the dp3 instruction, [299](#)

with the dp4 instruction, [300](#)

not supported in mova, [252](#)

saturate modifier, [367](#)

saturated colors, [26-27](#)

saturation, shifting color values to maintain, [28-29](#)

saturation instruction modifier, [270](#)

scalar argument, computing the sine and cosine values for, [264](#)

scalar mask, [364](#)

scalar pipe, [93](#)

scalar pipeline, [369](#)

scalar processing (alpha), [370](#)

scalar product. See [dot product](#)

scalar triple product, [20](#), [21](#)

scalar values

 computing power functions for, [255](#)

 computing the base two logarithm of, [303-304](#)

 raising to a power, [320-321](#)

scalars, [10](#)

scale by two modifier, [359](#)

scaled color values, [27](#)

scenes

 eliminating bright areas in, [30](#)

 making darker with colored light, [26](#)

Schlick's simplification, for the specular exponential term, [51-53](#)

Schwab, John, [142](#)

scuff map, [174](#), [175](#), [176](#), [177](#)

scuff texture, applying, [175](#)

sensitivity, of the human eye to color graduations and edge detection, [23](#)

serial register, setting more than one, [133](#)

`SetDestinationRegisters()` section, of the pseudocode with each vertex instruction, [192](#)

`SetFVF()` function, in DirectX 9, [129](#), [132](#), [133](#)

`SetPixelShader()` function, calling, [132](#), [133](#)

`SetPixelShaderConstants()` function, [136](#)

 calling, [297](#)

setting pixel shader constant registers, [284](#)

SetShaderConstant () function, in DirectX 8, [133](#)

SetSourceRegisters () section, of the pseudocode with each vertex instruction, [192](#)

SetStreamSource () function, [101](#)

SetTexture () function, [108](#), [287](#)

SetTextureStage () function, [291](#)

SetTextureStageState () function, [108](#)

setup codes, for vertex shaders, [150-153](#)

SetVertexDeclaration () function, [122](#), [133](#)

SetVertexShader () function, [128](#)

- calling, [132](#)

- calling with a null argument, [132](#)

- calling with the shader interface from CreateVertexShader (), [133](#)

SetVertexShaderConstant () function, [100](#), [198](#)

SetVertexShaderConstantF () function, for DirectX [9](#), [100](#)

sgc instruction, [261-262](#)

sgn macro, [263](#)

shader buffer, size of, [89](#)

shader buffet, [149-189](#)

shader constants

- in DirectX 9, [134](#)

- setting, [133-136](#), [212](#)

Shader Debugger, from NVIDIA, [140-141](#), [142](#)

shader handle, creating a token array into, [130](#)

shader language

- commands in, [173](#)

- single-instruction multiple-data nature of, [268](#)

shader reference, [191-371](#)

- online version of, [192](#)

shader register window, of Shader Studio, [142](#), [144](#)

shader registers, [106](#)

shader revision history, by DirectX version, [193](#)

shader source, placing in memory, [130](#)

Shader Studio, [142](#), [144](#)

shader token arrays, [130](#)

shader version declaration, [274](#)

ShaderLab tool, [139](#)

shaders, [3](#)

- assembling, [130-132](#)

- basics of getting ready to use, [113-114](#)

- capabilities of, [89](#)

- checking for supported versions of, [116-117](#)

- creating, [130](#)

- existing graphics pipeline and, [90-92](#)

- flavors of, [90](#)

- loading, [132](#)

- optimizing, [90](#)

- philosophy of, [204](#)

- programming, [90](#)

- setting, [132](#)

- using in conjunction with the FFP, [95](#)

shading, [35](#), [50](#)

shading languages

- era of, [4](#)

- higher level, [90](#)

shadowed rays, of the Cook-Torrance model, [74](#)

shift left modifier, [367](#)

shift right modifier, [367](#)

"shininess" parameter, [162](#)

"shininess" value, [46](#)

Siggraph conference, [146](#)

signed scaling modifier, [358](#)

signed textures, [339](#)

signs, computing for each element in a register, [263](#)

silhouette edge shader, [178-180](#)

simulated Fresnel term, adding with minimal effort, [187](#)

`sincos` macro, [264](#), [324-325](#)

sine and cosine, computing for a scalar value, [324-325](#)

sine value, computing for a scalar argument, [264](#)

single-instruction multiple data nature, of the shader language, [268](#)

singly tiled texture, texture coordinates for, [203](#)

sketching style, [82](#), [83](#)

slope distribution function, [75](#)

slope distribution, Gaussian distribution modeling, [75](#)

`slt` instruction, [265-266](#)

Snell's law, [59-60](#)

software vertex processing, specifying, [115](#)

solid color objects, generating, [153](#)

source absolute value, [270](#)

source bias, [357-358](#)

source colors, adding two, [288-289](#)

source invert, [357](#)

source negation, [270-271](#), [356-357](#)

source registers

- corrupted after using `texbem`, [329](#), [330](#)
- multiplying, [316-317](#)
- multiplying element by element, [253](#)
- negating entire, [270-271](#), [356-357](#)
- storing into destination registers, [250-251](#), [315](#)
- subtracting two, [326](#), [327](#)
- swizzling, [361-362](#)

source replication/selection, with pixel shaders, [360](#)

source scale 2X, [359](#)

source signed scaling, [358-359](#)

source swizzle, [269](#)

source texture register modifiers, [362-363](#)

source texture register selectors, [361-362](#)

sparkling, [99](#)

spectrum spreading effect, of refraction, [62-63](#)

- specular and diffuse lighting coefficients, computing, [227-229](#)
- specular color, [34](#)
 - added by pixel shaders, [95](#)
- specular exponential term, Schlick's simplification for, [51-53](#)
- specular highlights, varying with the eye direction, [352](#)
- specular light, [44-49](#)
- specular light reflection, power term approximating the distribution of, [46](#)
- specular lighting, [34](#)
 - Phong's equation for, [45-48](#)
- specular lighting term, generating, [162](#)
- specular reflection and environment mapping performing, [348-350](#)
 - performing assuming a nonconstant view direction, [351-352](#)
- specular reflection, parts of, [78](#)
- specular term
 - for a light shining on a vertex, [172](#)
 - using multiple, [37](#)
- specular vector component, FVF vertex shader register mapping order, [199](#)
- sprites, generating z value for, [343](#)
- static data, vertex buffer strategies for, [125](#)
- stride, of the vertex stream, [119](#)
- stride value, specifying for a DirectX 9 stream, [127](#)
- stylized rendering, [84](#), [85](#)
- sub instruction, [266-267](#), [326](#), [327](#)
- subroutine, indicating the end of, [259](#)
- subtraction, notation for, [11](#)
- surface boundary, interaction of light with, [34](#)
- surface models, physically based, [74-76](#)
- surface normals, affine transformation of, [103](#)
- surface reflections, in the Cook-Torrance surface model, [75](#)
- surface roughness parameter, of the Oren-Nayar model, [57](#)
- surfaces
 - calculating the color reflected off of, [35](#)

interaction with light, [36-38](#)

interaction with light sources, [25](#)

minimizing the shininess of, [34](#)

swizzle modifier

with $m3 \times 2$, [238](#)

with $m3 \times 3$, [240](#)

with $m3 \times 4$, [241](#)

with $m4 \times 3$, [243](#)

with $m4 \times 4$, [245](#)

swizzles, [269](#)

T

table fog, versus vertex fog, [204](#)

TAMs (tonal art maps), [82](#), [83](#), [84](#)

Taylor series expansion, [264](#), [325](#)

temperature correction, for refractive index, [64](#)

temporary $r0$ register, for single color output, [281](#)

temporary registers ($r[n]$), [200](#), [286](#)

tessellation processing, preparing the vertical pipeline for, [122](#)

tex instruction, [326-327](#)

tex3x3pad instruction, using texm3x3 with, [346](#)

texbem instruction, [328-329](#)

texbem1 instruction, [329-330](#)

texcoord instruction, [330-331](#)

compared to texcrd, [332](#)

reading texture coordinates, [203-204](#)

texcrd instruction, [332-333](#)

passing texture coordinates, [285](#)

reading texture coordinates, [203-204](#)

texdepth instruction, [333-334](#)

texdp3 instruction, [334-335](#)

texdp3tex instruction, [336-337](#)

texkill instruction, [337-338](#)

texld instruction, [285](#), [338-340](#)

texldb instruction, [340-341](#)

texldp instruction, [341-342](#)

texm3x2depth instruction, [343-344](#)

texm3x2pad instruction, [344-345](#)

 using texm3x2depth after, [343](#)

 using texm3x2tex instruction with, [345](#)

texm3x2tex instruction, [345-346](#)

texm3x3*, compared to texdp3tex, [337](#)

texm3x3 instruction, [346-347](#), [348](#)

texm3x3pad instruction, [347-348](#)

texm3x3spec instruction, [348-350](#)

 term3x3pad as a setup for, [348](#)

 using constant registers with, [284-285](#)

texm3x3tex instruction, [350-351](#)

texm3x3vspec instruction, [351-352](#)

texreg2ar instruction, [352-353](#)

texreg2gb instruction, [354-355](#)

texreg2rgb instruction, [355-356](#)

text, applying as a decal, [165-166](#)

textdepth instruction, [286](#)

texture

 blending a multicolor object, [166](#)

 as data matrices, [111](#)

 layering on multiple, [111](#)

 sampling, [338-340](#), [341](#)

 sampling using the (a,r) texture coordinates, [352-353](#)

 sampling using the (g, b) texture coordinates, [354](#)

 sampling using the (r,g,b) texture coordinates, [355](#)

 shaders applying two or more, [174](#)

 sorting objects by, [137](#)

 specifying as a render target, [287-288](#)

texture address computation operation, syntax for, [274](#)

texture addressing, for pixel shaders, [276](#)

texture addressing instructions for pixel shaders, [273-274](#)

preceding color-blending instructions in pixel shader versions 1.0 through 1.3, [274](#)

texture coordinate data, passing from the source into the destination register as color data, [332-333](#)

texture coordinate registers

partial precision declaration modifiers for, [369](#)

in PS 1.4, [285](#)

using as extra temporary registers, [286](#)

texture coordinate values, passing as color, [330](#)

texture coordinates

morphing for one of the textures, [175](#)

for pixel shaders, [107](#)

remapping, [353,354,356](#)

for a singly tiled texture, [203](#)

specifying multiple, [126](#)

texture coordinates vector component, FVF vertex shader register mapping order, [199](#)

texture instructions, for pixel shaders, [108](#)

texture operations, of pixel shaders, [94](#)

texture pipeline, dualistic nature of, [93](#)

texture register modifiers, with PS 1.4, [361](#)

texture registers (t[n]), [284-285](#)

for pixel shaders, [282](#)

valid range in PS 1.4 and 2.0, [278](#)

values in, [285](#)

texture sampler register, declaring the use of, [296](#)

texture sampling stage registers (s[n]), [286-287](#)

texture sampling states, pixel shader versions active in, [276, 278](#)

texture stage states, pixel shader versions active in, [276, 277](#)

texture stages

assigning textures to, [108](#)

sampling the color from, [326](#)

`TextureStageState` construct, sampling-specific state provided in, [286-287](#)

three-component cross product, computing, [209-210,294-295](#)

three-component dot product, computing, [214,298-299](#)

TnL (texture and lighting) part of the FFP, replaced by vertex shaders, [92](#)

token array, creating, [120-121](#)

token element-size array, creating, [120](#)

tonal art maps (TAMs), [82,83,84](#)

toon shading style, [84](#)

Torrance-Sparrow model, applied by Oren Nayar, [55](#)

total internal reflection, critical angle for, [61,70](#)

Toy Story, rendering of, [4](#)

traditional 3D hardware-accelerated lighting models, [38-58](#)

transformation matrix, applying to a shape and normals, [102-103](#)

transformations

performing, [97-106](#)

types of, [103-104](#)

transformed vertex position, placing into `oPos` and `r0`, [150](#)

transforming the normal of the vertex, by the inverse transpose model-view-projection matrix, [150](#)

transmitted ray. See [refracted ray](#)

transparent, dielectrics as, [36](#)

transparent-opaque boundary, [36](#)

transparent-transparent boundary, [36](#)

triangle, formed by two vectors, [19](#)

Trowbridge-Reitz model, modeling microfacets as ellipsoids, [76](#)

two-component dot product plus a scalar add, computing, [297](#)

U

unconditional function call, making to the instruction label, [207](#)

uniform scalings, in the world/model matrix, [104](#)

unit cube, [99](#)

unit vectors, [10, 14-15](#)

unpolarized light, fraction transmitted and reflected, [66](#)

unsigned textures, [339](#)

untransformed vertex, creating, [123](#)

V

v-shaped facets, of the Cook-Torrance model, [74-75](#)

vector mask, [364](#)

vector operation, dp3 instruction as a, [299](#)

vector pipe, [93](#)

vector pipeline, [369](#)

vector processing (color), [370](#)

vector product. See [cross product](#)

vector triple product, [20](#)

vectors, [10](#), [13](#)

- angle between, [15](#)

- mathematical properties of, [13-20](#)

- normalizing, [151](#)

- relating across cross products, [20](#)

version 1.0 for pixel shaders. See [PS 1.0 through 1.3](#)

version 1.4 for pixel shaders. See [PS 1.4](#)

version 2.0 for pixel shaders. See [PS 2.0](#)

version 3.0 for pixel shaders, [279](#)

versions, defining for vertex shader code, [267-268](#)

vertex, [12](#)

- setting to a flat color, [101](#)

- specifying the look of, [195](#)

vertex and pixel shaders, techniques requiring processing by both, [181-189](#)

vertex buffer formats, customized, [126](#)

vertex buffers

- allocating dynamic, [125](#)

- associating with streams, [128-129](#)

- creating, [123-126](#)

- sizing, [125](#)

- strategies for dynamic and static data, [125-126](#)

vertex color registers (v[u]), for pixel shaders, [282-283](#)

vertex color shading, [154-155](#)

vertex constant registers, setting in DirectX 8, [133-134](#)

vertex data

- creating well-formed, [128-129](#)
- specifying the source of, [126](#)

vertex data streams, [126-127](#)

vertex declarations, decoupled from vertex shaders in DirectX 9, [132](#)

vertex elements, [117-119](#)

- assembling, [118](#)
- mapping to input registers, [210-211](#)

vertex fog

- applying, [169-172](#)
- indicating intensity, [204](#)
- versus table fog, [204](#)

vertex format flag, [118-119](#)

vertex input, setting up to the shader, [101](#)

vertex instructions, C-like pseudocode included with, [191-192](#)

vertex normals

- vs. face normals, [21-22](#)
- placing a transformed and normalized vertex normal value in r1, [151](#)

vertex operations, [96](#)

vertex positions, transforming into clip-space coordinates, [96](#)

vertex shader boolean constants, setting the value of, [213](#)

vertex shader code, defining the version of, [267-268](#)

vertex shader constant registers, querying the number of, [136](#)

vertex shader constants

- setting in DirectX 8, [133-134](#)
- setting in DirectX 9, [134-135](#)

vertex shader declarator macros, creating the token array, [120-121](#)

vertex shader floating point constants, setting the value of, [211-212](#)

vertex shader input declaration, [97](#)

vertex shader input registers, with corresponding FVF codes, [119](#)

vertex shader instructions, [204-268](#)

vertex shader integer constants, setting the value of, [212-213](#)

vertex shader interface declaration, [119-122](#)

Vertex Shader Reference, [195-271](#)

vertex shader registers, [106](#)

vertex shader version, specifying, [195](#)

vertex shaders, [90](#), [95-107](#), [150-180](#), [195](#)

- ambient shading, [155-157](#)

- assembling, [130-132](#)

- assigning a per vertex color, [154-155](#)

- checking for supported versions of, [116](#)

- common code for, [150-153](#)

- compared to pixel shaders, [108](#)

- components of, [150-153](#)

- constant color, [153-154](#)

- creating and using, [130](#)

- input to, [92](#), [129](#)

- for non-orthogonal transformation matrices, [105-106](#)

- only job of, [92](#)

- for orthogonal transformation matrices, [104-105](#)

- output of, [93](#), [96](#), [129](#)

- overview of, [195](#)

- passing data to, [197](#)

- referencing individual pieces of a register, [273](#)

- replacing sections of the fixed function

- pipeline (FFP), [91-92](#)

- revision history of, [196](#)

- self-contained, [102](#)

- technical overview, [92-93](#)

vertex streams

- associating a token element-size arrays with, [120](#)

- as input to vertex shaders, [92](#), [93](#)

- loading into vertex registers, [129](#)

- specifying more than one, [121](#)

- using multiple, [127](#)

vertex transform and copy setup code, [150](#)

vertical pipeline, preparing for tessellation processing, [122](#)

vertices. See [vertex](#)

view coordinate system, getting the object's vertices into, [98](#)

view direction normalized vector, creating, [151-152](#)

view direction, of specular light, [46](#)

view frustum, [99](#)

view transformation

- multiplying, [98](#)

- output of, [99](#)

viewer, placing the object's vertices in the same space as, [98](#)

viewer position. *See also* [eye position taking](#), [151](#)

viewing parameters, setup of, [99](#)

vn. *See* [input vertex registers](#)

vs instruction, [267-268](#)

W

w element, of the source register, [220](#), [221](#)

warm air-cold air interface, of a road heated by the sun, [61-62](#)

wave theory of light, advanced unintentionally by Poisson, [67](#)

wavelength of light, index of refraction as a function of, [62](#)

Web sites, on DirectX and shader programming, [144-145](#)

well-formed vertex data, [128-129](#)

white light, passing through a prism, [62](#), [63](#)

world and view matrices, concatenating into a single matrix, [98-99](#)

world coordinates

- creating a light directional vector in, [159](#)

- getting to clip coordinates, [98](#)

world/model transformations, consisting only of rotations and translations, [104](#)

world transformation matrix, multiplying an object's vertices by, [98](#)

write masks, [268-269](#)

- destination, [363-367](#)

write only registers, [200](#)

WVP (world-view-projection) matrix

- creating a temporary matrix containing, [100](#)

- loading the concatenated, [100](#)

X-Y

_x2 modifier

added to version 1.4, [281](#)

adding to a register, [359](#)

`_x8` modifier, [280](#)

Z

z-fighting, [99](#)

zFar value, [99](#)

zNear value, [99](#)

List of Figures

Chapter 2: Preliminary Math

[Figure 2.1:](#) The dot product can be used to calculate the angle between two vectors.

[Figure 2.2:](#) The dot product lets you determine the relationship of two vectors.

[Figure 2.3:](#) The cross-product of $\mathbf{a} \times \mathbf{b}$ is vector \mathbf{c} , perpendicular to both.

[Figure 2.4:](#) The right-hand rule.

[Figure 2.5:](#) The cross product lets you determine the area of the parallelogram formed by two vectors.

[Figure 2.6:](#) The scalar triple product lets you determine volumes.

[Figure 2.7:](#) The normal \mathbf{n} of a triangle.

[Figure 2.8:](#) Vertex normals and face normals.

[Figure 2.9:](#) The 1931 CIE diagram shows the gamuts of the eye and the lesser gamuts of output devices.

[Figure 2.10:](#) Multiplying (modulating) color values results in a color equal to or less than (darker) the original two.

[Figure 2.11:](#) Adding colors can result in colors that are outside the displayable range.

[Figure 2.12:](#) The results of three strategies for dealing with the same oversaturated color.

[Figure 2.13:](#) The ColorSpace tool interface.

Chapter 3: Mathematics of Lighting and Shading

[Figure 3.1:](#) Light reflecting from a rough and smooth surface of a conductor.

[Figure 3.2:](#) Light reflecting from a rough and smooth surface of a dielectric showing some penetration.

[Figure 3.3:](#) Subsurface scattering typical of pigment-saturated translucent coatings.

[Figure 3.4:](#) A simple shader to simulate metallic paint: (a) shows the two-tone paint shading pass; (b) shows the specular sparkle shading pass; (c) shows the environment mapping pass; (d) shows the final composite image

[Figure 3.5:](#) Ambient light provides illumination, but no surface details.

[Figure 3.6:](#) Diffuse light decreases as the angle between the light vector and the surface normal increases.

[Figure 3.7:](#) Diffuse shading brings out some surface details.

[Figure 3.8:](#) When diffuse and ambient terms are combined, you get more detail and a more natural-looking scene. The final color is the combination of the ambient and diffuse colors.

[Figure 3.9:](#) Specular light's intensity follows the reflection vector.

[Figure 3.10:](#) The relationship between the normal \mathbf{n} , the light vector \mathbf{v} , the view direction \mathbf{v} , and the reflection vector \mathbf{r} .

[Figure 3.11:](#) Phong's specular term for various values of the "shininess" term. Note that the values never get above 1.

[Figure 3.12:](#) A specular term just shows the highlights.

[Figure 3.13:](#) The half-angle vector is an averaging of the light and view vectors.

[Figure 3.14:](#) Blinn-Phong specular on the left, Phong specular on the right.

[Figure 3.15:](#) A combination of ambient, diffuse, and specular illumination.

[Figure 3.16:](#) A scene with light attenuation. The white sphere is the light position.

[Figure 3.17:](#) Schlick's term for specular looks very much like the more expensive Phong term.

[Figure 3.18:](#) Schlick's vs. Phong's specular terms.

[Figure 3.19:](#) The full moon is an good example of something that doesn't show Lambertian diffuse shading.

[Figure 3.20:](#) The same dirt field showing wildly differing reflection properties.

[Figure 3.21:](#) A soybean field showing differing reflection properties.

[Figure 3.22:](#) Light being reflected and refracted through a boundary.

[Figure 3.23:](#) The refracted ray's angle is less than the incoming ray's.

[Figure 3.24:](#) The critical angle.

[Figure 3.25:](#) Index of refraction as a function of wavelength for quartz.

[Figure 3.26:](#) Index of refraction as a function of wavelength for sapphire.

[Figure 3.27:](#) The wavelength dependence of the index of refraction in action.

[Figure 3.28:](#) The perpendicular nature of the magnetic and electrical fields of a light wave.

[Figure 3.29:](#) Poisson's bright spot.

[Figure 3.30:](#) The reflection and transmission curves for the parallel and perpendicular waves in the air-glass interface.

[Figure 3.31:](#) The averaged reflectance and transmittance values for the air-glass interface.

[Figure 3.32:](#) A glass-air interface shows that we reach a point where all light is reflected internally.

[Figure 3.33:](#) The averaged reflectance and transmittance curves for the glass-air interface.

[Figure 3.34:](#) The three types of surface reflections that can occur in the Cook-Torrance surface model.

[Figure 3.35:](#) A BRDF is a function of an incident and reflection angle and two rotational angles.

[Figure 3.36:](#) A precomputed BRDF texture using the values for gold.

[Figure 3.37:](#) Illustrating different anisotropic features.

[Figure 3.38:](#) The pen-and-ink style.

[Figure 3.39:](#) Engraving style [OSTROMOUKHOV 1999].

[Figure 3.40:](#) Tonal art maps [PRAUN 2001].

[Figure 3.41:](#) Rendered using tonal art maps.

[Figure 3.42:](#) The stylized rendering [GOOCH 1998].

[Figure 3.43:](#) Cel shading found in the game Jet Set Radio Future. The cel shading on the character gives it a unique look.

[Figure 3.44:](#) The 3D scene before the rendering of graftals [Kowalski 1999].

[Figure 3.45:](#) The 3D scene after the addition of graftals [Kowalski 1999].

Chapter 4: Introduction to Shaders

[Figure 4.1:](#) Vertex and pixel shaders replace sections of the fixed function pipeline.

[Figure 4.2:](#) DirectX 8 introduced vertex streams.

[Figure 4.3:](#) Pixel shader hardware can perform different operations simultaneously on a color vector and an alpha scalar.

[Figure 4.4:](#) Vertex operations take input from vertex stream(s) and constants, and place the results in output registers.

[Figure 4.5:](#) Surface normals prior to an affine transformation.

[Figure 4.6:](#) Surface normals following an affine transformation.

[Figure 4.7:](#) Vertex shader (and 2.0+ pixel shader) registers are made of four float vector elements.

[Figure 4.8:](#) Pixel shaders take color inputs and texture coordinates to generate a single output color value.

[Figure 4.9:](#) Pixel shader registers prior to version 2.0 consisted of a four-element vector made of (at least) 8-bit floating point elements; 2.0 pixel shaders are 32-bit floats.

Chapter 5: Shader Setup in DirectX

[Figure 5.1:](#) Vertex streams get loaded into vertex registers by a render call.

Chapter 6: Shader Tools and Resources

[Figure 6.1:](#) RenderMonkey in action.

[Figure 6.2:](#) NVIDIA Effects Browser.

[Figure 6.3:](#) NVIDIA Shader Debugger.

[Figure 6.4:](#) The D3D Shader Debugger that shipped with the DirectX 9 SDK.

[Figure 6.5:](#) Shader Studio.

[Figure 6.6:](#) ColorSpace lets you see the effects of clamping vs. clipping vs. scaling when two colors oversaturate.

Chapter 7: Shader Buffet

[Figure 7.1:](#) Constant vertex color gives color to an object but no hint of depth.

[Figure 7.2:](#) Assigning a per vertex color gives more control at the expense of more stream data.

[Figure 7.3:](#) Ambient shading gives color plus shading, but the unilluminated areas are black.

[Figure 7.4:](#) Infinite lights are the most common type of lights. The light vector for all vertices is the same.

[Figure 7.5:](#) A positional light means calculating a unique light direction vector for each vertex. For light sources close to a surface, this overhead is sometimes necessary to get the light looking correct.

[Figure 7.6:](#) Blinn-Phong specular on the left and then the ambient, diffuse, and specular terms combined on the right.

[Figure 7.7:](#) The decal we're going to wrap the object with.

[Figure 7.8:](#) Decal texturing slaps a texture onto a surface with no other coloring effects.

[Figure 7.9:](#) To blend a texture with a surface color, you'll need to modulate the texture color.

[Figure 7.10:](#) Fog is done after the texture stage. In vertex shaders, you can just set the vertex fog intensity. Here, the fog is red and the pin is standing out of the fog.

[Figure 7.11:](#) Point shaders are somewhat limited since point size has only a fixed maximum value.

[Figure 7.12:](#) The bowling pin texture map.

[Figure 7.13:](#) The scuff map.

[Figure 7.14:](#) The pin with the texture map and the scuff map added on top of it.

[Figure 7.15:](#) A simple silhouette shader.

[Figure 7.16:](#) Cartoon shading using traditional lighting equations and a three-stage color graduation.

[Figure 7.17:](#) Taking a bump map to generate normal perturbations that look like real geometry except at the edges of the object.

[Figure 7.18:](#) A simplified Fresnel shader used to generate a shiny specular when an object is illuminated from behind.

[Figure 7.19:](#) The Fresnel shader output added to the ambient, diffuse, and specular output.

Part II: Pixel Shader Reference

[Figure 8.1:](#) Constant, texture addressing, and arithmetic instructions for pixel shaders 1.0 through 1.3.

[Figure 8.2:](#) Phase instruction in pixel shaders 1.4.

[Figure 8.3:](#) Pixel shader register construction.

List of Tables

Chapter 2: Preliminary Math

[Table 2.1:](#) OPERAND NOTATION

[Table 2.2:](#) OPERATOR NOTATION

Chapter 5: Shader Setup in DirectX

[Table 5.1:](#) FFP Vertex Shader Register Mapping

Part I: Vertex Shader Reference

[Table 8.1:](#) VVF Vertex Shader Register Mapping

Part II: Pixel Shader Reference

[Table 8.1:](#) Directx 9 Texture Stage State

[Table 8.2:](#) Directx 9 Texture Sampling State

[Table 8.3:](#) Vertex Color Register Properties

[Table 8.4:](#) Constant Register Propertis

[Table 8.5:](#) Texture Register Properties

[Table 8.6:](#) Temporary Register Properties

List of Sidebars

Chapter 2: Preliminary Math

[WHY YOU MIGHT WANT 128-BIT COLOR](#)

Chapter 3: Mathematics of Lighting and Shading

POISSON'S BRIGHT SPOT