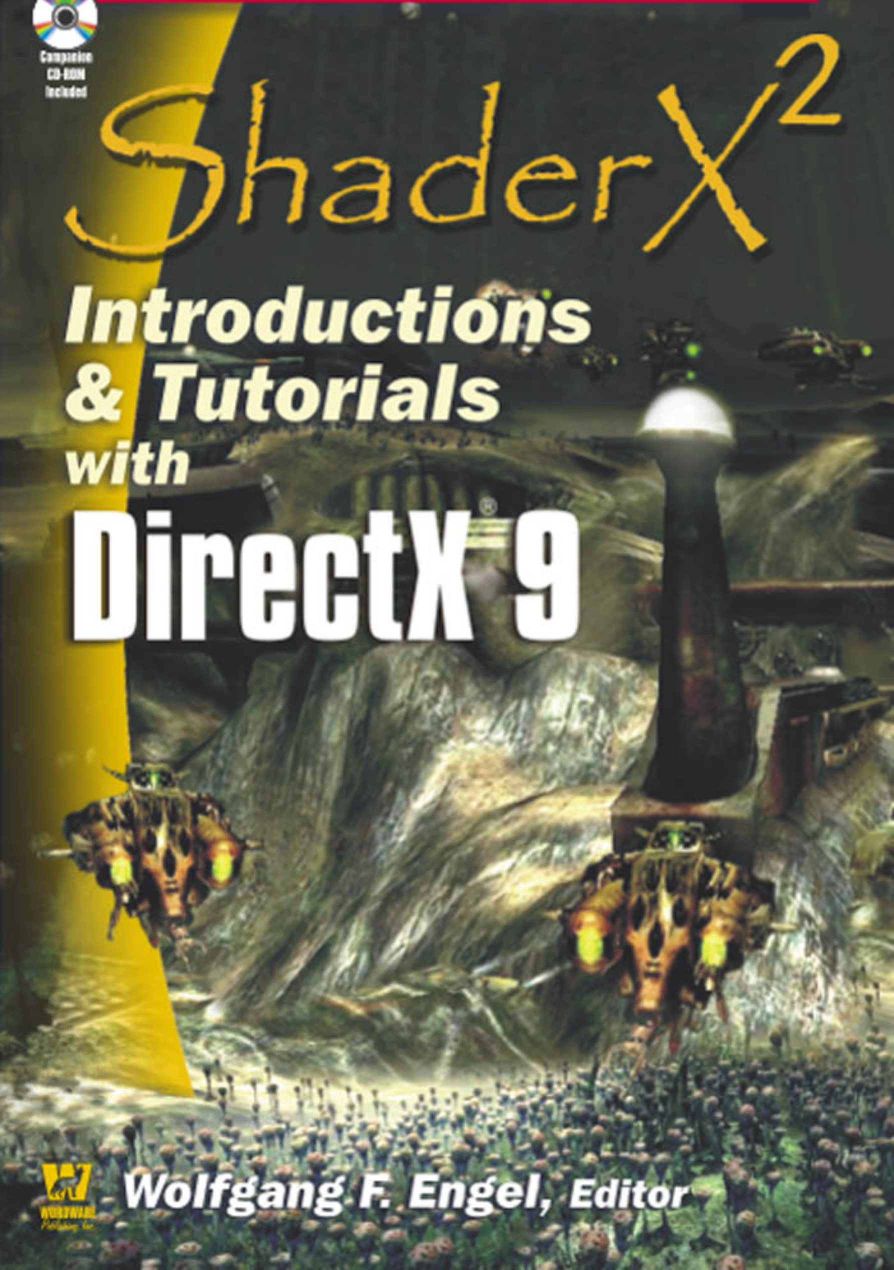
WORDWARE GAME DEVELOPER'S LIBRARY

Companion CD-ROM Included

ShaderX²

Introductions & Tutorials with DirectX® 9



Wolfgang F. Engel, Editor

ShaderX^2: Introductions & Tutorials with DirectX 9

Edited by

Wolfgang F. Engel

ShaderX^2: Introductions & Tutorials with DirectX 9

Edited by
Wolfgang F. Engel

Wordware Publishing, Inc.

Library of Congress Cataloging-in-Publication Data

ShaderX^2 : introductions and tutorials with DirectX 9 / edited by
Wolfgang Engel.
p. cm.
Includes bibliographical references and index.
ISBN 1-55622-902-X (paperback, companion CD-ROM)
1. Computer games--Programming. 2. Three-dimensional display systems.
3. DirectX. I. Engel, Wolfgang F.
QA76.76.C672S47 2003
794.8'16693--dc22 2003016311
CIP

2320 Los Rios Boulevard
Plano, Texas 75074

Printed in the United States of America

ISBN 1-55622-902-X

10 9 8 7 6 5 4 3 2 1
0307

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

(972) 423-0090

Articles

Contents

Preface

After the tremendous success of *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, I planned to do another book with an entirely new set of innovative ideas, techniques, and algorithms. The call for authors led to many proposals from nearly 80 people who wanted to contribute to the book. Some of these proposals featured introductory material and others featured much more advanced themes. Because of the large amount of material, I decided to split the articles into introductory pieces that are much longer but explain a lot of groundwork and articles that assume a certain degree of knowledge. This idea led to two books:

ShaderX^2: Introductions & Tutorials with DirectX 9
ShaderX^2: Shader Programming Tips & Tricks with DirectX 9

The first book (this one) helps the reader get started with shader programming, whereas the second book features tips and tricks that an experienced shader programmer will benefit from.

As with *Direct3D ShaderX*, Javier Izquierdo Villagrán (nurbs1@jazzfree.com) prepared the drafts for the cover design of both books with in-game screen shots from Aquanox 2, which were contributed by Ingo Frick, the technical director of Massive Development.

A number of people have enthusiastically contributed to both books:

Wessam Bahnassi
Andre Chen
Muhammad Haggag
Kenneth L. Hurley
Eran Kampf

Brian Peltonen
Mark Wang

Additionally, the following *ShaderX*^{2} authors proofread several articles each:

Dean Calver
Nicolas Capens
Tom Forsyth
Shawn Hargreaves
Jeffrey Kiel
Hun Yen Kwoon
Markus Nuebel
Michal Valient
Oliver Weichhold

These great people spent a lot of time proofreading articles, proposing improvements, and exchanging e-mails with other authors and myself. Their support was essential to the book development process, and their work led to the high quality of the books. Thank you!

Another big thank you goes to the people in the Microsoft Direct3D discussion group (http://DISCUSS.MICROSOFT.COM/archives/DIRECTXDEV.html). They were very helpful in answering my numerous questions.

As with *Direct3D ShaderX*, there were some driving spirits who encouraged me to start this project and hold on through the seven months it took to complete it:

Dean Calver (Eclipse)
Jason L. Mitchell (ATI Research)
Natasha Tatarchuk (ATI Research)
Nicolas Thibieroz (PowerVR)
Carsten Wenzel (Crytek)

Additionally, I have to thank Thomas Rued from DigitalArts for inviting me to the Vision Days in Copenhagen, Denmark, and for the great time I had there. I would like to thank Matthias Wloka and Randima Fernando from nVidia for lunch at GDC 2003. I had a great time.

As usual, the great team at Wordware made the whole project happen: Jim Hill, Wes Beckwith, Heather Hill, Beth Kohler, and Paula Price took over after I sent them hundreds of megabytes of data.

There were other numerous people involved in this book project that I have not mentioned. I would like to thank them here. It was a pleasure working with so many talented people.

Special thanks goes to my wife, Katja, and our daughter, Anna, who spent a lot of evenings and weekends during the last seven months without me, and to my parents, who always helped me to believe in my strength.

— Wolfgang F. Engel

P.S.: Plans for an upcoming project named *ShaderX*^{3} are already in progress. Any comments, proposals, and suggestions are highly welcome (wolf@shaderx.com).

About the Authors

Kristof Beets (kristof.beets@powervr.com)
Kristof took his first steps in the 3D world by running a technical 3D fan site, covering topics such as the differences between traditional and tile-based rendering technologies. This influenced his electrical engineering studies in such a way that he wrote his thesis about wavelet compression for textures in Direct3D, a paper that won the Belgian Barco Prize. He continued his studies, obtaining a master's degree in artificial intelligence. In the meantime he worked as a technical editor for Beyond3D, writing various technical articles about 3D hardware, effects, and technology. As a freelance writer he wrote the "FSAA Explained" document for 3Dfx Interactive to explain the differences between various types of full-screen anti-aliasing. This document resulted in a full-time job offer at 3Dfx. Currently he is working as a developer relations engineer for PowerVR Technologies, which includes research into new graphical algorithms and techniques.

Aaron Burton (aaron.burton@powervr.com)
Aaron has been a developer relations engineer at PowerVR Technologies since he received his Honours degree in information systems engineering in 1998. His first computer was a VIC 20, though his fascination for 3D graphics began with the Atari ST. At PowerVR he has been able to indulge this interest by developing a variety of demos, benchmarks, and debug/performance tools, and supporting developers in creating faster and better games. When he's not climbing, he works on projects such as ray-tracing and real-time 3D demos.

Gim Guan Chua (ggchua@mail.com)
Blackbox Technologies is an experimental platform for innovative usage of interactive 3D. It uses OpenGL and a component-based

software architecture to add programmable behaviors (and properties) to generic 3D objects, and lets them exist without a 2D window frame. Creator Gim Guan Chua is a freelance graphics programmer based in Singapore. He has been developing 3D applications for more than six years and likes to dabble in 3D modeling in his spare time. His web site is http://toybox.150m.com.

Wolfgang F. Engel (wolfgang.engel@shaderx.com)
Wolfgang is the editor and co-author of *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, the author of *Beginning Direct3D Game Programming*, and a co-author of *OS/2 in Team*, for which he contributed the introductory chapters on OpenGL and DIVE. Wolfgang has written several articles in German journals on game programming and many online tutorials that were published on www.gamedev.net and his own web site, www.direct3d.net. During his career in the game industry he built up two game development units with four and five people that published six online games for the biggest European TV show, *Wetten das..?*. As a member of the board or as a CEO of different companies, he was responsible for several game projects.

Hun Yen Kwoon (ykhun@PacketOfMilk.com)
Hun Yen Kwoon is an electrical engineering graduate from the National University of Singapore. After spending 16 years in the education system, he decided he wanted to be a programmer more than an electrical engineer. He promptly joined an IT business solutions company and developed an online debit system for a local bank before realizing that Java is boring. He is now working as a software engineer with Silicon Illusions in Singapore. His work involves 3D visualization software engineering, SSE/SSE2, OpenGL, and Direct3D. Recently he has also been fiddling with game networking architecture and dead-reckoning techniques. What kind of work can be more exciting?

Jason L. Mitchell (JasonM@ati.com)
Jason is the team lead of the 3D Application Research Group at ATI Research, makers of the Radeon family of graphics processors. Working on the Microsoft campus in Redmond, Jason has worked with Microsoft for several years to define key new Direct3D

features. Prior to working at ATI, Jason did work in human eye tracking for human interface applications at the University of Cincinnati, where he received his master's degree in electrical engineering in 1996. He received a bachelor's degree in computer engineering from Case Western Reserve University in 1994. In addition to this book's article on HLSL programming and an article on advanced image processing for *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9*, Jason has written for the *Game Programming Gems* books, *Game Developer* magazine, Gamasutra.com, and academic publications on graphics and image processing. He regularly presents at graphics and game development conferences around the world. His home page can be found at http://www.pixelmaven.com/jason/.

Markus Nuebel (markus.nuebel@t-online.de)
Markus holds a master's degree in computer science and has been programming professionally for over eight years. Several years ago he discovered his passion for graphics and game programming. He has been into shader programming since nVidia launched cg and spends every free minute expanding his knowledge of interesting graphic programming algorithms.

Craig Peeper (CraigP@microsoft.com)
Craig Peeper is the lead developer for D3DX at Microsoft and has been on the team since DirectX 7. D3DX provides user-mode functionality for Direct3D, including mesh optimization, texture processing, and the High Level Shading Language compiler/runtime. Prior to his work on D3DX, Craig worked in Microsoft Graphics Research.

Natasha Tatarchuk (Natasha@ati.com)
Natasha Tatarchuk is a software engineer working in the 3D Application Research Group at ATI Research, where she is the programming lead for the RenderMonkey IDE project. She has been in the graphics industry for over six years, working on 3D modeling applications and scientific visualization prior to joining ATI. Natasha graduated from Boston University with a bachelor's degree in

computer science, a bachelor's degree in mathematics, and a minor in visual arts.

Nicolas Thibieroz (nicolas.thibieroz@powervr.com)
Like many kids of his generation, Nicolas Thibieroz discovered video games on the Atari VCS 2600. He quickly became fascinated by the mechanics behind those games, and started programming on the C64 and Amstrad CPC before moving on to the PC world. Nicolas realized the potential of real-time 3D graphics while playing Ultima Underworld. This game inspired him in such a way that both his school placement and final year projects were based on 3D computer graphics. After obtaining a bachelor's degree in electronic engineering in 1996 he joined PowerVR Technologies where he is now responsible for developer relations. His duties include supporting game developers, writing test programs and demos, and generally keeping up to date with the latest 3D technology.

Michal Valient (valiant@host.sk)
Michal received a degree in computer graphics at the Faculty of Mathematics, Physics and Informatics, Comenius University, Slovakia, in June 2003 after finishing his master's thesis about special effects for computer games. He is continuing with Ph.D. studies at the university. Previously he worked as director of development for a bigger company, but the call of real-time rendering was too strong and now he is fully concentrated in this area. Michal currently works for Caligari Corporation. His home page is at http://www.dimension3.host.sk.

Introduction

This book is a collection of articles that explain the foundations of shader programming, from the High Level Shading Language and version 3.0 shader models to shadow mapping and stencil shadow volumes. The following provides a brief overview of these articles:

Jason L. Mitchell and Craig Peeper, one of the creators of HLSL and the compiler, have written the best introduction to HLSL there is in "Introduction to the DirectX High Level Shading Language." Because it comes from the official source, this article covers everything that an HLSL programmer needs and a lot more.

The vs_3_0 and ps_3_0 shader models will be available in third-generation shader graphics hardware. These shader versions are much more flexible and powerful than the previous versions, offering vertex texturing capabilities, predication, static and dynamic flow control, vertex stream frequency, and much more. Nicolas Thibieroz, Kristof Beets, and Aaron Burton from PowerVR have written an introduction to this shader model that explains every new feature and includes a source snippet.

Michal Valient's article "Advanced Lighting and Shading with Direct3D 9" covers some more advanced lighting models including Phong, Oren-Nayar, and Cook-Torrance. He implements these algorithms with ps_1_4, ps_2_0, ps_3_0, and HLSL. This is the most extensive treatment of this topic available.

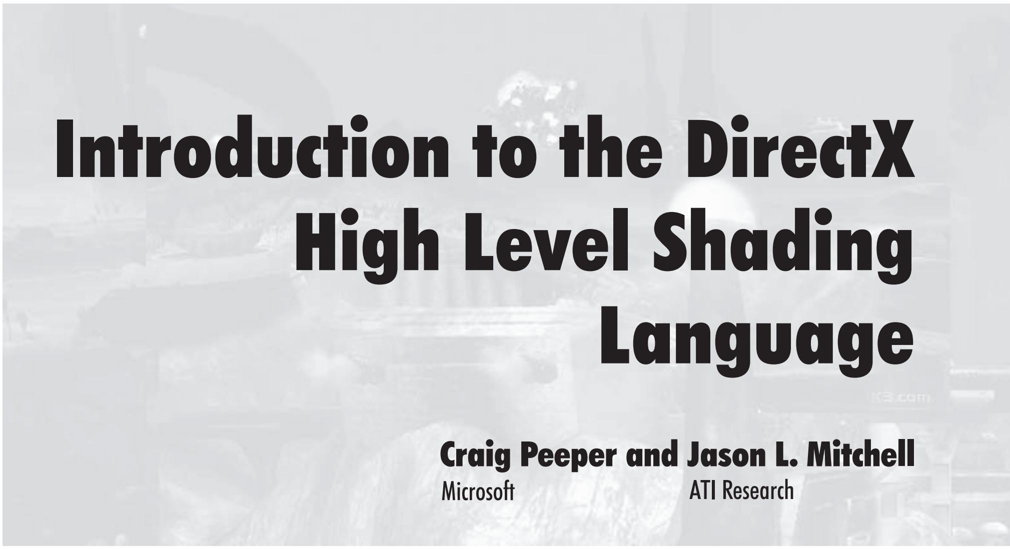
There are several different ways to use fog to produce a specific mood in games. Markus Nuebel shows all possible ways to implement fog in a way that is easy to understand. The six example programs make using fog as easy as possible.

Michal Valient's second contribution is the article "Shadow Mapping with Direct3D 9." With the release of DirectX 9 and its floating-point textures, using shadow maps for shadows leads to a much better visual experience. Michal shows how to implement shadow mapping in the most efficient and most flexible way and gives tips on how to debug an application.

The most comprehensive treatment of shadow volumes available is contained in the article “The Theory of Stencil Shadow Volumes” by Hun Yen Kwoon. It covers every aspect of the various ways of programming shadow volumes. Six example programs give you a head start on implementing shadow volumes in minutes.

ATI’s RenderMonkey is a shader development tool that helps to reduce the workload of programmers and artists. One of its creators, Natalya Tatarchuk, explains how to use it and discusses its feature set.

A topic that is seldom covered elsewhere is the necessity of creating geometric data in the art pipeline that is shader-friendly. Gim Guan Chua has written an article describing this task and provides a step-by-step explanation of how to do it.



Introduction to the DirectX High Level Shading Language

Craig Peeper and Jason L. Mitchell
Microsoft ATI Research

Introduction

One of the most empowering new components of DirectX 9 is the High Level Shading Language (HLSL). Using this standard high-level language, shader writers can think at the algorithm level while implementing shaders rather than worry about meddlesome hardware details, such as register allocation, register read-port limits, instruction co-issuing, and so on. In addition to freeing the developer from hardware details, the HLSL also has all of the usual advantages of a high-level language, such as easy code reuse, improved readability, and the presence of an optimizing compiler. Many of the chapters in this book and in *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9* (also from Wordware Publishing) utilize shaders that are written in HLSL. As a result, it will be much easier for you to understand and work with those shaders after reading this introductory chapter.

In this chapter, we outline the basic structure of the language itself, as well as strategies for integrating HLSL shaders into your application.

A Simple Example

Before presenting an exhaustive description of the HLSL, let's first have a look at one HLSL vertex shader and one HLSL pixel shader taken from an application that renders simple procedural wood. The first HLSL shader shown below is a simple vertex shader:

```
float4x4 view_proj_matrix;
float4x4 texture_matrix0;

struct VS_OUTPUT
{
   float4 Pos     : POSITION;
   float3 Pshade  : TEXCOORD0;
};


VS_OUTPUT main (float4 vPosition : POSITION)
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   // Transform position to clip space
   Out.Pos = mul (view_proj_matrix, vPosition);

   // Transform Pshade
   Out.Pshade = mul (texture_matrix0, vPosition);

   return Out;
}
```

The first two lines of this shader declare a pair of 4×4 matrices called `view_proj_matrix` and `texture_matrix0`. Following these global-scope matrices, a structure is declared. This `VS_OUTPUT` structure has two members: a `float4` called `Pos` and a `float3` called `Pshade`.

The `main` function for this shader takes a single `float4` input parameter and returns a `VS_OUTPUT` structure. The `float4` input `vPosition` is the sole input to the shader, while the returned `VS_OUTPUT` struct defines this vertex shader's output. For now, don't worry about the `POSITION` and `TEXCOORD0` keywords following these parameters and structure members. These are called *semantics*, and their meaning is discussed later in this chapter.

Looking at the actual code body of the `main` function, you can see that an intrinsic function called `mul` is used to multiply the input `vPosition` vector by the `view_proj_matrix` matrix. This intrinsic is commonly used in vertex shaders to perform vector-matrix multiplication. In this case, `vPosition` is treated as a column vector, since it is the second parameter to `mul`. If the `vPosition` vector were the first parameter to `mul`, it would be treated as a row vector. (The `mul` intrinsic and other intrinsics are discussed in more detail later in the chapter.) Following the transformation of the input position `vPosition` to clip space, `vPosition` is multiplied by another matrix called `texture_matrix0` to generate a 3D texture coordinate. The results of both of these transformations have been written to members of a `VS_OUTPUT` structure, which is returned. A vertex shader must always output a clip-space position at a minimum. Any additional values that are output from the vertex shader are interpolated across the rasterized polygon and available as inputs to the pixel shader. In this case, the 3D `Pshade` is passed from the vertex to the pixel shader via an interpolator.

Below, we see a simple HLSL procedural wood pixel shader. This pixel shader, which is written to work with the vertex shader that we just described, will be compiled for the ps_2_0 target.

```
float4 lightWood; // xyz == Light Wood Color
float4 darkWood;  // xyz == Dark Wood Color
float  ringFreq;  // ring frequency

sampler PulseTrainSampler;

float4 hlsl_rings (float4 Pshade : TEXCOORD0) : COLOR
{
   float scaledDistFromZAxis = sqrt(dot(Pshade.xy, Pshade.xy)) * ringFreq;
   float blendFactor = tex1D (PulseTrainSampler, scaledDistFromZAxis);
   return lerp (darkWood, lightWood, blendFactor);
}
```

The first few lines of this shader are the declaration of a pair of floating-point 4-tuples and one scalar `float` at global scope. Following these variables, a sampler called `PulseTrainSampler` is declared. Samplers are discussed in more detail later in the chapter, but for now you can just think of a sampler as a window into video memory with an associated state defining things like filtering and texture coordinate addressing modes. With variable and sampler declarations out of the way, we can move on to the body of the shader code. You can see that there is one input parameter called `Pshade`, which is interpolated across the polygon. This is the value that was computed at each vertex by the vertex shader above. In the pixel shader, the Cartesian distance from the shader-space *z*-axis is computed, scaled, and used as a 1D texture coordinate to access the texture bound to the `PulseTrainSampler`. The scalar color that is returned from the `tex1D()` sampling function is used as a blend factor to blend between the two constant colors (`lightWood` and `darkWood`) declared at the global scope of the shader. The 4D vector result of this blend is the final output of the pixel shader. All pixel shaders must return a 4D RGBA color at a minimum. We discuss additional optional pixel shader outputs later in the chapter.

Assembly Language and Compile Targets

Now that we have seen a few HLSL shaders, we can discuss briefly how the language relates to Direct3D, D3DX, assembly shader models, and your application. Shaders were first added to Direct3D in DirectX 8.0. At that time, several virtual shader machines were defined — each roughly corresponding to a particular graphics processor produced by each of the top 3D graphics hardware vendors. For each of these virtual shader machines, an assembly language was designed. In DirectX 8.0 and DirectX 8.1, programs written to these shader models (named vs_1_1 and ps_1_1 through ps_1_4) were relatively short and generally written by developers directly in the appropriate assembly language. As shown on the left side of Figure 1, the application passes this human-readable assembly language code to the D3DX library via `D3DXAssembleShader()` and gets back a binary representation of the shader, which would in turn be passed to Direct3D via `Create-PixelShader()` or `CreateVertexShader()`. For more on the details of the legacy assembly shader models, please refer to the many resources available online and offline, including *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks* and the DirectX SDK.

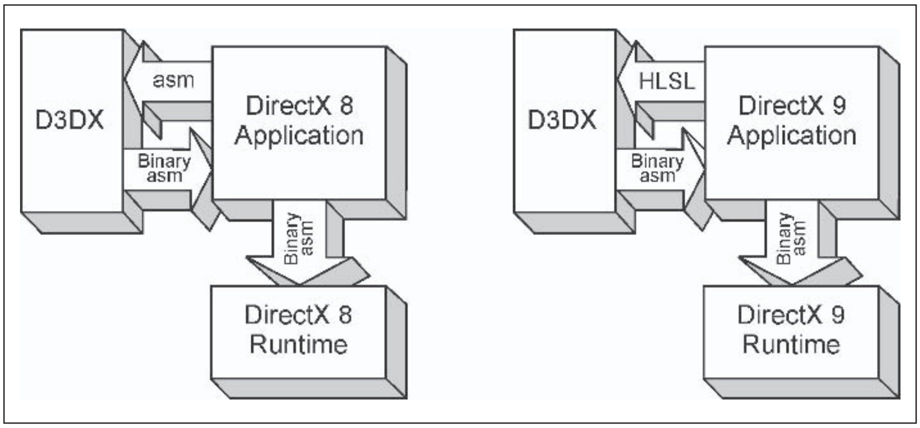


Figure 1: Use of D3DX for assembly and compilation in DirectX 8 and DirectX 9

As shown on the right side of Figure 1, the situation in DirectX 9 is very similar in that the application passes an HLSL shader to D3DX via the `D3DXCompileShader()` API and gets back a binary representation of the compiled shader, which is in turn passed to Direct3D via `CreatePixelShader()` or `CreateVertexShader()`. The binary asm code that's generated is only a function of the compile target chosen, not the specific graphics device in the user's or developer's system. That is, the binary asm that is generated is vendor-neutral and will be the same no matter where you compile or run it. In fact, the Direct3D runtime itself does not know anything about HLSL — only the binary assembly shader models. This is nice because it means that the HLSL compiler can be updated independently of the Direct3D runtime. In fact, between press time and the release of the first printing of this book in late summer 2003, Microsoft plans to release a DirectX SDK update, which will contain an updated HLSL compiler.

In addition to the development of the HLSL compiler in D3DX, DirectX 9 also introduced additional assembly-level shader models to expose the functionality of the latest generation of 3D graphics hardware. Application developers can feel free to work directly in the assembly languages for these new models (vs_2_0, vs_3_0, ps_2_0, and ps_3_0), but we expect most developers to move wholesale to HLSL for shader development.

Hardware Realities

Of course, just because you can write an HLSL program to express a particular shading algorithm doesn't mean that it will run on a given piece of hardware. As we discussed earlier, an application calls D3DX to compile an HLSL shader to binary asm via the `D3DXCompileShader()` API. One of the parameters to this API entrypoint is a parameter that defines which of the assembly language models (or *compile targets*) the HLSL compiler should use to express the final shader code. If an application is doing HLSL shader compilation at run time (as opposed to offline), the application could examine the capabilities of the Direct3D device and select the compile target to match. If the algorithm expressed in the HLSL shader is too complex to execute on the selected compile target, compilation *will fail*. This means that while HLSL is a huge benefit to shader development, it does not free developers from the realities of shipping games to a target audience that owns graphics devices of varying capabilities. As a game developer, you still have to manage a tiered approach to your visuals, writing better shaders for better graphics cards and more basic versions for older cards. With well-written HLSL, however, this burden can be eased significantly.

Compilation Failure

As mentioned above, failure of a given HLSL shader to compile for a particular compile target is an indication that the shader is too complex for the compile target. This can mean that the shader either requires too many resources or it requires some capability, such as dynamic branching, that is not supported by the chosen compile target. For example, an HLSL shader could be written to access a given texture map six times in a shader. If this shader is compiled for the ps_1_1 compile target, compilation will fail since the ps_1_1 model supports only four textures. Another common source of compilation failure is exceeding instruction count of the chosen compile target. An algorithm expressed in HLSL may simply require too many instructions to be executed by a given compile target.

It is important to note that the choice of compile target does not restrict the HLSL syntax that a shader writer can use. For example, a shader writer can use for loops, subroutines, if-else statements, etc., and still compile for targets that don't natively support looping, branching, or if-else statements. In such cases, the compiler will unroll loops, inline function calls, and execute both branches of an if-else statement, selecting the proper result based upon the original value used in the if-else statement. Of course, if the resulting shader is too long or otherwise exceeds the resources of the compile target, compilation will fail.

The Command-line Compiler — fxc

Rather than compile HLSL shaders using D3DX on the customer's machine at application load time or at first use, many developers choose to compile their shaders from HLSL to binary asm before they even ship. This keeps their HLSL source away from prying eyes. It also ensures that all of the shaders their app runs will have gone through their internal quality assurance process. A convenient utility that allows developers to compile shaders offline is the fxc command-line compiler, which is provided in the DirectX 9 SDK. This utility has a number of convenient options that you can use to not only compile your shaders on the command line but also generate disassembled code for the specified compile target. Studying the disassembled output can be very educational during development if you want to optimize your shaders or just generally get to know the virtual shader machine's capabilities at a more detailed level. These command-line options are summarized in the following table.

Command-line Option	Description
-T *target*	compile target (default: vs_2_0)
-E *name*	entrypoint *name* (default: main)
-Od	disable optimizations
-Vd	disable validation
-Zi	enable debugging information
-Zpr	pack matrices in row-major order
-Zpc	pack matrices in column-major order
-Fo *file*	output object file
-Fc *file*	output listing of generated code
-Fh *file*	output header containing generated code
-D *id* = *text*	define macro
-nologo	suppress copyright message

Now that you understand the context in which the HLSL compiler can be used for shader development, let's discuss the actual mechanics of the language. As we progress, it is important to keep the notion of a *compile target* and the varying capabilities of the underlying assembly shader models in mind.

Language Basics

Now that you have a sense of what HLSL vertex and pixel shaders look like and how they interact with the low-level assembly shaders, we can discuss some of the details of the language itself.

Keywords

Keywords are predefined identifiers that are reserved for the HLSL language and cannot be used as identifiers in your program. Keywords marked with an asterisk (*) are case insensitive.

```
asm*          bool          compile        const
decl*         do            double         else
```

`extern`	`false`	`float`	`for`
`half`	`if`	`in`	`inline`
`inout`	`int`	`matrix*`	`out`
`pass*`	`pixelshader*`	`return`	`sampler`
`shared`	`static`	`string*`	`struct`
`technique*`	`texture*`	`true`	`typedef`
`uniform`	`vector*`	`vertexshader*`	`void`
`volatile`	`while`		

The following keywords are currently unused but reserved for potential future use:

`auto`	`break`	`case`	`catch`
`char`	`class`	`compile`	`const`
`const_cast`	`continue`	`default`	`delete`
`dynamic_cast`	`enum`	`explicit`	`friend`
`goto`	`long`	`mutable`	`namespace`
`new`	`operator`	`private`	`protected`
`public`	`register`	`reinterpret_cast`	`short`
`signed`	`sizeof`	`static_cast`	`switch`
`template`	`this`	`throw`	`try`
`typename`	`union`	`unsigned`	`using`
`virtual`			

Data Types

The HLSL has support for a variety of data types, from simple scalars to more complex types, such as vectors and matrices.

Scalar Types

The language supports the following scalar data types:

Data Type	Representable Values
`bool`	true or false
`int`	32-bit signed integer
`half`	16-bit floating-point value
`float`	32-bit floating-point value
`double`	64-bit floating-point value

If you are already familiar with the assembly-level programming models, you should know that graphics processors do not currently have native support for all of these data types. As a result, integers may need to be emulated using floating-point hardware. This means that integer operations that go outside the range of integers that can be expressed as floats on these platforms are not guaranteed to function as expected. Additionally, not all target platforms have native support for half or double values. If the target platform does not, these will be emulated using float.

Vector Types

You will often find yourself declaring vector variables in your HLSL shaders. There are a variety of ways that these vectors can be declared, including the following:

Vector	Declared as
`vector`	A vector of dimension 4; each component is of type float.
`vector`<*type*, *size*>	A vector of dimension *size*; each component is of scalar type *type*.

The most common way that you see shader authors declare vectors, however, is by using the name of a type followed by an integer from 2 to 4. To declare a 4-tuple of `floats`, for example, you could use any of the following vector declarations:

```
float4 fVector0;
float  fVector1[4];
vector fVector2;
vector <float, 4> fVector3;
```

To declare a 3-tuple of `bools`, for example, you could use any of the following declarations:

```
bool3 bVector0;
bool  bVector1[3];
vector <bool, 3> bVector2;
```

Once you have defined a vector, you may access its individual components by using the array access syntax or a swizzle. In the swizzle case, the components must come from either the $\{x, y, z, w\}$ or $\{r, g, b, a\}$ namespace (but not both). For example:

```
float4 pos = {3.0f, 5.0f, 2.0f, 1.0f};
float  value0 = pos[0]; // value0 is 3.0f
float  value1 = pos.x;  // value1 is 3.0f
float  value2 = pos.g;  // value2 is 5.0f
float2 vec0   = pos.xy; // vec0 is {3.0f, 5.0f}
float2 vec1   = pos.ry; // INVALID because of bad swizzle
```

It should be noted that the ps_2_0 and lower pixel shader models do not have native support for arbitrary swizzles. Hence, concise high-level code that uses swizzles can result in fairly nasty binary asm when compiling to these targets. You should familiarize yourself with the native swizzles available in these assembly models.

Matrix Types

Another very common type of variable that you will find yourself using in HLSL shaders is matrices, which are 2D arrays of data. Like scalars and vectors, matrices may be composed of any of the basic data types: `bool`, `int`, `half`, `float`, or `double`. Matrices may be of any size, but you will typically find shader writers using matrices with up to four rows and columns. Recall that the example vertex shader shown at the beginning of the chapter declared two 4×4 `float` matrices at global scope:

```
float4x4 view_proj_matrix;
float4x4 texture_matrix0;
```

Naturally, other dimensions of matrices can be used. For example, we could declare a floating-point matrix with three rows and four columns in a variety of ways:

```
float3x4           mat0;
matrix<float, 3, 4> mat1;
```

Like vectors, the individual elements of matrices can be accessed using array or structure/swizzle syntax. For example, the following array indexing syntax can be used to access the top-left element of the matrix `view_proj_matrix`:

```
float fValue = view_proj_matrix[0][0];
```

There is also a structure syntax defined for access to and swizzling of matrix elements. For zero-based row-column position, you can use any of the following:

_m00, _m01, _m02, _m03
_m10, _m11, _m12, _m13
_m20, _m21, _m22, _m23
_m30, _m31, _m32, _m33

For one-based row-column position, you can use any of the following:

_11, _12, _13, _14
_21, _22, _23, _24
_31, _32, _33, _34
_41, _42, _43, _44

Matrices can also be accessed using array notation. For example:

```
float2x2 fMat = {3.0f, 5.0f,  // row 1
                 2.0f, 1.0f}; // row 2

float  value0 = fMat[0];      // value0 is 3.0f
float  value1 = fMat._m00;    // value1 is 3.0f
float  value2 = fMat._12      // value2 is 5.0f
float  value3 = fMat[1][1]    // value3 is 1.0f
float2 vec0   = fMat._21_22;  // vec0 is {2.0f, 1.0f}
float2 vec1   = fMat[1];      // vec1 is {2.0f, 1.0f}
```

Type Modifiers

There are a couple of optional type modifiers in the HLSL that you may want to use in your shaders. The familiar `const` type modifier is used to specify a variable whose value cannot be changed by the shader code. Using such a variable on the left side of an assignment (i.e., as an *lval*) will result in a compilation error.

The row_major and col_major type modifiers can be used to specify the expected layout of a matrix within the hardware constant store. The row_major type modifier indicates that each row of the matrix will be stored in a single constant register. Likewise, using col_major indicates that each column of the matrix will be stored in a single constant register. Column major is the default.

Storage Class Modifiers

Storage class modifiers inform the compiler about the intended scope and lifetime of a given variable. These modifiers are optional and may appear in any order, as long as they appear before the variable type.

As in C, a variable may be declared as static or extern. (These two modifiers are mutually exclusive.) At global scope, the static storage class modifier indicates that the variable is only to be accessed by the shader and not by the application via the API. Any non-static variable that is declared at global scope may be modified by the application through the API. As with C, using the static modifier at *local* scope indicates that the variable contains data that is to persist between invocations of the declaring function.

The extern modifier can be used on a global variable to indicate that it can be modified from outside of the shader via the API. This is redundant, however, as this is the default behavior for variables declared at global scope.

The shared modifier is used to specify that a given global variable is to be shared between effects.

A variable that is uniform is assumed to have been set externally to the HLSL shader (i.e., via the Set*ShaderConstant*() API). Global variables are treated as if they were declared uniform. Such variables are not assumed to be const, however, as their values can be modified in the shader.

For example, say you declare the following variables at global scope:

```
extern float translucencyCoeff;
const  float gloss_bias;
static float gloss_scale;
float diffuse;
```

The variables `diffuse` and `translucencyCoeff` are settable by the `Set*ShaderConstant*()` API and can be modified by the shader itself. The `const` variable `gloss_bias` is settable by the `Set*ShaderConstant*()` API but cannot be modified in the shader code. Finally, the `static` variable `gloss_scale` is not settable by the `Set*ShaderConstant*()` API but can be modified within the shader only.

Initializers

As we have shown in some of the preceding examples, it is possible to initialize variables at declaration time in the same manner used in C. For example:

```
float2x2 fMat = {3.0f, 5.0f,  // row 1
                 2.0f, 1.0f}; // row 2
float4   vPos = {3.0f, 5.0f, 2.0f, 1.0f};
float fFactor = 0.2f;
```

Working with Vectors

In HLSL, there are a few "gotchas" to look out for when performing math on vectors. Fortunately, most of them are quite intuitive, given that we are writing shaders for 3D graphics. For example, standard binary operators are defined to work *per component*:

```
float4 vTone = vBrightness * vExposure;
```

Assuming `vBrightness` and `vExposure` are both of type `float4`, this is equivalent to:

```
float4 vTone;
vTone.x = vBrightness.x * vExposure.x;
vTone.y = vBrightness.y * vExposure.y;
vTone.z = vBrightness.z * vExposure.z;
vTone.w = vBrightness.w * vExposure.w;
```

Note that this is *not* a dot product between the 4D vectors vBrightness and vExposure. Additionally, multiplying matrix variables in this way does not result in a matrix multiply. Dot products and matrix multiplies are applied via the intrinsic function mul(), which we discuss later in the chapter.

Constructors

Another language feature that you often see in HLSL shaders is the constructor, which is similar to C++ but has some enhancements to deal with complex data types. Example uses of constructors include:

```
float3   vPos     = float3(4.0f, 1.0f, 2.0f);
float    fDiffuse = dot(vNormal, float3(1.0f, 0.0f, 0.0f));
float4   vPack    = float4(vPos, fDiffuse);
```

Constructors are commonly used when a shader writer wants to temporarily define a quantity with literal values (as in dot(vNormal, float3(1.0f, 0.0f, 0.0f)) above) or when a shader writer wants to explicitly pack smaller data types together (as in float4(vPos, fDiffuse) above). In this case, the float4 constructor takes in a float3 and a float and returns a float4 with the data packed together.

Type Casting

To aid in shader writing and the efficiency of the generated code, it is a good idea to be familiar with HLSL's type casting behavior. Type casting often happens in order to promote or demote a given variable to match a variable to which it is being assigned. For example, in the following case, a literal float 0.0f is being cast to a float4 {0.0f , 0.0f , 0.0f , 0.0f } to initialize vResult.

```
float4   vResult = 0.0f;
```

Similar casting can occur when assigning a higher dimensional data type like a vector or matrix to a lower dimensional data type. In these cases, the extra data is effectively omitted. For example, we may write the following code:

```
float3   vLight;
float    fFinal, fColor;
fFinal = vLight * fColor;
```

In this case, `vLight` is cast to a `float` by using only the first component in the multiply with the scalar float `fColor`. In this case, `fFinal` is equal to `vLight.x * fColor`.

It is a good idea to be familiar with the following table of type casting rules for HLSL:

Type of Cast	Casting Behavior
Scalar-to-scalar	**Always valid**. When casting from bool type to an integer or floating-point type, false is considered to be zero and true is considered to be one. When casting from an integer or floating-point type to bool, a zero value is considered to be false and a nonzero value is considered to be true. When casting from a floating-point type to an integer type, the value is rounded toward zero. This is the same truncation behavior as in C.
Scalar-to-vector	**Always valid**. This cast operates by replicating the scalar to fill the vector.
Scalar-to-matrix	**Always valid**. This cast operates by replicating the scalar to fill the matrix.
Scalar-to-structure	This cast operates by replicating the scalar to fill the structure.
Vector-to-scalar	**Always valid**. This selects the first component of the vector.
Vector-to-vector	The destination vector must not be larger than the source vector. The cast operates by keeping the leftmost values and truncating the rest. For the purposes of this cast, column matrices, row matrices, and numeric structures are treated as vectors.
Vector-to-matrix	The size of the vector must be equal to the size of the matrix.
Vector-to-structure	This is valid if the structure is not larger than the vector, and all components of the structure are numeric.
Matrix-to-scalar	**Always valid**. This selects the upper-left component of the matrix.
Matrix-to-vector	The size of the matrix must be equal to the size of the vector.
Matrix-to-matrix	The destination matrix must not be larger than the source matrix in both dimensions. The cast operates by keeping the upper-left values and truncating the rest.
Matrix-to-structure	The size of the structure must be equal to the size of the matrix, and all components of the structure are numeric.
Structure-to-scalar	The structure must contain at least one member.

Type of Cast	Casting Behavior
Structure-to-vector	The structure must be at least the size of the vector. The first components must be numeric, up to the size of the vector.
Structure-to-matrix	The structure must be at least the size of the matrix. The first components must be numeric, up to the size of the matrix.
Structure-to-object	The structure must contain at least one member. The type of this member must be identical to the type of the object.
Structure-to-structure	The destination structure must not be larger than the source structure. A valid cast must exist between all respective source and destination components.

Structures

As we showed in the first example shader, it is often convenient to be able to define structures in HLSL shaders. For example, many shader writers will define an output structure in their vertex shader code and use this structure as the return type from their vertex shader's `main` function. (It is less common to do this with a pixel shader since most pixel shaders have only one `float4` output.) An example structure taken from the NPR Metallic shader that we discuss later is shown below:

```
struct VS_OUTPUT
{
   float4 Pos   : POSITION;
   float3 View  : TEXCOORD0;
   float3 Normal: TEXCOORD1;
   float3 Light1: TEXCOORD2;
   float3 Light2: TEXCOORD3;
   float3 Light3: TEXCOORD4;
};
```

Structures may be declared for general use in an HLSL shader as well. They follow the type casting rules outlined above.

Samplers

For each different texture map that you plan to sample in a pixel shader, you must declare a *sampler*. Recall the `hlsl_rings()` shader described earlier:

```
float4 lightWood; // xyz == Light Wood Color
float4 darkWood;  // xyz == Dark Wood Color
float  ringFreq;  // ring frequency

sampler PulseTrainSampler;

float4 hlsl_rings (float4 Pshade : TEXCOORD0) : COLOR
{
    float scaledDistFromZAxis = sqrt(dot(Pshade.xy, Pshade.xy)) * ringFreq;
    float blendFactor = tex1D (PulseTrainSampler, scaledDistFromZAxis);
    return lerp (darkWood, lightWood, blendFactor);
}
```

In this shader, we declared a sampler called `PulseTrainSampler` at global scope and passed it as the first parameter to the `tex1D()` intrinsic function (we discuss intrinsics in the next section). An HLSL sampler has a very direct mapping to the API concept of a sampler and, in turn, to the actual silicon in the 3D graphics processor, which is responsible for addressing and filtering textures. A sampler must be defined for every texture map that you plan to access in a given shader, but you may use a given sampler multiple times in a shader. This usage is very common in image processing applications, as discussed in *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9*, since the input image is often sampled multiple times with different texture coordinates to provide data to a filter kernel expressed in shader code. For example, the following shader uses the rasterizer to convert a height map to a normal map with a pair of Sobel filters:

```
sampler InputImage;

float4 main( float2 topLeft    : TEXCOORD0, float2 left        : TEXCOORD1,
             float2 bottomLeft : TEXCOORD2, float2 top         : TEXCOORD3,
             float2 bottom     : TEXCOORD4, float2 topRight    : TEXCOORD5,
             float2 right      : TEXCOORD6, float2 bottomRight : TEXCOORD7):
                                                                 COLOR
{
   // Take all eight taps
   float4 tl = tex2D (InputImage, topLeft);
   float4  l = tex2D (InputImage, left);
   float4 bl = tex2D (InputImage, bottomLeft);
   float4  t = tex2D (InputImage, top);
   float4  b = tex2D (InputImage, bottom);
   float4 tr = tex2D (InputImage, topRight);
   float4  r = tex2D (InputImage, right);
   float4 br = tex2D (InputImage, bottomRight);

   // Compute dx using Sobel operator:
   //
   //           -1 0 1
   //           -2 0 2
   //           -1 0 1
   float dX = -tl.a - 2.0f*l.a - bl.a + tr.a + 2.0f*r.a + br.a;

   // Compute dy using Sobel operator:
   //
   //           -1 -2 -1
   //            0  0  0
   //            1  2  1
   float dY = -tl.a - 2.0f*t.a - tr.a + bl.a + 2.0f*b.a + br.a;

   // Compute cross product and renormalize
   float4 N = float4(normalize(float3(-dX, -dY, 1)), tl.a);

   // Convert signed values from -1..1 to 0..1 range and return
   return N * 0.5f + 0.5f;
}
```

This shader uses only one sampler, `InputImage`, but samples from it eight times using the `tex2D()` intrinsic function.

Intrinsics

As mentioned in the preceding section, there are a number of *intrinsics* built into the DirectX High Level Shading Language for your convenience. Many intrinsics, such as mathematical functions, are provided for convenience, while others, such as the `tex1D()` and `tex2D()` functions mentioned above, are necessary for accessing texture data via samplers.

Math Intrinsics

The math intrinsics listed in the table below will be converted to micro operations by the HLSL compiler. In some cases, such as `abs()` and `dot()`, these intrinsics will map directly to single assembly-level operations, while in other cases, such as `refract()` and `step()`, they will map to multiple assembly instructions. There are even a couple of cases, notably `ddx()`, `ddy()`, and `fwidth()`, that are not supported for all compile targets. The math intrinsics are shown below:

Intrinsic	Description
abs(*x*)	Absolute value (per component).
acos(*x*)	Returns the arccosine of each component of *x*. Each component should be in the range [–1, 1].
all(*x*)	Tests if all components of *x* are nonzero.
any(*x*)	Tests if any component of *x* is nonzero.
asin(*x*)	Returns the arcsine of each component of *x*. Each component should be in the range [–π/2, π/2].
atan(x)	Returns the arctangent of x. The return values are in the range [–π/2, π/2].
atan2(*y*, *x*)	Returns the arctangent of *y*/*x*. The signs of *y* and *x* are used to determine the quadrant of the return values in the range [–π, π]. atan2 is well-defined for every point other than the origin, even if *x* equals 0 and *y* does not equal 0.
ceil(*x*)	Returns the smallest integer that is greater than or equal to *x*.
clamp(*x*, *min*, *max*)	Clamps *x* to the range [*min*, *max*].
clip(*x*)	Discards the current pixel, if any component of *x* is less than 0. This can be used to simulate clip planes, if each component of *x* represents the distance from a plane. This is the intrinsic that you use when you want to generate an asm `texkill`.
cos(*x*)	Returns the cosine of *x*.
cosh(*x*)	Returns the hyperbolic cosine of *x*.
cross(*a*, *b*)	Returns the cross product of two 3D vectors *a* and *b*.
D3DCOLORtoUBYTE4(*x*)	Swizzles and scales components of the 4D vector *x* to compensate for the lack of UBYTE4 stream component support in some hardware.
ddx(*x*)	Returns the partial derivative of *x* with respect to the screen-space *x*-coordinate.

Intrinsic	Description
ddy(*x*)	Returns the partial derivative of *x* with respect to the screen-space *y*-coordinate.
degrees(*x*)	Converts *x* from radians to degrees.
determinant(*m*)	Returns the determinant of the square matrix *m*.
distance(*a*, *b*)	Returns the distance between two points *a* and *b*.
dot(*a*, *b*)	Returns the dot product of two vectors *a* and *b*.
exp(*x*)	Returns the base-e exponent e^x.
exp2(a)	Base-2 exponent (per component).
faceforward(*n*, *i*, *ng*)	Returns –*n* * sign(dot(*i*, *ng*)).
floor(*x*)	Returns the greatest integer that is less than or equal to *x*.
fmod(*a*, *b*)	Returns the floating-point remainder *f* of *a* / *b* such that *a* = *i* * *b* + *f*, where *i* is an integer, *f* has the same sign as *x*, and the absolute value of *f* is less than the absolute value of *b*.
frac(*x*)	Returns the fractional part *f* of *x*, such that *f* is a value greater than or equal to 0 and less than 1.
frexp(*x*, out *exp*)	Returns the mantissa and exponent of *x*. frexp returns the mantissa, and the exponent is stored in the output parameter *exp*. If *x* is 0, the function returns 0 for both the mantissa and the exponent.
fwidth(*x*)	Returns abs(ddx(*x*))+abs(ddy(*x*)).
isfinite(*x*)	Returns true if *x* is finite; false otherwise.
isinf(*x*)	Returns true if *x* is +INF or –INF; false otherwise.
isnan(*x*)	Returns true if *x* is NAN or QNAN; false otherwise.
ldexp(*x*, *exp*)	Returns $x * 2^{exp}$.
len(*v*)	Vector length.
length(*v*)	Returns the length of the vector *v*.
lerp(*a*, *b*, *s*)	Returns *a* + *s*(*b* – *a*). This linearly interpolates between *a* and *b*, such that the return value is *a* when *s* is 0 and *b* when *s* is 1.
log(*x*)	Returns the base-e logarithm of *x*. If *x* is negative, the function returns indefinite. If *x* is 0, the function returns +INF.
log10(*x*)	Returns the base-10 logarithm of *x*. If *x* is negative, the function returns indefinite. If *x* is 0, the function returns +INF.
log2(*x*)	Returns the base-2 logarithm of *x*. If *x* is negative, the function returns indefinite. If *x* is 0, the function returns +INF.
max(*a*, *b*)	Selects the greater of *a* and *b*.

Intrinsic	Description
min(*a*, *b*)	Selects the lesser of *a* and *b*.
modf(*x*, out *ip*)	Splits the value *x* into fractional and integer parts, each of which has the same sign as *x*. The signed fractional portion of *x* is returned. The integer portion is stored in the output parameter *ip*.
mul(*a*, *b*)	Performs matrix multiplication between *a* and *b*. If *a* is a vector, it is treated as a row vector. If *b* is a vector, it is treated as a column vector. The inner dimension $a_{columns}$ and b_{rows} must be equal. The result has the dimension $a_{rows} \times b_{columns}$.
normalize(*v*)	Returns the normalized vector *v* / length(*v*). If the length of *v* is 0, the result is indefinite.
pow(*x*, *y*)	Returns x^y.
radians(*x*)	Converts *x* from degrees to radians.
reflect(*i*, *n*)	Returns the reflection vector *v*, given the entering ray direction *i* and the surface normal *n*, such that *v* = i – 2 * dot(*i*, *n*) * *n*.
refract(*i*, *n*, *eta*)	Returns the refraction vector *v*, given the entering ray direction *i*, the surface normal *n*, and the relative index of refraction *eta*. If the angle between *i* and *n* is too great for a given *eta*, refract returns (0,0,0).
round(*x*)	Rounds *x* to the nearest integer.
rsqrt(*x*)	Returns 1 / sqrt(*x*).
saturate(*x*)	Clamps *x* to the range [0, 1].
sign(*x*)	Computes the sign of *x*. Returns –1 if *x* is less than 0, 0 if *x* equals 0, and 1 if *x* is greater than 0.
sin(*x*)	Returns the sine of *x*.
sincos(*x*, out *s*, out *c*)	Returns the sine and cosine of *x*. sin(*x*) is stored in the output parameter *s*. cos(*x*) is stored in the output parameter *c*.
sinh(*x*)	Returns the hyperbolic sine of *x*.
smoothstep(*min*, *max*, *x*)	Returns 0 if *x* < *min*. Returns 1 if *x* > *max*. Returns a smooth Hermite interpolation between 0 and 1 if *x* is in the range [*min*, *max*].
sqrt(*x*)	Square root (per component).
step(*a*, *x*)	Returns (*x* = *a*) ? 1 : 0.
tan(*x*)	Returns the tangent of *x*.
tanh(*x*)	Returns the hyperbolic tangent of *x*.
transpose(*m*)	Returns the transpose of the matrix *m*. If the source is dimension $m_{rows} \times m_{columns}$, the result is dimension $m_{columns} \times m_{rows}$.

Texture Sampling Intrinsics

There are 16 texture sampling intrinsics used for sampling texture data into a shader. There are four types of textures (1D, 2D, 3D, and cube map) and four types of loads (regular, with derivatives, projective, and biased) with an intrinsic for each of the 16 combinations:

Intrinsic	Description
tex1D(*s*, *t*)	**1D texture lookup**. *s* is a sampler. *t* is a scalar.
tex1D(*s*, *t*, *ddx*, *ddy*)	**1D texture lookup, with derivatives**. *s* is a sampler. *t*, *ddx*, and *ddy* are scalars.
tex1Dproj(*s*, *t*)	**1D projective texture lookup**. *s* is a sampler. *t* is a 4D vector. *t* is divided by its last component before the lookup takes place.
tex1Dbias(*s*, *t*)	**1D biased texture lookup**. *s* is a sampler. *t* is a 4D vector. The mip level is biased by *t*.w before the lookup takes place.
tex2D(*s*, *t*)	**2D texture lookup**. *s* is a sampler. *t* is a 2D texture coordinate.
tex2D(*s*, *t*, *ddx*, *ddy*)	**2D texture lookup, with derivatives**. *s* is a sampler. *t*, *ddx*, and *ddy* are 2D vectors.
tex2Dproj(*s*, *t*)	**2D projective texture lookup**. *s* is a sampler. *t* is a 4D vector. *t* is divided by its last component before the lookup takes place.
tex2Dbias(*s*, *t*)	**2D biased texture lookup**. s is a sampler. *t* is a 4D vector. The mip level is biased by *t*.w before the lookup takes place.
tex3D(*s*, *t*)	**3D volume texture lookup**. *s* is a sampler. *t* is a 3D texture coordinate.
tex3D(*s*, *t*, *ddx*, *ddy*)	**3D volume texture lookup, with derivatives**. *s* is a sampler. *t*, *ddx*, and *ddy* are 3D vectors.
tex3Dproj(*s*, *t*)	**3D projective volume texture lookup**. *s* is a sampler. *t* is a 4D vector. *t* is divided by its last component before the lookup takes place.
tex3Dbias(*s*, *t*)	**3D biased texture lookup**. *s* is a sampler. *t* is a 4D vector. The mip level is biased by *t*.w before the lookup takes place.
texCUBE(*s*, *t*)	**Cube map lookup**. *s* is a sampler. *t* is a 3D texture coordinate.
texCUBE(*s*, *t*, *ddx*, *ddy*)	**Cube map lookup, with derivatives**. *s* is a sampler. *t*, *ddx*, and *ddy* are 3D vectors.
texCUBEproj(*s*, *t*)	**Projective cube map lookup**. *s* is a sampler. *t* is a 4D vector. *t* is divided by its last component before the lookup takes place.

Intrinsic	Description
texCUBEbias(*s*, *t*)	**Biased cube map lookup**. *s* is a sampler. *t* is a 4D vector. The mip level is biased by *t*.w before the lookup takes place.

The `tex1D()`, `tex2D()`, `tex3D()`, and `texCUBE()` intrinsics are the most commonly used to sample textures. The texture loading intrinsics that take ddx and ddy parameters compute texture LOD using these explicit derivatives, which would typically have been previously calculated with the `ddx()` and `ddy()` math intrinsics. These are particularly important when writing procedural pixel shaders, but they are not supported on ps_2_0 or lower compile targets.

The `tex*proj()` intrinsics are used to do projective texture reads, where the texture coordinates used to sample the texture are divided by the last component prior to accessing the texture. Of these, `tex2Dproj()` is the most commonly used, since it is necessary for projective shadow maps and similar effects.

The `tex*bias()` intrinsics are used to perform biased texture sampling, where the bias can be computed per pixel. This is typically done to induce some over-blurring of the texture for a special effect. For example, as discussed in *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9*, the pixel shader used on the motion-blurred balls in the Radeon 9700 Animusic Pipe Dream demo uses the `texCUBEbias()` intrinsic to access the cubic environment map of the local scene:

```
...
   // Blur reflection by extension amount.
   float3 vCubeLookup = vReflection + i.Pos/fEnvMapRadius;
   float4 cReflection = texCUBEbias(tCubeEnv, float4(vCubeLookup,
          fBlur * fTextureBlur)) * vReflectionColor;
...
```

In this code snippet, `fBlur * fTextureBlur` is stored in the fourth component of the texture coordinate used in the `texCUBEbias()` call and determines the bias to be used when accessing the cube map.

Now that we have introduced some of the mechanics of the language, we can discuss how data is input to and output from HLSL shaders in DirectX 9.

Shader Inputs

Vertex and pixel shaders have two types of input data: *varying* and *uniform*. The varying input is the data that is unique to each execution of a shader. For a vertex shader, the varying data (i.e., position, normals, etc.) comes from the vertex streams. The uniform data (i.e., material color, world transform, etc.) is constant for multiple executions of a shader. If you are familiar with the assembly models, uniform data is specified in constant registers and varying data in the v/t registers in vertex and pixel shaders.

Uniform Input

Uniform data can be specified by two methods in HLSL. The most common method is to declare global variables and use them within the vertex or pixel shaders. Any use of a global variable within a shader will result in the addition of the variable to a list of uniform variables required by the shader. The second method is to mark an input parameter of the top-level shader function as uniform. This marking specifies that the given variable should be added to the list of uniform variables used by the shader. Both of these methods are illustrated in the following code snippet:

```
// Declare a global uniform variable
// Appears in constant table under name 'UniformGlobal'
float4 UniformGlobal;

// Declare a uniform input parameter
// Appears in constant table under name '$UniformParam'
float4 main( uniform float4 UniformParam ) : POSITION
{
   return UniformGlobal * UniformParam;
}
```

The uniform variables used by a shader are communicated back to the application via the *constant table*. The constant table is a symbol table that defines how the uniform variables used by a shader must be loaded into the constant registers prior to shader execution.

NOTE The uniform input function parameters appear in the constant table with a $ prepended, unlike the global variables. The $ is required to avoid name collisions between "local" uniform inputs and global variables of the same name.

The constant table contains the constant register locations of all uniform variables used by the shader. The table also includes the type information and the default value, if specified, for each constant table entry. The following is an example of what a constant table looks like when printed out. The constant table generated by the compiler is stored in a compact binary form. The API to interpret the table at run time will be discussed later in the section on HLSL integration without the use of D3DX Effects.

Here is the textual printout of a constant table emitted by fxc.exe for a sample shader:

```
//
// Generated by Microsoft (R) D3DX9 Shader Compiler
//
//  Source: hemisphere.fx
//  Flags: /E:VS /T:vs_1_1
//

// Registers:
//
//      Name         Reg   Size
//      ------------ ----- ----
//      Projection   c0       4
//      WorldView    c4       3
//      DirFromLight c7       1
//      DirFromSky   c8       1
//      $bHemi       c18      1
//      $bDiff       c19      1
//      $bSpec       c20      1
//
```

```
//
// Default values:
//
//    DirFromLight
//      c7   = { 0.577, -0.577, 0.577, 0 };
//
//    DirFromSky
//      c8   = { 0, -1, 0, 0 };
```

Varying Input

Varying data is specified by marking the input parameters of the top-level shader function with an input semantic. All top-level shader inputs must either be marked as varying by using semantics or marked with the keyword "uniform" to indicate the value is constant for the execution of the shader. If a top-level shader input is not marked with a semantic or "uniform" keyword, the shader will fail to compile.

The input semantic is a name used to link the given shader input to an output of the previous stage of the graphics pipeline. For example, the input semantic `POSITION0` is used by vertex shaders to specify where the position data from the vertex buffer should be linked.

Pixel and vertex shaders have different sets of input semantics due to the different parts of the graphics pipeline that feed into each shader unit. Vertex shader input semantics describe the per-vertex information to be loaded from a vertex buffer into a form that can be consumed by the vertex shader (i.e., positions, normals, texture coordinates, colors, tangents, binormals, etc.). These input semantics directly map to the combination of the `D3DDECLUSAGE` enum and `UsageIndex` that is used to describe vertex data elements in a vertex buffer.

Pixel shader input semantics describe the information that is provided per pixel by the rasterization unit. This data is generated by interpolating between the outputs of the vertex shader for each vertex of the current primitive. The basic pixel shader input semantics link the input color and texture coordinate information to input parameters.

Input semantics can be assigned to shader input by two methods. The first method is by appending a colon (:) and the input semantic name to the input parameter declaration. The second method is to define an input structure with input semantics assigned to each element of the input structure. Both of these styles are used in the example shaders in this chapter and throughout the ShaderX books.

Here is an input semantic example:

```
// Declare an input structure with a semantic binding
struct InStruct
{
    float4 Pos1 : POSITION1
};

// Declare the Pos variable as containing position data
float4 main( float4 Pos : POSITION0, InStruct In ) : POSITION
{
    return Pos * In.Pos1;
}

// Declare the Col variable as containing the interpolated COLOR0 value
float4 mainPS( float4 Col : COLOR0 ) : COLOR
{
    return Col;
}
```

Here are the vertex shader input semantics:

Semantic	Description
POSITION*n*	Position
BLENDWEIGHT*n*	Blend weights
BLENDINDICES*n*	Blend indices
NORMAL*n*	Normal vector
PSIZE*n*	Point size
COLOR*n*	Color
TEXCOORD*n*	Texture coordinates
TANGENT*n*	Tangent
BINORMAL*n*	Binormal
TESSFACTOR*n*	Tessellation factor

Here are the pixel shader input semantics:

Semantic	Description
`COLOR`*n*	Color
`TEXCOORD`*n*	Texture coordinates

n is an optional integer (as an example: `PSIZE0`, `DIFFUSE1`, etc.).

Shader Outputs

Vertex and pixel shaders provide output data to the subsequent graphics pipeline stage. Output semantics are used to specify how data generated by the shader should be linked to the inputs of the next stage. For example, the output semantics for a vertex shader are used to link the outputs with the interpolators in the rasterizer to generate the input data for the pixel shader. The pixel shader outputs are the values provided to the alpha blending unit for each of the render targets or the depth value to be written to the depth buffer.

Vertex shader output semantics are used to link the shader to both the pixel shader and the rasterizer stage. The `POSITION` output is a required output from each vertex shader that is consumed by the rasterizer and not exposed to the pixel shader. `TEXCOORD`*n* and `COLOR`*n* denote outputs that are made available to the pixel shader post interpolation.

Pixel shader output semantics bind the output colors of a pixel shader with the correct render target. The colors output from the pixel shader are linked to the alpha blend stage, which determines how the destination render targets are modified. The `DEPTH` output semantics can be used to change the destination depth value at the current raster location.

NOTE `DEPTH` and multiple render targets (also known as "MRT") are only supported with some shader models.

The syntax for output semantics is identical to the syntax for specifying input semantics. The semantics can either be specified directly on parameters declared as out parameters or assigned during the definition of a structure that is either returned as an out parameter or the return value of the function.

Here are the vertex shader output semantics:

Semantic	Description
POSITION	Position
PSIZE	Point size
FOG	Vertex fog
COLOR*n*	Color (example: COLOR0)
TEXCOORD*n*	Texture coordinates (example: TEXCOORD0)

Here are the pixel shader output semantics:

Semantic	Description
COLOR*n*	Color for render target *n*
DEPTH	Depth value

n is an optional integer (as an example: TEXCOORD3, COLOR0).

The following code snippets illustrate the variety of ways in which data can be output from HLSL shaders:

```
// Declare an output structure with a semantic binding
struct OutStruct
{
float2 Tex2 : TEXCOORD2
};

// Declare the Tex0 out parameter as containing TEXCOORD0 data
float4 main(out float2 Tex0 : TEXCOORD0, out OutStruct Out ) : POSITION
{
    Tex0 = float2(1.0, 0.0);
    Out.Tex2 = float2(0.1, 0.2);
    return float4(0.5, 0.5, 0.5, 1);
}

// Declare the Col variable as containing the interpolated COLOR0 value
float4 mainPS( out float4 Col1 : COLOR1) : COLOR
{
    // write out to render target 1 using out parameter
    Col1 = float4(0.0, 0.0, 0.0, 0.0);
    // write to render target 0 using the declared return destination
    return float4(1.0, 0.9722, 0.3333334, 0);
}
```

```
struct PS_OUT
{
    float4 Color: COLOR;
    float  Depth: DEPTH;
};

//
// Three different ways to output from a pixel shader:
//

PS_OUT PSFunc1() { ... }

void PSFunc2(out float4 Color : COLOR,
             out float Depth : DEPTH)
{
  ...
}

void PSFunc3(out PS_OUT Out)
{
  ...
}
```

An Example Shader

Now that we've discussed the language itself and how it connects with the rest of the graphics pipeline via inputs and outputs, we can discuss an example shader called NPR Metallic. We call it this since it was designed to look like a metallic surface that would exist in a world rendered in a cel-animation style (see Figure 2). This effect ships with the RenderMonkey shader development environment discussed in the "Shader Development Using RenderMonkey" article in this book and is available on the ATI Developer Relations web site (www.ati.com/developer).

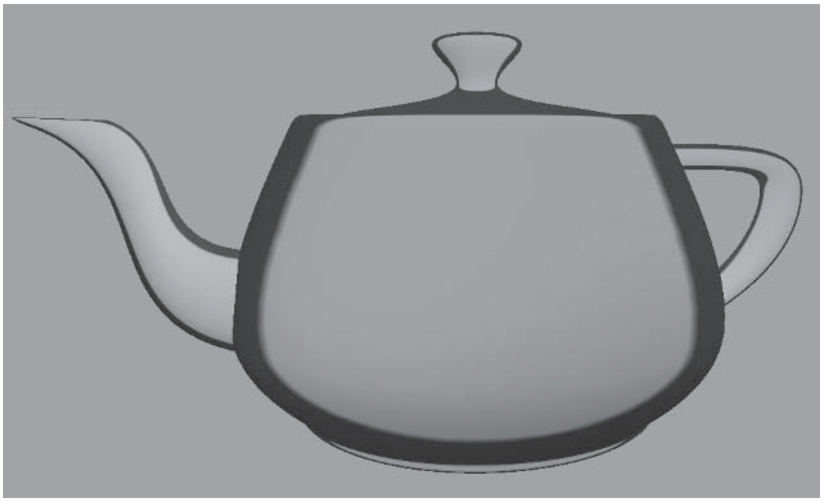


Figure 2: NPR Metallic

First, let's look at the NPR Metallic vertex shader written in HLSL:

```
float4x4 view_proj_matrix;

float4 view_position;
float4 light0;
float4 light1;
float4 light2;

struct VS_OUTPUT
{
   float4 Pos   : POSITION;
   float3 View  : TEXCOORD0;
   float3 Normal: TEXCOORD1;
   float3 Light1: TEXCOORD2;
   float3 Light2: TEXCOORD3;
   float3 Light3: TEXCOORD4;
};

VS_OUTPUT main( float4 inPos    : POSITION,
                float3 inNorm   : NORMAL )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;
   // Output transformed vertex position:
   Out.Pos = mul(view_proj_matrix, inPos);

   Out.Normal = inNorm;

   // Compute the view vector:
   Out.View = normalize(view_position - inPos);

   // Compute vectors to three lights from the current vertex position:
   Out.Light1 = normalize(light0 - inPos);   // Light 1
   Out.Light2 = normalize(light1 - inPos);   // Light 2
   Out.Light3 = normalize(light2 - inPos);   // Light 3

   return Out;
}
```

The first thing that we see in this vertex shader is the declaration of a matrix and a set of floats at global scope: `view_proj_matrix`, `view_position`, `light0`, `light1`, and `light2`. These are all implicitly uniform variables that are externally settable by the API and modifiable in the shader itself.

Following these global variables, we see the definition of a structure called `VS_OUTPUT`, which is also the return type of our `main` function. This means that this vertex shader will output five 3D texture coordinates in addition to the required 4D position.

Looking at the `main` function, we can see that the vertex shader takes a 4D vector as input position, a 3D vector as input normal, and a 2D vector as a texture coordinate. The input position, `inPos`, is transformed by the `view_proj_matrix` using the `mul()` intrinsic, while the normal, `inNorm`, is passed through to the output untouched.

Finally, 3D vectors from the object space vertex position to the three lights and the view position are all computed. These 3D vectors are passed to the `normalize()` intrinsic to guarantee that they are of unit length. These normalized 3D vectors are all output from the vertex shader as 3D texture coordinates that will be interpolated across the polygon.

To reinforce the earlier discussion about compile targets and assembly models, let's compile this shader and have a look at the assembly output. First, we write the above code into a file called

NPRMetallic.vhl. Next, we can compile it on the command line with fxc:

```
fxc -nologo -T vs_1_1 -Fc -Vd NPRMetallic.vhl
```

Because this vertex shader does not require flow control, we select the vs_1_1 compile target. We also set the flags to generate a code file and disable validation. A portion of the generated code file is shown here:

```
// Parameters:
//     float4 light0;
//     float4 light1;
//     float4 light2;
//     float4 view_position;
//     float4x4 view_proj_matrix;
//
// Registers:
//     Name             Reg   Size
//     ---------------- ----- ----
//     view_proj_matrix c0       4
//     view_position    c4       1
//     light1           c5       1
//     light2           c6       1
//     light0           c7       1

    vs_1_1
    dcl_position v0
    dcl_normal v1
    mul r0, v0.x, c0
    mad r2, v0.y, c1, r0
    mad r4, v0.z, c2, r2
    mad oPos, v0.w, c3, r4
    add r1, -v0, c4
    dp4 r1.w, r1, r1
    rsq r1.w, r1.w
    mul oT0.xyz, r1, r1.w
    add r8, -v0, c7
    dp4 r8.w, r8, r8
    rsq r8.w, r8.w
    mul oT2.xyz, r8, r8.w
    add r3, -v0, c5
    add r10, -v0, c6
    dp4 r3.w, r3, r3
    rsq r3.w, r3.w
    mul oT3.xyz, r3, r3.w
    dp4 r10.w, r10, r10
    rsq r10.w, r10.w
    mul oT4.xyz, r10, r10.w
    mov oT1.xyz, v1
```

At the top of the code file, we see the parameters to this vertex shader. That is, we see the global scope variables that will need to be set from the API for this shader to work properly in a given application. The next section shows the hardware registers to which these parameters must be loaded by the application for the assembly shader to work properly. Next, we have the shader code itself, which was compiled to 21 assembly instructions. We don't go through all of the code, but you should take note of the `dcl_position` and `dcl_normal` statements, which are a direct result of the `POSITION` and `NORMAL` semantics on the inputs to the shader's `main` function. Additionally, note the storage of final results in the `oPos`, `oT0`, `oT1`, `oT2`, `oT3`, and `oT4` registers. This is caused by the return type of the function being a structure whose members are tagged with the corresponding semantics. While not strictly necessary, knowing how to use fxc to generate assembly code from HLSL and how to read through it can be beneficial at some stages of development, particularly when trying to write more optimal HLSL.

Now that we have used the vertex shader to transform the geometry into clip space and define the values that will be interpolated across the polygons, we can move on to the pixel shader, which will make use of all of these interpolated quantities.

The following is the NPR Metallic pixel shader:

```
float4 Material;

sampler Outline;

float4 main( float3 View:   TEXCOORD0,
             float3 Normal: TEXCOORD1,
             float3 Light1: TEXCOORD2,
             float3 Light2: TEXCOORD3,
            float3 Light3: TEXCOORD4 ) : COLOR
{
   // Normalize input normal vector:
   float3 norm = normalize (Normal);

   float4 outline = tex1D(Outline, 1 - dot (norm, normalize(View)));

   float lighting = (dot (normalize (Light1), norm) * 0.5 + 0.5) +
                    (dot (normalize (Light2), norm) * 0.5 + 0.5) +
                    (dot (normalize (Light3), norm) * 0.5 + 0.5);

   return outline * Material * lighting;
}
```

As before, we see that this shader has declared some variables at global scope. In this case, we have a 4D vector `Material`, which defines material values for the object to be rendered, and a single sampler `Outline`, which we use to access a special texture used for outlining the object. The five 3D texture coordinates computed in the vertex shader are the inputs to the `main` function of this pixel shader and define the view vector, the normal vector, and three light vectors.

Since the texture coordinates are *linearly* interpolated across the polygon, it is possible for them to contain non-normalized values at a given pixel. Thus, this shader first renormalizes the interpolated normal vector using the `normalize()` intrinsic. Subsequently, the outline texture is sampled using the dot product of the normalized normal and view vectors. The lighting is then computed by summing a series of scaled and biased dot products of the normal with normalized light vectors.

In the last line of this pixel shader, we return the product of the variables `outline`, `Material`, and `lighting`. The first two of these are 4D vectors, while the last is a scalar. If you recall from our earlier discussion of type casting, the multiplication of the scalar by a vector temporarily promotes the scalar to a vector whose components are all equivalent to the original scalar. That is, the following two expressions are equivalent:

```
return outline * Material * lighting;
return outline * Material * float4(lighting, lighting, lighting, lighting);
```

Thus, the end result is that all of the channels are multiplied by the scalar `lighting`, giving us the final result you see in Figure 2.

As we did with the NPR Metallic vertex shader, we generate a code file for the pixel shader using fxc:

```
fxc -nologo -T ps_2_0 -Fc -Vd NPRMetallic.phl
```

This compilation uses the same flags as before but is compiled for the ps_2_0 target. The resulting 29-instruction shader is shown below:

```
// Parameters:
//     float4 Material;
//     sampler Outline;
//
// Registers:
//     Name         Reg   Size
//     ------------ ----- ----
//     Material     c0       1
//     Outline      s0       1

    ps_2_0
    def c1, 1, 0, 0, 0.5
    dcl t0.xyz
    dcl t1.xyz
    dcl t2.xyz
    dcl t3.xyz
    dcl t4.xyz
    dcl_2d s0
    dp3 r0.w, t1, t1
    rsq r2.w, r0.w
    mul r9.xyz, r2.w, t1
    dp3 r9.w, t0, t0
    rsq r9.w, r9.w
    mul r4.xyz, r9.w, t0
    dp3 r9.w, r9, r4
    add r11.xy, -r9.w, c1.x
    texld r6, r11, s0
    dp3 r9.w, t2, t2
    rsq r9.w, r9.w
    mul r1.xyz, r9.w, t2
    dp3 r9.w, r1, r9
    mad r9.w, r9.w, c1.w, c1.w
    dp3 r8.w, t3, t3
    rsq r10.w, r8.w
    mul r5.xyz, r10.w, t3
    dp3 r0.w, r5, r9
    mad r9.w, r0.w, c1.w, r9.w
    add r9.w, r9.w, c1.w
    dp3 r2.w, t4, t4
    rsq r11.w, r2.w
    mul r1.xyz, r11.w, t4
    dp3 r8.w, r1, r9
    mad r10.w, r8.w, c1.w, r9.w
    add r5.w, r10.w, c1.w
    mul r6, r6, r5.w
    mul r0, r6, c0
    mov oC0, r0
```

As before, the variables (in this case, the constant `Material` and the sampler `Outline`) are listed at the top of the file. These must be set properly by the application via the API in order for the shader to function correctly.

After the `ps_2_0` instruction, there is a `def` instruction of some magic constants. This `def` instruction is a free instruction that appears in the actual assembly instruction stream that defines constants that will be used by the subsequent ALU operations. This kind of constant definition is generally the result of literal values appearing in the HLSL shader, as in the following statements taken from the NPR Metallic pixel shader:

```
...
1 - dot (norm, normalize(View)
...
dot (normalize (Light1), norm) * 0.5 + 0.5
...
```

Following this constant definition, there are five 3D texture coordinate declarations of the form `dcl t`*`n`*`.xyz`. As in the vertex shader, these are a result of the semantics of the input parameters to this HLSL shader's `main` function. Following the texture coordinate declarations, there is a sampler declaration — `dcl_2d s0`. This indicates that a 2D texture must be bound to sampler zero. This may seem odd since the `tex1D()` intrinsic was used in the

HLSL shader. This discrepancy exists since there is no such thing as a 1D texture in the Direct3D API or shader assembly language. The `tex1D()` intrinsic is actually just a way for the HLSL shader writer to indicate to the compiler that only one component of the texture coordinate needs to be populated, shaving off an assembly instruction in some cases.

Now that you are familiar with some of the correspondence between HLSL and assembly code, we can discuss optimization strategies so that you can be sure that you are writing the best HLSL possible.

Optimization

While the DirectX HLSL compiler has an excellent optimizer built in, there are things that you can do as an HLSL coder to help shave off a few more cycles here and there. While this is probably more of an academic exercise in the long term, it may or may not make the difference between being able to target a legacy 1.*x* shader model using HLSL.

The most important thing to remember about writing high-performance shaders is that the compiler is required to do what you ask it to. That is, if you write your shader to require a certain number of math operations or a particular value in an output component, it needs to perform those operations. The compiler is smart about removing dead code, but it cannot know about values that do not ultimately matter due to circumstances outside of a given shader. For example, if the pixel shader is not using the second texture coordinate, the vertex shader probably shouldn't compute it. The HLSL compiler, of course, has no way of knowing this when you compile the vertex shader. Additionally, you may know that you will always use an $n \times 1$ function lookup texture at a given sampler, and hence it is not necessary to compute the second texture coordinate for use in the sampling intrinsic. If you use the `tex2D()` intrinsic, however, the HLSL compiler requires you to compute the second texture coordinate even though it is ultimately unnecessary. The compiler is designed to build an assembly program that does exactly what you asked without making any visual quality versus performance trade-offs.

Another extremely important objective for high-performance shaders is to make sure that a computation only runs at the required frequency. If you can get away with doing a calculation per vertex rather than per pixel, then do so. The biggest wins often come from these types of operations. The same optimization is true for operations on values that are uniform (i.e., operations that do not change for the entire execution of the shader). An example of this would be pre-multiplying the world ambient color value by an object's material ambient value and passing their product to the shader instead of redundantly calculating the product per vertex or per pixel.

The following sections go into some detail on how language features are mapped into assembly constructs. While it is not necessary to understand how to write vertex or pixel shader assembly, it can be quite helpful to understand the basic limitations and efficiencies of the assembly models. Understanding key assembly features is essential to generating compact and efficient shaders.

Matrix Data Type Usage

One of HLSL's more obvious departures from the C standard is the introduction of vector and matrix data types. The data types were added to enable easier writing of code and enable intrinsic functions to work properly, but correct usage of the data types allows for better code generation as well. The usage of vector types enables the compiler to more easily use all of the capabilities of the vector instructions. The compiler will automatically vectorize scalar operations when possible, but in general it is better to write your HLSL code in a vector-friendly form for best performance.

Although you can implement shaders with arrays of vectors instead of matrices, the recommended way to store a matrix is with a matrix data type. By using a matrix data type, the compiler has the choice to store internal matrices in either column major or row major order, depending on how the matrix is used. This optimization can be quite useful for situations in which a matrix is generated in either a pixel or vertex shader. As mentioned earlier, for input matrices, the compiler always uses either column major or row major storage format based on a compiler flag, with column major being the default method.

Integer Data Type Usage

It is important to understand the `int` data type and use it correctly in HLSL. It is very easy to generate extra instructions by using the `int` data type in places that it should not be used. The `int` data type was added to HLSL to make relative addressing familiar as well as efficient. A problem with using `float` data types for addressing purposes without truncation rules is that incorrect access to arrays can occur. For example, if the index variable is 2.5 and a `float4x4` matrix is being accessed, half of matrix 2 and half of matrix 3 will be used instead of truncating to access matrix 2 before accessing the matrix. In order to fix this, all `float`s that are used for accessing arrays must be rounded before being multiplied by the size of each element. This can be an expensive operation, since correct C rounding rules are not easily accomplished using the available instructions.

In order to avoid unwanted rounding or truncation, the `int` data type was added to mark values as being integer values. In order to properly avoid treating input data incorrectly as floating-point data, all inputs that are going to be used as integers should be defined as `int`s. For example, matrix palette indices read from a vertex stream component should be marked as `int`s. Declaring an input as `int` is a "free" operation in that no truncation is performed and the value is assumed to be an integer value. If the input is not declared as an `int`, the shader will not do what you expect. If, on the other hand, you cast a `float` to `int` in your shader or use a `float` for addressing purposes, a truncation will happen. Casting non-`int` intermediate values to `int` will also result in truncation overhead.

The following is code generated with a `float` index versus an `int` index:

```
OutPos = mul(Pos, WorldArray[Index]);

// Index declared as float        // Index declared as int
frc r0.w, r1.w                     mul r0.w, c60.x, r1.w
add r2.w, -r0.w, r1.w              mova a0.x, r0.w
mul r9.w, r2.w, c61.x              m4x4 oPos, v0, c0[a0.x]
mova a0.x, r9.w
m4x4 oPos, v0, c0[a0.x]
```

Flow Control and Performance

The most current vertex and pixel shader hardware does not support flow control. The hardware is designed to run a shader linearly, executing each instruction once. Newer hardware supports limited forms of flow control: *static branching*, *predicated instructions*, *static looping*, *dynamic branching*, and *dynamic looping*. Since HLSL can be compiled down to any or all of the models that support various degrees of flow control, it must be taken into consideration when writing shaders designed to run on more restricted models. As mentioned earlier, no restrictions are placed on the syntax of HLSL based on the compile target, but compile-time errors will occur if a shader is impossible to implement on the compile target used.

Loops are a form of flow control that occur quite often in shaders. Some hardware allows for either static or dynamic looping, but most require linear execution. On the models that do not support looping, all loops must be unrolled. While this can be an expensive operation, it can be used to generate excellent code with minimal effort. A good example is the DepthOfField sample from the DirectX 9 SDK that uses unrolled loops even for ps_1_1 shaders. In order to write high-performance shaders, you should keep this in mind — either for using the compiler to do the unrolling work for you or realizing when shaders will become unbounded and perform poorly or exceed instruction limits.

Using if statements can have large performance implications due to the lack of support for branching in most assembly-level shader models. In models that do not support any form of branching, both sides of an if must be executed and the output chosen based on which side of the if would have been taken. Having come from the CPU programming world, this form of execution is a bit different than most HLSL shader writers would expect. Common optimizations that you would use on a CPU to avoid expensive operations will not work as expected on shader models that don't support branches, since both the expensive path and the cheap path will be executed. Some shader models support different levels of branching: *predicated instructions*, *static if blocks*, and *dynamic if blocks*.

Example using if in vs_1_1:

```
if (Value > 0)
    Position = Value1;
else
    Position = Value2;
```

Assembly generated:

```
// calculate lerp value based on Value > 0
mov r1.w, c2.x
slt r0.w, c3.x, r1.w
// lerp between Value1 and Value2
mov r7, -c1
add r2, r7, c0
mad oPos, r0.w, r2, c1
```

The most common branching support in current hardware shading models is *static branching*. Static branching is a capability in a shader model that allows for blocks of code to be switched on or off based on a Boolean shader constant. This is a very convenient method for enabling/disabling potentially expensive code paths based on the type of object currently being rendered. Between draw calls, you can decide the various features that you want to support with the current shader and then set the Boolean flags required to get that behavior. The best part about this method is that any instructions that are "disabled" by the Boolean constant are completely skipped during execution. The disadvantage is that you can only change the if blocks that are enabled/disabled at a low frequency (i.e., between draw calls). In contrast, using the execute-both-sides approach, it is possible to dynamically choose between the outputs of the two paths dynamically at a per-pixel or per-vertex level.

The most familiar branching support is *dynamic branching*. The dynamic branching support offered by some shader models is very similar to that offered by a standard CPU. The performance hit is the cost of the branch plus the cost of the instructions on the side of the branch taken. This execution cost is comparable to what most people are familiar with optimizing for in CPU-side code. The problem with this form of branching is that it is not available on most hardware and is currently only available for vertex shaders. Optimizing shaders that work with these models is very similar to optimizing code that runs on a CPU.

Importance of Input Type Declarations

The *type* of an input to a shader is used differently than you might expect. The method in which data is loaded into the registers either from a vertex buffer into a vertex shader or from the vertex shader output to the pixel shader input registers is well-defined in the Direct3D spec. That is, shader input values are always expanded into a vector of four `floats`. This means that the data type declaration is more of a hint than a specification of how the data is loaded into the shader. Taking advantage of this provides a couple of optimization opportunities.

A common optimization used by shader assembly writers is to take advantage of the way in which data is expanded when loaded into registers. For example, in vertex shaders, the w component will be set to 1.0 if no w component is present in the vertex buffer. The y and z components will be set to 0.0 if not present in the vertex buffer. The most common place that this is useful is the position in vertex shaders. It is very common to need the w component to be set to 1.0 when multiplying by the world matrix, but the vertex buffer typically only contains x, y, and z components. If the position input parameter is declared as a `float3`, then an extra instruction to copy a 1.0 into the w component would be required.

If the parameter were declared as a `float4`, then the w component would be set to 1.0f by the hardware loading the input registers. The compiler cannot do this type of optimization automatically, since this optimization requires knowledge of what data is in the vertex buffer.

Another optimization is to make sure to declare all input parameters with the appropriate type for their usage in the shader. For example, if the incoming data is integer and the data is going to be used for addressing purposes, then it is important to declare the parameter as an `int` to avoid truncation. The subtle issue with declaring inputs as `int`s is that the values in the input should truly be integer values. Otherwise, the generated code might not run correctly due to the optimizations that the compiler will make based upon the assumption that the input data is truly integer data.

Precision Issues (`logp`, `expp`, `lit`)

A good understanding of precision is necessary for writing shaders that give correct results and reasonable performance. With most shader compile targets, the internal precision is fixed and needs to be taken into account for correct results. For example, the ps_1_*x* models have relatively low-precision fixed-point registers. Raising a number to even a low power for specular highlights can quickly generate banding.

In some other models, such as vs_1_1 and vs_2_0, there are low-precision versions of some instructions. Specifically, `logp`, `expp`, and `lit` can be used as low-precision versions of `log`, `exp`, and `pow`. On some hardware, the performance difference between the low- and high-precision variants is not significant. Since `log` and `exp` count for ten instruction slots each and `logp` and `expp` only count as one instruction each, it is possible to balloon the size of the generated asm code and potentially run out of instructions, particularly on the vs_1_1 compile target. Accessing these low-precision instructions is accomplished by declaring the output to be either cast to or stored into the low-precision data type called `half`. A low-precision output from an operation informs the compiler that the operation should be performed with the lowest precision possible. Some pixel shader hardware can take advantage of performing other operations at a lower precision as well.

Here is an example of `log` versus `logp`:

```
float LogValue = log(Value);             float LogValue = (half)log(Value)

// counts as 10 instructions             // counts as 1 instruction on
//                 on vs_1_1             //                      vs_1_1
log r0, c0;                              logp r0, c0
```

Using the ps_1_x Compile Targets

The original pixel shader models (ps_1_1, ps_1_2, ps_1_3, and ps_1_4) offer a large degree of programmability, but they have some restrictions in what can be done. It is possible to efficiently target the ps_1_*x* compile targets using HLSL, but it is imperative for the shader writer to understand the underlying set of limitations. This is important in order to write high-performance shaders and, more importantly, to even get your shader to compile. Instruction count is probably the first limitation that most people hit, but this is usually due to ignoring other limitations of the ps_1_*x* compile targets.

The first thing to remember about the ps_1_*x* compile targets is that the target hardware does not have arbitrary swizzles. This limitation means that the compiler must use extra instructions anytime that a swizzle is required. The extra instructions generated can quickly cause programs to overrun the total instructions possible in these compile targets. The ps_1_1 through ps_1_3 targets do not support arbitrary write masks or replicate swizzles (i.e., `.r`, `.g`, `.b`, or `.a`) and can cause the same situation. The ps_1_4 compile target does have support for replicate swizzles and arbitrary write masks. Even with these limitations, it is quite easy to write interesting and complex shaders. This is just something to keep in mind when writing HLSL code targeted at the ps_1_*x* compile targets.

While the ps_1_*x* targets naturally have disadvantages relative to the newer pixel shader models, one advantage that they do have is the existence of "free" source and dest modifiers (i.e., the ability to clamp values to the 0 to 1 range, take the complement of a source, negate a source, bias a source, etc.). These modifiers are extremely handy when generating shaders that accomplish a lot in a small number of instructions. The compiler automatically matches all modifiers that it can, but it is helpful if the HLSL shader writer thinks in terms of using these modifiers to accomplish certain operations. In fact, some intrinsics were added to HLSL to make this type of shader writing easier. For example, it is recommended that you use the `saturate()` intrinsic when trying to generate a free `_sat` modifier in a pixel shader.

We now present a series of HLSL code sequences that generate free source modifiers when compiling to ps_1_*x* targets.

The _bx2 Modifier

There are a number of different HLSL code sequences that can be used to cause the HLSL compiler to generate `_bx2` modifiers. Any of the following `main` functions will cause the compiler to generate a `_bx2` modifier:

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, Tex*2 - 1);
}

float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    float3 val = Tex*2;
    val = val -1;
    return dot(Col,val);
}

float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, (Tex -.5f)*2);
}
```

All of these main functions generate the same asm shader:

```
ps_1_1
texcoord t0
dp3 r0, v0, t0_bx2
```

It is important to note that the Tex*2 -1 version is recommend because it generates more optimal code in ps_2_0 targets and beyond.

The _bias Modifier

The following code causes the HLSL compiler to generate a _bias modifier:

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, (Tex - .5f));
}
```

This main function generates the following assembly shader:

```
ps_1_1
texcoord t0
dp3 r0, v0, t0_bias
```

Note that _bias cannot be done in ps_1_1, ps_1_2, or ps_1_3 unless the source is known to be in the range of 0 to 1. That is, it must have been previously saturated.

The _x2 Modifier (ps_1_4 only)

The following code causes the HLSL compiler to generate an _x2 source modifier:

```
float4 main( float3 Col : COLOR0, float3 Tex : TEXCOORD0 ) : COLOR0
{
    return dot(Col, Tex*2);
}
```

This HLSL code results in the following asm shader code:

```
ps_1_4
texcrd r0.xyz, t0
dp3 r0, v0, r0_x2
```

The _x2, _x4, _x8, _d2, _d4, and _d8 Destination Write Modifiers

A set of destination write modifiers exists in the ps_1_*x* models, and it is possible to write HLSL code to cause the compiler to generate them in the resulting asm. The modifiers to double (_x2), quadruple (_x4), and halve (_d2) the result of the instruction are supported on ps_1_1 through ps_1_3 models, while the ps_1_4 model supports all six of the modifiers — _x2, _x4, _x8, _d2, _d4, and _d8. The following code will generate the corresponding modifiers for N = 2, 4, 8, 0.5, 0.25, or 0.125:

```
static const float N = 2;

float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
   return (Col[0] + Col[1] )*N;
}
```

The above HLSL code results in the following asm output:

```
ps_1_1
add_x2 r0, v0, v1
```

The Complement Modifier

It is also possible to write HLSL code that allows the compiler to generate a complement modifier when compiling to a ps_1_*x* target. Note that this only works if the quantity being complemented is known to be in the 0 to 1 range (i.e., the quantity has previously been saturated). The following HLSL code causes the compiler to generate a free complement modifier:

```
float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
   return (1-Col[0]) * (Col[1]);
}
```

This HLSL code results in the following asm shader:

```
ps_1_1
mul r0, 1-v0, v1
```

The Saturate Modifier

The following two shaders generate a `_sat` modifier. Note that this modifier is available on all pixel shader compile targets:

```
float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
    return saturate(Col[0]);
}
```

```
float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
    return clamp(Col[0],0,1);
}
```

Both of these HLSL shaders result in the following asm shader:

```
ps_1_1
mov_sat r0, v0
```

The Negate Modifier

The following shader generates a negate modifier, which is also available on all shader targets.

NOTE On ps_1_*x*, constant registers cannot be directly negated and hence will not result in a single free negation, since the constant will have to be moved to a temp before it can be negated.

```
float4 main( float4 Col[2] : COLOR0 ) : COLOR0
{
    return -Col[0];
}
```

This HLSL code will result in the following asm shader:

```
ps_1_1
mov r0, -v0
```

Strategy for Targeting ps_1_x

The best strategy that we have found to optimize for ps_1_*x* compile targets is to first write your shader on ps_2_0, since this allows for quick and easy prototyping on ps_2_0 capable hardware. Once the shader is working as desired, cross-compile it for the desired ps_1_*x* model. Using the disable validation option, `-Vd` for fxc.exe, you can see how many instructions the shader would have if there were no instruction limits on the chosen ps_1_*x* model. If the shader did not fit, you can at least see what you are up against and begin paring away the least necessary pieces of your shader to get an efficient ps_1_*x* fallback up and running.

Now that we have presented HLSL shaders in detail, we can discuss the issues involved in integrating HLSL shader support into an application. HLSL can be integrated into your engine with or without the use of D3DX Effects, and we discuss both approaches. It is also worth mentioning that it is possible to start experimenting with HLSL without writing any application code by using a shader development environment, such as RenderMonkey. For more on RenderMonkey, please consult the "Shader Development Using RenderMonkey" article in this book.

Integration into an Engine Using D3DX Effects

The D3DX Effects framework is a very useful component of the D3DX library that is gaining more attention from professional developers. Naturally, in DirectX 9, D3DX Effects was updated to include support for HLSL. If you aren't familiar with D3DX Effects, it is an abstraction designed to conveniently encapsulate rendering special effects in 3D applications. Effects can encapsulate rendering states as well as shaders written in asm or HLSL, including fallback versions targeted at legacy hardware. A given *effect* is generally stored in a single .fx or .fxl file, and the file itself can contain multiple versions of the effect called *techniques*. For example, you may want to create a more basic version of a given effect that you can use on older hardware with legacy shader support or no shaders at all. An excellent example of this kind of use of techniques is the Water sample in the DirectX SDK. This sample uses several different techniques that are targeted at different generations of hardware. Of course, the more basic techniques that require fewer textures and generally less sophistication don't look as impressive, but that's the point; D3DX Effects let you manage this quality/speed trade-off very naturally.

Effect Files

We don't go into all of the facets of effects here, but you should understand the basic structure of an effect file in order to see how it can be used with HLSL. A typical effect file might look something like this:

```
// Lighting constants
VECTOR g_Leye;
float4 GlobalAmbient = 0.5;
float Ka = 1;
float Kd = 0.8;
float Ks = 0.9;
float roughness = 0.1;
float noiseFrequency;

MATRIX matWorldViewProj;
MATRIX matWorldView;
MATRIX matITWorldView;
MATRIX matWorld;
MATRIX matTex0;

TEXTURE tVolumeNoise;
TEXTURE tMarbleSpline;

sampler NoiseSampler = sampler_state
{
   Texture = (tVolumeNoise);

   MinFilter = Linear;
   MagFilter = Linear;
   MipFilter = Linear;
   AddressU  = Wrap;
   AddressV  = Wrap;
   AddressW  = Wrap;
   MaxAnisotropy = 16;
};

sampler MarbleSplineSampler = sampler_state
{
   Texture = (tMarbleSpline);

   MinFilter = Linear;
   MagFilter = Linear;
   MipFilter = Linear;
   AddressU  = Clamp;
   AddressV  = Clamp;
   MaxAnisotropy = 16;
};

float3 snoise (float3 x)
{
    return 2.0f * tex3D (NoiseSampler, x) - 1.0f;
}


float4 ambient(void)
{
   return GlobalAmbient;
}


float4 soft_diffuse(float3 Neye, float3 Peye)
{
   // Compute normalized vector from vertex to light in eye space  (Leye)
   float3 Leye = (g_Leye - Peye) / length(g_Leye - Peye);

   float NdotL = dot(Neye, Leye) * 0.5f + 0.5f;

   // N.L
   return float4(NdotL, NdotL, NdotL, NdotL);
}


float4 specular(float3 NNeye, float3 Peye, float k)
{
    // Compute normalized vector from vertex to light in eye space  (Leye)
    float3 Leye = (g_Leye - Peye) / length(g_Leye - Peye);

    // Compute Veye
    float3 Veye = -(Peye / length(Peye));

    // Compute half-angle
    float3 Heye = (Leye + Veye) / length(Leye + Veye);

    // Compute N.H
    float NdotH = clamp(dot(NNeye, Heye), 0.0f, 1.0f);

    float NdotH_2  = NdotH    * NdotH;
    float NdotH_4  = NdotH_2  * NdotH_2;
    float NdotH_8  = NdotH_4  * NdotH_4;
    float NdotH_16 = NdotH_8  * NdotH_8;
    float NdotH_32 = NdotH_16 * NdotH_16;

    return NdotH_32 * NdotH_32;
}

float4 hlsl_bluemarble (float3 P : TEXCOORD0, float3 Peye : TEXCOORD1, float3
          Neye : TEXCOORD2) : COLOR
{
   float4 Ct;
   float4 Ci;
   float3 NNeye;
   float marble;
   float f;

   // Divide down to nice frequency
   P = P/16;

   marble = -2.0f * snoise(P * noiseFrequency) + 0.75f;

   NNeye = normalize(Neye);

   // Cubic interpolation of f along color spline (gloss in alpha)
   Ct = tex1D (MarbleSplineSampler, marble);

   // Color from illumination
   Ci = Ct * (Ka * ambient() + Kd * soft_diffuse(NNeye, Peye)) + Ct.w * Ks *
            specular(NNeye, Peye, roughness);
   return Ci;
}

VERTEXSHADER asm_marble_vs =
decl {}
asm
{
   vs.1.1

   dcl_position v0
   dcl_normal   v3

   m4x4 oPos, v0, c[0]           // Transform position to clip space

   m4x4 r0, v0, c[17]            // Transformed Pshade (using texture matrix 0)
   mov oT0, r0

   m4x4 oT1, v0, c[4]            // Transform position to eye space
   m3x3 oT2.xyz, v3, c[8]        // Transform normal to eye space
};


technique technique_hlsl_bluemarble
{
   pass P0
   {
      // Only need to map variable names to hardware
      // registers like this for asm shaders:
      VertexShaderConstant[0]  = <matWorldViewProj>;
      VertexShaderConstant[4]  = <matWorldView>;
      VertexShaderConstant[8]  = <matITWorldView>;
      VertexShaderconstant[12] = <matWorld>;
      VertexShaderConstant[17] = <matTex0>;

      VertexShader = <asm_marble_vs>;
      PixelShader  = compile ps_2_0 hlsl_bluemarble();

      CullMode = CCW;
   }
}
```

We now explain this example effect file from the bottom up. The very last block of code in this effect file defines a technique called `technique_hlsl_bluemarble`, which has only one rendering pass.

This single pass will use a vertex shader written in assembly language and a pixel shader written in HLSL. The first several lines in this pass declare five different matrices, which will be loaded into specific hardware constant registers from high-level effect variables when this pass is invoked. This explicit mapping is only done in the effect file for asm shaders. There are no explicit mappings done like this for the pixel shader, since it is written in HLSL. The next line declares the vertex shader to be used in this pass, an assembly shader called `asm_marble_vs`:

```
    VertexShader = <asm_marble_vs>;
```

The following line defines the pixel shader, which will be compiled for the ps_2_0 target using the `hlsl_bluemarble()` function as its main entrypoint:

```
    PixelShader  = compile ps_2_0 hlsl_bluemarble();
```

The block of code preceding the technique definition is the vertex shader written by hand in assembly language. Above this is `hlsl_bluemarble`, the main entrypoint for our HLSL pixel shader. If you take a look at this code, you can see that, in addition to the `tex1D()` intrinsic, this function calls several other utility functions, such as `ambient()` and `soft_diffuse()`. These utility functions are defined earlier in this effect, and since we're compiling for the ps_2_0 target, they are inlined into the resulting assembly.

If you look above the utility functions, you can see the declaration of a pair of samplers called `NoiseSampler` and `Marble-SplineSampler`. These are declared just as before except that when used in an effect file, they can also be followed by the bracketed code defining the addressing and filtering sampler state to be used. Textures may also be defined in effect files, as shown above the sampler declarations. At the very top of the effect, we see the declaration of a series of global variables, which are settable from the application level.

The Effect API

Now that we have written an effect and stored it in a file, we wish to use it from our application code. Naturally, the first thing that we do is create the effect using the `D3DXCreateEffectFromFile()` API. Assuming this succeeds, we can use the effect API to set the appropriate variables needed by our effect. For example, we can set the matrices with the `SetMatrix()` entrypoint:

```
// Set all the matrices
m_pEffect->SetMatrix ("matWorldViewProj", &m_matWorldViewProj);
m_pEffect->SetMatrix ("matWorldView", &m_matWorldView);
m_pEffect->SetMatrix ("matITWorldView", &m_matITWorldView);
m_pEffect->SetMatrix ("matWorld", &m_matWorld);
m_pEffect->SetMatrix ("matTex0", &m_ObjectParameters.m_matTex0);
```

We could also set any `floats` and vectors similarly:

```
m_pEffect->SetFloat ("noiseFrequency ", &m_fNoiseFreq);
m_pEffect->SetVector("g_Leye", &g_Leye);
```

Likewise, with textures:

```
m_pEffect->SetTexture("tVolumeNoise",  m_pVolumeNoiseTexture);
m_pEffect->SetTexture("tMarbleSpline", m_pMarbleColorSplineTexture);
```

With all of the proper constants set up, we can set the desired technique and render all of its passes (in this case, just one):

```
m_pEffect->SetTechnique(m_pEffect->GetTechniqueByName("technique_hlsl_
          bluemarble"));

m_pEffect->Begin(&cPasses, 0);
for (iPass = 0; iPass < cPasses; iPass++)
{
   m_pEffect->Pass(iPass);

   // Render geometry
}
m_pEffect->End();
```

As you can see, this is a straightforward process that hides several unnecessary burdens from the application. For example, the application never needs to know into what hardware constant register to load g_Leye or to which sampler the noise texture should be bound. These details are all managed by the D3DX Effects framework.

Integration into an Engine without Using D3DX Effects

We have found that some ISVs prefer not to wed their code too closely to D3DX because of cross-platform development or overhead concerns. As a result, while the use of D3DX Effects for HLSL shader management is very convenient, it is not required. Of course, giving up the convenience of D3DX Effects means that the application will have to take responsibility for tracking and setting up the appropriate constants and samplers prior to rendering with a given shader. Let's discuss how this is done.

Since you won't be creating D3DX Effects that trigger compilation of HLSL code, you must invoke the HLSL compiler explicitly in your application. In fact, this is very similar to the application code that you would write for the use of assembly shaders, except you call one of the D3DXCompileShader*() routines instead of one of the D3DXAssembleShader*() routines. You then pass the resulting asm code to the appropriate CreatePixelShader() or CreateVertexShader() entrypoint, just as you would for an assembly shader that was assembled rather than compiled. An example of this usage is shown in the following code snippet:

```
if (FAILED (hr = D3DXCompileShaderFromFile (g_strVHLFile, NULL, NULL, "main",
         "vs_1_1", NULL, &pCode, NULL, &m_VS_ConstantTable)))
{
   return hr;
}

if (FAILED (hr = m_pd3dDevice->CreateVertexShader ((DWORD*)pCode->
         GetBufferPointer(), &m_HLSLVertexShader)))
{
   return hr;
}
```

Notice in the above code that the D3DXCompileShader*() routines have some additional parameters not found in the D3DXAssemble-Shader*() routines. Specifically, it is necessary to specify the name of the main entrypoint for the shader as well as the compile target ("main" and "vs_1_1" above). You can also optionally specify values of #defines, include files, and flags to control generation of debug information, optimization, validation, and matrix data ordering. All of these inputs are passed to the D3DXCompileShader*() routines via the first six parameters. The last three parameters are pointers to buffers that get filled in by the compiler — the binary assembly code, human-readable error messages (optional), and the constant table. The binary assembly code gets passed to CreatePixelShader() or CreateVertexShader(), while the constant table must be used by the application to know how to load the proper constant data prior to executing a given HLSL shader. We devote the remainder of this discussion to the final parameter returned from the D3DXCompileShader*() routine, as this is the most critical piece to understand when integrating HLSL shaders into an application without the use of effects. You can refer to the documentation for discussion of the other parameters.

The Constant Table

The constant table returned from the D3DXCompileShader*() routine is used to map high-level constants and samplers to specific hardware constants and samplers. Non-static variables declared at global scope are considered input parameters to the compiled shader and must be properly initialized in order for the shader to execute correctly. The constant table provides this mapping. Typically, it is most convenient for an application to use the ID3DXConstantTable interface directly, as this does not require the application to parse the actual data structures of the constant table. The ID3DXConstantTable interface provides a number of convenient methods for looking up *handles* of known HLSL variables based upon their ASCII names. The appropriate values for these HLSL variables may then be set as shown in the following code snippet:

```
D3DXHANDLE handle;

if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "ringFreq"))
{
   m_PS_ConstantTable->SetFloat(m_pd3dDevice, handle, m_fRingFrequency);
}

if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "lightWood"))
{
   m_PS_ConstantTable->SetVector(m_pd3dDevice, handle, &lightWood);
}
```

Likewise, textures and sampler states must be set up correctly, as shown in the following code snippet:

```
if (handle = m_PS_ConstantTable->GetConstantByName(NULL, "NoiseSampler"))
{
   m_PS_ConstantTable->GetConstantDesc(handle, &constDesc, &count);

   if (constDesc.RegisterSet == D3DXRS_SAMPLER)
   {
      m_pd3dDevice->SetTexture (constDesc.RegisterIndex,
               m_pVolumeNoiseTexture);

      // Set sampler states appropriate for the Noise Sampler
      m_pd3dDevice->SetSamplerState (constDesc.RegisterIndex, …, …);

   }
}
```

The implication of this, of course, is that render states, texture stage states, and sampler states must be maintained by the application and are in no way encapsulated in the HLSL shader code as they would be using D3DX Effects.

Of course, particularly in any kind of shader-authoring tool, there may be no *a priori* application knowledge of the names of variables or samplers expected. In this case, it is necessary to use the `ID3DXConstantTable::GetDesc()` method to determine the number of constants in the constant table. Subsequently, the application can use the `ID3DXConstantTable::GetConstantElement()` method rather than the `ID3DXConstantTable::GetConstantByName()` method used in the code snippets above. In general, it is a good idea to familiarize yourself with the `ID3DXConstantTable` interface if you intend to integrate support for HLSL shaders into your application without the use of D3DX Effects.

SDK Updates

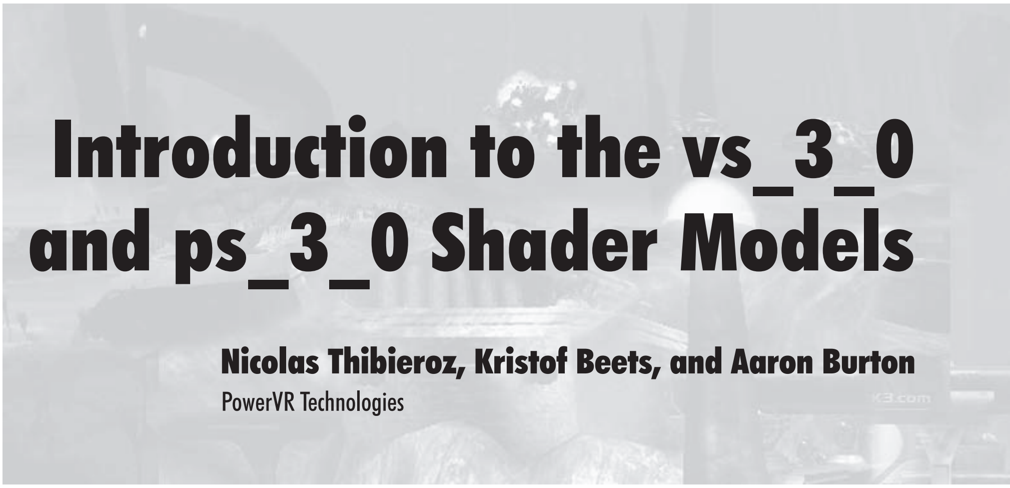
Since the release of DirectX 9.0 and the subsequent DirectX 9.0a patch, Microsoft has committed to releasing periodic SDK updates for developers. These SDK updates do *not* contain Direct3D run-time changes, but they do include upgrades to important D3DX tools, including the HLSL compiler. It is highly recommended that you keep up to date with the latest release of DirectX SDK updates so that you are using the latest compiler revision and generating the best possible asm from your HLSL source.

Conclusion

We have presented a detailed description of the Direct3D High Level Shading Language (HLSL), which is one of the most significant new features of DirectX 9.0. We have presented an introduction to the mechanics of the language itself and reinforced key concepts with sample shaders. We have also given some insight into the compilation process and how you can best write shaders for optimal performance. We hope this introduction has provided you with a solid foundation so that you can understand the HLSL shaders presented in later chapters and begin integrating HLSL shaders into your own projects.

Acknowledgments

Thanks goes to ATI's 3D Application Research Group for providing the sample HLSL shaders. Thanks to Dan Baker and Loren McQuade of Microsoft for their feedback and specifically their contributions to the section on optimizations. Thanks also to Mark Wang and Wolfgang Engel for valuable comments that resulted in greater clarity.



Introduction to the vs_3_0 and ps_3_0 Shader Models

Nicolas Thibieroz, Kristof Beets, and Aaron Burton
PowerVR Technologies

Introduction

DirectX 9 introduces the new shader model 2.0 whose capabilities clearly exceed their DirectX 8 counterparts. However, the same DirectX 9 release also includes the 3.0 shading model whose advanced vertex and pixel processing features open the door to a plethora of new techniques and effects previously not possible in real-time 3D rendering. While the "extended" shader model 2.x offers some functionality common to its 3.0 counterpart, its availability depends on a number of capabilities that may or may not be exposed, depending on implementations. Vertex and pixel shaders 3.0 raise the bar and require a base feature set for 3D acceleration hardware supporting this model, making it easier to determine the capabilities of the rendering device. They also share the same unified structure and syntax, making the writing of shaders an intuitive and straightforward process. For this reason, a vs_3_0 program must always be associated with a ps_3_0 program and vice versa. This article describes the new features of this shader model in detail while giving practical examples of effects that can be implemented with it.

Features Common to vs_3_0 and ps_3_0

Flexible Input and Output Declarations

New to the 3.0 shader model is the obligation to declare input (for both vertex and pixel shaders) and output (vertex shader only) registers prior to their utilization. However, declared registers come with the added facility of being able to contain more than one declaration type. This means that even if the number of inputs to a shader program exceeds the limit of input registers allowed, inputs can be packed together into registers during the declaration phase of the shader. As an example, a vertex shader 3.0 declaration could accept position, normal, tangent, binormal, two colors, eight blending weights, eight blending indices, and 14 2D texture coordinates by optimal use of input registers packing. With the 2.0 model, such a declaration would only be possible by prepacking all this data in the vertex structure using 16 total inputs. Outputs are declared the same way as the inputs. An example of a vs_3_0 declaration is given below:

```
vs_3_0

; Declare inputs
dcl_position0    v0.xyzw
dcl_normal0      v1.xyz
dcl_tangent0     v2.xyz
dcl_binormal0    v3.xyz
dcl_blendweight0 v1.w
dcl_blendweight1 v2.w
dcl_blendweight2 v3.w
dcl_texcoord0    v4.xy
dcl_texcoord1    v4.wz

; Declare outputs
dcl_position     o0.xyzw
dcl_texcoord0    o1.xy
dcl_texcoord1    o1.zw
dcl_texcoord2    o2.xyz
dcl_fog          o2.w
dcl_psize        o3
```

As the number of vertex shader outputs and pixel shader inputs is the same (12), this feature is not as useful in the pixel shader as it is in the vertex shader. The preferred method of selecting pixel shader inputs is arbitrary source swizzling, which is covered later in this article.

Flexible input and output declarations also allow different vertex and pixel shaders to be paired together without having to ensure they all use exactly the same register assignment. This can be a useful feature when dealing with a large number of vertex and pixel shader programs.

Predication

The predicate register (`p0`) is a set of four Boolean flags (one per x, y, z, and w channel) that is basically a "dynamic write mask." It enables shader instructions to be performed on a per-channel basis based on the results of previous calculations. The flags in the predicate register are set with the `setp_comp p0, src1, src2` instruction, where `comp` is a comparison mode (greater than, less than, etc.), `p0` is the predicate register, and `src1` and `src2` are two input registers. The comparison is performed four times on the corresponding components of the source registers, and the results are stored in the Boolean flags of the predicate register. For example, the following code sets the predicate register components to (false, true, false, false):

```
def c0, 0.0f, 2.0f, -4.0f, 1.0f
def c1, 4.0f, -8.0f, -2.0f, 1.0f
mov r0, c0
setp_gt p0, r0, c1
```

Once the predicate register is set, its contents can be used to allow or prevent per-channel operations to be carried out. To enable predication, `(p0)` is added in front of the corresponding arithmetic or texture instruction. For example, based on the predicate register contents defined above, only the `.y` component of the destination register `r0` is affected by the result of the following instruction:

```
(p0) mul r0, r1.x, c1
```

A negate modifier (`!`) and single-component replicate swizzle can also be used with the predicate register. In the following example (and using the same predicate contents as before), all the components of `r0` receive the multiplication results:

```
(!p0.z) mul r0, r1.x, c1
```

Using predication as a dynamic write mask has its uses; for very short branching sequences, it should be preferred instead of the dynamic branching instructions like `if_comp`. The predicate register also uses fewer temporary registers compared to the equivalent non-predicated sequence of instructions, which can help compiler optimizations and may produce better code.

Static and dynamic flow control instructions like `loop`, `if_comp`, etc., may not be used in predication mode, although the predicate register can be used as a branching condition using the dedicated flow control instructions `if_pred`, `callnz_pred`, and `break_pred`. A replicate swizzle *must* be used with those instructions in order to determine which component triggers the branch.

Static and Dynamic Flow Control

The vs_2_0 model supports static flow control (i.e., branching instructions like loops and subroutines that are called based on values in constant registers). Static flow control allows the combining of different code paths into long shaders, reducing the number of shader state changes in the process. Static loops can also be useful when a fixed number of iterations are performed (e.g., looping through a set number of lights). The only difference in the 3.0 model with regard to static flow control is the nesting level. While vs_2_0 does not support nesting (i.e., having loops within loops), both vs_3_0 and ps_3_0 support static flow control with a nesting depth of 4. However, the real power of the 3.0 model comes with the ability to support *dynamic* flow control.

Dynamic flow control is a way to specify different code paths based on the comparison of registers whose contents are modified dynamically within the shader program. There are two main advantages to this feature: flexibility and performance. Flexibility because different branching instructions are now executed at the vertex or pixel level, allowing complex code trees to be implemented. Performance because code can now be run only for the vertices or pixels that require it (although the performance gained from unexecuted code may vary depending on hardware implementations). Dynamic branching instructions can be nested up to 24 levels deep; a description of the instructions follows:

- **`if_comp`**: Conditionally performs the next sequence of instructions based on a comparison. The `else/endif` instructions are used to delimit the if blocks.
- **`if_pred`**: Conditionally performs the next sequence of instructions based on the value of the predicate register. The `else/endif` instructions are used to delimit the if blocks.
- **`callnz_pred`**: Conditionally calls a subroutine based on the value of the predicate register. The `ret` instruction is used to return from the subroutine.
- **`break_pred`**: Conditionally breaks from a `loop/endloop` or `rep/endrep` block based on the value of the predicate register.
- **`break_comp`**: Conditionally breaks from a `loop/endloop` or `rep/endrep` block based on a comparison.

A typical application of dynamic branching is the common ($\vec{N}.\vec{L}$) calculation (dot product of the normal and light vector). Depending on the result of the dot product, the rest of the lighting equation may or may not be calculated, improving performance in the process. The following pixel shader illustrates this:

```
ps_3_0

; User-defined constants
def c0, 0.0f, 0.0f, 0.0f, 1.0f

; Declare samplers
dcl_2d          s1        ; Normal map

; Declare inputs
dcl_texcoord0   v0.xy     ; Texture coordinates
```

```
dcl_texcoord1     v1.xyz     ; Un-normalized light vector

texld r2, v0, s1             ; Retrieve pixel normal
nrm r1, v1                   ; Normalize light vector

dp3 r0.w, r2, r1             ; Light calculation (N.L)

if_gt r0.w, c0.x             ; if (N.L)>0

  ; Performs the rest of the lighting equation: specular,
  ; attenuation, light maps, etc. r3 contains the final pixel
  ; color

else

  ; Output black (or any other ambient color)
  mov r3, c0

endif

mov oC0, r3                 ; Output pixel color
```

The same principle can be applied to shadows (in or out of shadow), light attenuation (distance from the light exceeds maximum range), etc. Many optimizations can be performed using dynamic branching.

NOTE For small portions of conditional code, it is usually preferable to use the predicate register or other comparison instructions than to start a dynamic branch. There may be a setup cost associated with dynamic branching, and so running a few instructions for all conditions could be faster than running fewer instructions in separate branches.

The `break` instructions are used to break from loops (using `loop/endloop` or `rep/endrep` instructions), which can be useful for iterative mathematical operations. By breaking when the right result is found, the remaining loop iterations are not executed, thus improving performance.

Dynamic flow control allows numerous new effects to be implemented in vertex or pixel shaders. Recursion, tree structures, ray tracing, etc., are all possible with dynamic flow control.

Arbitrary Swizzle

Arbitrary source swizzling is now supported for both vs_3_0 and ps_3_0 (arbitrary source swizzling was not supported in ps_2_0). This feature allows the selection of source components to be specified in *any* order and eliminates the need to copy or modify registers when their component arrangement does not match the format required for the next instruction.

Arbitrary source swizzling is compatible with texture instructions (in both vertex and pixel shaders), thus, texture coordinates can be selected in any order from a given set of coordinates. This is very useful when filter kernels are involved, as several sample points can be fetched simply by using source swizzles on the texture coordinates. The following example fetches five samples in an x-shaped kernel from a single set of 2D texture coordinates:

```
;----------------------------------------------------------------
; Constants specified by the app
;   c0 =  -1/TextureWidth,  1/TextureHeight,
;          1/TextureWidth, -1/TextureHeight
;----------------------------------------------------------------
ps_3_0

; Declare samplers
dcl_2d s0            ; Texture to sample from

; Declare inputs
dcl_texcoord0 v0.xy  ; Texture coordinates UV

; Prepare all possible texture coordinate values
add r0, v0.xyxy, c0  ; r0 = (U-texel, V+texel,
                     ;       U+texel, V-texel)

; Fetches all 5 samples ('X' shape)
texld r1, v0, s0     ; Texel at (U, V)
texld r2, r0.xw, s0  ; Texel at (U-texel, V-texel)
texld r3, r0.zw, s0  ; Texel at (U+texel, V-texel)
texld r4, r0.xy, s0  ; Texel at (U-texel, V+texel)
texld r5, r0.zy, s0  ; Texel at (U+texel, V+texel)
```

Interestingly, arbitrary source swizzling also works on the sampler registers. It is possible to swap or replicate color channels by using the appropriate swizzle with the sampler register. For instance, the following instruction changes the channel ordering from the default RGBA to ABGR when sampling a texel:

```
texld r0, v0, s0.abgr
```

Arbitrary source swizzles not only improve performance by avoiding copy or replicate instructions, but they also make shader code more readable by doing so.

Destination Write Masks on Texture Instructions

Both vs_3_0 and ps_3_0 support destination write masks on texture instructions, and so only the selected color channels in the destination register are updated with the results of the texture sampling. This allows the contents of masked components to be preserved during a texture instruction. The following example combines destination write masks and arbitrary source swizzling to sample two 32-bit textures of D3DFMT_R16G16F format containing position data (`XY` in `s0`, `ZW` in `s1`) and directly store the results into a destination register:

```
ps_3_0

; Declare samplers
dcl_2d          s0     ; Contains XY data
dcl_2d          s1     ; Contains ZW data

; Declare inputs
dcl_texcoord0     v0.xy  ; Texture coordinates

; Retrieve position data
texld r0.xy, v0, s0      ; Sample RG data into r0.xy
texld r0.zw, v0, s0.abrg ; Sample RG data into r0.zw

; r0.xyzw now contains position data
```

NOTE The predicate register can also be used to specify dynamic write masks on a texture sampling instruction.

vs_3_0 Features

Registers

A total of 32 temporary registers (r0...r31) are available in the vs_3_0 model, compared to a mere 12 for the vs_2_0 model. This number of registers provides more storage for complex mathematical functions as well as extra parameters for subroutines (see the "Static and Dynamic Flow Control" section above).

To increase the flexibility of the shader, the 12 output registers have been renamed to `oX` (`o0-o11`) and can now contain any float values that will be iterated and supplied to the pixel shader. Of those, only ten are custom four-component output registers, as one register must be declared as the output position and the remaining one may only be used for point sprite size. For more information on vertex shader declarations, see the "Flexible Input and Output Declarations" section in this article.

The loop counter register `aL`, used in vs_2_0 to index constants within a loop, can now also be used to relatively address both input and output registers. This enables the same piece of code to operate on a set of different inputs. This can be useful, for instance, to apply the same transformations to a set of vertex positions or output the results of per-vertex light vector calculations to texture coordinates. The following code gives an example of output register indexing in a vs_3_0 program.

```
;----------------------------------------------------------------
; Constants specified by the app
; c0-c3   = Global transformation matrix (World*View*Projection)
; c12-c19 = Model space positions of light sources
;----------------------------------------------------------------
vs_3_0

; Declare constant integer for looping
```

```
defi i0, 8, 2, 1, 0         ; Loop 8 times, starting from 2 and
                            ; incrementing by 1 each iteration
def c4, 0, 0, 0, 1          ; Static constant

; Declare input registers
dcl_position0 v0            ; Input position

; Declare output registers
dcl_position0 o0.xyzw       ; Output position
dcl_texcoord0 o2.xyzw
dcl_texcoord1 o3.xyzw
dcl_texcoord2 o4.xyzw
dcl_texcoord3 o5.xyzw
dcl_texcoord4 o6.xyzw
dcl_texcoord5 o7.xyzw
dcl_texcoord6 o8.xyzw
dcl_texcoord7 o9.xyzw       ; Per-vertex light vector xyz and
                            ; distance w for 8 lights

; Vertex transformation
m4x4 o0.xyzw, v0, c0        ; Transform vertices by WVP matrix

; Set r0.w to 1 (used in distance calculation later on)
mov r0.w, c4.w

; Lighting pre-processing
loop aL, i0                 ; Start loop, aL will loop from 2 to 9

  ; Compute vertex-to-light vectors and distance
  sub r0.xyz, c[aL+10], v0  ; Subtract model space light
                            ; position from vertex position
  nrm r1, r0                ; Normalize vector
  rcp r1.w, r1.w            ; 1/(1/distance) = dist(light, vertex)

  mov o[aL], r1             ; Store result in corresponding texture
                            ; coordinate output
endloop
```

Other registers (16 input registers, 256 constant float registers, 16 constant integer registers, 16 constant Boolean registers, address register) remain unchanged compared to vs_2_0.

Instructions

The new vs_3_0 model supports a minimum of 512 instructions in a vertex shader program compared to 256 for the vs_2_0 model. Note that the number of executed instructions can potentially be made larger by the use of loops and subroutines within the vertex shader. Supporting longer shaders not only enables more operations to be performed like advanced animation, complex vertex lighting, etc., but also concatenating different shaders into a larger one reduces or even eliminates vertex shader state changes, improving performance.

The _abs source modifier is a new addition to vs_3_0. It forces the absolute value of a source register to be used in an instruction. Note that it takes precedence over the negate modifier (-) so that a negative value can always be guaranteed. Here are a few examples:

```
add r0, r8_abs.x, c10       ; Adds the absolute value of r8.x
                            ; and c10 together
mad r0, r1, r2, -v_abs[2]   ; Multiplies r1 and r2 and subtracts
                            ; the absolute value of v2. Note
                            ; that –v2_abs also works.
```

In an effort to unify the vertex and pixel shader models, the _sat instruction modifier that was available in ps_2_0 has been included in vs_3_0. Applying this modifier clamps the result to the [0,1] range:

```
sub_sat r0, r0, r1          ; Subtracts r1 from r0 and clamps
                            ; the result to the [0,1] range
```

New instructions in the vs_3_0 model that relate to dynamic branching are discussed in the "Static and Dynamic Flow Control" section.

Texture Sampling

The 2.0 model introduced basic texture sampler functionality to the vertex shader unit. This access was limited to a single texture with a fixed set of texture coordinates — either read directly from the vertex stream (which supports filtering) or derived from the vertex index (which supports point sampling only) and only in combination with n-patches.

The 3.0 model introduces true vertex texturing support, which is texture access from the vertex shader at the same level of functionality and flexibility existing in the pixel shader unit. Using this new functionality is also very similar to using textures in the pixel shader; textures (`SetTexture`) and sampler states (`SetSamplerState`) simply have to be set for the four available vertex texture sampler stages (D3DVERTEXTEXTURESAMPLER0, D3DVERTEXTEXTURESAMPLER1, D3DVERTEXTEXTURESAMPLER2, and D3DVERTEXTEXTURESAMPLER3) with the same arguments used for regular textures. These samplers also need to be declared as part of the shader program using the `dcl_textureType s#` syntax, where the texture type can be `2d`, `cube`, or `volume`. The only difference with textures in the pixel shader is that anisotropic filtering is not supported for vertex textures. Also, because the rate of change information is not available, the shader or application has to compute the level of detail (LOD) and provide that information as a parameter to the actual texture sampling instruction. Hence, only the `texldl` instruction is supported, for which the particular mipmap level (LOD) to sample has to be specified as the fourth component of the texture coordinate.

Given that texture sampling is now implemented using an instruction (unlike the 2.0 model, where the sampled data appears in an input register), it is now possible to modify the texture coordinates and LOD *before* sampling, meaning that procedural texture coordinates are possible as well as dependent texture reads (using the result of one texture read to read into another texture). The number of reads and dependent reads is unlimited in the 3.0 model.

Vertex texturing allows the implementation of huge lookup tables, effectively using the texture as a massive data storage area that can be accessed freely from within the vertex shader. Up to four variables can be fetched from the table per read (RGBA components). Completely flexible displacement mapping (reading a value from a texture and using it to displace a vertex — e.g., along its normal vector) is also possible. This functionality is no longer limited to point sampling (pre-sampled displacement mapping in vs_2_0) or geometry with n-patches enabled. The following is a vertex shader example performing displacement mapping:

```
;--------------------------------------------------------------
; Constants specified by the app
;    c0-c3   = Global transformation matrix
;    c11.x   = Scaling factor for displacement
;--------------------------------------------------------------
vs_3_0

; Samplers
dcl_2d           s0        ; Declare sampler

; Input registers
dcl_position     v0        ; Vertex position
dcl_normal       v3        ; Normal vector
dcl_texcoord0    v4        ; Texture coordinate

; Output registers
dcl_position0    o0.xyzw   ; Final vertex position
dcl_texcoord0    o1.xy     ; Texture coordinate

; Sample texture
texldl r0, v4, s0          ; Sample displacement scalar from
                           ; texture

; Displacement mapping
mul r2, v3, c11.x          ; Create displacement vector
                           ; (based on normal vector)
mul r2.xyz, r2, r0.x       ; Multiply unit displacement vector
                           ; by displacement scalar
add r0.xyz, v0, r2         ; Displace vertex position

; Vertex transformation
m4x4 o0, r0, c0            ; Transform vertices
mov  o1.xy, v4             ; Output texture coordinate
```

A form of geometry loopback can also be implemented where a complex vertex shader (e.g., very complex skinning and lighting models) is executed once, and the resulting vertex information is stored out into several textures using a trivial pixel shader program. It is then possible to read this vertex information back and send it to the pixel shader multiple times to implement some complex multipass effect. This same principle can also be used to implement geometry images, as described by Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe [Gu], where impressive geometry compression is achieved by using textures as data storage for a model's vertex positions and normals. Similarly, it is also possible to generate procedurally animated geometry, where an object's vertex positions and normals are stored within a texture that is then processed recursively by a complex pixel shader program to create a procedurally animated object. This principle is explained in detail in the article "Cloth Animation with Pixel and Vertex Shader 3.0" in *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9*.

Vertex Stream Frequency

Vertex stream frequency is a DirectX 9 feature requiring full support of the vs_3_0 model. Setting a vertex stream frequency allows vertex data to be fetched at different rates so that the same input data can be used for more than a single vertex in the vertex shader. Setting a stream frequency is achieved by using the following function:

```
HRESULT IDirect3DDevice9::SetStreamSourceFreq(UINT StreamIndex,
                                              UINT Frequency);
```

`StreamIndex` indicates which stream is to have its frequency set, while `Frequency` is the frequency to which it will be set.

One practical usage of vertex stream frequencies is vertex compression. A 3D model can be separated into "chunks" of vertices; each chunk is composed of full-precision 3D coordinates, indicating the chunk position, and a number of lower-precision "offset" vertices. The vertex shader adds the base position to each of the offset values to generate the untransformed vertex. The first stream is given a frequency indicating how many offset vertices are to use the same base position data, while the frequency of the second stream remains unchanged. Figure 1 illustrates this principle for a given set of 16 vertices.

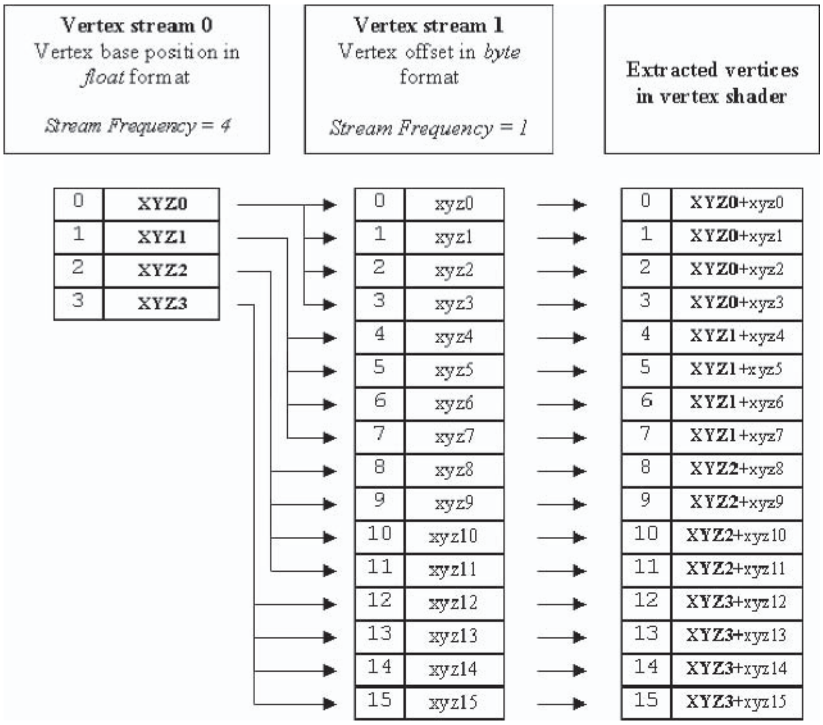


Figure 1: Example of vertex compression with stream frequencies

Another typical usage of stream frequencies is to use a vertex stream to control the animation of individual (or groups of) triangles in a vertex buffer. For example, explosions can be controlled at the triangle level by setting the desired animation data for vertices in a control stream set to a frequency of 3 (one for each triangle in the model vertex buffer). The vertex shader then transforms each triangle vertex in the model using the animation data in the control stream. The frequency can be set to higher values so that blocks of triangles can be transformed together.

Any type of vertex data can be shared between groups of vertices. For instance, a vertex stream containing triangle normals could be set up with a frequency of 3 to avoid duplicating the normal vector across all three vertices defining a face in the associated triangle list. Hierarchical sub-mesh information could also be stored using this feature by using several streams of various frequencies.

Future versions of DirectX might implement vertex stream *stepping* as well as frequency, enabling geometry instancing to be performed by looping streams multiple times.

ps_3_0 Features

Registers

The ps_3_0 model supports 32 temporary registers and 256 constant registers (224 float, 16 integer, and 16 Boolean). This increase enables more data to be manipulated or stored compared to the ps_2_0 model, which only supports 12 temporary and 32 constant registers. While ps_2_0 supported eight float and two integer input registers, all ten input registers of ps_3_0 are now in float format. Thus, interpolated colors from the vertex shader can be passed as float, increasing their precision in the process. Predication and static/dynamic flow control are controlled by two additional registers — `p0` and `aL`. Note that input register indexing can also be performed using the loop counter register `aL`.

A face register is now available in ps_3_0, which is used to indicate whether the incoming pixel is part of a front- or back-facing triangle (front-facing triangles are defined by a *clockwise* vertex ordering). Typical usages are two-sided lighting and volume algorithms. The sign of the `vFace` register determines whether the pixel is front- or back-facing, and the `if_cmp` and `setp` instructions are used to test for the sign of the face register. The following example sets front-facing pixels to red and back-facing pixels to green using predication (note that the `vFace` register must be declared prior to being used in a pixel shader program):

```
ps_3_0

; Declare face register
dcl    vFace
```

```
; Declare constant
def c0, 0, 0, 0, 1

; Set predicate to true if front-facing, false otherwise
setp_gt p0.x, vFace, c0.x

; Set front faces to red and back faces to green
(p0.x)  mov oC0, c0.wxx
(!p0.x) mov oC0, c0.xwx
```

Another useful register present in the ps_3_0 model is the position register `vPos`. Once declared, this register contains the current pixel position in screen coordinates. As such, only the `x` and `y` components of `vPos` are valid. This facility is interesting for all sorts of post-process effects operating on a surface containing a rendered scene. For example, deferred shading algorithms can use the `vPos` register to retrieve the current pixel position of a volume and thus directly use it as texture coordinates to sample data in screen-aligned textures. As a simple example, the following code renders every second horizontal line with a different color:

```
ps_3_0

; Declare position register
dcl    vPos.xy

; Declare constant
def c0, 1, 0, 0, 0.5

; Divide position by 2
mul r0.xy, vPos, c0.w

; Retrieve fractional part
frc r0.xy, r0

; Set predicate to true if fraction != 0
setp_ne p0.xy, r0, c0.y

; Output different colors based on predicate register
(p0.y)  mov oC0, c0.xyyx     ; Output red
(!p0.y) mov oC0, c0.yxyx     ; Output green
```

Instructions

As with the vs_3_0 model, ps_3_0 supports a minimum instruction count of 512. This is a considerable increase compared to the ps_2_0 model that only supports a minimum of 96 instructions (64 arithmetic and 32 texture instructions). Indeed, complex shaders like shadow mapping with percentage-closer filtering or large filter kernels could already exceed the ps_2_0 limit. Also, these 512 instructions could be arithmetic or texture instructions, as there is no restriction on their type. Note that the number of executed instructions can potentially be made larger by the use of loops and subroutines within the pixel shader.

One obvious advantage to supporting that many instructions is the reduction in pixel shader state changes. By using static flow control, several pixel shaders can be combined into a longer one, and the corresponding code path can be chosen based on a dynamic constant. The increase in performance by reduction of shader state changes is even more significant when the scene uses a large number of different shaders.

The `_abs` source modifier present in vs_3_0 is also available in ps_3_0. It forces the absolute value of a source register to be used in an instruction. For a code example using this modifier, see the "Instructions" paragraph of the "vs_3_0 Features" section.

Ps_3_0 contains new texture instructions. The selection of a particular mipmap level can be forced by using the `texldl` instruction and setting the desired MIP level into the `w` component of the source texture coordinates. A blend between MIP levels can be achieved by setting a fractional value for `w`. This feature can be useful for micro or detail texturing or to customize texture filtering.

Gradient instructions are a new feature of the ps_3_0 model. These new instructions are `dsx`, `dsy`, and `texldd`. Gradient instructions are used to detect the rate of change of a given register across adjacent pixels in the horizontal (`dsx`) and vertical (`dsy`) directions. The `texldd` instruction can then be used to sample a pixel according to the horizontal and vertical rates of changes of the texture coordinates passed to the function. Gradient instructions are generally used to determine the mipmap levels applied to a sampled texel so that custom filtering can be applied. As an example, the following shader determines the rates of change in texture coordinates and feeds them to the `texldd` instruction:

```
ps_3_0

; Samplers
dcl_2d          s0         ; Scene contained in texture

; Input Registers
dcl_texcoord0   v0         ; Texture coordinate

; Compute the horizontal and vertical rates of change in
; adjacent texture coordinates
dsx r1, v0                 ; Horizontal
dsy r2, v0                 ; Vertical

; Sample pixel
texldd r0, v0, s0, r1, r2
```

Centroid is an instruction modifier used to adjust the texture sampling location when multisampling is used. This is used to avoid artifacts when a multisampled triangle edge does not cover the center of a pixel but does cover the center of at least one sub-pixel of the multisampled mask. Centroid is used by appending the `_centroid` modifier to a texture instruction. The following is an example of the pixel shader code that can be used on a scene with multisampling enabled:

```
ps_3_0

; Samplers
dcl_2d          s0         ; Texture

; Input registers
dcl_texcoord0   v0         ; Texture coordinate

; Sample texel at centroid
texld_centroid r0, v0, s0
```

Unlimited Texture Samples and Dependent Reads

The ps_3_0 model completely removes any and all texture-read limits. Shaders can now read from a texture any number of times with coordinates calculated from any source and with unlimited complexity. The previous 2.0 model only allowed 32 texture instructions and four dependent reads to be performed within a pixel shader program. These new abilities open the hardware to iterative/recursive algorithms. For example, it is now possible to write a pixel shader that performs ray-tracing operations through a volume texture, enabling shadows or reflective surfaces to be calculated. See the "Rendering Voxel Objects with ps_3_0" article in *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9* for an example of ray tracing through a volume texture. Other applications include single-pass blur filters or spatial convolution, both with an unlimited kernel size, and many other image-processing algorithms.

Conclusion

The 3.0 shader model is a huge step forward compared to the previous 2.0 model. New features have been introduced while register and instruction limits have been increased dramatically, allowing for much more advanced effects to be implemented. Simplicity has also been greatly enhanced by unifying the vertex and pixel shader models and allowing more flexibility on instructions and registers.

References

[Gu] Gu, X., S. Gortler, and H. Hoppe, "Geometry Images," ACM SIGGRAPH '02, pp. 355-361, http://research.microsoft.com/~hoppe/.

Advanced Lighting and Shading with Direct3D 9

Michal Valient
Caligari Corporation

Introduction

As promised, DirectX 9 has a lot of new functionality, mainly in the programmable pipeline. Floating-point support in pixel shaders gives us what we missed in Direct3D 8 — precision in this major part of rendering. Larger shaders and flow control allow more effects. New types of textures (16-bit per component and floating-point components) give us an extra bit of detail. Of course, new hardware is on the market (or heading to the market) — ATI Radeon 9700, nVidia GeForceFX, and cards from S3, 3DLabs, and other companies.

This article discusses the new possibilities of Direct3D 9. We begin with classic per-pixel shading. First we improve it for version 2.0 shaders — great quality and no more lookup textures. Then we utilize 3.0 shaders to show how to do four spotlights in one pass with dynamic flow control and relative addressing.

We continue with per-pixel environment bump mapping — DirectX 8.1 is presented first (with pixel shader 1.4), and then the new shaders 2.0 version is presented. The Fresnel term is added for a more impressive and realistic effect.

The end of the article is reserved for two lighting models that are not commonly used in real-time computer graphics. This is mainly due to limitations of the hardware prior to the new versions of DirectX. The Oren-Nayar generalization of the Lambertian diffuse model is implemented with 2.0 shaders. It brings more reality to materials like clay and porcelain. The specular part of the Cook-Torrance model is presented with both pixel shader 1.4 and 2.0 for visual comparison. This model produces very good results for metallic surfaces.

The following sections are organized similarly: The whole shader is presented at the beginning of the section, and then it is broken into pieces with necessary explanations. New shader concepts (syntax) are explained in depth.

Per-Pixel Phong

This section covers the possibilities of Phong lighting with new shader models. It is targeted mainly at people upgrading to DirectX 9 from a previous version. Because of this, knowledge of the Phong lighting equation (only a brief review is available here), concepts of per-pixel shading, and DirectX 8 is expected. Most of this can be found in [1] (this article is a direct extension). Other sources of information are [2] and [3]. A shader reference is available on the MSDN DirectX web pages (http://msdn.microsoft.com/directx).

Phong's Lighting Equation

The equation that we use includes only a diffuse and specular term. Both are attenuated with a spotlight cone.

$$L_{\text{Phong}} = I_{\text{spotlight}} \otimes (I_{diffuse} \otimes m_{diffuse} + I_{specular} \otimes m_{specular})$$

$$I_{diffuse} = (n \bullet 1)$$

$$I_{specular} = (r \bullet v)^{shininess}$$

$$r = 2(n \bullet 1)n - 1$$

…where **n** is the surface normal vector, **l** is the vector from the surface point to the light, and **v** is the vector from the surface point to the viewer's position. Every vector is assumed to be normalized.

$\mathbf{m}_{\mathbf{diffuse}}$ is the color of diffuse material at a given pixel while $\mathbf{m}_{\mathbf{specular}}$ is the color of specular material at a given pixel.

$\mathbf{I}_{\mathbf{spotlight}}$ is used to simulate a spotlight. In our case, we use additional texture, which is projected in the spotlight's direction on every object. Think of it as a projector.

Vertex and Pixel Shaders 2.0

Direct3D 9 introduced a new evolutionary step in shader architecture — the 2.0 version.

Vertex shaders can be larger, and static flow control is now possible. This includes support for if-then-else (with constant Boolean registers), loop and repeat (with constant integer registers as loop counters), and subroutine support (also Boolean register dependent calls). Of course, a couple of new instructions and macros are available.

Pixel shaders are far more modified. A major improvement is floating-point precision all over the pipeline. Version 2.0 is an extension of version 1.4, so none of the instructions (like texm-3x3pad, texm3x3tex, or texm3x3vspec) from versions prior to 1.4 survived. This is a good step because the version 1.3 style of coding was more CISC-like (powerful instructions), but with so little space in the shader, we had less freedom. On the other hand, version 1.4 and newer are comparable to RISC style (a small set of simple instructions and a lot of freedom). Shaders can now contain 64 arithmetic and 32 texture instructions. The instruction set is comparable to that in vertex shaders but without any flow control. Of course, texture sampling instructions are available. The pixel shader can now have four color outputs, so we can update four independent render targets at one time. The depth buffer is another possible output.

Vertex Shader 2.0

As you can see, the new vertex shader differs only in minor ways from version 1.1, which was commonly used to set up per-pixel shading.

Here is a vertex shader for per-pixel Phong lighting:

```
vs_2_0

// Constant registers
//------------------------------
// c0-c3     - world space transposed
// c4-c7     - world * view * projection
// c8        - Light position (in world space)
// c9        - Eye position (in world space)
// c10-c13   - Spotlight projection matrix

// Input registers
//------------------------------
dcl_position  v0
dcl_normal    v1
dcl_texcoord  v2
dcl_tangent   v3

// Output
//------------------------------
// oT0 - texture coordinates
// oT1 - Light vector (in tangent space)
// oT2 - eye vector (in tangent space)
// oT3 - spotlight texture coordinates

//The following code outputs position and texture coordinates
//------------------------------
m4x4  oPos, v0, c4       //vertex clip position
mov   oT0.xy, v2.xy      //Texture coordinates for color texture
m4x4  r8, v0, c0         //Transform vertex into world position

//The following code generates tangent space base vectors
//------------------------------
m3x3  r11.xyz, v1, c0    //N to world space
mov   r11.w, v1.w
m3x3  r9.xyz, v3, c0     //T to world space
mov   r9.w, v3.w
```

```
crs   r10.xyz, r9, r11   //Cross product - binormal B=NxT

//Computes light and eye vectors and projector's texture coordinates
//-----------------------------
add   r0, c8, -r8        //Build the light
nrm   r1, r0             //normalize vector
m3x3  oT1.xyz, r1, r9    //to tangent space

add   r0, c9, -r8        //build the eye vector
nrm   r1, r0             //normalize vector
m3x3  oT2.xyz, r1, r9    //to tangent space

m4x4  oT3.xyzw, v0, c10  //compute projector texture coordinates
```

Here is the first change that we can find in the declaration of input registers:

```
dcl_position  v0
dcl_normal    v1
dcl_texcoord  v2
dcl_tangent   v3
```

In Direct3D 8, vertices were declared only at the time of shader creation outside the shader. We specified which input register in the shader would be loaded with which part of data. In Direct3D 9, we have two declarations:

- **Outside the shader with the `SetVertexDeclaration` method.** In this phase, we define for each input element the stream from which it will be loaded — offset in bytes from the start of the stream to the data element, type of data (i.e., float, float3 for vector, etc.), and the semantic of the element (i.e., position, normal, tangent, binormal, etc. — this will be used later in the shader).
- **Inside the shader.** Here we specify the target register for data with the specified semantic (i.e., position, normal, tangent, binormal, etc. — same as outside the shader).

This allows us to write shaders without expectations of a specific input structure and specify a new vertex format for each model still using the same shader. In the example above, we load one position, texture coordinate, normal, and binormal to the first four input registers, but in the vertex stream, this data can be anywhere and even sorted in a different way (i.e., normal, texcoord, position, and tangent).

Later in the tangent space base vectors computation, we use the new `crs` macro instruction to do cross product instead of using the `mul r0,r9.zxyw,r11.yzxw; mad r10,r9.yzxw,r11.zxyw,-r0` pair known from previous versions. This command takes two instruction slots and is most likely expanded to `mul-mad` internally. Also note that we have to explicitly fill the w component of every base vector because this version of the shader does not allow us to use a component that was not filled previously, and `crs` uses all four components of input registers.

The following is the creation of a tangent space base:

```
m3x3  r11.xyz, v1, c0    //N to world space
mov   r11.w, v1.w
m3x3  r9.xyz, v3, c0     //T to world space
mov   r9.w, v3.w
crs   r10.xyz, r9, r11   //Cross product - binormal B=NxT
```

When light and eye vectors are computed, the `nrm` macro instruction is used to normalize the vector instead of the previously used three instructions.

Here is the transformation of a normalized vector to tangent space:

```
add   r0, c8, -r8        //Build the light vector
nrm   r1, r0             //normalize vector
m3x3  oT1.xyz, r1, r9    //to tangent space
```

The last shader instruction is used to compute the spotlight texture coordinates for this vertex. The matrix has the following form:

$$M_{\text{Spotlight}} = M_{\text{ObjectToWorld}} * M_{\text{SpotView}} * M_{\text{SpotProjection}} * \begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{pmatrix}$$

$\mathbf{M}_{\mathbf{ObjectToWorld}}$ is a matrix that transforms vertices from object to world space. $\mathbf{M}_{\mathbf{SpotView}}$ is a spotlight's view transformation matrix, while $\mathbf{M}_{\mathbf{SpotProjection}}$ is a spotlight's perspective transformation matrix. Because of similarity with the camera, they can be easily computed with the `D3DXMatrixLookAtLH` and `D3DXMatrixPerspectiveFovLH` functions. The last matrix in the previous equation is used to shift coordinates from range [–1...1] (output of clipping matrix) to range [0...1]. Note that this matrix negates the y coordinate because in texture space, y has a value of 0.0 on the top and 1.0 on the bottom and clipping space has –1.0 at the "top of the space" and 1.0 at the bottom.

Usage of macro instructions all over the shader is preferred over the usage of their inline versions. This is because they are not expanded by the Direct3D runtime but by the driver. If the hardware supports a specific macro, it is executed directly; if not, it is safely replaced with supported instructions.

Pixel Shader 2.0

While the previous shader differs only in minor ways from version 1.1, pixel shader changes are much more notable.

The following is a pixel shader for per-pixel Phong lighting.

```
PS_2_0

// Constant registers
//-----------------------------
// c0 - diffuse texture multiplier (multiplied with light color)
// c1 - specular texture multiplier (multiplied with light color)
// c2 - specular shininess (shi, shi, shi, 1.0f)
def c31, 2.0f, 1.0f, 0.0f, 4.0f     //helper constant

// Used input registers
//-----------------------------
dcl     t0.xy                  //texture coordinates
dcl     t1.xyz                 //light vector
dcl     t2.xyz                 //eye vector
dcl     t3.xyzw                //projector texture coordinates

// Used input texture samplers
```

```
//------------------------------
dcl_2d  s0                    //diffuse texture (gloss in alpha)
dcl_2d  s1                    //normal texture
dcl_2d  s3                    //spotlight texture

// Output
//------------------------------
// oC0 - output color
//

// Set up needed vectors - load and normalize
//------------------------------
texld   r0, t0, s1            //load normal vector
mad     r1, r0, c31.r, -c31.g //bias normal to range -1,1
nrm     r11, r1               //r11 = normalized normal
mov     r1.xyz, t1
nrm     r10, r1               //r10 = normalized light vector
mov     r1.xyz, t2
nrm     r9, r1                //r9 = normalized eye vector

// Compute diffuse and specular intensities
//------------------------------
dp3     r0.r, r11, r10        //r0 = (n.l)
mul     r1, r0.r, c31.r       //r1 = 2*(n.l)
mad     r1, r1, r11, -r10     //r1=2(n.l)n-l – reflectance vector
dp3_sat r1, r1, r9            //r1 = (r.v)
pow     r0.g, r1.r, c2.r      //r0.g = (r.v)^shi – specular term

cmp     r0, r0.r, r0, c31.b   //if (n.l)<0 do not light anything

// Modulate texture with computed intensities
//------------------------------
texld   r1, t0, s0            //load diffuse texture (gloss is in alpha)
texldp  r4, t3, s3            //load projector map (perspective correct)

mul     r2, r1.a, r0.g        //multiply specular intensity with gloss
mul     r2, r2, c1            //… and with material's specular color
mul     r3, r1, r0.r          //multiply diffuse intensity with texture
mul     r3, r3, c0            //… and with material's diffuse color
add     r0, r2, r3            //combine it together
mul     r0, r0, r4            //modulate it with spotlight texture

mov     oC0, r0               //color output
```

Changes start from the very beginning. We have to specify input registers from the vertex shader. In the previous version, these registers had to be loaded with special `texcrd` instructions into a temporary register before they could be used in mathematical instructions. Now we just declare them as used with the `dcl` instruction. (With individual `.xyzw` components, if we fail to specify components, the compiler will assume `.xyzw`. If these are not filled in the vertex shader, the run-time shader linker will fail.) Then we can use them as read-only registers freely across the shader.

The following is a declaration of input registers.

```
dcl     t0.xy                   //texture coordinates
dcl     t1.xyz                  //light vector
dcl     t2.xyz                  //eye vector
dcl     t3.xyzw                 //projector texture coordinates
```

Similarly, we have to declare used texture samplers (simply said, texture stages that we want to use) with the type of used texture (`_2d`, `_3d`, or `_cube`). Again, failing to specify correct input types will generate linker errors at run time.

The following is a declaration of input texture samplers.

```
dcl_2d  s0                      //diffuse texture (gloss in alpha)
dcl_2d  s1                      //normal texture
dcl_2d  s3                      //spotlight texture
```

After setup, our first step is to load a normal vector. We are using the `texld` instruction, but it differs from the pixel shader 1.4 version. In fact, instead of specifying the output register/source texture stage and texture coordinate (`texld r0, t0` stands for "load texel at coordinates t0 from texture stage 0 to register r0"), we use three registers — output temporary register, texture coordinates, and input sampler register (this follows the syntax of all instructions in the shaders; the first parameter is an output register, and then there are the inputs). Note that texture coordinates can be specified by an input register from the vertex shader `tn` or by a temporary register `rn` (dependent read), and texture sampling can occur anywhere in the shader.

As in the previous version, we need to expand the components of a normal vector from range [0...1] to range [–1...1]. Because there is no longer a `_bx2` instruction modifier, we need to do this manually by multiplication with 2.0 and subtraction of 1.0 (`mad r1, r0, 2.0, -1.0` in the shader after replacing constants with values).

Now comes the big difference. Thanks to the power of pixel shader 2.0 shaders, we can now normalize each input vector. Mark Kilgard wrote in [4] that doing trilinear filtering of a normal texture denormalizes the resulting vector (which is correct), but the result is acceptable because it "simulates" dimming of bumps with increased distance from the viewer. When Kilgard saw images done with shaders that were using normalization per pixel, I think he changed his mind. Normalization is done with the `nrm` macro instruction (the same syntax and behavior as in the vertex shader). Note that we had to use `mov r1.xyz, t1; nrm r0, r1` instead of the simple `nrm r0, t1` because when `D3DXAssembleShader` is used to load and compile the shader at run time, the application crashes with an error inside the call (this should be corrected in the next version of the SDK). The command-line compiler psa.exe handles this correctly.

The following is a preparation of input vectors.

```
texld   r0, t0, s1              //load normal vector

mad     r1, r0, c31.r, -c31.g  //bias normal to range -1,1
nrm     r11, r1                 //r11 = normalized normal
mov     r1.xyz, t1
nrm     r10, r1                 //r10 = normalized light vector
mov     r1.xyz, t2
nrm     r9, r1                  //r9 = normalized eye vector
```

The next part of the shader is easy to understand. First we compute the reflection vector (because every input vector is normalized, the reflection vector is normalized too). Then we can do a power calculation in the shader with the `pow` macro instruction — no more lookup textures. If we use `mul r0.a, r0.a, 100.0; pow r0.g, r1.r, r0.a` instead of `pow r0.g, r1.r, c2.r` found in the shader, we can have the per-pixel shininess parameter at virtually no additional cost (if we stored it in the alpha component of a normal texture).

The last instruction in this block (cmp) is used to disable back lighting of pixels that are not visible from the light source (the angle between n and l is greater than π/2, and therefore the dot product is less than zero). In this case we reset the entire register r0 to zero. To make it clear, r0.r holds diffuse intensity, and r0.g holds specular intensity.

The following is a computation of diffuse and specular intensities.

```
dp3     r0.r, r11, r10         //r0 = (n.l)
mul     r1, r0.r, c31.r        //r1 = 2*(n.l)
mad     r1, r1, r11, -r10      //r1=2(n.l)n-l – reflectance vector
dp3_sat r1, r1, r9             //r1 = (r.v)
pow     r0.g, r1.r, c2.r       //r0.g = (r.v)^shi – specular term

cmp     r0, r0.r, r0, c31.b    //if (n.l)<0 do not light anything
```

In the last section of the shader, we load the decal texture with the texld instruction, and we load spotlight texture with perspective correct division with the texldp instruction (this instruction is the same as doing texld r3, t3_dw.xyw in pixel shader 1.4). In the next few instructions, we modulate intensities with light constants and textures.

Note that in pixel shader 2.0 the output register is oCn instead of r0 from the previous version, and the pixel shader can have up to four outputs.

The following is the final color output.

```
texld   r1, t0, s0             //load diffuse texture (gloss is in alpha)
texldp  r4, t3, s3             //load projector map (perspective correct)

mul     r2, r1.a, r0.g         //multiply specular intensity with gloss
mul     r2, r2, c1             //… and with material's specular color
mul     r3, r1, r0.r           //multiply diffuse intensity with texture
mul     r3, r3, c0             //… and with material's diffuse color
add     r0, r2, r3             //combine it together
mul     r0, r0, r4             //modulate it with spotlight texture
mov     oC0, r0                //color output
```

HLSL Version

Here we provide the High Level Shading Language version of the above shaders to show the simplicity of such programming. The vertex shader comes first and then the pixel shader.

The following is the HLSL vertex shader for Phong lighting.

```
// Used input structure
//-----------------------------
struct VS_INPUT {
    float4 vPosition : POSITION; //position in object space
    float3 vNormal   : NORMAL;   //normal
    float2 tcCoord   : TEXCOORD; //texture coordinates
    float3 vTangent  : TANGENT;  //tangent
};

// Used output structure
//------------------------------
struct VS_OUTPUT {
    float4 vClipPos: POSITION;  //Clipping space position
    float2 tcCoord : TEXCOORD0; //texture coordinates
    float3 vLight  : TEXCOORD1; //light vector
    float3 vEye    : TEXCOORD2; //eye vector
    float4 tcpSpot : TEXCOORD3; //perspective spotlight coordinates
};

// Constant registers
//------------------------------
float4x4 mToWorld : register(c0);   //world space transposed
float4x4 mToClip  : register(c4);   //world * view * proj
float4 pLight     : register(c8);   //Light position (world space)
float4 pEye       : register(c9);   //Eye position (world space)
float4x4 mSpot    : register(c10);  //Spotlight projection matrix

// function    : main
// description : vertex shader function
// return      : VS_OUTPUT
// param:
//    VS_INPUT input : vertex shader input
//-----------------------------
VS_OUTPUT main(const VS_INPUT input) {
    VS_OUTPUT output;
    //The following code outputs the position and texture coordinates
    //------------------------------
    output.vClipPos = mul(input.vPosition, mToClip);
    output.tcCoord = input.tcCoord;
    float4 pVertexWorld = mul(input.vPosition, mToWorld);

    //The following code generates the tangent space base vectors
    //------------------------------
    float3x3 mToTangent;
    mToTangent[0] = mul(input.vTangent, (float3x3)mToWorld);
    mToTangent[2] = mul(input.vNormal, (float3x3)mToWorld);
    mToTangent[1] = cross(mToTangent[0], mToTangent[2]);

    //Compute light and eye vectors and the projector's texture coordinates
    //------------------------------
    float3 vToLight = normalize(pLight - pVertexWorld);
    output.vLight = mul(mToTangent, vToLight);

    float3 vToEye = normalize(pEye - pVertexWorld);
    output.vEye = mul(mToTangent, vToEye);

    output.tcpSpot = mul(input.vPosition, mSpot);
    return output;
}
```

The following is the HLSL pixel shader for Phong lighting.

```
// Used input structure
//------------------------------
struct PS_INPUT {
    float2 tcCoord : TEXCOORD0; //input texture coordinates
    float3 vLight  : TEXCOORD1; //light vector
    float3 vEye    : TEXCOORD2; //eye vector
    float4 tcpSpot : TEXCOORD3; //perspective spotlight coordinates
};

// Used output structure
//------------------------------
struct PS_OUTPUT {
    float4 vColor  : COLOR0;    //render target 0
};

// Used input texture samplers
//------------------------------
```

```
sampler smplTexture : register(ps,s0);  //decal texture
sampler smplNormal  : register(ps,s1);  //normal texture
sampler smplSpot    : register(ps,s3);  //spotlight texture

// function    : main
// description : pixel shader function
// return      : PS_OUTPUT
// param:
//    PS_INPUT input : pixel shader input - output from VS
//    float3 colDiff : c0 - diffuse texture multiplier
//    float3 colSpec : c1 - specular texture multiplier
//    float shininess: c2 - specular shininess
//------------------------------
PS_OUTPUT main(const PS_INPUT input,
               uniform float3 colDiff : c0,
               uniform float3 colSpec : c1,
               uniform float shininess : c2) {
    PS_OUTPUT output;

    // Load and normalize input vectors
    //------------------------------
    float3 vNormal = tex2D(smplNormal, input.tcCoord).xyz;
    vNormal = normalize(2.0 * vNormal - 1.0);   //bias and normalize
    float3 vLight = normalize(input.vLight);
    float3 vEye = normalize(input.vEye);

    // Compute diffuse and specular intensities
    //------------------------------
    float normalDotLight = dot(vNormal, vLight);
    float3 vLightReflect = 2.0*normalDotLight*vNormal - vLight;
    float eyeDotReflect = saturate(dot(vEye, vLightReflect));
    float specularIntensity = pow(eyeDotReflect, shininess);
    float4 tmpOutput = {0.0f, 0.0f, 0.0f, 1.0f};
    if (normalDotLight>0.0f) {
        float4 tDecal = tex2D(smplTexture, input.tcCoord);
        float4 tSpot = tex2Dproj(smplSpot, input.tcpSpot);

        float3 diffuse = normalDotLight * colDiff * tDecal;
        float3 specular = specularIntensity * colSpec * tDecal.a;

        tmpOutput.xyz = tSpot * (diffuse + specular);
    }
    output.vColor = tmpOutput;
    return output;
}
```

Quality Comparison

The possibility of normalization and higher precision in pixel shaders gives us far better results than with previous versions. We can compare the next images, but the difference is more visible in motion. Light intensity is more stable in motion in the pixel shader 2.0 version.

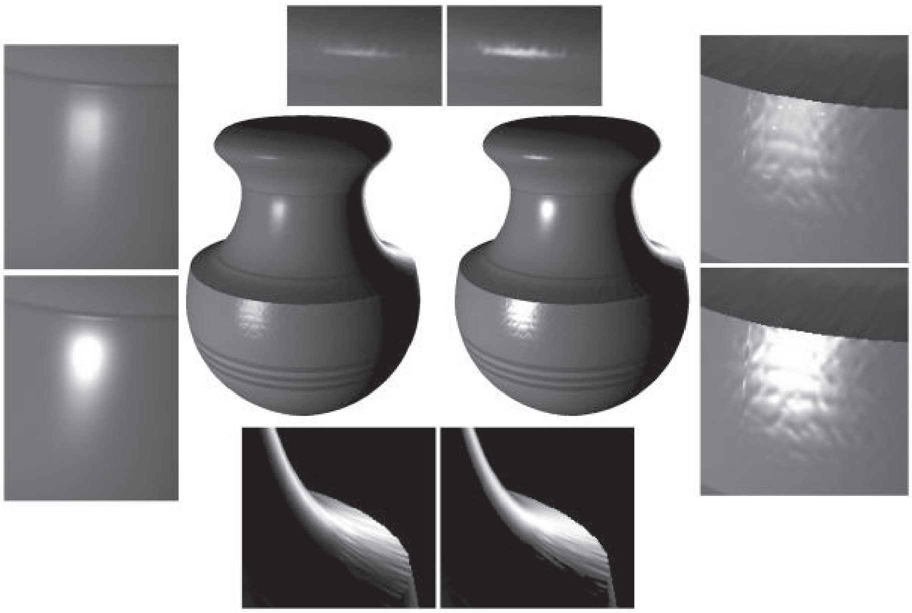


Figure 1: This image shows a vase rendered with pixel shader 1.4 on the left and pixel shader 2.0 on the right. In the small frames, you can see details of highlights.

Vertex and Pixel Shaders 3.0

To make the development of shaders under Direct3D 9 a bit more adventurous, there is another version of shaders available — version 3.0. It differs slightly from 2.0 in syntax (setup stage of shaders and registers) and much more in shader possibilities. And yes, there are also version 2.x (or 2.eXtended) shaders, but to put it simply, it is somewhere between 2.0 and “3.0 in syntax of 2.0,” and its features depend on device capabilities. We can tell that a device that is capable of accelerating version 3.0 can accelerate full-featured shaders 2.x, and because of this, it is not very useful for educational purposes.

Vertex shader 3.0 can be even larger than the previous version. The shader has a minimum of 512 instructions, but due to the flow control, the device has to be capable of executing a much larger number of instructions (at least 65,536). Dynamic flow control is possible in shaders (loops can be exited depending on the value of a temporary or special predicate register). Other features include texture lookup in the vertex shader and extension of relative indexing from constants to inputs and outputs.

Pixel shader 3.0 follows the way of vertex shaders. Dynamic and static flow control is possible, and the instruction count limits are the same. There is no limit on the texture instruction count and no limit on the dependent texture reads. New input registers are introduced — the pixel position on the screen and face orientation register. Also, the gradient instructions are new — the rate of change of the input registers can be inspected.

In the following examples, the capabilities of the highest shader version is used to compute four spotlights in a single pass, all done in one loop. There is no visual change from version 2.0.

Vertex Shader 3.0

The following is a vertex shader for four spotlights in one pass.

```
vs_3_0

// Constants
//-----------------------------
// i0        - light iteration loop data
// c0-c3     - world space transposed
// c4-c7     - world * view * proj
// c8        - Eye position (in world space)
// c9 >>     - start of lighting data :
//              VECTOR4     LightPos;
//              MATRIX4x4   Projector;
//              COLOR4      LightColor (r,g,b, multiplier);
//                          Colors are expected to be < 1.0f.
// c255=(size_of_light_struct, reset, increment, start_index)
```

```
def c255, 6.0f, 0.0f, 1.0f, 9.0f    //loop counter data

// Input
//------------------------------
dcl_position   v0
dcl_normal     v1
dcl_texcoord   v2
dcl_tangent    v3

// Output
//------------------------------
dcl_position0 o0         //clip space coordinates
dcl_texcoord0 o1.xy      //texture coordinates
dcl_texcoord1 o2.xyz     //eye vector
dcl_texcoord2 o3         //Light vector 1
dcl_texcoord3 o4         //Projector texture coordinates 1
dcl_texcoord4 o5         //Light vector 2
dcl_texcoord5 o6         //Projector texture coordinates 2
dcl_texcoord6 o7         //Light vector 3
dcl_texcoord7 o8         //Projector texture coordinates 3
dcl_texcoord8 o9         //Light vector 4
dcl_texcoord9 o10        //Projector texture coordinates 4

//The following code outputs position and texture coordinates
//------------------------------
m4x4 o0, v0, c4          //vertex clip position
mov o1.xy, v2.xy         //Texture coordinates for color texture
m4x4 r8, v0, c0          //Transform vertex into world position

//The following code generates tangent space base vectors
//------------------------------
m3x3 r11.xyz, v1, c0     //N to world space
mov r11.w, v1.w
m3x3 r9.xyz, v3, c0      //T to world space
mov r9.w, v3.w
crs r10.xyz, r9, r11     //The cross product - binormal NxT

//Compute normalized eye vector and transform it to tangent space
//------------------------------
add r0, c8, -r8          //build the eye vector
nrm r6, r0               //normalize vector
m3x3 o2.xyz, r6, r9      //eye vector to tangent space
```

```
//In the following loop we are computing normalized light vectors
//and transforming them to tangent space
//-----------------------------
mov r0.y, c255.y                    //reset constant addressing counter
loop aL, i0                         //Loop for lights
    //Index = Counter * DataSize + DataStart
    mad r0.x, r0.y, c255.x, c255.w  //light data index
    mova a0, r0.x                   //fill address register

    add r1, c[a0.x], -r8            //Build the light vector
    nrm r6, r1                      //normalize vector
    m3x3 o3[aL].xyz, r6, r9         //light vector to tangent space

    m4x4 o3[aL+1], v0, c[a0.x+1]    //transform vertex with light matrix
                                    //(get projector texture coordinates)

    add r0.y, r0.y, c255.z          //Increment const addressing counter
endloop
```

In this shader version, there is only one set of output registers. Previously, we had `oD#` for color, `oFog`, `oPos` for clip space position, `oPts` for point size, and `oT#` for texture coordinates. Now only the `o#` registers are available, but they can be used in any way. Therefore, the semantics of every used output register has to be declared at the beginning of the shader in the same way that it was done for inputs. Exactly one `dcl_position0` always has to be declared to specify the clipping space vertex position.

The following is a declaration of outputs.

```
dcl_position0 o0                    //clip space coordinates
dcl_texcoord0 o1.xy                 //texture coordinates
dcl_texcoord1 o2.xyz                //eye vector
dcl_texcoord2 o3                    //Light vector 1
dcl_texcoord3 o4                    //Projector texture coordinates 1
dcl_texcoord4 o5                    //Light vector 2
dcl_texcoord5 o6                    //Projector texture coordinates 2
dcl_texcoord6 o7                    //Light vector 3
dcl_texcoord7 o8                    //Projector texture coordinates 3
dcl_texcoord8 o9                    //Light vector 4
dcl_texcoord9 o10                   //Projector texture coordinates 4
```

After the vertex transformation, the tangent space base creation and eye vector computation (discussed earlier) result in new code.

The following is the main loop.

```
mov r0.y, c255.y                    //reset constant addressing counter
loop aL, i0                         //Loop for lights
    //Index = Counter * DataSize + DataStart
    mad r0.x, r0.y, c255.x, c255.w  //light data index
    mova a0, r0.x                   //fill address register

    add r1, c[a0.x], -r8            //Build the light vector
    nrm r6, r1                      //normalize vector
    m3x3 o3[aL].xyz, r6, r9         //light vector to tangent space

    m4x4 o3[aL+1], v0, c[a0.x+1]    //transform vertex with light matrix
                                    //(get projector texture coordinates)

    add r0.y, r0.y, c255.z          //Increment const addressing counter
endloop
```

We used two relative addressing registers in this loop: aL for output registers and a0 for input constant data.

First let's discuss the loop execution schema and its preparation. We know that starting from register o3 we produce the following output: o[3+2i], which is a vector of light i, and o[3+2i+1], which is the projector texture coordinates for this light. Outside the shader we set up the i0 integer constant register with this information for the loop. The light count is stored in i0.x, starting at the aL value in i0.y (= 0 because we use o3[aL], which is the same as o[3+aL]) and the aL step in i0.z (= 2).

NOTE In the DirectX 9.0 SDK documentation, i0.x and i0.y meanings are switched, but loop works as described here.

The execution of loop aL, i0 can be expressed with the following pseudocode.

```
RemainingLoops = i0.x;
LoopCounter = i0.y;
LoopStep = i0.z
while (RemainingLoops > 0) {
   aL = LoopCounter;
   do_some_code();
   LoopCounter = LoopCounter + LoopStep;
   RemainingLoops = RemainingLoops - 1;
}
```

Light data is stored in constant registers; `c[i]` is the light position in world space, `c[i+1]`…`c[i+4]` is the 4x4 light projector matrix, and `c[i+5]` holds light color data (unused here). The relative address computation that was used for constants and the entire lighting can be seen in the following pseudocode (the original assembler lines are in the comments):

```
//c255.x - number of registers occupied by one light
LightStructureSize = 6;
//c255.w – c[9] is first constant register with light
LightDataStart = 9;

//mov r0.y, c255.y
LightDataCounter = 0;
while (...) {
    //mad r0.x, r0.y, c255.x, c255.w
    LightDataIndex=LightDataCounter*LightStructureSize+LightDataStart;
    //mova a0, r0.x
    a0 = LightDataIndex;
    do_light_computation();
    //add r0.y, r0.y, c255.z
    LightDataCounter = LightDataCounter + 1;
}
```

After this preparation, we can see that the light vector and projector texture coordinates computation consists of the following four instructions and is the same as in the previous versions, except for the use of relative addressing:

```
add r1, c[a0.x], -r8            //Build the light vector
nrm r6, r1                      //normalize vector
m3x3 o3[aL].xyz, r6, r9         //light vector to tangent space
m4x4 o3[aL+1], v0, c[a0.x+1]    //transform vertex with light matrix
                                //(get projector texture coordinates)
```

Pixel Shader 3.0

The following pixel shader might look complicated, but after an explanation, it's quite simple:

```
ps_3_0

// Constant registers
//-----------------------------
// c0       - diffuse texture multiplier
// c1       - specular texture multiplier
// c2       - specular shininess (shi, shi, shi, 1.0f)
// c3       - special constants - see below
// c4 >>    - light colors
def c3, 2.0f, 1.0f, 0.0f, 0.0f
def c223, 2.5f, 1.5f, 0.5f, 0.0f    //light color comparison indexes

// Used input registers
//-----------------------------
dcl_texcoord0 v0.xy                 //texture coordinates
dcl_texcoord1 v1.xyz                //eye vector
dcl_texcoord2 v2                    //Light vector 1
dcl_texcoord3 v3                    //Projector texture coordinates 1
dcl_texcoord4 v4                    //Light vector 2
dcl_texcoord5 v5                    //Projector texture coordinates 2
dcl_texcoord6 v6                    //Light vector 3
dcl_texcoord7 v7                    //Projector texture coordinates 3
dcl_texcoord8 v8                    //Light vector 4
dcl_texcoord9 v9                    //Projector texture coordinates 4

// Used input texture samplers
//-----------------------------
dcl_2d      s0                      //diffuse texture (gloss in alpha)
dcl_2d      s1                      //normal texture
dcl_2d      s3                      //spotlight texture

// Output
//-----------------------------
// oC0 - output color

// Set up needed vectors - load and normalize
//-----------------------------
texld       r0, v0, s1                 //load normal
```

```
mad         r1, r0, c3.r, -c3.g        //bias normal to range -1,1
nrm         r11, r1                    //r11 = normalized normal
mov         r1.xyz, v1                 //load eye vector
nrm         r9, r1                     //r9 = normalized eye vector

//In the following loop, lighting contribution will be computed for lights
//-----------------------------
mov         r8, c3.b                   //reset overall diffuse output
mov         r7, c3.b                   //reset overall specular output
mov         r6, c3.b                   //reset iteration counter - used
                                       //to get correct light color
loop        aL, i0                     //This is the light loop
    mov         r1.xyz, v2[aL]         //load light vector
    nrm         r10, r1                //r10 = normalized light vector

    dp3         r0, r11, r10           //r0 = (n.l)
    if_gt       r0.r, c3.b             //Light only if (n.l)>0
        mul         r1, r0, c3.r       //r1 = 2*(n.l)
        mad         r1, r1, r11, -r10  //reflection vector - r1=2(n.l)n-l
        dp3_sat     r1, r1, r9         //r1 = (r.v)
        pow         r0.g, r1.r, c2.r   //r1 = (r.v)^shi

        //We have to use the following code to get the correct light color.
        //Pixel shader 3.0 cannot address constants relatively.
        mov r3, c4
        setp_gt p0, r6.r, c223
        (p0.z) mov r3, c5
        (p0.y) mov r3, c6
        (p0.x) mov r3, c7

        mov         r4, v2[aL+1]      //move coordinate to temp register
        texldp      r1, r4, s3        //load projector texture
        mul         r1, r1, r3        //modulate with light color
        mul         r1, r1, r3.a      //modulate with light intensity
        mad         r8, r0.r, r1, r8  //modulate diffuse intensity with
                                      //spotlight and add to overall
        mad         r7, r0.g, r1, r7  //modulate specular intensity with
                                      //spotlight and add to overall
    endif
    add         r6, r6, c3.g          //increment iteration counter + 1
endloop

// Compute resulting color
```

```
//------------------------------
texld       r1, v0, s0         //load diffuse texture
mul         r8, r8, c0         //multiply overall diffuse with
                               //material diffuse color
mul         r8, r8, r1         //modulate it with texture
mul         r7, r7, c1         //multiply overall specular with
                               //material specular color
mad         r7, r7, r1.a, r8   //modulate it with gloss map and
                               //add computed diffuse color
mov         oC0, r7            //output the color
```

First note the change in declaration of input registers; we have to specify a semantic. To obtain the correct results, semantics used in the pixel shader have to match the output semantic in the vertex shader.

The following is a declaration of the input semantic.

```
dcl_texcoord0 v0.xy                //texture coordinates
dcl_texcoord1 v1.xyz               //eye vector
dcl_texcoord2 v2                   //Light vector 1
dcl_texcoord3 v3                   //Projector texture coordinates 1
dcl_texcoord4 v4                   //Light vector 2
dcl_texcoord5 v5                   //Projector texture coordinates 2
dcl_texcoord6 v6                   //Light vector 3
dcl_texcoord7 v7                   //Projector texture coordinates 3
dcl_texcoord8 v8                   //Light vector 4
dcl_texcoord9 v9                   //Projector texture coordinates 4
```

After this well-known vector setup (this time without the light vector), we enter the main loop. We go step by step through it. Just before the loop, we reset all registers used in the loop. Register `r8` is used to accumulate diffuse intensity while register `r7` is used to accumulate specular intensity. Register `r6` is used to indicate the loop index and is incremented by one with the last instruction in the loop (`add r6, r6, c3.g`). The loop works the same way described for the vertex shader, even with the same constant values in our case. The first step in the loop is to load and normalize the current light vector.

The following is the main loop with a light vector setup.

```
mov         r8, c3.b            //reset overall diffuse output
mov         r7, c3.b            //reset overall specular output
mov         r6, c3.b            //reset iteration counter - used
                                //to get correct light color
loop        aL, i0              //This is the light loop
    mov         r1.xyz, v2[aL]  //load light vector
    nrm         r10, r1         //r10 = normalized light vector
    .
    .
    .
    add         r6, r6, c3.g    //increment iteration counter + 1
endloop
```

Later in the loop we compute diffuse intensity. With flow control available, we can do if-then-else constructions, so why not use it to ignore unlit pixels (those where the dot product of n and l is less than 0) with the `if_gt x, y` instruction standing for `if x>y`? Now any further computation is done only for lit pixels. In the first four instructions after `if_gt`, we compute the reflection vector and specular power the same way that we did in the previous version.

The following is a computation of light intensities only for lit pixels.

```
dp3         r0, r11, r10            //r0 = (n.l)
if_gt       r0.r, c3.b              //Light only if (n.l)>0
    mul         r1, r0, c3.r        //r1 = 2*(n.l)
    mad         r1, r1, r11, -r10   //reflectance vector - r1=2(n.l)n-l
    dp3_sat     r1, r1, r9          //r1 = (r.v)
    pow         r0.g, r1.r, c2.r    //r1 = (r.v)^shi
    .
    .
    .
endif
```

The next block of code uses predicates to obtain the correct light color. We use the `r6` register (the "loop counter") to discover this. This is due to the inability of relative addressing of the constants in pixel shader 3.0.

```
        mov r3, c4
        setp_gt p0, r6.r, c223
        (p0.z) mov r3, c5
        (p0.y) mov r3, c6
        (p0.x) mov r3, c7
```

The following is a determination of the correct light color in pseudocode.

```
LightColor = c4;  //c4 - color of first light

//compute predicates (setp_gt p0, r6.r, c223)
bool IsSecondLight = (LightIndex > 0.5) ? true : false;
bool IsThirdLight = (LightIndex > 1.5) ? true : false;
bool IsFourthLight = (LightIndex > 2.5) ? true : false;

if (IsSecondLight) LightColor = c5;    //c5 - color of second light
if (IsThirdLight) LightColor = c6;     //c6 - color of third light
if (IsFourthLight) LightColor = c7;    //c7 - color of fourth light
r3 = LightColor;
```

In the last section of the loop, we sample a projector texture for this light. Intensities are then modulated with light color, light intensity, and a spotlight texture sample, and then the result is added to the overall intensity.

NOTE We had to use `mov r4, v2[aL+1]; texldp r1, r4, s3` instead of a simple `texldp r1, v2[aL+1], s3` because it just did not work correctly in the loop — it probably was not translated as a dependent read and always returned the same value.

The following is a computation of one light combination.

```
mov         r4, v2[aL+1]       //move coordinate to temp register
texldp      r1, r4, s3         //load projector texture
mul         r1, r1, r3         //modulate with light color
mul         r1, r1, r3.a       //modulate with light intensity
mad         r8, r0.r, r1, r8   //modulate diffuse intensity with
                               //spotlight and add to overall
mad         r7, r0.g, r1, r7   //modulate specular intensity with
                               //spotlight and add to overall
```

After the loop comes the standard stuff; we load a decal texture and combine accumulated intensities to the final lighting for a given pixel as shown below.

```
texld      r1, v0, s0        //load diffuse texture
mul        r8, r8, c0        //multiply overall diffuse with
                             //material diffuse color
mul        r8, r8, r1        //modulate it with texture
mul        r7, r7, c1        //multiply overall specular with
                             //material specular color
mad        r7, r7, r1.a, r8  //modulate it with gloss map and
                             //and computed diffuse part
mov        oC0, r7           //output the color
```

Per-pixel Environment Bump Mapping with Fresnel Term

Environment mapping (EM) is widely used to simulate reflective surfaces or give surfaces a metallic look. The main concept is to use texture as a source of reflection data, compute the reflection vector of the camera around the surface normal, and use a lookup to this texture. There are three types of EM:

- **Paraboloid EM:** One texture is used (named sphere map because the reflection is stored as seen on a completely reflective sphere). This texture stores information from one hemisphere around the object. Therefore, we can handle reflections only from the hemisphere facing the camera to get expected results. Without shaders, sphere maps are hard to update interactively.
- **Dual paraboloid EM:** Two sphere maps are used, so a complete sphere around the object can be covered.
- **Cube map EM:** Six textures (or one cube map) are used. Each texture represents one side of a cube around an object. A cube map is easy to update (even single sides of a cube can be updated) and easy to use. It is a native format, and it can be sampled with (x,y,z) coordinates representing vector (x,y,z) from the center of the cube.

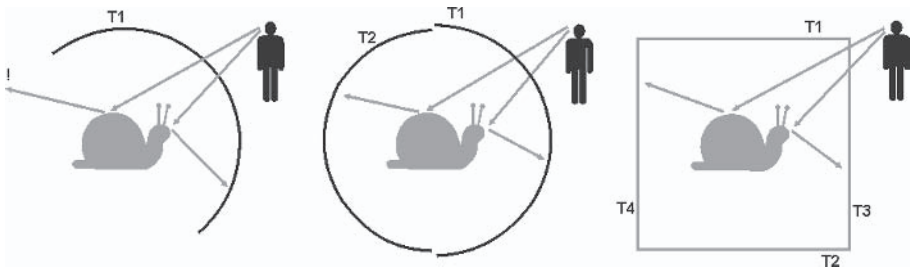


Figure 2: 2D visualization of paraboloid EM, dual paraboloid EM, and cube map EM (left to right)

Each type of EM can be done per vertex or per pixel; it depends on where we compute the reflection vector — in the vertex or pixel shader. In this text, we use cube maps and per-pixel computation of the reflection vector, and the resulting effect is enhanced with the Fresnel term. Versions using pixel shader 1.4 and 2.0 are shown.

Mathematical Background

We all know from Phong shading how to compute light's reflection vector r. Now we do the same, but for view vector v:

$$r = 2(n \bullet v)n - v$$

Until now, all computations in the pixel shader were in tangent space because normals were stored that way. Now we are using a cube map, and it has its own space — cube map space. Every vector (x,y,z) in that space points to a texel in one of the cube sides. We know that we need to do all of our computations in one space. The simplest way here is a transfer to cube map space; the eye vector can be transformed to this space in the vertex shader and the normal vector in the pixel shader. To do this, we need to prepare a transformation matrix from tangent space to cube space in the vertex shader. In most cases, cube map space is the same as world space (just pass the other matrix to the vertex shader and use it instead of the cube transformation matrix where applicable).

We have a matrix that transforms the vector from object space into tangent space formed from tangent t, binormal b, and normal n; let's call it $\mathbf{M}_{\mathbf{O\_to\_T}}$.

$$M_{\mathrm{O\_to\_T}} = \begin{pmatrix} t_x & b_x & n_x & 0 \\ t_y & b_y & n_y & 0 \\ t_z & b_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

We also have a matrix that transforms vectors from object space into cube space; it is passed into the shader as a constant. Let's call it $\mathbf{M}_{\mathbf{O\_to\_C}}$. With these we can compute tangent-to-cube space transformation $\mathbf{M}_{\mathbf{T\_to\_C}}$.

Our two matrices give us the following two equations (note that the subscript part in a vector name indicates the space in which the vector is defined):

$$\mathrm{v}_{\mathrm{tangent}} = \mathrm{v}_{\mathrm{object}} * \mathrm{M}_{\mathrm{O\_to\_T}}$$
$$\mathrm{v}_{\mathrm{cube}} = \mathrm{v}_{\mathrm{object}} * \mathrm{M}_{\mathrm{O\_to\_C}}$$

By multiplying the first one with $\mathbf{M}_{\mathbf{O\_to\_T}}^{-1}$ from the right, we get:

$$\mathrm{v}_{\mathrm{tangent}} * \mathrm{M}_{\mathrm{O\_to\_T}}^{-1} = \mathrm{v}_{\mathrm{object}}$$

By simple replacement of $\mathbf{v}_{\mathbf{object}}$, we have:

$$\mathrm{v}_{\mathrm{cube}} = \mathrm{v}_{\mathrm{tangent}} * \mathrm{M}_{\mathrm{O\_to\_T}}^{-1} * \mathrm{M}_{\mathrm{O\_to\_C}}$$

…which means our searched matrix is:

$$\mathrm{M}_{\mathrm{T\_to\_C}} = \mathrm{M}_{\mathrm{O\_to\_T}}^{-1} * \mathrm{M}_{\mathrm{O\_to\_C}}$$

Unfortunately, to use this matrix easily in the pixel shader, we need to transpose it (`m4x4`, or similar macro instructions, need this). Because $\mathbf{M}_{\mathbf{O\_to\_T}}$ is orthogonal (it is a rotation matrix), its transpose is equal to its inverse, and we use this to get the following result:

$$\mathrm{M}_{\mathrm{T\_to\_C}}{}^{\mathrm{T}} = (\mathrm{M}_{\mathrm{O\_to\_T}}{}^{-1} * \mathrm{M}_{\mathrm{O\_to\_C}})^{\mathrm{T}} = \mathrm{M}_{\mathrm{O\_to\_C}}{}^{\mathrm{T}} * (\mathrm{M}_{\mathrm{O\_to\_T}}{}^{-1})^{\mathrm{T}}$$
$$= \mathrm{M}_{\mathrm{O\_to\_C}}{}^{\mathrm{T}} * (\mathrm{M}_{\mathrm{O\_to\_T}}{}^{\mathrm{T}})^{\mathrm{T}} = \mathrm{M}_{\mathrm{O\_to\_C}}{}^{\mathrm{T}} * \mathrm{M}_{\mathrm{O\_to\_T}}$$

Luckily, needed matrices are in the required form in the vertex shader. $\mathbf{M}_{\mathbf{O\_to\_C}}$ is passed already transposed to the shader, and $\mathbf{M}_{\mathbf{O\_to\_T}}$ is created inside the shader the same way that it was for per-pixel Phong shading (even in the transposed form needed for `m3x3` macros).

We can enhance the resulting reflection with the Fresnel term. It describes the amount of light reflected to the viewer and the amount of light refracted when it strikes a material boundary. The maximum amount of light is reflected when the angle between the surface normal and the eye vector is near π/2, and the minimum is reflected when it is near 0. A good explanation of Fresnel reflection, its equation, and usable approximations is in [5]. We use Schlick's approximation:

$$R(\theta) = R(0) + (1 - R(0)) * (1 - \cos(\theta))^5$$
$$R(0) = \frac{(n_1 - n_2)^2}{(n_1 + n_2)^2}$$

In previous equations, θ is the angle between the eye vector and half vector (between eye and light source). The half vector describes the normal of a surface, which reflects the light ray directly to the eye. In our case, this vector is replaced with a surface normal. R(0) is the Fresnel reflection for zero angle θ, n_1 is the index of the refraction of material from which light comes (commonly air or vacuum), and n_2 is the refraction index of the surface material. For example, a vacuum has a refraction index *ri* = 1.0, air is *ri* = 1.000293, water is *ri* = 1.333333, and diamond is *ri* = 2.417.

Vertex Shader

First we show a version for the DirectX 8 class of hardware, and we begin with the vertex shader. For clearer code and a simpler explanation, version 2.0 of the vertex shader is used (version 1.1 can be extracted by replacing the macro instructions `nrm` and `crs` with respective code known from older shaders).

The following is a 2.0 vertex shader for environment mapping.

```
vs_2_0

// Constant registers
//------------------------------
// c0-c3     - cube map space transposed (might be world space)
// c4-c7     - cube * view * proj
// c8        - Eye position (in cube space)
// c9        - Adjustment factor for cube map
//

// Input registers
//------------------------------
dcl_position v0
dcl_normal v1
dcl_texcoord v2
dcl_tangent v3

// Output
//------------------------------
// oT0 - tex coord
// oT1 - eye vector in cube space
// oT2 - 1st row of tangent-to-cube matrix
// oT3 - 2nd row of tangent-to-cube matrix
// oT4 - 3rd row of tangent-to-cube matrix
// oT5 - vertex modifier
//------------------------------

//Output clip space position, texture coordinates
//Compute eye vector in cube space
//------------------------------
m4x4    oPos, v0, c4      //vertex clip position
mov     oT0.xy, v2.xy     //Texture coordinates for color texture
```

```
m4x4    r8, v0, c0         //Transform vertex into cube map space
add     r0, c8, -r8        //Eye vector
nrm     r1.xyz, r0         //Normalize it
mov     oT1.xyz, r1        //Output it

//Create tangent space basis and matrix from tangent to cube space
//-----------------------------
mov     r9, v3             //Copy tangent, then
crs     r10.xyz, r9, v1    //do cross product to compute binormal and
mov     r11, v1            //then copy normal - matrix is in r9,r10,r11

m3x3    oT2.xyz, c0, r9    //Create transformation matrix (transposed)
m3x3    oT3.xyz, c1, r9
m3x3    oT4.xyz, c2, r9

mul oT5, r8, c9.xxxx       //reflection vector adjustment
```

The first notable thing is that the eye vector is computed in cube space (eye position is passed to the vertex shader in that space, and a vertex is transformed into it as well). We pass the vector without any further transform to the pixel shader.

Later we can create an object-to-tangent space transformation matrix in registers r9, r10, r11, but this time we do not transform the normal and tangent to world space before computation of the binormal. Also note that this matrix is created in transposed form, which is needed for the m3x3 macro. With the last three m3x3 instructions, we are creating the transposed $\mathbf{M}_{\mathbf{T\_to\_C}}$ matrix, as described in the previous section, and we output it to the pixel shader.

The very last instruction requires a little bit of an explanation. Because for cube map lookup we are using vector (x,y,z) from the center of the cube, two points with the same vector (in our case, the reflection vector) will produce the same lookup result. If we have a group of points with the same normal (plane is a typical example), reflection vectors will be almost the same and reflection for the plane will be only one color. To prevent this, we have to modify the reflection vector so it points to the correct place. Take a look at the following figure:

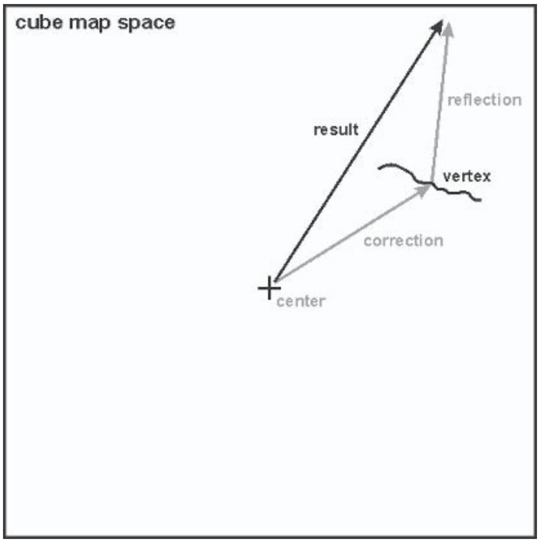


Figure 3: This figure shows modification of a reflection vector.

This figure shows that if we add a vector from the center to the vertex position (all in cube space) to the reflection vector, it points to the correct place and starts at the cube center. In real situations, the reflection vector is not guaranteed to end exactly at the "cube boundary," as shown above, and therefore the resulting vector will likely point to another place, but the results that are produced are good and acceptable.

The last shader instruction prepares this correction vector (remember that the vertex position in cube map space is also a vector from the center to this point) by multiplying it with a scalar constant. This is needed if the cube space transformation is replaced with world space transformation (a very common situation) and (x,y,z) ranges in this space are greater than [–1…1] (valid ranges in the cube map). We compute this tweak ratio (which can be 1/*greatest_coordinate_in_world_space*) outside the vertex shader to get the vertex into cube space range. Of course, a simple `mul` can be replaced with the more sophisticated `mad` (for additional center adjustments).

Pixel Shader 1.4

Let's take a look at the pixel shader. After the usual load of the normal and eye vector (and also the transformation matrix — transposed), we transform the normal into cube space (three successive dot products have the functionality of `m3x3` vector-matrix multiplication). Note that in these instructions the `_bx2` modifier is used for the normal to bias it to the range [–1...1]. Next we compute the eye reflection vector. The last instruction in the first phase is an adjustment of the reflection vector that was already mentioned. Further explanation continues after the shader.

The following is a 1.4 pixel shader for environment mapping.

```
ps_1_4

// Constant registers
//------------------------------
// c0 - R(0)
def c3, 1.0f, 1.0f, 1.0f, 1.0f
def c4, 0.0f, 0.0f, 0.0f, 1.0f

// Used input registers
//------------------------------
// t0 - color / bump coordinates
// t1 - eye vector in cube space
// t2,t3,t4 – tangent_to_cube matrix
// t5 - reflection vector shift

// Used input texture stages
//------------------------------
// stage0 - ambient texture
// stage1 - normal texture
// stage3 – fresnel lookup texture

// Output
//------------------------------
// r0 - output color
//

texld  r1, t0                //normal vector (n)
texcrd r2.rgb, t1.xyz        //eye vector (v)
texcrd r3.rgb, t2           //1st row of tangent-to-cube matrix
texcrd r4.rgb, t3           //2st row of tangent-to-cube matrix
texcrd r5.rgb, t4           //3rd row of tangent-to-cube matrix
texcrd r0.rgb, t5           //vector shift

//multiply normal with transform matrix
dp3 r3.r, r3, r1_bx2        //transform normal to cube space
dp3 r3.g, r4, r1_bx2
dp3 r3.b, r5, r1_bx2

//compute eye reflection vector
dp3 r1.rgb, r3, r2          //r1 = dot(normal, eye)
mad r4.rgb, r1_x2, r3, -r2  //reflectance vector r2=2(n.e)n-e
add r4.rgb, r4, r0          //Shift it

phase
texld  r0, t0               //diffuse texture(n)
texld  r2, r4               //cube map lookup
texld  r3, r1               //Fresnel lookup

mul r0.rgb, r0, r1.r        //to simulate diffuse lighting
lrp_sat r4, c0.r, c0.g, r3  //prepare Fresnel value (with R(0))
//mul_sat r4, r4, r0.a      //modulate with gloss ratio
lrp r0.rgb, r4.a, r2, r0    //compute final color
```

In the second phase, we read environment reflection from the cube texture at coordinates specified with the reflection vector computed before. Then we use the dot product of the normal and eye vector (computed in the first phase) for lookup into the texture holding one part of the Fresnel reflection approximation — $(1–\cos(\theta))^5$. We could compute it in the shader, but limited precision will produce even more errors, and lookup to 1D texture is very fast. We use the computed dot product one more time to simulate diffuse light, so the scene won't look flat (this is not the situation in a game, where the lighting is done in a separate pass).

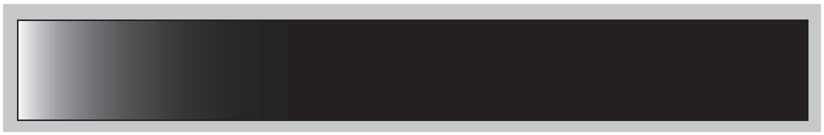


Figure 4: Part of Fresnel reflection intensity: $(1–x)^5$ where x is expected to be $\cos(\theta)$

The `lrp_sat` instruction (saturated linear interpolation, `lrp dest, src0, src1, src2` means `dest = src0*src1 + (1-src0)*src2`) is used to finalize the Fresnel equation (after replacing constants, we will get $R = R(0)*1.0 + (1–R(0))\,.\mathit{LookupValue}$). Then comes modulation with gloss ratio (disabled in this case) and final interpolation between diffuse color and reflection, depending on the value of the Fresnel function.

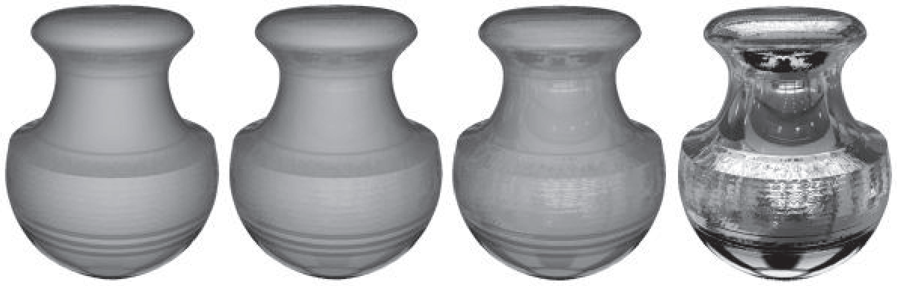


Figure 5: This image shows a reflection of materials with various indexes of refraction — water, flint glass, diamond, and full reflection. Note how the amount of reflection increases with the angle.

Pixel Shader 2.0

The power of pixel shader 2.0 gives us the ability to compute everything in the shader and skip Fresnel texture lookup. Here are the main changes: At the start, we normalize input vectors, and Schlick's approximation of Fresnel term is computed directly in the shader. The rest of the shader is almost identical to the previous one.

The following is a 2.0 pixel shader for environment mapping.

```
ps_2_0

// Constant registers
//------------------------------
// c0 - refraction index
def c1, 2.0f, 1.0f, 5.0f, 0.0f

// Used input registers
//------------------------------
dcl      t0.xy                  //texture coordinates
dcl      t1.xyz                 //eye vector (v)
dcl      t2.xyz                 //1st row of tangent-to-cube matrix
dcl      t3.xyz                 //2nd row of tangent-to-cube matrix
dcl      t4.xyz                 //3rd row of tangent-to-cube matrix
dcl      t5.xyz                 //reflection vector shift

// Used input texture samplers
//-----------------------------
dcl_2d   s0                     //diffuse texture (gloss in alpha)
dcl_2d   s1                     //normal texture
dcl_cube s2                     //cube map texture

// Output
//-----------------------------
// oC0 - output color
//

// Set up needed vectors - load and normalize
//-----------------------------
texld    r0, t0, s1             //load normal
mad      r1, r0, c1.r, -c1.g    //bias normal to range -1,1
m3x3     r0.xyz, r1, t2         //transform to cube space
nrm      r11, r0                //r11 = normalized normal
mov      r1.xyz, t1
nrm      r10, r1                //r10 = normalized eye vector

// Compute eye reflection vector
//-----------------------------
dp3      r9.r, r11, r10         //r9 = dot(normal, eye)
mul      r8.r, r9.r, c1.r       //r1 = 2*(n.v)
mad      r8.rgb, r8.r, r11, -r10 //reflectance vector - r1=2(n.v)n-v
add      r8.rgb, r8, t5         //Shift it

// Compute Fresnel term (Schlick's approximation) F=IR+(1-IR)*(1-(n.l))^5
//-----------------------------
sub      r1.r, c1.g, r9.r         //r1 = 1 - n.v
pow      r0.r, r1.r, c1.b         //r0 = (1 - n.v)^5
lrp      r7.rgb, c0.r, c0.g, r0.r  //final F = IR*1 + (1 - IR)*r0

// Texture lookups and final modulations
//-----------------------------
texld    r0, t0, s0             //diffuse texture
texld    r1, r8, s2             //cube map lookup
```

```
mul      r0.rgb, r0, r9.r        //to simulate diffuse lighting (n.v)
//mul      r7.rgb, r7, r0.a      //modulate Fresnel term with gloss
lrp      r1.rgb, r7.r, r1, r0    //compute final color
mov      oC0, r1                 //output the color
```

The new pixel shader produces results almost identical to the older one. With direct picture-to-picture comparison, very little shift in position of reflection and slightly stronger reflection at glancing angles can be seen. This is all due to normalization of vectors per pixel, and none is notable in moving the environment of a game.

HLSL Version

These programs are translations of pixel and vertex shaders 2.0 into HLSL. The following HLSL vertex shader is for environment mapping.

```
// Used input structure
//------------------------------
struct VS_INPUT {
    float4 vPosition : POSITION; //position in object space
    float3 vNormal   : NORMAL;   //normal
    float2 tcCoord   : TEXCOORD; //texture coordinates
    float3 vTangent  : TANGENT;  //tangent
};

// Used output structure
//------------------------------
struct VS_OUTPUT {
    float4 vClipPos  : POSITION;  //Clipping space position
    float2 tcCoord   : TEXCOORD0; //texture coordinates
    float3 vEye      : TEXCOORD1; //eye vector
    float3x3 mToWorld: TEXCOORD2; //from tangent to world space
    float3 vAdjust   : TEXCOORD5; //perspective spotlight coordinates
};

// Constant registers
//------------------------------
float4x4 mToCube  : register(c0);   //cube map space transposed
float4x4 mToClip  : register(c4);   //world * view * proj
float4 pEye       : register(c8);   //Eye position (cube space)
```

```
float  Adjustment : register(c9);   //Adjustment factor for cube map

// function    : main
// description : vertex shader function
// return      : VS_OUTPUT
// param:
//    VS_INPUT input : vertex shader input
//-----------------------------
VS_OUTPUT main(const VS_INPUT input) {
    VS_OUTPUT output;

    //Following code outputs position and texture coordinates
    //-----------------------------
    output.vClipPos = mul(input.vPosition, mToClip);
    output.tcCoord = input.tcCoord;
    float4 pVertexCube = mul(input.vPosition, mToCube);  //To cube space
    output.vEye = normalize(pEye - (float3)pVertexCube);

    //Create tangent space basis and matrix from tangent to cube space
    //-----------------------------
    float3x3 mToTangent;
    mToTangent[0] = input.vTangent;
    mToTangent[2] = input.vNormal;
    mToTangent[1] = cross(mToTangent[0], mToTangent[2]); //binormal NxT
    output.mToWorld = mul(mToTangent, mToCube);
    output.vAdjust = pVertexCube * Adjustment;           //vector adjustment
    return output;
}
```

The following HLSL pixel shader is for environment mapping.

```
// Used input structure
//-----------------------------
#pragma pack_matrix(col_major)
struct PS_INPUT {
    float2 tcCoord  : TEXCOORD0;  //texture coordinates
    float3 vEye     : TEXCOORD1;  //eye vector
    float3x3 mToWorld: TEXCOORD2; //from tangent to world space 1
    float3 vAdjust  : TEXCOORD5;  //perspective spotlight coordinates
};

// Used output structure
//-----------------------------
struct PS_OUTPUT {
    float4 vColor  : COLOR0;    //render target 0
};

// Used input texture samplers
//-----------------------------
sampler smplTexture : register(ps,s0);  //decal texture
sampler smplNormal  : register(ps,s1);  //normal texture
sampler smplCube    : register(ps,s2);  //cube map texture

// function    : main
// description : pixel shader function
// return      : PS_OUTPUT
// param:
//    PS_INPUT input : pixel shader input - output from VS
//    float refindex: c0 - R(0) for Fresnel term
//-----------------------------
PS_OUTPUT main(const PS_INPUT input,
               uniform float refindex : register(c0)) {
    PS_OUTPUT output;

    //load and normalize
    //-----------------------------
    float3 vNormal = tex2D(smplNormal, input.tcCoord).xyz;
    vNormal = 2.0 * vNormal - 1.0;              //bias to [-1...1]
    vNormal = mul(vNormal, input.mToWorld);     //to world space
    vNormal = normalize(vNormal);
    float3 vEye = normalize(input.vEye);

    //compute adjusted eye reflection vector
    //-----------------------------
    float eDotN = dot(vEye, vNormal);
    float3 vEyeReflected = 2* eDotN * vNormal – vEye + input.vAdjust;

    float4 cube = texCUBE(smplCube, vEyeReflected);
    float4 color = tex2D(smplTexture, input.tcCoord);

    float Fresnel = lerp(pow(1 - dot(vNormal, vEye), 5), 1, refindex);
    Fresnel = Fresnel * color.a;     //reflection only on shiny parts

    float4 diffuse = color * dot(vNormal, vEye);  //diffuse simulation
    output.vColor = lerp(diffuse, cube, Fresnel);
    return output;
}
```

Background for Advanced Models

For the next two lighting models, we need to explain a few new concepts; both models rely on spherical coordinates, roughness of a surface, and masking or shadowing of light. These are explained in the next few sections.

Spherical Coordinates

Since lighting equations are more about directions (light and view vectors, normal vectors, etc.) than positions, it is often better to describe a vector not in Cartesian coordinates $\mathbf{v} = (v_x, v_y, v_z)$ but with a pair of angles θ (polar or elevation) and ϕ (azimuth) and length of vector. A polar angle is an angle between a vector and one base vector, and an azimuth angle is an angle between a vector projected into a plane defined by the remaining two base vectors and one of these vectors. This is good for a description of the lighting equation vectors, since they are normalized, so we can just ignore the length parameter. In the case of lighting, base vectors will almost always be n, t, and b, polar angles will be computed in respect to normal n, and azimuth angles are relative to the tangent t. The following figure shows this situation.

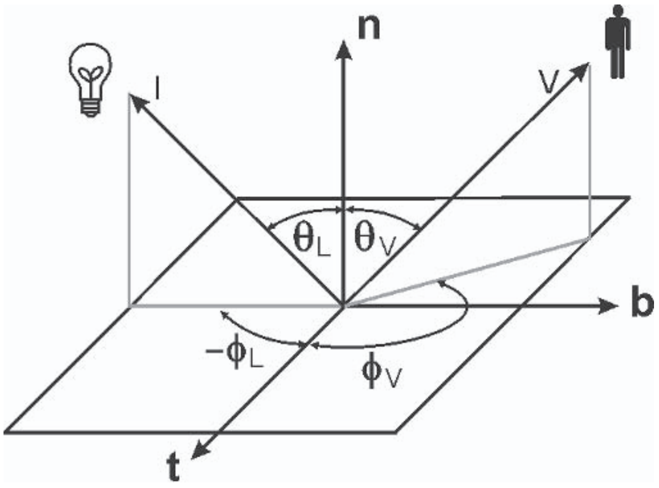


Figure 6: Spherical coordinates

The relationship between Cartesian (v_x, v_y, v_z) and spherical (θ_v, ϕ_v) coordinates for normalized vector v is shown in the following equation:

$$v_x = \cos(\phi_v) * \sin(\theta_v)$$
$$v_y = \sin(\phi_v) * \sin(\theta_v)$$
$$v_z = \cos(\theta_v)$$
$$\phi_v = \arctan\left(\frac{v_x}{v_y}\right)$$
$$\theta_v = \arctan\left(\frac{\sqrt{v_x^2 + v_y^2}}{v_z}\right) = \arccos\left(\frac{v_z}{\sqrt{v_x^2 + v_y^2 + v_z^2}}\right) = \arccos(v_z)$$

It is obvious that the dot product between the normal and light (or eye) vector is nothing more than a cosine of a polar angle.

Roughness of a Surface

Both of the models presented use a micro-facet model to simulate structure (roughness) of a surface — every piece of surface is composed of tiny facets. The roughness model is called a v-cavities model because the structure of the surface is modeled with cavities in the shape of a V. It was introduced by Torrance-Sparrow in [6].

While facets in every model have other properties, there is something in common. The resulting intensity of a surface piece depends on the sum of the facet intensities. The importance of this approach is shown in Figure 7 (taken from the later Oren-Nayar model), which shows that the surface composed of totally Lambertian micro-facets (independent of the position of a viewer) is not Lambertian when viewed from a distance where several micro-facets are covered by one pixel.

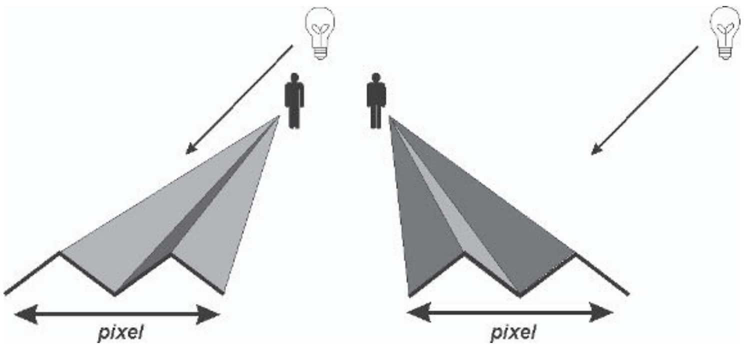


Figure 7: The brightness of a pixel that covers a surface patch with Lambertian micro-facets depends on the viewer's position. On the left, the viewer "sees" brighter facets and the resulting pixel is brighter. On the right, the pixel is darker.

Due to the complexity of the computation, actual rendering does not use real facets. Instead, models use a probability distribution function that predicts the approximate number of facets with a specific normal (heading the specific way). These functions are described later with a respective model.

Masking and Shadowing

With the introduction of a micro-facet model, masking and shadowing of individual facets should be taken into account. If a facet blocks a portion of light reflected from another facet, it is masking. If a facet blocks incoming light so another facet is shadowed by this one, it is shadowing.

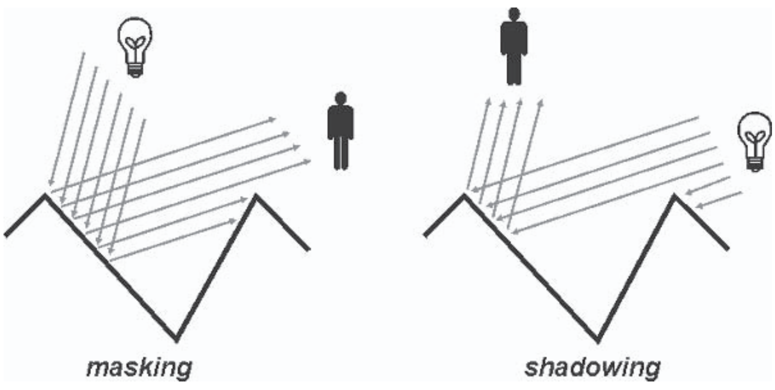


Figure 8: Facet masking and shadowing effects

The equation for computing the masking-shadowing term (or GAF for geometric attenuation factor) is:

$$GAF = \min\left(1, \frac{2(n \bullet h)(n \bullet v)}{(v \bullet h)}, \frac{2(n \bullet h)(n \bullet l)}{(v \bullet h)}\right)$$

The h vector has the same meaning that it had previously in the Fresnel term equation. GAF is used as an attenuator for intensity of a fully lit facet or surface patch.

The Oren-Nayar Model

Michael Oren and Shree K. Nayar published a lighting model in 1992 (also referred to as the O-N model, or just O-N, later in the text) that enhances the standard Lambertian model (commonly used to compute diffuse lighting). You can find a very detailed description in [7] and [8]. We provide only a short introduction and the math needed for future shader development here.

The Lambertian model depends only on the light position and surface normal. While this can be correct for some smooth surfaces, it is false for rough surfaces, such as clay or concrete. This is because a viewer's position cannot be ignored in these cases. The following figure shows why.

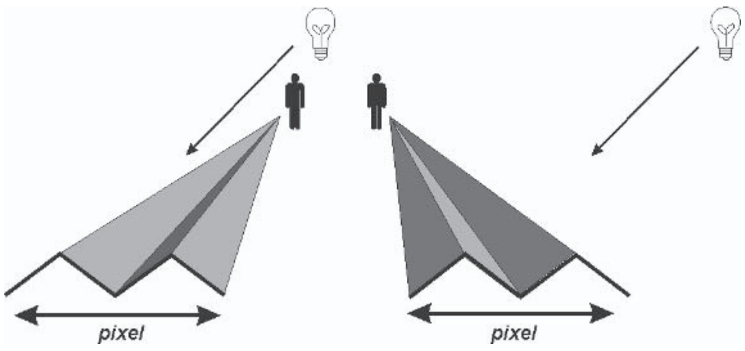


Figure 9: Detail of a rough surface. On the left, the viewer "sees" more of the brighter facets and the resulting pixel is brighter. On the right, the pixel will be darker.

The O-N model assumes that the surface consists of tiny microfacets, each perfectly Lambertian. This model takes into account masking and shadowing of facets and also adds an interreflection factor — light bouncing between adjacent faces.

The model uses Gauss distribution function D_{Gauss} with zero mean and standard deviation σ to describe the roughness of the surface. We use σ as a roughness parameter. If it is zero, all facet normals are aligned with a surface normal. The greater σ is, the deeper the cavities are. Here you can see the Gauss distribution function equation, where C is the normalization constant and θ is the polar angle of the facet normal with respect to the surface patch normal:

$$D_{Gauss} = C * e^{-\frac{\theta^2}{\sigma^2}}$$

Overall brightness of a surface patch is integral to the intensities of all its facets — some masked by others, some shadowed, and some lit by the reflection from other ones. To compute this integral in real time, we have to wait for a new generation of hardware. The authors of this model (knowing its complexity) simplified it into a single equation (interested readers should look into the original papers), which is unfortunately still far from usability in game-oriented computer graphics applications. But authors simplified it even more, ignoring the interreflection factor and terms contributing only little to the final intensity. Now the hardware is powerful enough, and we can use it in the pixel shader for real-time lighting and shading. Here is a simplified O-N equation:

$$I_{O-N} = \cos(\theta_L) * (A + B * \max(0, \cos(\phi_V - \phi_L)) * \sin(\alpha) * \tan(\beta))$$

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

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

$$\alpha = \min(\theta_L, \theta_V)$$

$$\beta = \max(\theta_L, \theta_V)$$

In the previous equation, θ_L means the polar angle for the light view vector, and θ_V means the polor angle for the view angle according to the surface normal. ϕ_L, ϕ_V are azimuth angles for the light and view vector, respectively, according to the tangent.

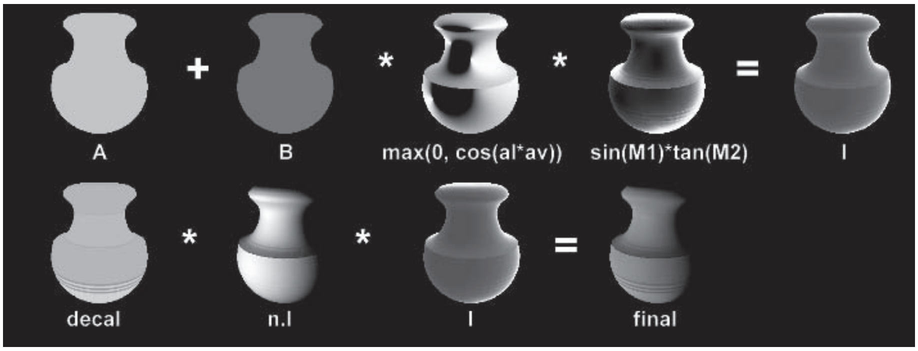


Figure 10: Visualization of the Oren-Nayar equation

Shaders

In this section we implement the Oren-Nayar lighting model with shaders version 2.0. We show and describe only the pixel shader because the vertex shader is identical to the one shown in the Phong shading section at the beginning of this article.

The following is the pixel shader for Oren-Nayar lighting.

```
ps_2_0

// Constant registers
//------------------------------
//  c0 - roughness (R) (should be redefined in material)
def     c31, 1.0f, 2.0f, 0.5f, 0.33f    //useful constants
def     c30, 0.45f, 0.09f, 0.0f, 0.0f   //useful constants
def     c29, 0.0f, 1.0f, 2.0f, 3.0f     //useful constants

// Used input registers
//------------------------------
dcl     t0.xy                   //texture coordinates
dcl     t1.xyz                  //light vector
dcl     t2.xyz                  //eye vector

// Used input texture samplers
```

```
//------------------------------
dcl_2d  s0                      //diffuse texture (gloss in alpha)
dcl_2d  s1                      //normal texture
dcl_2d  s2                      //lookup

// Output
//------------------------------
// oC0 - output color

// Load and normalize input vectors
//------------------------------
texld   r0, t0, s1              //load normal
mad     r1, r0, c31.g, -c31.r   //bias normal to range -1,1
nrm     r11, r1                 //r11 = normalized normal
mov     r1.xyz, t1
nrm     r10, r1                 //r10 = normalized light vector
mov     r1.xyz, t2
nrm     r9, r1                  //r9 = normalized eye vector

// A = 1 - 0.5 * R^2 / (R^2 + 0.33)
//------------------------------
mul     r0, c0.r, c0.r          //R^2
add     r1, r0, c31.a           //R^2 + 0.33
rcp     r1, r1.r                //1 / (R^2 + 0.33)
mul     r0, r1.r, r0            //R^2 / (R^2 + 0.33)
mad     r8, r0.r,-c31.b, c31.r  //1 - 0.5 * R^2 / (R^2 + 0.33)

// B = 0.45 * R^2 / (R^2 + 0.09)
//------------------------------
mul     r0, c0.r, c0.r          //R^2
add     r1, r0, c30.g           //R^2 + 0.09
rcp     r1, r1.r                //1 / (R^2 + 0.09)
mul     r0, r1.r, r0            //R^2 / (R^2 + 0.09)
mul     r7, r0.r, c30.r         //r8 = 0.45 * R^2 / (R^2 + 0.09)

// CX = Max(0, cos (l',v'))
//------------------------------
dp3     r1, r10, r11       //these four instructions are projecting the
mul     r1, r11, r1        //light vector to the plane defined by T and B
sub     r1, r10, r1        //equation is : l' = normalize(l - n * (n.l))
nrm     r0, r1             //in our case r0 = normalize(r10 - r9 * r1)

dp3     r2, r9, r11        //these four instructions are projecting the
```

```
mul     r2, r11, r2         //eye vector to the plane defined by T and B
sub     r2, r9, r2          //equation is : v' = normalize(v - n * (n.v))
nrm     r1, r2              //in our case r1 = normalize(r11 - r9 * r2)

dp3     r6, r0, r1          //(l'.v') = (r0.r1)
max     r6, r6, c29.r       //only positive values

// DX = texture lookup for sin(a)*tan(b); a=max(0-r,0-i); b=min(0-r,0-i)
//-----------------------------
dp3     r1.x, r10, r11           //n.l
dp3     r1.y, r9, r11            //n.v
texld   r0, r1, s2               //look up
mov     r5, r0.r

// complete it - A + B*CX*DX
//------------------------------
mul     r0, r5, r6               //CX*DX
mul     r0, r0, r7               //B*CX*DX
add     r4, r0, r8               //A + B*CX*DX

// Load texture, compute diffuse part and combine it all to output
//------------------------------
dp3_sat r1, r11, r10             //n.l
texld   r0, t0, s0               //load diffuse and specular texture
mul     r0, r0, r1               //compute diffuse texture
mul     r0, r0, r4               //modulate by A + B*CX*DX
mov     oC0, r0
```

Right after setup and renormalization of vectors, we compute the A and B parameters of the equation. Because these depend only on the roughness parameter and are constant for the shader, they should be computed outside it in a real game situation to gain some speed.

Then we compute $\max(0, \cos(\phi_V - \phi_L))$. To do this, we project eye and light vectors to the plane described by t and b using the next equation:

$$V_{projected} = v - n(n \bullet v)$$

This can also be done by resetting their z component, but we will lose the bump map in that case. After renormalization of both modified vectors (v' and l'), we can replace $\cos(\phi_V - \phi_L)$ with cos(**v'** • **l'**). To see why this works, take a look at Figure 6 in the section titled "Spherical Coordinates."

After this, we compute the sin(α)*tan(β) part. We know that we can easily compute cosines of polar angles (dot product with normal). Sine or tangent is a bit more difficult to do with the constraints of shaders 2.0 because we would need to do arccos (we do not have enough instruction slots), or we would need to replace sin and tan with cosines. To solve this, we transfer the entire computation to the 2D lookup function, where the *x* coordinate is the cosine of θ_L (**n** • **l**) and *y* is the cosine of θ_V (**n** • **v**). Textures accept only coordinates from range [0…1], but this is not a problem because θ_L and θ_V are both less than π/2 and positive. (θ_V is always less than π/2 because we render only pixels facing toward the camera. Cases where θ_L is greater than π/2 can be ignored, as in this case the cos(θ_L) at the beginning of the equation will reset the whole computation to zero.) So both dot products will be in the range [0…1].

Here, the texture function can use arccos to restore angles from inputs. Then the maximum and minimum are chosen and sin(α)*tan(β) is computed. Because the result of this function could not be in the range [0…1], we use a new floating-point texture format — only one channel (red) with full 32-bit float precision.

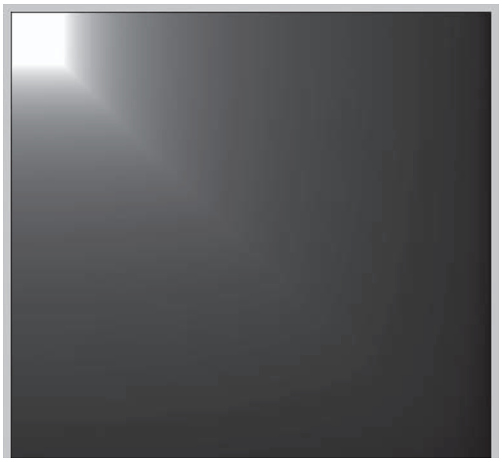


*Figure 11: Lookup texture for sin(α)*tan(β). The result is divided by 9.0 to show greater range.*

Later in the shader we complete these contributions and modulate the result with the standard Lambertian diffuse and color texture.

Due to its complexity, it is hard to port this shader into the pixel shader 1.4 version. However, this could be done if we replaced azimuth angles with respective polar angles everywhere in computation. Then we could use one 3D lookup texture with three parameters (**n•l**, **n•v**, *roughness*) and compute all the lighting with one dependent texture lookup.

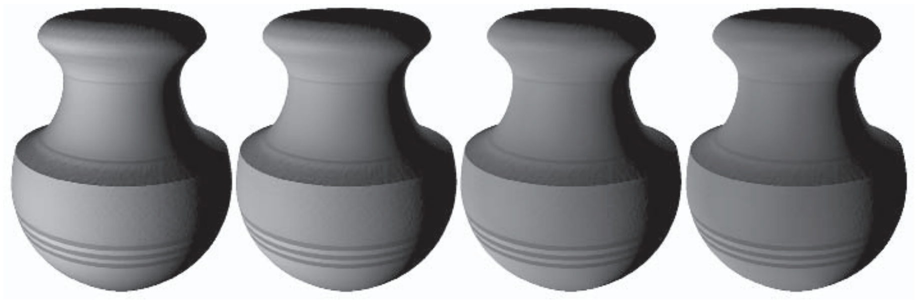


Figure 12: A vase with various roughness values 0.0, 0.3, 0.6, and 1.0 (left to right). If roughness is 0, the model is identical to Lambertian. Note how the vase gets flatter with increased roughness.

HLSL Version

The following HLSL vertex shader is for Oren-Nayar lighting.

```
// Used input structure
//-----------------------------
struct VS_INPUT {
    float4 vPosition : POSITION; //position in object space
    float3 vNormal   : NORMAL;   //normal
    float2 tcCoord   : TEXCOORD; //texture coordinates
    float3 vTangent  : TANGENT;  //tangent
};

// Used output structure
//-----------------------------
struct VS_OUTPUT {
    float4 vClipPos: POSITION;  //Clipping space position
    float2 tcCoord : TEXCOORD0; //texture coordinates
    float3 vLight  : TEXCOORD1; //light vector
    float3 vEye    : TEXCOORD2; //eye vector
};

// Constant registers
//------------------------------
float4x4 mToWorld : register(c0);   //world space transposed
float4x4 mToClip  : register(c4);   //world * view * proj
float4 pLight     : register(c8);   //Light position (world space)
float4 pEye       : register(c9);   //Eye position (world space)

// function    : main
// description : vertex shader function
// return      : VS_OUTPUT
// param:
//    VS_INPUT input : vertex shader input
//------------------------------
VS_OUTPUT main(const VS_INPUT input) {
    VS_OUTPUT output;

    //The following code outputs position and texture coordinates
    //------------------------------
    output.vClipPos = mul(input.vPosition, mToClip);
    output.tcCoord = input.tcCoord;
    float4 pVertexWorld = mul(input.vPosition, mToWorld);

    //The following code generates tangent space base vectors
    //------------------------------
    float3x3 mToTangent;
    mToTangent[0] = mul(input.vTangent, (float3x3)mToWorld);
    mToTangent[2] = mul(input.vNormal, (float3x3)mToWorld);
    mToTangent[1] = cross(mToTangent[0], mToTangent[2]);

    //Compute light and eye vectors
    //------------------------------
    float3 vToLight = normalize(pLight - pVertexWorld);
    output.vLight = mul(mToTangent, vToLight);

    float3 vToEye = normalize(pEye - pVertexWorld);
    output.vEye = mul(mToTangent, vToEye);
}
```

The following HLSL pixel shader is for Oren-Nayar lighting.

```
// Used input structure
//-----------------------------
struct PS_INPUT {
    float2 tcCoord : TEXCOORD0; //input texture coordinates
    float3 vLight  : TEXCOORD1; //light vectpor
    float3 vEye    : TEXCOORD2; //eye vector
};

// Used output structure
//-----------------------------
struct PS_OUTPUT {
    float4 vColor  : COLOR0;    //render target 0
};

// Used input texture samplers
//-----------------------------
sampler smplTexture : register(ps,s0);  //decal texture
sampler smplNormal  : register(ps,s1);  //normal texture
sampler smplLookUp  : register(ps,s2);  //lookup texture for sin.tan

// function    : main
// description : pixel shader function
// return      : PS_OUTPUT
// param:
//    PS_INPUT input : pixel shader input - output from VS
//    float roughness : c0 - roughness of a surface
//-----------------------------
PS_OUTPUT main(const PS_INPUT input, uniform float roughness : c0) {
    PS_OUTPUT output;

    // Load and normalize input vectors
    //-----------------------------
    float3 vNormal = tex2D(smplNormal, input.tcCoord).xyz;
    vNormal = normalize(2.0*vNormal - 1.0);      //bias
    float3 vLight = normalize(input.vLight);
    float3 vEye = normalize(input.vEye);

    // A = 1 - 0.5 * R^2 / (R^2 + 0.33)
    // B = 0.45 * R^2 / (R^2 + 0.09)
    //-----------------------------
    float roughness2 = roughness*roughness;
    float A = 1.0f - 0.5f * roughness2 / (roughness2 + 0.33f);
    float B = 0.45f * roughness2 / (roughness2 + 0.09f);

    // CX = Max(0, cos (l',v'))
    //-----------------------------
    float  normalDotLight  = dot(vNormal, vLight);
    float3 vLightProjected = normalize(vLight - vNormal*normalDotLight);

    float  normalDotEye  = dot(vNormal, vEye);
    float3 vEyeProjected = normalize(vEye - vNormal*normalDotEye);

    float CX = saturate(dot(vLightProjected, vEyeProjected));

    // DX = texture lookup for sin*tan
    //-----------------------------
    float2 tcLookup = {normalDotLight, normalDotEye};
    float DX = tex2D(smplLookUp, tcLookup);

    // completize it - (n.l)*texture*(A + B*CX*DX)
    //-----------------------------
    output.vColor = saturate(normalDotLight)*
                    tex2D(smplTexture, input.tcCoord)*(A+B*CX*DX);
    return output;
}
```

Cook-Torrance Model

This model was published in 1981 [9] and is based on the Torrance-Sparrow model from 1967 [6]. The Cook-Torrance (C-T) model is often used to evaluate specular highlights of metals and plastics. It is a physically based method and surpasses Phong's model because it was developed using measured data from real materials and uses physically measurable factors, such as energy and wavelength.

The Cook-Torrance model uses:

- A micro-facet model for surface roughness
- Fresnel's equation to compute the amount of reflection and color shift of highlight

- The geometric attenuation factor for micro-facet self shadowing and masking

Some interesting conclusions about the model are:

- The specular highlight usually has the color of the material and not the color of the light.
- The Fresnel equation predicts a color shift of the specular component at glancing angles.
- Some types of materials (painted objects and plastics) have specular and diffuse components that do not have the same color.

We use Beckman's distribution function $D_{Beckman}$ for surface roughness. In the following equation, m stands for surface roughness and θ is the angle between the normal and half vector $(\cos(\theta)=(\mathbf{n \bullet h}))$.

$$D_{Beckman} = \frac{1}{m^2 \cos^4 \theta} e^{-\left(\frac{\tan^2 \theta}{m^2}\right)}$$

$D_{Beckman}$ can be changed into a more shader-friendly form with well-known goniometric expressions:

$$\tan^2 \theta = \frac{\sin^2 \theta}{\cos^2 \theta} = \frac{1-\cos^2 \theta}{\cos^2 \theta} = \frac{1=(n \bullet h)^2}{(n \bullet h)^2}$$

The resulting equation then is:

$$D_{Beckman} = \frac{1}{m^2 (n \bullet h)^4} e^{-\left(\frac{1-(n \bullet h)^2}{m^2 (n \bullet h)^2}\right)}$$

Because we are using Schlick's approximation of Fresnel's equation, which works for non-polarized lights, we have to ignore color shifts in specular highlights. The following equation represents a specular component of the Cook-Torrance lighting model. *F* stands for Fresnel term, *GAF* for geometric attenuation factor, and *D* for distribution function.

$$I_{C-T} = \frac{F * GAF * D}{\pi (n \bullet l)(n \bullet v)}$$

Note that (**n•l**) is ignored in our shader because the previous equation has a form used for BRDF, where computed intensity (of every used lighting model) is multiplied with (**n•l**). In our case, there is no need to multiply and divide by the same term.

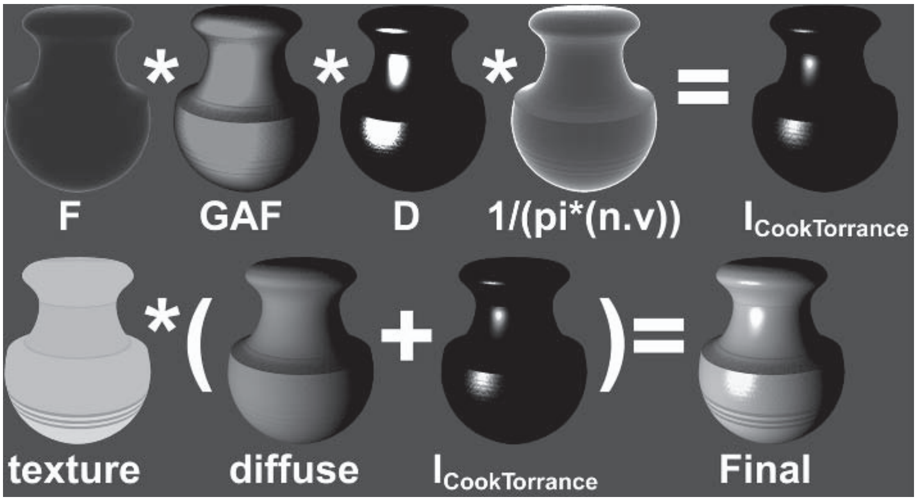


Figure 13: An illustration of the Cook-Torrance equation. The intensity range is from 0 (black) to 3 (white) to show how D contributes to the final lighting.

We attempt to port this model to Direct3D first with 2.0 shaders and then with 1.4 shaders.

Shaders 2.0

The vertex shader for Direct3D 9 is almost the same as the shader for Phong shading, except we are also computing the half vector with the last instructions.

The following is the vertex shader 2.0 for Cook-Torrance lighting.

```
vs_2_0

// Constant registers
//------------------------------
// c0-c3     - world space transposed
// c4-c7     - world * view * proj
// c8        - Light position (in world space)
// c9       - Eye position (in world space)

// Input registers
//------------------------------
dcl_position v0
dcl_normal v1
dcl_texcoord v2
dcl_tangent v3

// Fixed temporary registers
//------------------------------
// r9, r10, r11 - tangent space basis
// r8 - vertex world position

// Output
//------------------------------
// oT0 - texture coordinates
// oT1 - Light vector (in tangent space)
// oT2 - eye vector (in tangent space)
// oT3 - half vector (in tangent space)

//The following code will output position and texture coordinates
//------------------------------
m4x4 oPos, v0, c4        //vertex clip position
mov oT0.xy, v2.xy        //Texture coordinates for color texture
m4x4 r8, v0, c0          //Transform vertex into world position

//The following code generates tangent space base vectors
//------------------------------
m3x3 r11.xyz, v1, c0     //transform normal N to world space
mov r11.w, v1.w
m3x3 r9.xyz, v3, c0      //transform tangent T to world space
mov r9.w, v3.w
crs r10.xyz, r9, r11     //The cross product to compute binormal NxT

//Computes light, eye, and half vectors
//------------------------------
add r0, c8, -r8          //Build the light vector
nrm r6, r0               //normalize vector
m3x3 oT1.xyz, r6, r9     //transform vector into tangent space

add r0, c9, -r8          //Build the eye vector
nrm r7, r0               //normalize vector
```

```
m3x3 oT2.xyz, r7, r9     //transform vector into tangent space

add r0, r6, r7           //build the half vector between light and eye
nrm r1, r0               //normalize vector
m3x3 oT3.xyz, r1, r9     //transform vector into tangent space
```

All of the lighting is done with the pixel shader. The entire shader is just a computation of previously specified equations. Everything can be seen from the comments, and therefore no further explanation is needed.

The following is the pixel shader for Cook-Torrance lighting.

```
ps_2_0

// Constant registers
//-----------------------------
def c0, 2.71828182845904523536028747135266f, //e
        3.14159265358979323846264338327950f, //pi
        4.0f,   //useful constants
        1.0f    //useful constants
def c1, 5.0f, 2.0f, 1.0f, 0.0f      //useful constants
//  c2 - roughness (should be redefined in material)
//  c3 - refraction index (should be redefined in material)

// Used input registers
//-----------------------------
dcl     t0.xy                //texture coordinates
dcl     t1.xyz               //light vector
dcl     t2.xyz               //eye vector
dcl     t3.xyz               //half vector between light and eye

// Used input texture samplers
//-----------------------------
dcl_2d  s0                   //diffuse texture (gloss in alpha)
dcl_2d  s1                   //normal texture

// Output
//-----------------------------
// oC0 - output color

// Load and normalize input vectors
//-----------------------------
texld   r0, t0, s1           //load normal
```

```
mad     r1, r0, c1.g, -c1.b   //bias normal to range -1,1
nrm     r11, r1               //r11 = normalized normal
mov     r1.xyz, t1
nrm     r10, r1               //r10 = normalized light vector
mov     r1.xyz, t2
nrm     r9, r1                //r9 = normalized eye vector
mov     r1.xyz, t3
nrm     r8, r1                //r8 = normalized half vector

// Compute Beckman's distribution function
// D = (1 / m^2*cos(A)^4) * e^(-tan(A) / m^2)
//-------------------------------
dp3     r1, r11, r8           //n.h
mul     r1, r1.r, r1.r        //x = (n.h)^2
mul     r2, c2.r, c2.r        //y = m^2
mul     r3, r2.r, r1.r        //z = m^2 * (n.h)^2

sub     r4, c0.a, r1.r        //1-x
rcp     r5, r3.r              //1 / z
mul     r4, r4.r, r5.r        //(1-x) / z
pow     r5, c0.r, -r4.r       // pow(e, -(1-x) / z)

mul     r3, r3.r, r1.r        //z*x
rcp     r4, r3.r              //1/(z*x)

mul     r1, r5.r, r4.r        //r1 will hold final D

// Compute Fresnel term (Schlick's approximation)
// F = IR + (1-IR)*(1 - (n.l))^5
//-------------------------------
dp3     r3, r11, r9             //n.v
sub     r3, c0.a, r3.r          //1 - n.v
pow     r3, r3.r, c1.r          //(1 - n.v)^5
lrp     r2, c3.r, c3.g, r3.r    //r2 will hold final F

// Compute self shadowing term
// G = min(1, X*(n.l), X*(n.v)); X = 2*(n.h) / (v.h)
//-------------------------------
dp3     r3, r11, r8             //n.h
dp3     r4, r9, r8              //v.h
mul     r3, r3.r, c1.g          //2.(n.h)
rcp     r5, r4.r                //1 / (v.h)
mul     r3, r3.r, r5.r          //X = 2.(n.h) / (v.h)
```

```
dp3     r4, r11, r10            //n.l
dp3     r5, r11, r9             //n.v
mul     r4, r4.r, r3.r          //second parameter of G : X*(n.l)
mul     r5, r5.r, r3.r          //third parameter of G : X*(n.v)
min     r4, r4.r, r5.r          //min of second and third parameters
min     r3, r4.r, c0.a          //min of previous and 1. We have final G

// Compute denominator part of lighting equation - 1 / (n.v)*pi
//-----------------------------
dp3     r5, r11, r9             //n.v
mul     r5, r5.r, c0.g          //(n.v)*pi
rcp     r4, r5.r                //r4 = 1 / (n.v)*pi

// Compute final Cook-Torrance specular term - (1 / (n.v)*pi) * D*F*G
//-----------------------------
mul     r5, r1.r, r2.r          //D.F
mul     r5, r5.r, r3.r          //D.F.G
mul     r5, r5.r, r4.r          //D.F.G / ((n.v)*pi)

// Load texture, compute diffuse part, and combine it all to output
//-----------------------------
dp3_sat r1, r11, r10            //n.l
texld   r0, t0, s0              //load diffuse and specular texture
mul     r2, r0, r0.a            //modulate texture with gloss map
mul     r1, r0, r1              //compute diffuse texture
mad     r0, r2, r5.r, r1        //compute specular + diffuse
mov     oC0, r0
```

Shaders 1.4

Here we attempt to port the Cook-Torrance specular highlight to the Direct3D 8.1 class of hardware. Only the pixel shader is described because the vertex shader is very similar to version 2.0.

Due to the very low precision of the pixel shader and only eight possible instructions per phase, we transferred the entire calculation to lookup textures. The first 2D texture stores the Fresnel part of the Cook-Torrance lighting equation divided by (**n•v**):

$$F(\mathrm{n} \bullet \mathrm{v}, R(0)) = \frac{R(0) + (1 - R(0))(1 - \mathrm{n} \bullet \mathrm{v})^5}{\mathrm{n} \bullet \mathrm{v}}$$

The second 2D texture stores Beckman's distribution function divided by π:

$$D(\mathrm{n} \bullet \mathrm{h}, m) = \frac{1}{\pi * m2(\mathrm{n} \bullet \mathrm{h})^4} e^{-\left(\frac{1-(\mathrm{n}\bullet\mathrm{h})^2}{m^2 (\mathrm{n}\bullet\mathrm{h})^2}\right)}$$

By multiplying the results from these functions, we get an almost complete C-T specular equation, except for the geometric attenuation term (we ignore it here). One problem is that these functions are greater than 1 for some parameters, and so we have to store as much of their range as possible. We do this by storing [0…1] in the red channel and then subtracting by one and storing the remainder (range [1…2]) in the green channel. We do something similar for the blue and alpha channels. In the pixel shader, we are able to restore function to range [0…4].

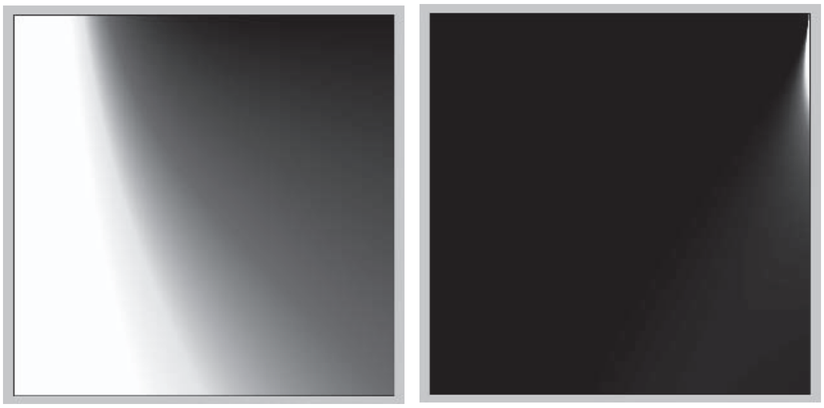


Figure 14: The image on the left is a visualizization of the Fresnel texture; on the right is the Beckman texture.

In the first phase of the shader, we prepare coordinates for lookup textures. Additionally, we compute the diffuse part of shading because in the second phase we do not have enough instructions left.

In the second phase, we sample both functions and unpack intensity from color channels by summing them in a red component. After that, we compute the final color value. More clever pack/unpack is also possible; the red channel will hold range

[0…1], and the green will hold [1…2]. The blue channel range [2…4] and the alpha range [4…6] will be stored. With the _x2 register modifier, we can multiply the register value by two without increasing the instruction count, and the range will then be [0…6].

The following is pixel shader 1.4 for Cook-Torrance lighting.

```
ps_1_4

// Constant registers
//-----------------------------
//  c2 - vector in form (roughness, 1.0, 1.0, 1.0)
//  c3 - vector in form (refraction_index, 1.0, 1.0, 1.0)

// Used input registers
//-----------------------------
// t0 - texture and normal coordinates
// t1 - light vector
// t2 - eye vector
// t3 - half vector

// Used input texture stages
//-----------------------------
// stage0 - diffuse texture
// stage1 - normal texture
// stage2 - f(n_dot_h,roughness) = Beckman(n_dot_h,roughness)/pi
// stage3 - f(n_dot_l,RI) = Fresnel(n_dot_v, RI)/n_dot_v


// Output
//-----------------------------
// r0 - output color

// Sample normal texture and load vectors from input
//-----------------------------
texld r0, t0                //load texture (gloss map in alpha)
texld r1, t0                //normal vector (n)
texcrd r2.rgb, t1.xyz       //Light vector (l)
texcrd r3.rgb, t2.xyz       //eye vector (v)
texcrd r4.rgb, t3.xyz       //half vector (h)

// Compute lookup texture coordinates
```

```
//------------------------------
dp3 r5.rgb, r1_bx2, r2      //n.l - for diffuse part

dp3_sat r2.rgb, r1_bx2, r4  //n.h - first parameter of Beckman lookup
mov r2.g, c2.r              //roughness (M) - 2nd parameter of lookup

dp3_sat r3.rgb, r1_bx2, r3  //n.v - for Fresnel equation
mov r3.g, c3.r              //index of refraction – 2nd lookup parameter

mul r1.rgb, r0, r5          //diffuse lighting
mul r4.rgb, r0, r0.a        //modulate texture with gloss

// 2nd phase - Sample diffuse texture and lookup in texture functions
//------------------------------
phase
texld r0, t0                //load texture (gloss map in alpha)
texld r2, r2                //Beckman distribution (B)
texld r3, r3                //Fresnel lookup (F)

// Expand Beckman and Fresnel to range [0...4] from RGB channels
//------------------------------
add r2.r, r2.r, r2.g        //unpack - R+G
add r2.g, r2.r, r2.b        //R+G+B
add r2.r, r2.r, r2.g        //R+G+B+A

add r3.r, r3.r, r3.g        //unpack - R+G
add r3.r, r3.r, r3.b        //R+G+B
add r3.r, r3.r, r3.a        //R+G+B+A

mul r2.r, r2.r, r3.r        //I(spec) = B * F
mad r0.rgb, r4, r2.r, r1    //spec_color * I(spec) + diffuse
```

HLSL Version

The following is the HLSL vertex shader for Cook-Torrance lighting.

```
// Used input structure
//------------------------------
struct VS_INPUT {
    float4 vPosition : POSITION; //position in object space
    float3 vNormal   : NORMAL;   //normal
    float2 tcCoord   : TEXCOORD; //texture coordinates
    float3 vTangent  : TANGENT;  //tangent
};

// Used output structure
//-----------------------------
struct VS_OUTPUT {
    float4 vClipPos: POSITION;  //Clipping space position
    float2 tcCoord : TEXCOORD0; //texture coordinates
    float3 vLight  : TEXCOORD1; //light vector
    float3 vEye    : TEXCOORD2; //eye vector
    float3 vHalf   : TEXCOORD3; //half vector
};

// Constant registers
//-----------------------------
float4x4 mToWorld : register(c0);   //world space transposed
float4x4 mToClip  : register(c4);   //world * view * proj
float4 pLight     : register(c8);   //Light position (world space)
float4 pEye       : register(c9);   //Eye position (world space)

// function    : main
// description : vertex shader function
// return      : VS_OUTPUT
// param:
//    VS_INPUT input : vertex shader input
//-----------------------------
VS_OUTPUT main(const VS_INPUT input) {
    VS_OUTPUT output;

    //The following code outputs position and texture coordinates
    //-----------------------------
    output.vClipPos = mul(input.vPosition, mToClip);
    output.tcCoord = input.tcCoord;
    float4 pVertexWorld = mul(input.vPosition, mToWorld);

    //The following code generates tangent space base vectors
    //-----------------------------
    float3x3 mToTangent;
    mToTangent[0] = mul(input.vTangent, (float3x3)mToWorld);
    mToTangent[2] = mul(input.vNormal, (float3x3)mToWorld);
    mToTangent[1] = cross(mToTangent[0], mToTangent[2]);

    //Compute light, eye, and half vectors
    //-----------------------------
    float3 vToLight = normalize(pLight - pVertexWorld);
    vToLight = mul(mToTangent, vToLight);
    output.vLight = vToLight;

    float3 vToEye = normalize(pEye - pVertexWorld);
    vToEye = mul(mToTangent, vToEye);
    output.vEye = vToEye;

    output.vHalf = normalize(vToLight + vToEye);
    return output;
}
```

The following is the HLSL pixel shader for Cook-Torrance lighting.

```
// Used input structure
//-----------------------------
struct PS_INPUT {
    float2 tcCoord : TEXCOORD0; //input texture coordinates
    float3 vLight  : TEXCOORD1; //light vector
    float3 vEye    : TEXCOORD2; //eye vector
    float3 vHalf   : TEXCOORD3; //eye vector
};

// Used output structure
//-----------------------------
struct PS_OUTPUT {
    float4 vColor  : COLOR0;    //render target 0
};

// Used input texture samplers
//-----------------------------
sampler smplTexture : register(ps,s0);  //decal texture
sampler smplNormal  : register(ps,s1);  //normal texture

// function    : main
// description : pixel shader function
// return      : PS_OUTPUT
// param:
//    PS_INPUT input : pixel shader input - output from VS
//    float3 roughness : c2 - roughness
//    float3 refindex  : c3 - R(0) for Fresnel term
//-----------------------------
```

```
PS_OUTPUT main(const PS_INPUT input,
               uniform float roughness : register(c2),
               uniform float refindex : register(c3)) {

    PS_OUTPUT output;

    // Load and normalize input vectors
    //------------------------------
    float3 vNormal = tex2D(smplNormal, input.tcCoord).xyz;
    vNormal = normalize(2.0f * vNormal - 1.0f);
    float3 vLight = normalize(input.vLight);
    float3 vEye = normalize(input.vEye);
    float3 vHalf = normalize(input.vHalf);

    // Beckman's distribution function D
    //------------------------------
    float normalDotHalf = dot(vNormal, vHalf);
    float normalDotHalf2 = normalDotHalf * normalDotHalf;
    float roughness2 = roughness * roughness;
    float exponent = -(1-normalDotHalf2) / (normalDotHalf2*roughness2);
    float e = 2.71828182845904523536028747135f;
    float D = pow(e,exponent) /
              (roughness2*normalDotHalf2*normalDotHalf2);

    // Compute Fresnel term F
    //------------------------------
    float normalDotEye = dot(vNormal, vEye);
    float F = lerp(pow(1 - normalDotEye, 5), 1, refindex);

    // Compute self shadowing term G
    //------------------------------
    float normalDotLight = dot(vNormal, vLight);
    float X = 2.0f * normalDotHalf / dot(vEye, vHalf);
    float G = min(1, min(X * normalDotLight, X * normalDotEye));

    // Compute final Cook-Torrance specular term
    // Load texture, compute diffuse part, and combine it all to output
    //------------------------------
    float pi = 3.1415926535897932384626433832f;
    float CookTorrance = (D*F*G) / (normalDotEye * pi);

    float4 color = tex2D(smplTexture, input.tcCoord);
    float4 specular = color * max(0.0f, CookTorrance) * color.a;
    float4 diffuse = color * max(0.0f, normalDotLight);
    output.vColor = diffuse + specular;
    return output;
}
```

Quality Comparison

From the following figure it is clear that the older version cannot compete with the newer one in terms of quality. On the other hand, with low roughness and high refraction, index results are much better than with the Phong shading version 1.4. Also note that the lack of geometry attenuation factor in the pixel shader 1.4 version causes errors in the intensity computation for high roughness — areas that are supposed to be dark are brighter. This can be seen below.

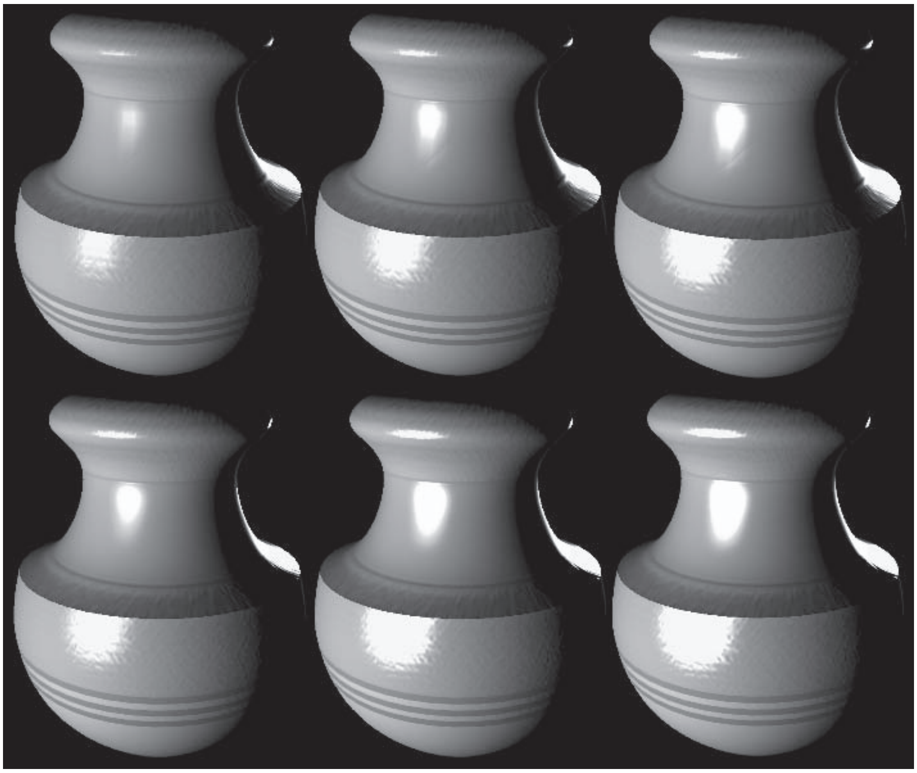


Figure 15: Rendering with various refraction index values with pixel shader 1.4 (top row) and pixel shader 2.0 (bottom row). Roughness is constant at 0.15. The index of refraction is 0.15, 0.45, and 0.85 (left to right). Note the visibility of the face edges and error (crack) in the middle of the large highlights in the 1.4 version. (See Color Plate 1.)

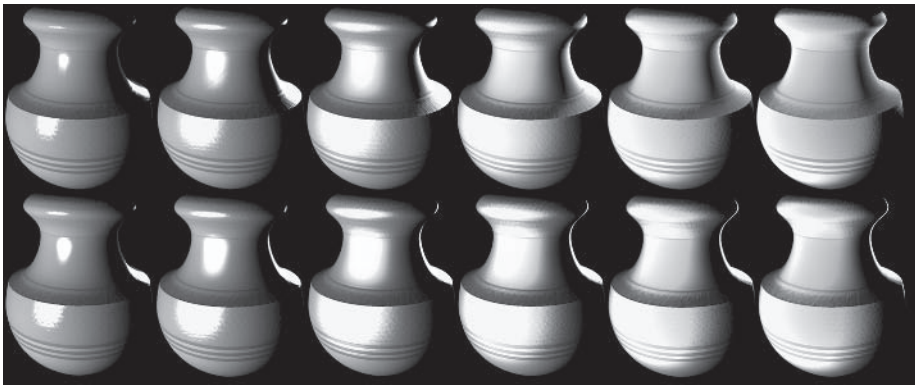


Figure 16: Rendering with various roughness values with pixel shader 1.4 (top row) and pixel shader 2.0 (bottom row). The refraction index is constantly 1.0 (full reflection). Roughness is 0.1, 0.2, 0.4, 0.6, 0.8, and 1.0 (left to right).

Conclusion

Previous examples showed that Direct3D 9 shaders are a great step toward more reality in games. What was not possible to put into a shader before (or was possible only with hacks and compromises) can now be done in full quality and in a single pass. The main advantage is floating-point precision in pixel shaders.

Vertex shaders 2.0 are great for games. With a huge instruction count and flow control, we can use one or two of them for a whole scene with features like per-vertex lighting and mesh skinning available via functions. Due to per-vertex lighting, there is no longer a need to switch the shader and do multipass rendering.

Pixel shader 2.0 architecture makes it possible to perform advanced lighting models per pixel. Sixty-four arithmetic and 32 texture instructions should be enough for most games to keep the number of required passes to the minimum. With shadows added into the scene, one light will still require at least one pass to render (the most common is two). The available instruction count can be used to render shadows for one light and some other per-pixel lights without shadows in one pass.

Shaders 3.0 are great in functionality, but to me the pixel shader is a bit unfinished because it lacks relative addressing of constant registers. With this enabled, an arbitrary number of lights could be computed per pixel (in world space, without shadow) in one pass thanks to loops (this can be done now with loop unrolling in design time, but it is not very handy). Also, a clear border could be defined between pixel and vertex shaders — vertex shaders for geometry transformation and pixel shaders for visualization. Today, with flow control and a huge instruction count, there can be one pixel shader for the whole scene with all required features available when needed.

Support for the shaders 2.x model in games is questionable. With capabilities changing from card to card, the game can use the shader's full potential only with some sort of run-time linker that merges fragments of prepared code, according to current device possibilities.

References

[1] Valient, M., "Project6 - Lighting," November 2002, http://www.dimension3.host.sk.

[2] Möller, T. and E. Haines, *Real-Time Rendering*, second edition, A.K. Peters, Natick, Massachusetts, 2002, http://www.realtime-rendering.com.

[3] Taylor, Philip, "Per-Pixel Lighting," MSDN article, November 2001, http://msdn.microsoft.com/directx.

[4] Kilgard, Mark J., "A Practical and Robust Bump-mapping Technique for Today's GPUs," nVidia Corporation paper, March 2000, http://developer.nvidia.com.

[5] Wloka, Matthias, "Fresnel Reflection Technical Report," nVidia Corporation paper, June 2002, http://developer.nvidia.com.

[6] Torrance, K. and E. Sparrow, "Theory for off-specular reflection from roughened surfaces," *Journal of the Optical Society of America*, September 1967, 57:1105-1114.

[7] Oren, Michael and Shree K. Nayar, "Generalization of Lambert's Reflectance Model," *Computer Graphics* (SIGGRAPH '94 Proceedings) 1994, pp. 239-246, http://www.cs.columbia.edu/~oren/.

[8] Oren, Michael and Shree K. Nayar, "Generalization of the Lambertian Model and Implications for Machine Vision," *International Journal of Computer Vision*, Vol. 14:3.

[9] Cook, Robert, L. Torrance, and E. Kenneth, "A reflectance model for computer graphics," *Computer Graphics* (SIGGRAPH '81 Proceedings) July 1981, Vol. 15:3, pp. 307-316.

Introduction to Different Fog Effects

Markus Nuebel
www.geocities.com/nuebelm

Introduction

Environmental fog is an effect that has been used in numerous games. Its popularity results from the fact that a great way to increase the realism of a scene is by adding an additional layer of ambience. This physical phenomenon itself is caused by dust and other particles suspended in the atmosphere, thereby scattering incoming light and emitting it back into the scene. Fog effects have become essential for depth culling, especially for large outdoor scenes, as it reduces the amount of popping caused by distant objects entering the scene at the far clipping plane.

Fog is a relatively cheap calculation, well supported by today's graphics hardware. It has become more and more a vital part of the gameplay. It allows creating different areas of visibility that can be easily incorporated into "line-of-sight" and "hiding" strategies.

In this article we start by taking a look at the standard fog effects: linear, exponential, and exponential squared fog. We create vertex and pixel shader equivalents for the fog calculations of the fixed-function pipeline (FFP). Although the shader versions are quite simple, they may help you develop your own variations and modifications of fog, which could result in the creation of some more interesting effects.

After dealing with these basic types, we take a look at layered fog. Layered fog is used to simulate ground fog and effects like smog, smoke, or clouds.

Finally we discuss a technique for real-time animated fog, which can be used to create non-uniform fog density over scenes and time.

All examples are based on DirectX but are applicable to other APIs as well.

The Theory behind Fog Calculations

The fog calculations are built around three main parameters:

- Fog color (C_{fog})
- Density (g)
- Distance (d)

Imagine a ray of light starting at a point in the scene and traveling toward the virtual camera. In a scene where no fog is present, the intensity of light along this ray is determined by the light intensity at the starting point.

In a foggy scene, the light intensity that reaches the virtual camera is influenced by absorption (light that is blocked by particles along the ray), out-scattering (light that is reflected toward other directions), and in-scattering (lights from other directions that are reflected along the ray). This influence is represented by the density factor g in our calculations.

Rays starting at points farther away from the camera have a far higher chance of getting influenced by fog particles than rays starting closer to the camera. So we have to take the distance d to the camera into account.

The final color, received at the camera and also written to the frame buffer, is determined by blending between the color of a scene point ($C_{current}$) and the color of the fog (C_{fog}). The used blend factor (*f*) has to be in the range [0.0, 1.0], meaning that a blend factor of 1.0 results in no fog influence (e.g., for points near to the camera), and a factor of 0.0 results in full fog influence (e.g., for points far in the distance).

Mathematically, this can be expressed by:

$$C_{final} = (f * C_{current}) + ((1-f) * C_{fog}))$$

...where C_{final} represents the final color written to the frame buffer.

Depending on the equation used to determine the factor *f*, various results can be achieved, both visually and with respect to performance issues. These also apply to determining the fog color C_{fog}. It can be constant for all rendered points or determined by additional calculations.

There are two commonly used approaches for determining the distance factor (*d*): plane-based and range-based.

- **Plane-based:** The camera space z-depth of a vertex/pixel is used as the distance factor. It is very cheap.
- **Range-based:** The exact distance between a vertex/pixel and the virtual camera is calculated. This can be done in either world space or camera space. It is more expensive, since this usually requires the evaluation of a square root.

Both ways of choosing *d* are used in the techniques described in the following sections.

Technique One: Linear Fog

Figure 1 shows a screen shot of linear fog applied to a terrain scene.

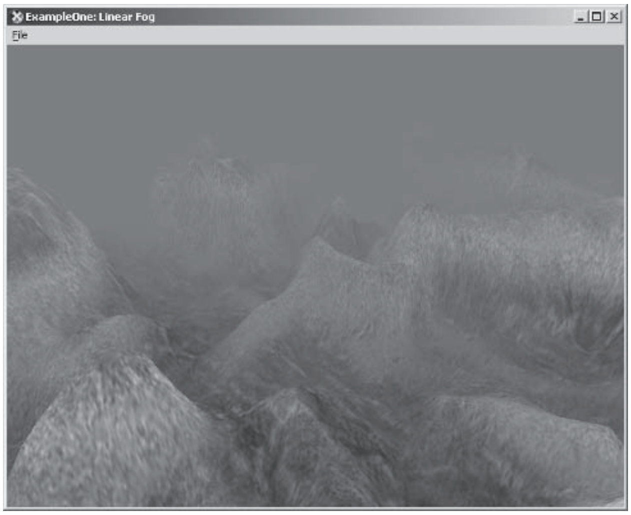


Figure 1: Linear fog (See Color Plate 2.)

Fog Equation

$$f = \frac{(Z_{FogEnd} - Z_{Depth})}{(Z_{FogEnd} - Z_{FogStart})}$$

Linear fog is the simplest calculation of all discussed methods. It assumes an equal distribution of fog particles throughout the scene.

Generally, you specify a starting value ($Z_{FogStart}$) and an ending value (Z_{FogEnd}), which control the way fog is applied to the scene.

For vertices/pixels that are between these boundaries, a blend value is obtained by a linear interpolation. The amount of fog present in the scene constantly increases with the distance to the virtual camera.

Implementation

Shader

For the implementation of linear fog, a vertex shader is used. The shader calculates the fog value according to the formula mentioned above and puts the result into the FOG register of the vertex shader ALU. This register is used by the FFP in the fog blending that takes place after the pixel processing of the graphics pipeline.

The complete vertex shader code follows:

```
VS_OUTPUT main(const VS_INPUT Input)
{
     float4        clpPos, camPos;

     // Init  output
     VS_OUTPUT     Out = (VS_OUTPUT) 0;

     // Retrieve fog parameters
     float fFogEnd     = fFog.x;
     float fFogStart   = fFog.y;

     // Calculate the clip space position
     Out.Position      = mul(Input.Position, matWorldViewProj);

     // Simply pass on the texture coords and the vertex color
     Out.Tex0.xy       = Input.Tex0.xy;
     Out.Diffuse       = Input.Diffuse;

     // Calculate vertex position in camera space
     camPos = mul(Input.Position, matWorldView);

     // Calculate the linear fog factor
     float fFogRange   = fFogEnd-fFogStart;
     float fVertexDist = fFogEnd - camPos.z;
     float f           = clamp((fVertexDist/fFogRange), 0.0f, 1.0f);

     //  Write the calculated factor to the FOG register
     Out.Fog = f;
```

```
    return Out;
}
```

Let's have a look at the code parts that are relevant for fog calculations.

The application supplies the shader with the settings for $Z_{FogStart}$ and Z_{FogEnd} by filling the first two components of the shader constant `fFog`. For better readability, these values are extracted and placed in variables.

```
float fFogEnd    = fFog.x;
float fFogStart  = fFog.y;
```

The linear fog interpolation is done by dividing the difference between the far fog end and the camera space z coordinate of a vertex ($Z_{FogEnd} - Z_{Depth}$) by the range over which fog is applied ($Z_{FogEnd} - Z_{FogStart}$).

```
float fFogRange    = fFogEnd-fFogStart;
float fVertexDist  = fFogEnd - camPos.z;
float f            = clamp((fVertexDist/fFogRange), 0.0f, 1.0f);
```

This calculation produces *f* values of 1.0 for vertices that are closer to the camera than $Z_{FogStart}$ and values of 0.0 for vertices with distance Z_{FogEnd} or greater.

The result is clamped to the range [0.0, 1.0] to stick to the conventions of the FOG register usage of the FFP. *f* is used in the fog blending phase of the FFP to mix between fragment color and fog color.

As you might have seen, the calculation above uses the plane-based approach in determining the distance by simply evaluating the camera space z coordinate of a scene point.

For a very obtuse field of view, this plane-based calculation may produce some visual anomalies when using a rotating camera. In this case, the problem is that distant objects may enter and leave the fog range during minor camera movements. Normally, this works quite well but depends on the special setting of an application.

We see the more accurate distance calculation later on when we discuss exponential squared fog.

Application Settings

There is indeed not much to say about the application that drives this shader — besides the fact that the fog calculation of the FFP has to be enabled prior to rendering the scene. This is necessary because we want to make the FFP use the value calculated by our vertex shader.

```
m_pd3dDevice->SetRenderState(D3DRS_FOGCOLOR, D3DCOLOR_XRGB(100,100,100));
m_pd3dDevice->SetRenderState(D3DRS_FOGENABLE,TRUE);
...
m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, m_nPolyCount);
...
m_pd3dDevice->SetRenderState(D3DRS_FOGENABLE,FALSE);
```

In addition to these settings, the combined world-view-projection matrix as well as a vector with fog parameters is provided to the shader.

```
m_pd3dDevice->SetVertexShaderConstantF(0, (float*)&m_matWorldViewProj, 4);
m_pd3dDevice->SetVertexShaderConstantF(10,  (float*)&fFog, 1);
```

Technique Two: Exponential Fog

Figure 2 shows a screen shot of exponential fog applied to terrain.

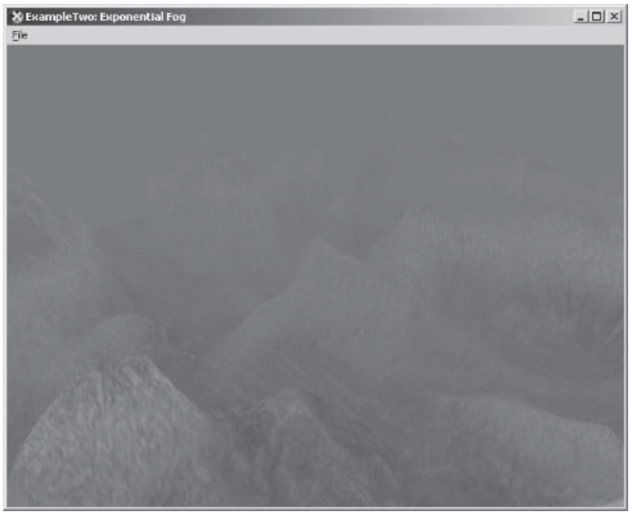


Figure 2: Exponential fog

Fog Equation

$$f = \frac{1}{e^{(d*g)}} = e^{-(d*g)} = e^{-(distance*density)}$$

e = base of natural logarithms (≈ 2.71828)

Exponential fog calculates the fog factor *f* based on the formula above. As you can see in Figure 3, the effect is a more rapid decrease in density than seen for linear fog.

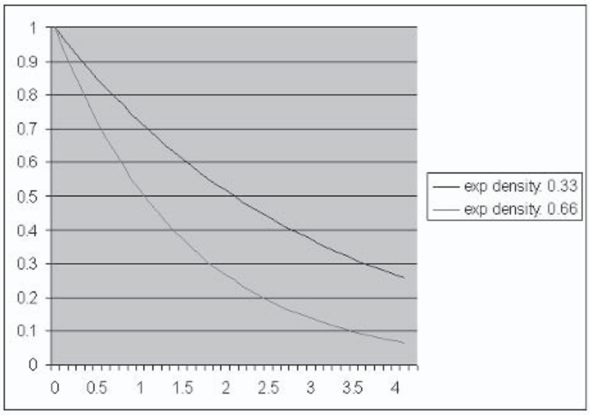


Figure 3: Exponential function for different densities

When we recall the fact that fog is caused by a reduction of light intensity along a ray from a point in the scene toward the virtual camera, we can see that exponential fog very much corresponds to the real-world phenomenon. In this simplified model, we assume a constant fog density over the scene. Think of light from a scene point traveling along a ray in multiple steps, where each step has a predefined unit length. By the time the light has crossed the first unit along this ray, its intensity is reduced by a constant factor (determined by the chosen density). The next unit starts with a reduced light intensity, which again is reduced by the same constant reduction factor applied to the previous ray.

When choosing very small segments, we end up with an exponential behavior that is well simulated by the use of the e-function.

Implementation

This example no longer relies on the fog blending of the FFP but uses a vertex shader to compute the fog factor and a pixel shader to do the blending of fragment color with fog color.

As with the linear fog implementation, we use the camera space z-depth as a distance factor. In contrast, the DirectX FFP determines the distance factor by using either the Z or the W value of a vertex. For more information on this implementation, please refer to [Rogers].

Shader

The complete vertex shader code follows:

```
VS_OUTPUT main(const VS_INPUT Input)
{
     float4        clpPos, camPos;
     // Init output
     VS_OUTPUT     Out = (VS_OUTPUT) 0;

     // Calculate the clip space position
     clpPos         = mul(Input.Position, matWorldViewProj);
     Out.Position   = clpPos;

     // Simply pass on the texture coords and the vertex color
     Out.Tex0.xy    = Input.Tex0.xy;
     Out.Diffuse    = Input.Diffuse;

     // Calculate vertex position in camera space
     camPos         = mul(Input.Position, matWorldView);

     // Extract the fog parameters
     float fDensity = fFog.x;
     float fFogEnd  = fFog.y;

     // Calculate the distance.
     // Camera space z coords scaled to have a value of 4 at distance: FogEnd
     float fDist   = camPos.z/fFogEnd*4;

     // Exp calculation
     float f       = exp(-fDist*fDensity);
```

```
    // Set the fog value
    Out.FogVal.x  = f;    // Passed to pixel shader using color register

    return Out;
}
```

The parameters of the shader are density *g* and fog end distance Z_{FogEnd}. A fog starting distance is not needed here, since fog blending starts immediately at the virtual camera. Both parameters are provided by the application in constant registers and assigned to some temporary variables using the following instructions:

```
    float fDensity = fFog.x;
    float fFogEnd  = fFog.y;
```

When you take a look at Figure 3, you can see that the function quickly decreases to 0 with increasing input values. For input values around 4.0, the function is already close to 0. We can use this fact to scale the camera space z-depth in a way that results in a value of 4.0 at distance Z_{FogEnd}.

```
    float fDist  = camPos.z/fFogEnd*4;
```

To calculate the value of an exponential function, the `exp()` library function is used on the product of scaled distance and density. The way in which the distance scaling is chosen ensures a fog factor of nearly 0.0 for vertices with distance Z_{FogEnd} and of nearly 1.0 for vertices near the camera.

```
    float f = exp(-fDist*fDensity);
```

The last thing we have to do is store the result in an output register so that it becomes accessible to a pixel shader.

```
    Out.FogVal.x  = f;    // Passed to PixelShader using color register
```

The variable `Out.Fog` is bound to the COLOR1 register of the vertex processing ALU, which is not used by any color value (e.g., specular) in this shader.

The pixel shader used in this example is very simple. It looks up the base texture of the fragment and does the blending, according to the fog value provided by the previous vertex program.

The complete fragment program follows:

```
PS_OUTPUT main(const PS_INPUT Input,   uniform sampler2D     baseTexture)
{
     // Init output
     PS_OUTPUT     Out = (PS_OUTPUT)  0;

     // Retrieve base texture
     float4 colorBase = tex2D(baseTexture, Input.Tex0);

     // Fog blending
     float  f      = Input.FogVal.x;
     Out.Color     = lerp(colorFog, colorBase*Input.Diffuse.xyzz, f);

     return Out;
}
```

The last two statements are the only ones of interest for our fog examination:

```
     float  f      = Input.FogVal.x;
     Out.Color     = lerp(colorFog, colorBase*Input.Diffuse.xyzz, f);
```

First, we extract the interpolated fog value from the COLOR1 register, which was prepared by the vertex program.

Then a `lerp` is performed between the fog color and the base color. Since we are no longer using the FFP for fog blending, the fog color is provided as constant input to the shader. It is no longer necessary to set the fog color as a renderstate. To enhance the visual output, the base color, which has been looked up from a texture, is modulated by the diffuse color of a fragment before the final blending takes place.

Application Settings

As we are not using the FFP for blending fog and fragment color, this part of the pipeline can be disabled for the whole application.

```
m_pd3dDevice->SetRenderState(D3DRS_FOGENABLE,FALSE);
```

The vertex shader is provided with the combined world-view-projection matrix and a vector containing fog parameters.

The pixel shader, on the other hand, needs the fog color that is used during blending.

```
m_pd3dDevice->SetVertexShaderConstantF(0, (float*)&m_matWorldViewProj, 4);
m_pd3dDevice->SetVertexShaderConstantF(10, (float*)&fFog, 1);
m_pd3dDevice->SetPixelShaderConstantF(5, (float*)&colFog, 1);
```

Technique Three: Exponential Squared Fog

Figure 4 shows a screen shot of exponential squared fog applied to a terrain scene.

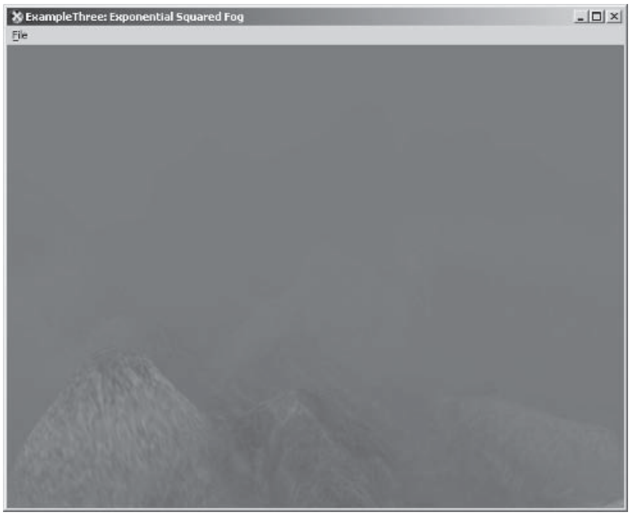


Figure 4: Exponential squared fog (See Color Plate 3.)

Fog Equation

$$f = \frac{1}{e^{(d*g)^2}} = e^{-(d*g)^2} = e^{-(distance*density)^2}$$

e = base of natural logarithms (≈ 2.71828)

Exponential squared fog is similar to the exponential fog that we discussed above. However, when looking at Figure 5, you can see that, compared with the exponential function, the squared function has a flatter slope for near distances followed by a steeper slope later on.

Looking at an application, this leads to a wider range of totally clear visibility surrounding the virtual camera, followed by an intense density increase toward Z_{FogEnd}.

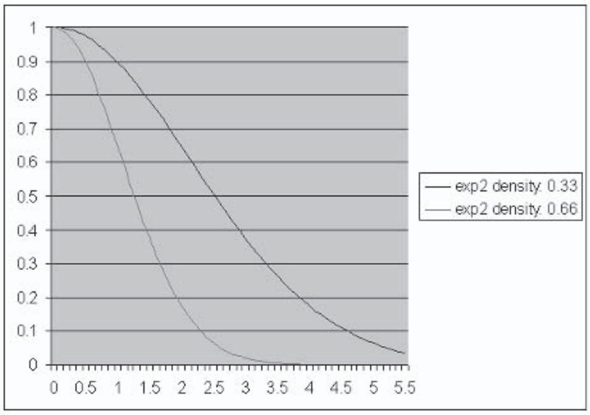


Figure 5: Exponential squared functions for different densities

On current generation graphics hardware, the slightly more complex exponent calculation should not produce a big performance hit, but you may want to keep this in mind when writing shaders for older boards, where a difference may be noticeable.

Implementation

Of course, the implementation is quite similar to the exponential fog version, but there are two main differences to point out.

In the first place, we do not use the camera space z-depth of a vertex as the distance factor. Secondly, the factor calculation is changed to include the additional multiplication.

Using the camera space z-depth as the distance factor may produce some artifacts because all vertices that are lying on the same z-plane are getting assigned the same distance factor. This is an approximation for the real distance and is in fact not correct. Taking the z-depth as distance is only right for vertices directly in front of the camera. As vertices move toward the outer screen regions, the error of this computation increases more and more. Depending on your application settings, this may not be noticeable, but for extremely wide fields of view and areas with a fast increasing slope of fog density (especially for exponential squared fog), this could cause some problems.

To circumvent this behavior, the exact distance of every vertex to the virtual camera should be calculated. This is what we will do in the implementation of this section's shader.

Shader

The complete vertex shader code follows:

```
VS_OUTPUT main(const VS_INPUT Input)
{
    float4          clpPos, worldPos;

    // Init output
    VS_OUTPUT    Out = (VS_OUTPUT) 0;

    // Calculate the clip space position
    clpPos           = mul(Input.Position, matWorldViewProj);
    Out.Position     = clpPos;

    // Simply pass on the texture coords and the vertex color
    Out.Tex0.xy      = Input.Tex0.xy;
    Out.Diffuse      = Input.Diffuse;
    // Extract the fog parameters
    float fDensity   = fFog.x;
    float fFogEnd    = fFog.y;

    // Calculate the vertex position in world space
    worldPos         = mul(Input.Position, matWorld);

    // Calculate the distance to the viewer in world space
    float fDistance  = distance(worldPos, vCamera);
    // The distance is scaled to have a value of 4 at distance: FogEnd
    float fDist      = fDistance/fFogEnd*4;

    // Exp squared calculation
    float fFog       = exp(-(fDist*fDensity)*(fDist*fDensity));

    // Set the fog value
    Out.FogVal.x     = fFog;    // Passed to pixel shader using color register

    return Out;
}
```

This time the distance factor is computed by calculating the exact distance between a vertex and the camera in world space. To get the world space position of a vertex, it is multiplied by the application's world matrix supplied in some of the constant registers. This is similar to the one already performed to transform the vertex into clip space, but the difference is that the world matrix is used instead of the combined world view matrix.

```
camPos         = mul(Input.Position, matWorldView);
```

Now, as we have computed the world space vertex position, we simply use the standard library function `distance()` to determine the exact distance between a vertex location and the camera. The coordinates of the virtual camera are supplied to the shader within the register bound to the `vCamera` variable.

```
float fDistance    = distance(worldPos, vCamera);
float fDist        = fDistance/fFogEnd*4;
```

As seen in the last section, the distance is scaled to reach a value of 4.0 at the distance Z_{FogEnd}.

The fragment program is identical to the one used in the last section, so please refer back to the exponential fog implementation for details.

Application Settings

Application settings are also very similar to those used by the exponential fog implementation in the last section. However, because of the changed distance computation, we have to provide the shader with the necessary world matrix and the position of the virtual camera.

```
m_pd3dDevice->SetVertexShaderConstantF(8, (float*)&m_matWorld, 4);
m_pd3dDevice->SetVertexShaderConstantF(12, (float*)&fFog, 1);
m_pd3dDevice->SetVertexShaderConstantF(13, (float*)&vCamera, 1);
```

Technique Four: Layered Fog

Figure 6 shows a screen shot of layered fog applied to a terrain scene.

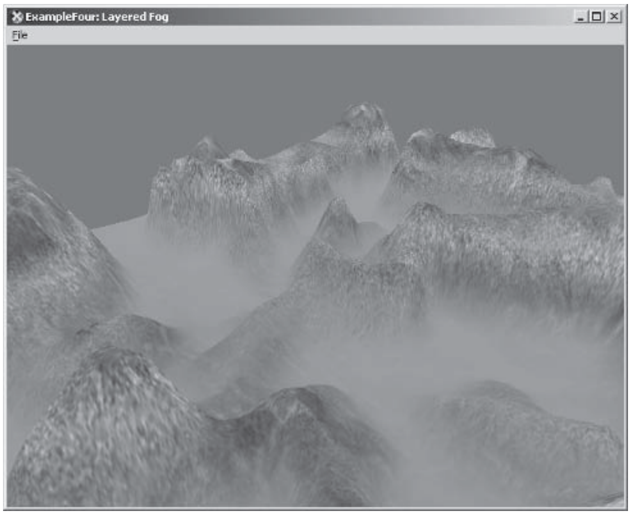


Figure 6: Layered fog (See Color Plate 4.)

Theory and Equations

In this section we take a look at layered fog, a simple kind of volumetric fog effect. The theory behind our implementation is based on a paper called "Fast Multi-Layer Fog." Please refer to [Legakis] for more details.

The idea behind layered fog is to extend the equations to include not only a function of distance but also a function of height.

This leads us to four important variables: Y_C, the height of the virtual camera, Y_P, the height of the scene point, X_C, the x coordinate of the camera, and X_P, the x coordinate of the scene point. See Figure 7.

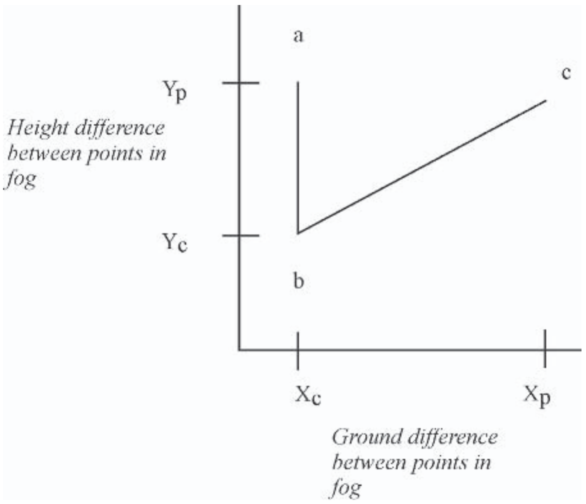

a
c
Yp
Height difference between points in fog
Yc
b
Xc
Xp
Ground difference between points in fog

Figure 7: Height and distance relation between virtual camera (a) and scene point (c).

The height difference ΔY is computed by subtraction of the y coordinates of the two points.

$$\Delta Y = abs(Y_C - Y_P)$$

The distance ΔD along the x-z plane is computed by:

$$\Delta D = \sqrt{(X_C - X_P)^2 + (Z_C - Z_S)^2}$$

What is left for our layered fog model is the computation of the fog density at various scene points. Modeling complex fog usually requires the integration of density along a line from the virtual camera to a point in the scene (e.g., $density(a,c)=\int_a^c density(t)dt$, where *density(a,c)* is the total density used in the computation of the fog factor for the line between the camera (c) and a scene point (a) and where *density(t)* defines the density at height t).

This is necessary because realistic fog normally does not have a constant density distribution. However, to simplify things, our implementation assumes a constant density increase for an increasing height difference between points inside the fog area.

This assumption allows us to choose $0.5*\Delta Y^2$ as the density integral.

Now we have gathered all the information needed to define a suitable density function:

$$density(a,c)=\sqrt{1+\left(\frac{\Delta D}{\Delta Y}\right)^2}\int_a^c density(t)dt=\sqrt{1+\left(\frac{\Delta D}{\Delta Y}\right)^2}\,\frac{(\Delta Y)^2}{2}$$

for $\Delta Y \neq 0$

$density(a,c) = 0$ for $\Delta Y=0$

Implementation

With all the theory in place, the implementation is a straightforward translation of the formulas described above into shader code. As with the previous techniques, we use a vertex shader to output the fog value at a vertex and a pixel shader that does the blending with the base texture.

Shader

The complete vertex program follows:

```
VS_OUTPUT main(const VS_INPUT Input)
{
    float4        clpPos, camPos, worldPos;
    float         fDistance;
```

```
// Init output
VS_OUTPUT    Out = (VS_OUTPUT) 0;

// Calculate the clip space position
clpPos           = mul(Input.Position, matWorldViewProj);
Out.Position     = clpPos;

// Simply pass on the texture coords and the vertex color
Out.Tex0.xy      = Input.Tex0.xy;
Out.Diffuse      = Input.Diffuse;

// Get fog parameter
float fFogTop    = fFog.x;
float fFogEnd    = fFog.y;
float fFogRange  = fFog.x;

// Calculate the world position
worldPos        = mul(Input.Position, matWorld);

// Calculate the distance to the viewer
fDistance       = distance(worldPos, vCamera);

// Project both points into the x-z plane
float4           vCameraProj, vWorldProj;
vCameraProj    = vCamera;
vCameraProj.y  = 0;
vWorldProj     = worldPos;
vWorldProj.y   = 0;

// Scaled distance calculation in x-z plane
float fDeltaD = distance(vCameraProj, vWorldProj)/fFogEnd*2.0f;

// Height-based calculations
float fDeltaY, fDensityIntegral ;
if(vCamera.y > fFogTop)
{
     if (worldPos.y < fFogTop)
     {
          fDeltaY = (fFogTop - worldPos.y)/fFogRange*2;
          fDensityIntegral = (fDeltaY * fDeltaY * 0.5f);
     }
     else
     {
            fDeltaY = 0.0f;
            fDensityIntegral = 0.0f;
        }
    }
    else
    {
        if (worldPos.y < fFogTop)
        {
            float fDeltaA = (fFogTop - vCamera.y)/fFogRange*2;
            float fDeltaB = (fFogTop - worldPos.y)/fFogRange*2;
            fDeltaY =abs(fDeltaA -fDeltaB);
            fDensityIntegral = abs((fDeltaA * fDeltaA * 0.5f) - (fDeltaB *
            fDeltaB * 0.5f));
        }
        else
        {
            fDeltaY = abs(fFogTop - vCamera.y)/fFogRange*2;
            fDensityIntegral = abs(fDeltaY * fDeltaY * 0.5f);
        }
    }

    float fDensity;
    if (fDeltaY != 0.0f)
    {
        fDensity = (sqrt(1.0f + ((fDeltaD / fDeltaY) * (fDeltaD / fDeltaY))))
        * fDensityIntegral;
    }
    else
    {
        fDensity = 0.0f;
    }

    float f= exp(-fDensity);

    // Set the fog value
    Out.FogVal.x  = f; // Passed to PixelShader using color register

    return Out;
}
```

Let's revisit the fog-related stuff, step by step.

The application supplies the shader with the parameter settings for the top of the fog, which is the height above the ground up to where we want ground mist to last. The second parameter is Z_{FogEnd}, which is used to determine the distance where scene points should be completely encompassed by the fog. We are also using a temporary variable called FogRange, which in our case is the same as the top of the fog because this section's ground fog implementation starts at height zero.

```
    float fFogTop    = fFog.x;
    float fFogEnd    = fFog.y;
    float fFogRange  = fFog.x;
```

To compute the distance between a point's world space position and the virtual camera in the x-z plane, both points are projected onto this plane by setting their y coordinate to 0.

```
    vCameraProj    = vCamera;
    vCameraProj.y  = 0;
    vWorldProj     = worldPos;
    vWorldProj.y   = 0;
```

Getting the distance between the two points is a matter of calling the library function `distance()`. As you have seen in the "Theory and Equations" section above, the distance ΔD contributes to the exponent of the exponential function. To get a desirable result, this factor, along with other factors that influence the exponent (e.g., ΔY), have to undergo some application-specific scaling. The scaling itself is influenced by the dimensions of the underlying world coordinate system. In our case, ΔD is scaled to have a value of 2.0 for points at distance Z_{FogEnd}.

```
    float fDeltaD  = distance(vCameraProj, vWorldProj)/fFogEnd*2.0f;
```

Regarding the calculation of ΔY and the density to use, we have to distinguish several cases. Thanks to the latest addition of control flow to shader languages, this is easily done using if-then-else statements.

First of all, we have to distinguish if the virtual camera is above or beneath the top of the fog. Let's start with the case that the camera is above the fog top.

Next we need to find out if the vertex that is currently being processed is above or below the fog top. If it is above the fog top, we do not want to apply fog to this vertex, and we simply set ΔY and density to 0.

If, on the other hand, the vertex is below the fog, we want to apply fog. According to our theory, ΔY is calculated as the difference between the top of the fog and the vertex height. This should give us more fog for vertices that reside closer to the ground and less fog for vertices that have just entered the fog area. As with ΔD, we apply our application-defined scaling to get proper results.

Afterward we are ready to determine the density with the formula $\frac{\Delta Y^2}{2}$.

```
        if (worldPos.y < fFogTop)
        {
             fDeltaY = (fFogTop - worldPos.y)/fFogRange*2;
             fDensityIntegral = (fDeltaY * fDeltaY * 0.5f);
        }
        else
        {
             fDeltaY = 0.0f;
             fDensityIntegral = 0.0f;
        }
```

With the second case, the virtual camera is located inside the fog, which means its height is between the ground (height zero) and the top of the fog.

For each processed vertex, we have to make the distinction of whether the vertex is above or below the fog top. Being above the fog top means that the camera is looking toward a point in a non-foggy area. So ΔY is computed as the difference between the top of the fog and the camera height.

In case the processed vertex is underneath the fog top, we have to calculate two densities. The first is computed for the line between the eye and the top of the fog, and the second is computed for the line between the vertex height and the top of the fog. The final value results from the difference between these two densities.

```
    if (worldPos.y < fFogTop)
    {
        float fDeltaA = (fFogTop - vCamera.y)/fFogRange*2;
        float fDeltaB = (fFogTop - worldPos.y)/fFogRange*2;
        fDeltaY =abs(fDeltaA -fDeltaB);
        fDensityIntegral = abs((fDeltaA * fDeltaA * 0.5f) - (fDeltaB *
        fDeltaB * 0.5f));
    }
    else
    {
        fDeltaY = abs(fFogTop - vCamera.y)/fFogRange*2;
        fDensityIntegral = abs(fDeltaY * fDeltaY * 0.5f);
    }
```

Now all terms have been computed that contribute to the density calculation. However, the case for ΔY being zero has to be handled separately, as we otherwise run into a division by zero error. The solution is to set the density to 0 in this case:

```
if (fDeltaY != 0.0f)
    fDensity = (sqrt(1.0f + ((fDeltaD / fDeltaY) * (fDeltaD / fDeltaY)))) *
    fDensityIntegral;
else
    fDensity = 0.0f;
```

With the last two lines, the fog value to be used by the pixel shader is computed using the library function `exp()` and handed over inside the COLOR1 register.

```
    float f = exp(-fDensity);

    Out.FogVal.x  = f; // Passed to pixel shader using color register
```

Application Settings

As seen in the other sections, the application provides the shader with the necessary settings for fFogTop, fFogEnd, fFogRange, and C_{Fog} in addition to the usual matrices and camera information.

Technique Five: Animated Fog

Figure 8 shows three screen shots of layered fog applied to a terrain scene. You should notice that the density of fog changes over the scene and it changes from screen shot to screen shot, which means that it varies over time.

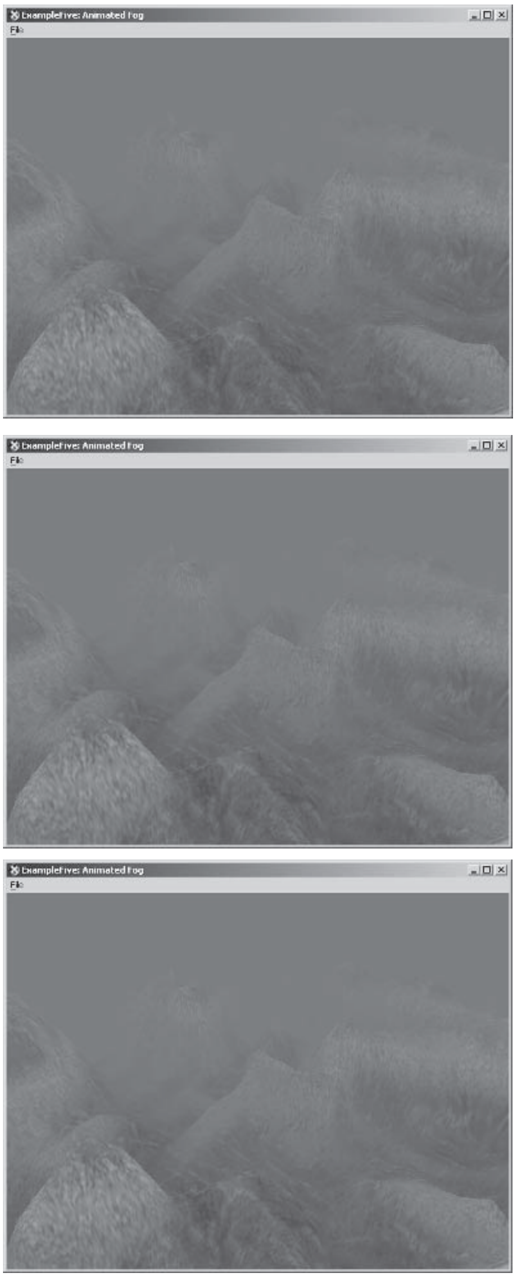

ExampleFive: Animated Fog
File
ExampleFive: Animated Fog
File
ExampleFive: Animated Fog
File

Figure 8: A series of screen shots for animated fog (See Color Plates 2, 3, and 4.)

Theory and Equations

The fog effect discussed in this section is based on the publication "Real-Time Animation of Realistic Fog," Biri, et al. Since the theory behind this technique extends the scope of this article, we only take a brief look at it and concentrate on the implementation. For additional information, please refer to [Biri].

The main idea behind animated fog is to approximate the different terms of fog equations, like absorption, scattering, and emission, by periodic function. Using a weighted sum of these functions and applying them over the complete scene results in a non-uniform fog distribution.

With the help of periodic functions, it is easy to add a kind of animation that can simulate the influence of wind to the fog. Modification of the animation settings even allows simulating changes in wind speed and direction.

To calculate the fog factor, we use the following formula:

$$f = \exp(-K(x,y,z)) = \exp\left(-\sum_{i=1}^{N} \chi_i(x,y,z) \right)$$

…where $K(x,y,z)$ is the sum of all periodic functions applied over the scene and χ_i represents a single periodic function. The input to the `exp()` function has to be negative; otherwise the fog calculation will not work.

According to [Biri], $K(x,y,z)$ is chosen as:

$K(x,y,z) =$

$$1 + \frac{1}{2}\cos(5\prod y) + \frac{1}{5}\cos(7\prod(y + 0.1x)) + \frac{1}{5}\cos(5\prod(y - 0.1x)) + \frac{1}{10}\cos(\prod x)\cos\left(\frac{\prod z}{2}\right)$$

Using world coordinates as input to this function gives us a very nice varying fog distribution over the whole scene. This works quite well for a static camera but results in some problems for a moving one. The periodic distribution caused by the trigonometric functions is very apparent and easy to notice.

To make this technique more applicable to a moving camera, we use the calculated factor $K(x,y,z)$ along with a normal exponential fog calculation.

This leads to the following fog calculation:

$$f = \exp(-dg) + K(x,y,z)$$

Up until now, we have achieved a non-uniform fog density of the scene, which seamlessly integrates with exponential fog. Points near the camera are less foggy than distant ones, even when moving the camera.

Animating the fog is nothing more than simply adding a time varying term to the density calculation. Depending on which term of $K(x,y,z)$ you choose to modify with a varying factor, you can control the direction of the fog animation and therefore the direction of simulated wind.

Of course, you can control the turbulence of the fog by choosing a constantly increasing animation term or by using a function to compute such a "wind factor."

Implementation

The implementation only uses a vertex shader to compensate for the performance hit of making extensive use of trigonometric functions. The achieved results are normally quite sufficient and do not justify calculations for every fragment.

Shader

The complete vertex shader code follows:

```
VS_OUTPUT main(const VS_INPUT Input)
{
    float4         clpPos;

    // Init output
    VS_OUTPUT      Out = (VS_OUTPUT) 0;

    // Calculate the clip space position
    clpPos         = mul(Input.Position, matWorldViewProj);
    Out.Position    = clpPos;

    // Simply pass on the texture coords and the vertex color
    Out.Tex0.xy      = Input.Tex0.xy;
    Out.Diffuse      = Input.Diffuse;

    // Get fog parameter
    float  fAnim     = fFog.x;
    float  fFogEnd   = fFog.y;
    float  fDensity  = fFog.z;

    // Calculate the distance. (Same as for exp-fog)
    // Camera space z coords scaled to have a value of 4 at distance FogEnd
    float4  camPos   = mul(Input.Position, matWorldView);
    float   fDist    = camPos.z/fFogEnd*4;

    // Exp fog calculation
    float fExpFog    = exp(-fDist*fDensity);

    // Animation is calculated based on world coordinates
    float4 worldPos  = mul(Input.Position, matWorld);

    // Do the animation: -(1+0.5*cos(5*PI*z)+0.2*cos(7Pi*(z+0.1*x))+0.2*cos
    //(5*PI*(z-0.05*x))+0.1*cos(PI+x)*cos(PI*y/2))
    float k = -1-0.5*cos(5*3.14*worldPos.z+fAnim)-0.2*cos
    (7*3.14*(worldPos.z+0.1*worldPos.x))-0.2*cos
    (5*3.14*(worldPos.z-0.05*worldPos.x))-0.1*cos
    (3.14*worldPos.x)*cos(3.14*worldPos.y/2);

    // Final fog is addition of exp and animation
    float f = fExpFog + (camPos.z/fFogEnd)/4.0f*k;

    // Set the fog value
    Out.Fog = f;

    return Out;
}
```

Most of the shader code should be familiar by now, so let's have a look at the interesting parts.

The shader uses the normal settings for fog end and density in its exponential fog calculation. The time varying term `fAnim` is used to modify *K*(*x,y,z*) and therefore animates the fog density over time.

```
    float  fAnim        = fFog.x;
    float  fFogEnd      = fFog.y;
    float  fDensity     = fFog.z;
```

The *K*(*x,y,z*) calculation is a direct translation of the equation discussed above into code. Notice that the time varying term `fAnim` is added to the first term, modifying the world space z coordinates. This achieves the effect of wind blowing along the world z-axis.

```
float k = -1-0.5*cos(5*3.14*worldPos.z+fAnim)-0.2*cos
(7*3.14*(worldPos.z+0.1*worldPos.x))-0.2*cos
(5*3.14*(worldPos.z-0.05*worldPos.x))-0.1*cos
(3.14*worldPos.x)*cos(3.14*worldPos.y/2);
```

To make the fog applicable for a moving camera, it is combined with an exponential fog factor. Notice that we again need to apply some application-dependent scaling to *K*(*x,y,z*) in order to achieve desirable results.

```
    // Final fog is addition of exp and animation
    float f = fExpFog + (camPos.z/fFogEnd)/4.0f*k;
```

Application Settings

Besides the usual parameters, the shader is provided with a value for the fog density and fog end, used by the `exp` fog calculation. Additionally, a single float value is passed to the shader that is constantly increased from frame to frame in order to achieve the discussed animation.

Conclusion

With our discussion of basic environmental fog, simple volumetric fog, and more interesting animated fog, we have seen much of what can be accomplished by adding fog to an application. For sure, each application will have different requirements and limitations regarding the usage of fog, but with the ground covered above, it should be easy to adapt and combine effects to achieve some unseen effects.

However, some closing notes must be mentioned here.

All of the pixel shaders used are very simple and do nothing more than a blend operation. Moving the fog calculation from the vertex shader to the pixel shader leads to more accurate results. But it must be decided on a case-by-case basis whether the additional computing cost (caused by doing computation per fragment instead of per vertex) is worth it. Generally, you will not notice a big difference because fog mostly deals with uniform colors. As mentioned, this is a decision to be made with your specific application requirements in mind.

Another point is the usage of constant terms in the discussed shaders. To have a direct connection to the equations that are defining a fog technique, most of the shaders are calculating constant terms inside the shader itself. For a real-world application, this should (of course) not be the case. Precomputing constant terms on the CPU and providing them to the GPU once per frame is far more efficient than doing repetitive calculations for each execution of the shader program.

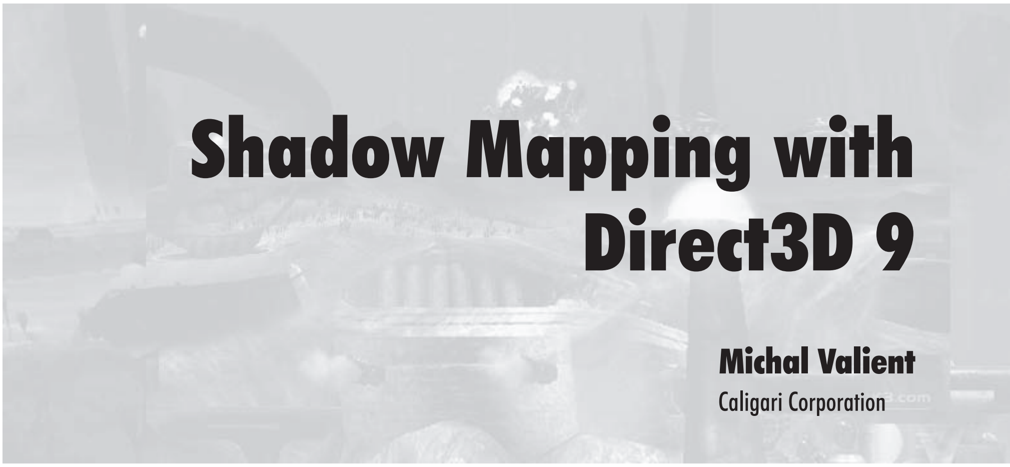
The last point that needs to be mentioned is the handling of fog color. All of our shaders use a constant fog color, but this is not a requirement. Doing additional calculations to determine the fog color for a vertex or fragment can result in some really interesting and exciting effects that can give your application the final touch.

References

[Biri] Biri, V., S. Micheling, and D.Arqués, “Real-Time Animation of Realistic Fog,” Thirteenth Eurographics Workshop on Rendering, 2002.

[Legakis] Legakis, J., “Fast Multi-Layer Fog,” ACM SIGGRAPH ’98 Conference Abstracts and Applications, p. 266.

[Rogers] Rogers, D., “Z-buffering, Interpolation and More W-buffering,” nVidia Corp., www.nvidia.com/developer.



Shadow Mapping with Direct3D 9

Michal Valient
Caligari Corporation

Introduction

Because we are using monitors to display information, the whole computer-generated world is turned into 2D before we see it. Then, only our brain (with great help from our memory and past experiences) can restore the feeling of three dimensions. Shadows are one of the most important guidelines in this process because they give us information about the position of objects in a scene.

This chapter covers the implementation of the shadow map algorithm with Direct3D 9 and offers some improvements. First we fight depth compare errors by using a back-faced shadow map; we store depths only for pixels facing away from the light. We improve standard percentage closer filtering (PCF) by additional bilinear filtering performed in the pixel shader. The result is a highly optimized pixel shader that uses 64 arithmetic instructions to produce smooth shadows with reduced aliasing problems even for low-resolution maps.

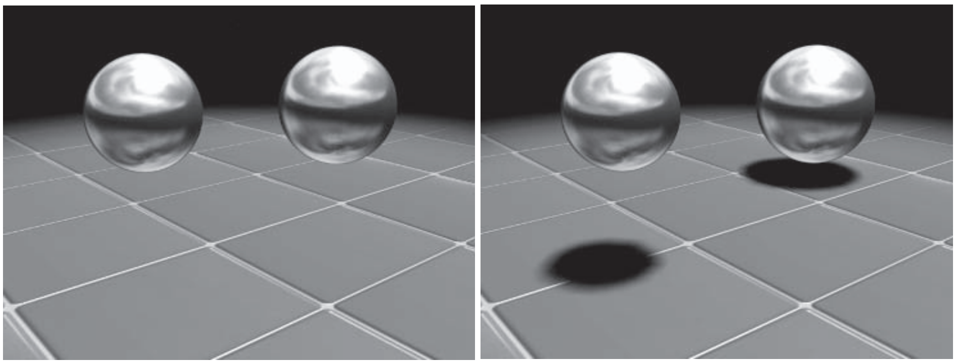


Figure 1: The importance of shadows

Shadow Algorithm

The implementation of shadows presented here uses a well-known algorithm published in 1978 by Lance Williams in [1]. Here, we briefly describe each step of the original technique. In the first step we render the image from the light's position and store the distance information for each pixel that is visible from the light's position into a texture called a shadow map. The original technique uses z-buffer information for the shadow map. In the second step we render the image from the camera's position. We project the shadow map texture onto the geometry just like we did for the spotlight (see the "Advanced Lighting and Shading with Direct3D 9" article). For each rendered pixel we compute its actual distance from the light and compare this value with the value stored in the texture. If the actual distance is greater, the pixel is shadowed by something nearer and we skip lighting. Otherwise, the pixel is not occluded and we can illuminate it. The algorithm is usable for directional lights or spotlights, but an extension for omnidirectional lights exists (see [3]). The advantage of this algorithm is that it is an image space algorithm and no knowledge of geometry is required. The main disadvantage is that it produces aliasing effects due to texture resolution. Before Direct3D 9, low precision of textures was one of the main problems in the implementation of this method in real time.

The implementation presented here does not use the z-buffer as the shadow map (although the z-buffer is still enabled while creating the shadow map), but we render to a 32-bit floating-point texture. The distance is computed in world space and everything between the spotlight's near and far clipping plane (in our case, derived from the 3D Studio Max Spotlight properties called Attenuation Near - Start and Attenuation Far - End) is mapped linearly into the range [0…1] with the following equation. This solution does not provide better quality (compared to plain usage of the z-buffer as the shadow map), but it is easier for prototyping, as we can easily test how the precision of texture affects the algorithm.

$$Z_{Stored} = \frac{(Z_{WorldSpace} - Z_{NearClippingPlane})}{(Z_{FarClippingPlane} - Z_{NearClippingPlane})}$$

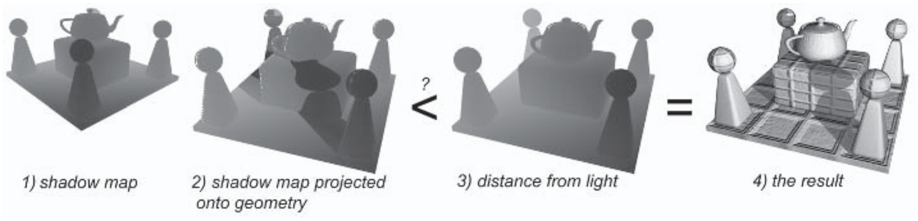


Figure 2: Shadow mapping algorithm

Depth Bias Problem

Performing a depth comparison on a shadow map introduces various errors that result in shadowing a pixel that is supposed to be lit. This is due to the shadow map magnification or minification during rendering. In the first case, one shadow map texel covers several nearby rendered pixels, resulting in some of them being incorrectly shadowed. In the second case, several texels cover one pixel and a depth comparison is likely performed with the wrong texel. The first image in Figure 3 shows this problem that is caused by minification of the shadow map. To fight these issues, the shadow map value is shifted slightly away from the light source by adding a value (called a depth bias) to the calculated distance. This minimizes depth comparison errors but finding a good bias for the whole scene is not a trivial task. The third image in Figure 3 shows what happens if the depth bias is too large.

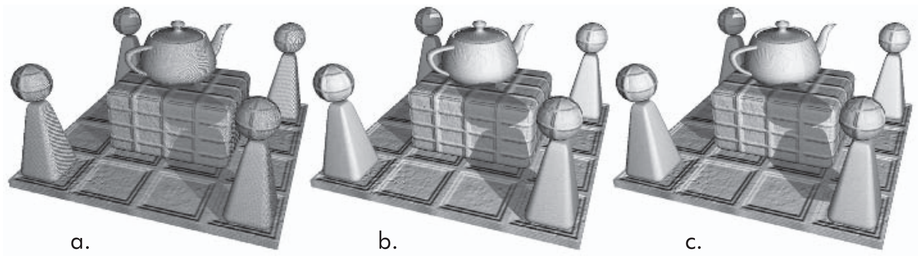


Figure 3: Depth bias issues
a) no depth bias b) just right depth bias c) too high bias

This implementation uses a different approach to fight depth test issues. Instead of finding a bias value, a shadow map is rendered with front-face culling enabled. Rendering only back faces (those facing away from the light source) causes lit faces to always pass the depth test. The depth bias issue can be ignored for back faces because these are already shadowed by a lighting and shading algorithm. Cases where the geometry is really supposed to be shadowed by another one are handled correctly unless occluder and occluded pixels are too close (but in this case, we cannot see a shadowed pixel anyway). This is not the most robust solution, but it works well for scenes with 2-manifold (or almost 2-manifold, like the teapot in our case) objects where the distance between front- and back-facing faces is not very small (with respect to the precision of the shadow map). Game geometry in most cases satisfies this need, and if not, it can be tuned during the authoring process. The solution can be easily improved if we add depth bias selectively only for objects that do not meet the criteria. Face culling is disabled, and the depth bias value is added for this geometry. This does not interfere with our implementation. The second image in Figure 4 shows that only 8-bit back-faced depth maps produce very good results.

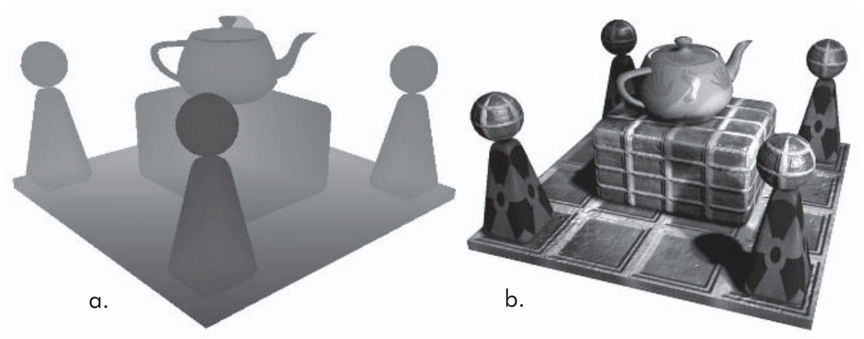


Figure 4: Back-facing depth map
a) back-facing depth map   b) scene rendered using back-faced depth maps that have 8-bit precision

Shadow Map Filtering

Since the quality of the algorithm depends on the resolution of the shadow map used, aliasing and blocky shadows can occur if the texture is small. We cannot fully eliminate this, but we can filter the map to get smoother shadows. Direct filtering of z values produces the wrong results, so a special algorithm called percentage closer filtering (PCF) was developed (see [2]). In the first step, we do depth comparison in the entire filtering kernel. Each lit pixel counts as one, and each shadowed pixel counts as zero. This forms a binary texture, which is then filtered. A sampled value gives us attenuation.

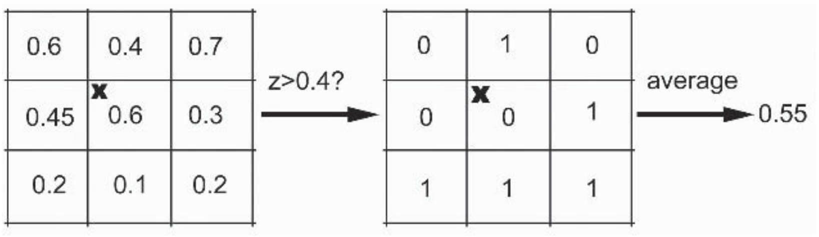


Figure 5: Percentage closer filtering

Until now, PCF was not possible in real time (or only in very limited form). With the power of Direct3D 9, we can perform it in the pixel shader. We implement a slightly improved version of the

PCF to gain even smoother results. After we obtain the binary result in the second part of the PCF for the 3x3 region, we use bilinear filtering on it to get four newly filtered values. For this, we use shadow map coordinates relative to the nearest top-left texel corner. First we perform a linear interpolation between columns of the kernel, and we get a temporary 2x3 kernel. Then we perform linear interpolation on rows and get four filtered values. In the last step, we average filtered values to get the final attenuation. Figure 6 illustrates the improved PCF, and Figure 7 illustrates bilinear filtering. Figure 8 shows the quality comparison.

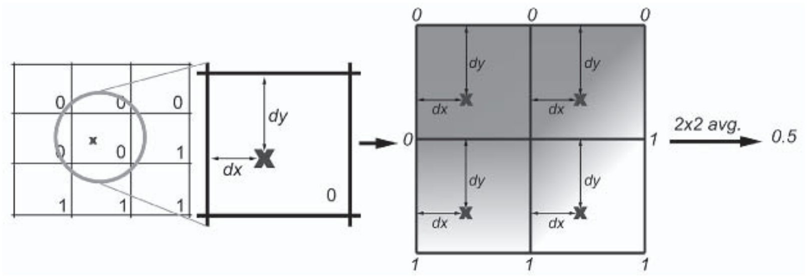


Figure 6: Improved PCF

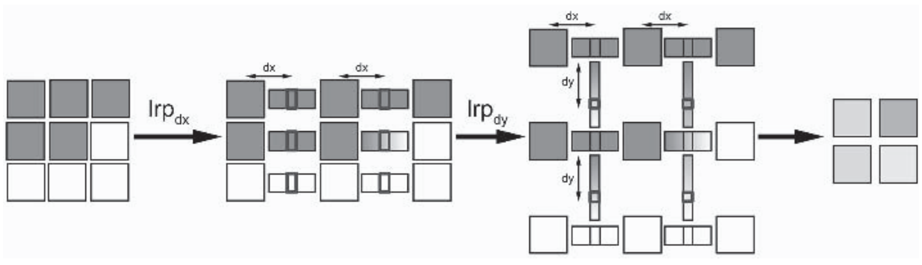


Figure 7: Bilinear filtering in the pixel shader

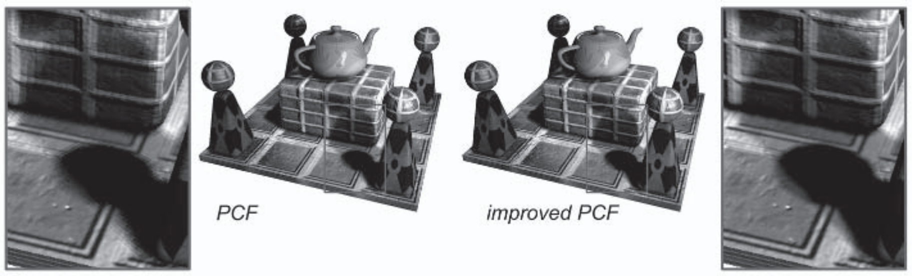


Figure 8: Standard and improved PCF comparison

Shaders for Shadow Map Creation

Shaders in this pass are very simple. The vertex shader just does a transformation into the clipping space (of the light) and into world space. Then we compute the distance from the light in world space, normalize it, and perform (optional) depth bias and output. The pixel shader just outputs this value.

The following is a vertex shader for shadow map generation.

```
vs_2_0

// Constant registers
//------------------------------
// c0-c3     - world * view * proj - this time for light
// c4-c7     - world space transposed
// c8        - Light position (In World Space)
// c9        - Z ranges – (ZNear, ZFar, 1/(ZFar-ZNear), DepthBias)
def c10, 0.5f, 0.0f, 0.0f, 0.0f

// Input registers
//------------------------------
dcl_position v0
dcl_normal v1
dcl_texcoord v2
dcl_tangent v3

//The following code outputs the position and texture coordinates
//------------------------------
m4x4 oPos, v0, c0        //vertex clip position
m4x4 r8, v0, c4          //Transform vertex into world position

//Compute distance from light and normalize it to [0...1]
//------------------------------
sub  r0, c8, r8          //Build the light vector from light source to vertex
dp3  r1.x, r0, r0        //length of vector^2
pow  r2.x, r1.x, c10.x   //sqrt(length^2)
sub  r3.x, r2.x, c9.x    //Dst - ZNear
mul  r4.x, r3.x, c9.z    //(Dst - ZNear)/(ZFar - ZNear) - normalized position
add  r4.x, r4.x, c9.w    //add depth bias – c9.w = 0 for no bias
mov  oT0, r4.x           //Output it
```

The following is a pixel shader for shadow map generation.

```
ps_2_0

// Used input registers
//------------------------------
dcl        t0                      //depth of pixel

// Output value
//------------------------------
mov        r0, t0                  //load floating point Z value
mov        oC0, r0                 //color output
```

Shaders for Final Rendering

The pixel shader is highly optimized to fit into the limit of 64 instructions without sacrificing quality. The vertex shader is the same as the one used in the “Advanced Lighting and Shading with Direct3D 9” article for Phong lighting with the addition of distance computation (the same algorithm used in the previous section).

The following is a vertex shader 2.0 model for rendering shadows.

```
vs_2_0

// Constant registers
//------------------------------
// c0-c3     - world space transposed
// c4-c7     - world * view * proj
// c8        - Light position (In World Space)
// c9        - Eye position (In World Space)
// c10-c13   - Spotlight projection matrix
// c14       - Light's Z ranges, c14.w = 0.5

// Input registers
//------------------------------
dcl_position v0
dcl_normal v1
dcl_texcoord v2
dcl_tangent v3
```

```
// Output
//------------------------------
// oT0 - texture coordinates
// oT1 - Light vector (in tangent space)
// oT2 - eye vector (in tangent space)
// oT3 - projective spotlight texture coordinates
// oT4 - distance from light

//The following code outputs position and texture coordinates
//------------------------------
m4x4 oPos, v0, c4          //vertex clip position
mov oT0.xy, v2.xy          //Texture coordinates for color texture
m4x4 r8, v0, c0            //Transform vertex into world position

//The following code generates tangent space base vectors
//------------------------------
m3x3 r11.xyz, v1, c0       //transform normal N to world space
mov r11.w, v1.w
m3x3 r9.xyz, v3, c0        //transform tangent T to world space
mov r9.w, v3.w
crs r10.xyz, r9, r11       //The cross product to compute binormal NxT

//Computes light and eye vectors and projector's texture coordinates
//------------------------------
sub r0, c8, r8             //Build the light vector from light source to vertex
nrm r1, r0                 //normalize vector
m3x3 oT1.xyz, r1, r9       //transform vector with N, T, NxT into tangent space

sub r0, c9, r8             //build the eye vector from vertex to camera source
nrm r1, r0                 //normalize vector
m3x3 oT2.xyz, r1, r9       //transform vector with N, T, NxT into tangent space

m4x4 oT3.xyzw, v0, c10     //compute projector texture coordinates

//Compute distance from light and normalize it to [0...1]
//------------------------------
sub r0, c8, r8             //Build the light vector from light source to vertex
dp3 r1.x, r0, r0           //length of vector^2
pow r2.x, r1.x, c14.w      //sqrt(length^2)
sub r3.x, r2.x, c14.x      //Dst - ZNear
mul r4.x, r3.x, c14.z      //(Dst - ZNear)/(ZFar - ZNear) - normalized position
mov oT4, r4.x              //Output it
```

The following is a pixel shader 2.0 model for rendering shadows.

```
ps_2_0

// 1 - Constant registers
//------------------------------
// c0 - diffuse texture multiplier (premultiplied with light color)
// c1 - specular texture multiplier (premultiplied with light const)
// c2 - specular shininess (shi, shi, shi, 1.0f)
// c3 - shadow texel adjust (x,y) and texture width/height (z,w)
// c4 - c11 - 3x3 filtering kernel
// c29 - texture width/height (x,y)
// c30 - texel divider - 1 / texels_in_kernel (our case is 4)
def c31, 2.0f, 1.0f, 0.0f, 4.0f     //helper constant

// 1 - Used input registers
//------------------------------
dcl         t0.xy             //texture coordinates
dcl         t1.xyz            //light vector
dcl         t2.xyz            //eye vector
dcl         t3.xyzw           //projector texture coordinates
dcl         t4.xyzw           //normalized distance from light

// 1 - Used input texture samplers
//------------------------------
dcl_2d      s0                //diffuse texture (gloss in alpha)
dcl_2d      s1                //normal texture
dcl_2d      s2                //shadow map
dcl_2d      s3                //spotlight texture

// 2 - Compute coordinates into shadow texture for actual pixel
//------------------------------
rcp         r11, t3.w         //1/w for projection divide
mad         r0, t3, r11, c3   //get texture coordinates (with adjusted center)
texld       r11, r0, s2       //sample shadow

// 3 - Compute coordinates for remaining of 3x3 filtering kernel
//------------------------------
add         r10, c4, r0       //Left column
add         r9, c5, r0
add         r8, c6, r0

add         r7, c7, r0        //Center column
add         r6, c8, r0

add         r5, c9, r0                //Right column
add         r4, c10, r0
add         r3, c11, r0

// 4 - Fill 3x3 filtering kernel
//------------------------------
texld       r10, r10, s2              //Left column
texld       r9, r9, s2
texld       r8, r8, s2

texld       r7, r7, s2                //Center column
texld       r6, r6, s2

texld       r5, r5, s2                //Right column
texld       r4, r4, s2
texld       r3, r3, s2

// 5 - Distance comparison – we get 3x3 binary kernel
//------------------------------
sub         r10.x, t4.x, r10.x        //Left column
sub         r10.y, t4.x, r9.x
sub         r10.z, t4.x, r8.x
cmp         r1.xyz, r10, c30.g, c30.r //distance comparison

sub         r9.x, t4.x, r7.x          //Center column
sub         r9.y, t4.x, r11.x
sub         r9.z, t4.x, r6.x
cmp         r2.xyz, r9, c30.g, c30.r //distance comparison

sub         r8.x, t4.x, r5.x         //Right column
sub         r8.y, t4.x, r4.x
sub         r8.z, t4.x, r3.x
cmp         r3.xyz, r8, c30.g, c30.r //distance comparison

// 6 - Bilinear filtering of 3x3 binary kernel
//------------------------------
mul         r0, r0, c29              //get coordinate in texture
frc         r0, r0                   //get fractional part only

lrp         r10.xyz, r0.x, r2, r1    //interpolate column 1 and 2
lrp         r11.xyz, r0.x, r3, r2    //interpolate column 2 and 3
lrp         r3.x, r0.y, r10.y, r10.x //interpolate row 1, column1
lrp         r3.y, r0.y, r10.z, r10.y //interpolate row 2, column1
lrp         r3.z, r0.y, r11.y, r11.x //interpolate row 1, column2
lrp         r3.w, r0.y, r11.z, r11.y //interpolate row 2, column2

dp4         r8.x, r3, c31.g          //accumulate to get average

// 7 - Setup needed vectors - load and normalize
//------------------------------
texld       r0, t0, s1              //load normal
mad         r1, r0, c31.r, -c31.g   //bias normal to range -1,1
nrm         r11, r1                 //r11 = normalized normal
mov         r1.xyz, t1
nrm         r10, r1                 //r10 = normalized light vector
mov         r1.xyz, t2
nrm         r9, r1                  //r9 = normalized eye vector

// 8 - Compute diffuse and specular intensities
//------------------------------
dp3         r0, r11, r10            //r0 = (n.l)
mul         r1, r0, c31.r           //r1.g = 2*(n.l)
mad         r1, r1, r11, -r10       //compute reflectance vector - r1 =
                                    // 2(n.l)n - l
dp3_sat     r1, r1, r9              //r1 = (r.v)
pow         r0.g, r1.r, c2.r        //r1 = (r.v)^shi
cmp         r0, r0.r, r0, c31.b     //if (n.l)<0 do not lit anything

// 9 - Modulate texture with computed intensities
//------------------------------
texld       r6, t0, s0          //load diffuse texture (gloss map is in alpha)
texldp      r4, t3, s3          //load projector texture (perspective correct)
mul         r2, r6.a, r0.g      //modulate specular intensity with gloss map…
mul         r2, r2, c1          //… and with material's specular and light color
mul         r3, r6, r0.r        //modulate diffuse intensity with texture…
mad         r0, r3, c0, r2      //… and with material's diffuse and light color
                                //and add specular
mul         r0, r0, r4          //modulate it with spotlight texture…
mul         r0, r0, r8.x        //… and with shadow

mov         oC0, r0             //color output
```

In the second part of the shader we compute the shadow map texture coordinates. These were computed in the vertex shader, and here we have to perform a perspective-correct texture lookup. We have to adjust coordinates to point exactly at the texel center to have correct shadows, and we use these in the next section to get the additional eight coordinates for a complete kernel. Because of this, we do manual division by *w* and then use `texld` instead of simple `texldp`. The `mad` instruction does this computation by multiplication with 1/*w* and addition of 0.5/*shadow_map_size* for texel center adjustment.

We compute coordinates for the remaining filtering kernel in the third part and load the distance information from the shadow map in the next one. In the fifth section we perform a comparison of distance stored in the texture and the actual one. We subtract the stored depth value from the actual one and store the result for each column into components of one register. Then we do a comparison of these values to 0 with one `cmp` instruction. If the value is less than 0, the actual pixel is not shadowed, and we remember a value of ¼. Otherwise, we store 0. Note that ¼ is stored here because in the final step we perform an average of four values.

In the beginning of the sixth section, we find the coordinates relative to the top-left corner of the shadow map texel. To do this, we multiply the coordinates by the dimensions of the texture and store only the fractional part. Then we perform bilinear interpolation. With the first two `lrp` instructions, we can interpolate whole columns with respect to the relative *x* coordinate. We interpolate final values from individual rows of these columns. The last instruction — `dp4` — performs a four-component dot product. Because we stored the interpolated values in one register and the other register contains vector (1,1,1,1), this dot product actually performs the sum of four values with one instruction. Since these values were already divided by four, this sum is their average — final light attenuation by shadow.

The rest of the shader performs the per-pixel Phong lighting shown in the "Advanced Lighting and Shading with Direct3D 9" article, and the only change is the modulation of the lighting result with shadow.

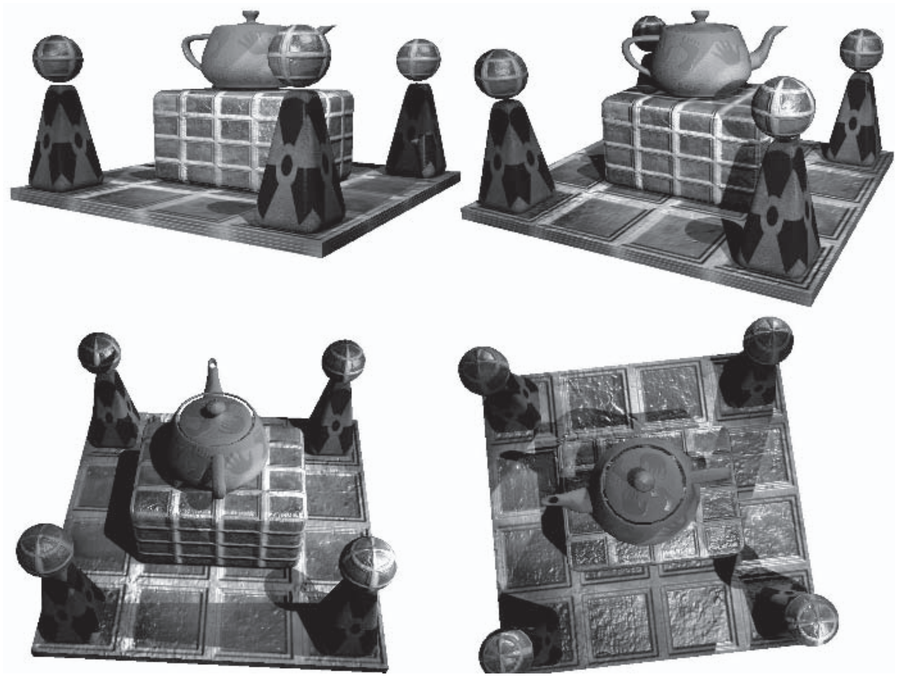


Figure 9: Shadow map results (See Color Plate 5.)

Conclusion

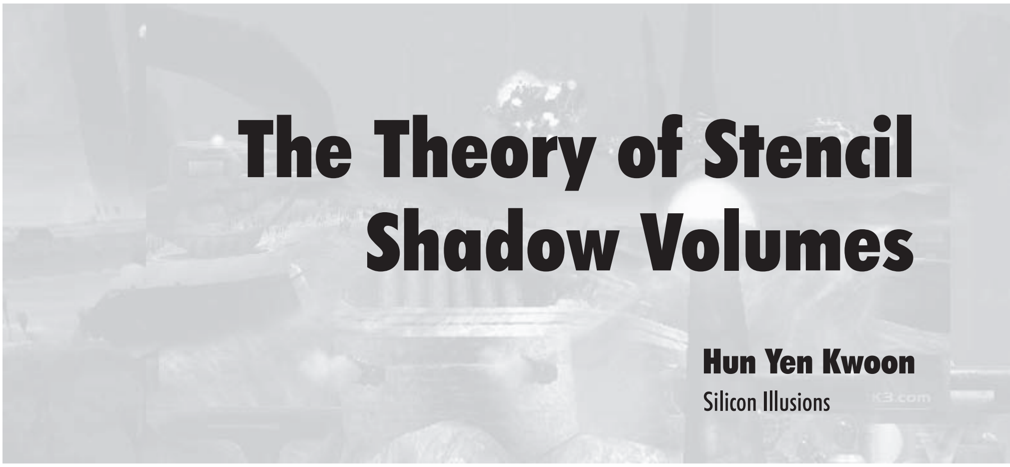
With the possibilities of Direct3D 9, we altered a classic shadow map algorithm so that it produces soft-edged shadows and minimizes depth bias issues. Bilinear filtering of the 3x3 binary kernel from the percentage closer filtering results in very soft shadows even for the low-resolution shadow map. Usage of the back-faced shadow map minimizes the depth compare errors and allows us to use lower-depth maps. This is vital for the Direct3D 8 class of hardware, where two channels of 8-bit textures are used to encode depth information.

References

[1] Williams, L., "Casting Curved Shadows on Curved Surfaces," *Computer Graphics* (SIGGRAPH '78 proceedings) August 1978, Vol. 12:3, pp. 270-274.

[2] Reeves, W.T., D.H. Salesin, and R.L. Cook, "Rendering Antialiased Shadows with Depth Maps," *Computer Graphics* (SIGGRAPH '87 proceedings) July 1987, pp. 283-291.

[3] Brabec, S., T. Annen, and H.P. Seidel, "Shadow Mapping for Hemispherical and Omnidirectional Light Sources," *Advances in Modelling, Animation and Rendering*, J. Vince, R. Earnshaw, eds., Springer: London, 2002, pp. 397-408.



The Theory of Stencil Shadow Volumes

Hun Yen Kwoon
Silicon Illusions

Introduction

One of the best visual improvements that we can make to a rendered scene is to add shadows. Shadows greatly enhance the realism of a rendered scene and provide viewers with important visual cues about object placement within the scene. Rendering cost, memory constraints, or hardware limitations sometimes make the generation of accurate shadows infeasible. However, it is often better to have at least some form of shadow, albeit a rough approximation. This is why some older games use patches of dark circular textures projected onto the surface that the game characters stand on to approximate shadowing by the characters. These types of hacks are no longer acceptable by today's gamers who have ever-increasing expectations.

Allan Watt [13] discussed four major approaches to shadow generation that include polygon projection with scan line testing, shadow polygon through visible surface, shadow volume, and shadow z-buffer. Of the four, only the shadow volume and shadow z-buffer approaches are still commonly employed today. This article concerns the shadow volume approach, which is fast becoming a fixture in newer games.

Although not a clear-cut winner, shadow volume implementation does provide several advantages over other shadowing techniques. It provides accurate hard shadows, and occluder self-shadowing is inherent in the technique. For a scene full of shadow casting occluders, shadow volume also provides accurate inter-occluder shadowing. Shadow volume is also fast gaining popularity with professional game and graphics developers. The extensive use of stencil shadow volumes in John Carmack's new Doom III engine and the impressive Power Render X game engine [4] are most notable. With this rising popularity comes a wave of enthusiastic hardware support from major graphics hardware vendors. Industry powerhouse nVidia, for example, has specifically added new capabilities to provide hardware support for shadow volume implementations in the GeForce family of consumer graphics cards. ATI Technologies Inc. has also included accelerated shadow volume rendering capabilities into its SMARTSHADER 2.0 technology that comes with graphics cards, such as the highly successful Radeon 9700. It is also possible to combine shadow volume implementation with other rendering techniques, such as projective texturing, volume texturing, or shadow mapping, to achieve highly realistic soft shadows or distance attenuated shadowing.

This article covers both the theoretical and practical aspects of stencil shadow volumes. For readers already well versed in the theory, the "Implementation on CPU" and "Implementation on GPU (Shaders)" sections provide details on implementation utilizing the CPU and GPU. For those unfamiliar with the shadow volume methodology, the "Shadow Volume Concept" and "Problems and Solutions" sections provide detailed discussions on the theories and algorithms.

Shadow Volume Concept

Frank Crow [1] first presented the idea of using shadow volumes for rendering shadows in 1977. Tim Heidmann [2] subsequently implemented Crow's shadow volume on IRIX GL by cunningly utilizing the stencil buffer commonly found in modern graphics hardware. The stencil buffer is used for counting the number of times that a ray from the eye enters and leaves the shadow volume. This is essentially a per-pixel test to determine whether an on-screen pixel is in shadow or not. The shadows generated are very accurate, albeit hard-edged. The use of the stencil buffer to implement shadow volumes also gave rise to the name "stencil shadow volumes." For general instructions on the uses of the stencil buffer, please refer to Mark Kilgard [3]. Let's first look at how shadow volume arises before we go into the details of its implementation.

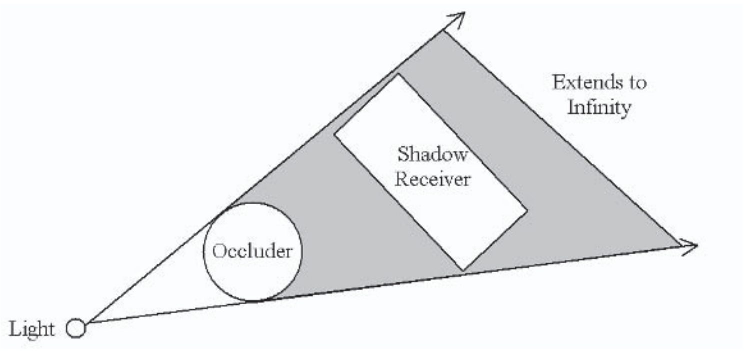


Figure 1: Occluder and shadow volume

Figure 1 shows a light source, an *occluder*, and a *shadow receiver*. The shaded region depicts the shadow volume generated by the occluder. We work on the basis that our light sources are attenuated omnidirectional point lights. This assumption is actually an added advantage of the stencil shadow volume technique. This is because generating shadows for omnidirectional light sources using view-dependent techniques such as shadow mapping or projective texturing is tricky, if not inefficient, on modern hardware. View-dependent techniques are very good for generating shadows created by directional light sources such as torchlight. However, the stencil shadow volume is more flexible and can be trivially altered to work for directional light sources. All the occluders that we work with are also assumed to be solid polygonal objects with no transparency or alpha that distorts the shadows generated. There is an added requirement that the occluders be made up of meshes that are closed volume; we discuss this requirement in more detail in the "Silhouette Determination" section. Lastly, we ignore the shadow volume projections and consequently the shadows of the shadow receivers. You probably noticed that in Figure 1 the shadow volume is supposed to extend to infinity. This is how the name "infinite shadow volumes" came about. Infinite shadow volumes help to solve a problem known as finite shadow cover, which we discuss in the "Finite Shadow Cover" section. The implementation of infinite shadow volume is presented in the "Vertex Shader Implementation (InfiniteGPU)" section.

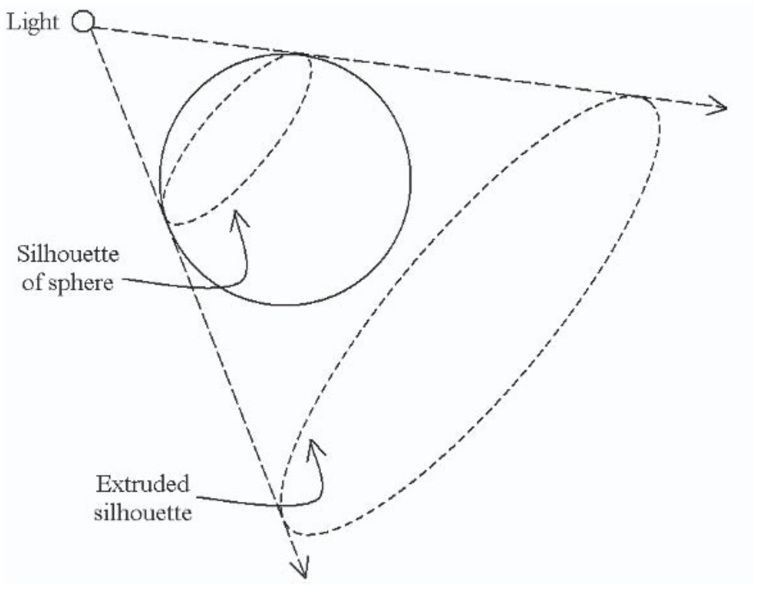


Figure 2: Silhouette of occluders

Figure 2 shows the silhouette of a sphere with respect to the light source. In essence, silhouettes are simply the outline of occluders as seen from the position of light sources. The shadow volume of an occluder is formed when we extrude the silhouette by a certain distance, finite or infinite, into the direction of incidental light rays originating from the light source. Using triangles as primitives in our meshes, a silhouette is simply made up of a chain of edges that consist of two vertices each. It should be noted at this point that shadow volume extrusion differs for different light sources. For point light sources (as depicted in Figure 2), the silhouette edges extrude exactly point for point. For infinite directional light sources, the silhouette edges extrude to a single point at infinity. We go into the details of determining silhouette edges and the creation of the shadow volumes in the "Implementation on CPU" and "Implementation on GPU (Shaders)" sections. The magnitude of the extrusion can be either finite or infinite.

There are two techniques for implementing stencil shadow volumes. The original technique is known as *depth-pass* while the other, a newer variant, is known as *depth-fail*. Let's look at how these two techniques differ in concept and implementation before we go into the problems that plague both of them.

Depth-pass (z-pass)

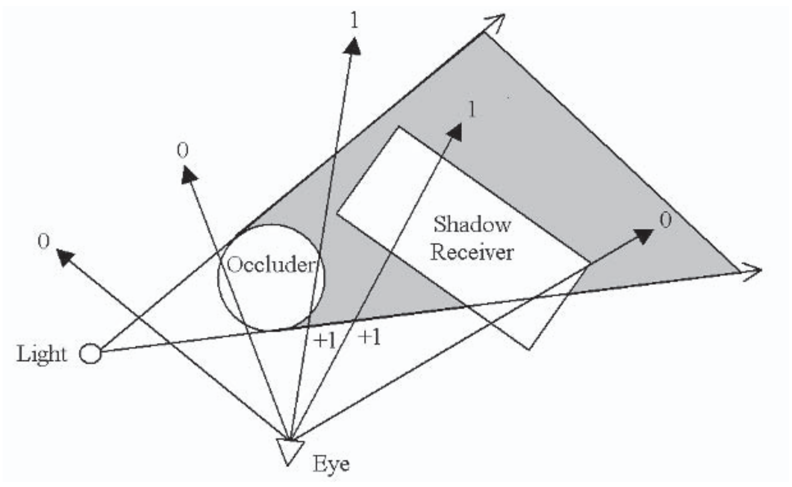


Figure 3: Depth-pass stencil operation

Figure 3 shows the numerous possible viewing directions of a player in the scene. The numbers at the end of the arrows are the values left in the stencil buffer after rendering the shadow volume. Fragments with non-zero stencil values are considered to be in shadow. The generation of the values in the stencil buffer is the result of the following stencil operations:

1. Render front faces. Increment stencil value for depth-pass. Do nothing for depth-fail. Disable draw to frame and depth buffer.
2. Render back faces. Decrement stencil value for depth-pass. Do nothing for depth-fail. Disable draw to frame and depth buffer.

The above algorithm is known as the depth-pass stencil shadow volume technique, since we manipulate the stencil values only when the depth test passes. Depth-pass is also commonly known as z-pass.

Let's assume that we had already rendered the objects onto the frame and depth buffer prior to the above stenciling operations. This means that the depth buffer would have been set with the correct values for depth testing (or z-testing if you like). The two leftmost rays originating from the eye position do not hit any part of the shadow volume (in gray), hence the resultant stencil values are 0, which means that the fragment represented by these two rays are not in shadow. Now let's trace the third ray from the left. When we render the front face of the shadow volume, the depth test would pass and the stencil value would be incremented to 1. When we render the back face of the shadow volume, the depth test would fail since the back face of the shadow volume is behind the occluder. Thus, the stencil value for the fragment represented by this ray remains at 1. This means that the fragment is in shadow since its stencil value is non-zero. To be convinced of the viability of the technique, the reader should inspect the derivation of the stencil values for the remaining two rays.

While going through the algorithm of the depth-pass technique, we are effectively doing per-pixel shadow volume counting to determine the number of times a ray representing a pixel enters and leaves the shadow volumes of the occluders. Not surprisingly, it is the same concept employed in ray-tracing techniques, whereby rays are projected to calculate the color values that represent on-screen pixels. In the case of stencil shadow volumes, we are only interested in whether a pixel is in shadow or not. Does shadow volume counting work for multiple overlapping shadow volumes?

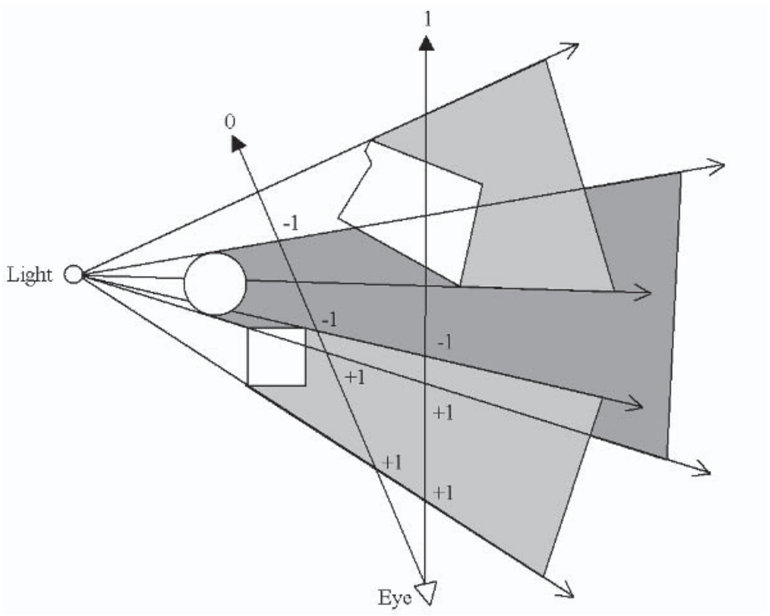


Figure 4: Multiple shadow volumes counting

Even when the shadow volumes are overlapping, as shown in Figure 4, shadow volume counting using the stencil buffer will still work. Any point on a geometric surface in a scene can only exist in three positions with respect to the shadow volumes. The point can be in front of the shadow volumes, behind the shadow volumes, or nested within the shadow volumes. For the first case whereby the point is in front of the shadow volumes, shadow volume counting gives a result of 0, as all depth tests fail, indicating that the point is not in shadow. The ray on the left in Figure 4 illustrates the second case, whereby the point is behind the shadow volume and thus not in shadow with a stencil value of 0.

The ray on the right illustrates the third case, whereby the point is nested within the shadow volumes and thus in shadow with a non-zero stencil value. The ingenuity of counting shadow volumes is that self-shadowing and inter-occluder shadowing is totally embedded into the algorithm!

John Carmack [5] and the team of Bill Bilodeau and Mike Songy [6] independently presented an alternative technique that is the direct reverse of the depth-pass stencil algorithm. Consequently, this alternative technique was aptly named the depth-fail technique. The depth-fail technique is also commonly known in the developer community as Carmack's Reverse. Why did John Carmack, Bill Bilodeau, and Mike Songy even bother to come up with an alternative stencil algorithm since the depth-pass technique seems to work great? Depth-pass works flawlessly, at least most of the time. However, when the eye point enters the shadow volume, the depth-pass algorithm fails utterly.

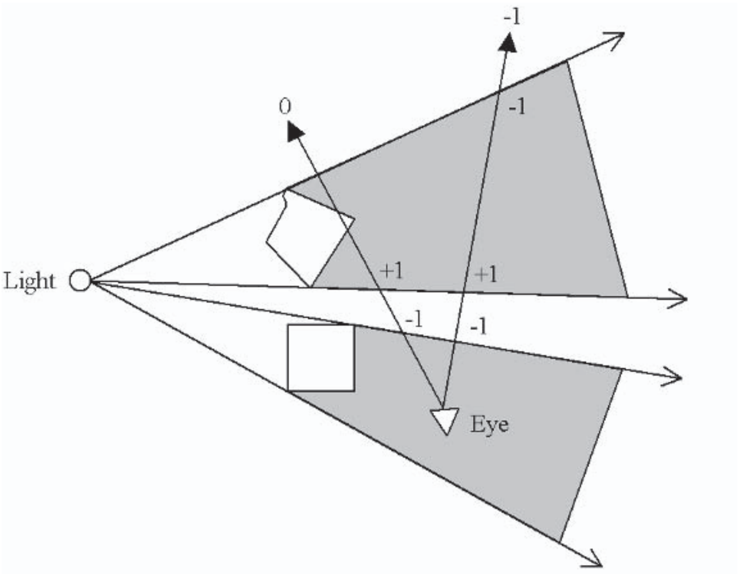


Figure 5: Depth-pass stencil operations fail when the eye point is within the shadow volume.

When the eye point is within the shadow volume, the front face of the shadow volume does not get rendered at all. This disrupts the shadow volume counting and results in erroneous values left in the stencil buffer. Figure 5 illustrates two cases in which the wrong shadow volume count is provided by the depth-pass stencil operations. The ray on the left should result in a non-zero stencil value, while the ray on the right should result in a stencil value of 0. Let's look into the mechanics of the depth-fail algorithm that allow it to handle this situation properly.

Depth-fail (z-fail)

When the eye point enters a shadow volume, the front faces of the shadow volume are clipped away by the near plane of the view frustum. This clipping is the culprit that causes disruption to the depth-pass shadow volume counting. To account for this clipping, we can perform the following extended stencil operations, which are derived from the depth-pass algorithm:

1. Render back faces. Increment stencil value for both depth-pass and depth-fail (effectively disabling depth test).
2. Render front faces. Decrement stencil value for both depth-pass and depth-fail (effectively disabling depth test).
3. Render back faces. Decrement stencil value for depth-pass. Do nothing for depth-fail.
4. Render front faces. Increment stencil value for depth-pass. Do nothing for depth-fail.

The purpose of the first two steps in the above stenciling operation is to leave positive values in the stencil buffer when the eye point is inside the shadow volume, thus accounting for the clipping of the front faces of the shadow volume. The third and fourth steps are actually the original depth-pass algorithms with the ordering reversed. Rearranging the steps, we get:

1. Render back faces. Increment stencil value for both depth-pass and depth-fail (effectively disabling depth test).
2. Render back faces. Decrement stencil value for depth-pass. Do nothing for depth-fail.
3. Render front faces. Decrement stencil value for both depth-pass and depth-fail (effectively disabling depth test).

4. Render front faces. Increment stencil value for depth-pass. Do nothing for depth-fail.

It is now obvious that some of the above steps actually cancel each other out. Simplifying the above stenciling operations, we get:

1. Render back faces. Increment stencil value for depth-fail. Do nothing for depth-pass. Disable draw to frame and depth buffer.
2. Render front faces. Decrement stencil value for depth-fail. Do nothing for depth-pass. Disable draw to frame and depth buffer.

The two-step stenciling operation above is the complete depth-fail algorithm. It is known as the depth-fail stencil shadow volume technique since we manipulate the stencil values only when the depth test fails. Depth-fail is also commonly known as z-fail. The depth-fail algorithm is really just the opposite of the depth-pass algorithm. The depth-fail stencil operations, however, do not falter when the eye point is in the shadow volume:

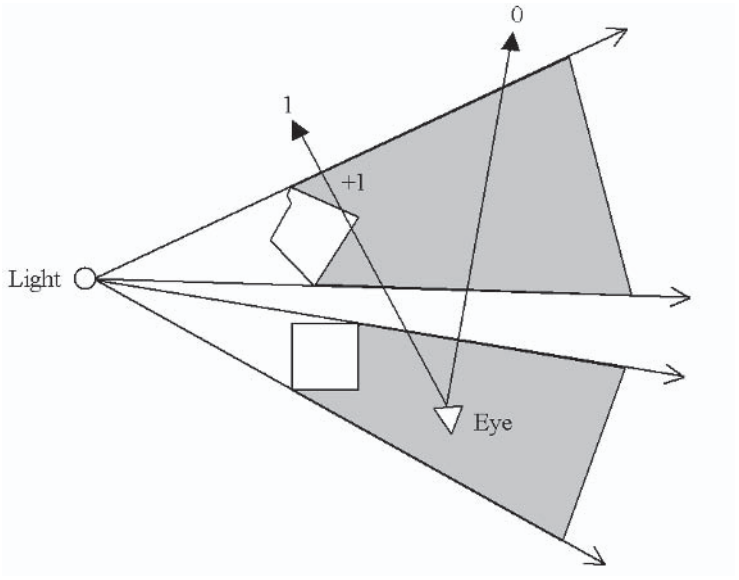


Figure 6: Depth-fail works even when the eye point is within the shadow volume.

Figure 6 again depicts the situation in which the eye point is within a shadow volume. However, by implementing the depth-fail stencil operations, the resultant values in the stencil buffer are correct. Figure 7 shows that the depth-fail algorithm would work for normal situations in which the eye point is outside the shadow volumes. The reader should inspect other possible scenarios to convince himself of the viability of the depth-fail algorithm.

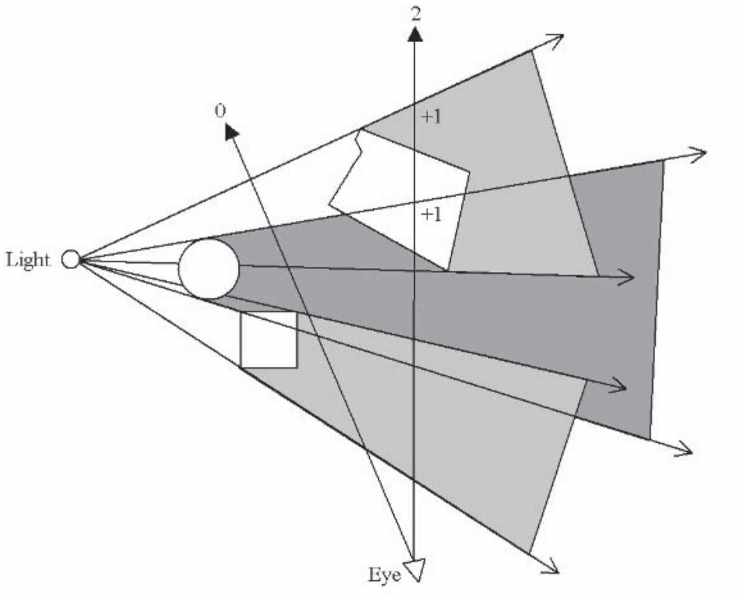


Figure 7: Multiple shadow volumes counting using depth-fail

To put non-zero values into the stencil buffer, the depth-fail technique depends on the failure to render the shadow volume's back faces with respect to the eye position. This means that the shadow volume must be a closed volume; the shadow volume must be capped at both the front and back ends (even if the back end is at infinity). Without capping, the depth-fail technique would produce erroneous results. Amazing as it may sound, you can even cap the shadow volume at infinity.

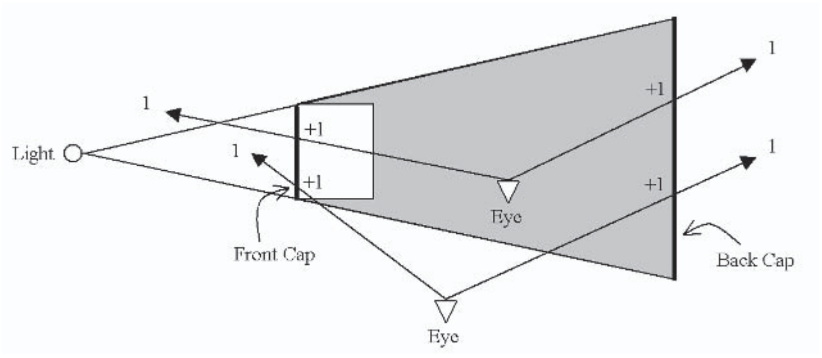


Figure 8: Capping for shadow volume

As shown in Figure 8, the front and back capping (bold lines) create a closed shadow volume. Both the front and back capping are considered back faces from the two eye positions. With depth-fail stenciling operations, the capping will create correct non-zero stencil values. There are a few ways to create the front and back capping. Mark Kilgard [7] described a non-trivial method of creating the front capping. The method basically involves the projection of the occluder's back-facing geometries onto the near clip plane and uses these geometries as the front capping. Alternatively, we can build the front capping by reusing the front-facing triangles with respect to the light source. The geometries used in the front capping can then be extruded, with their ordering reversed, to create the back capping. Reversing the ordering ensures that the back capping faces outward from the shadow volume. In fact, we must always ensure that the primitives (in our case, triangles) that define the entire shadow volume are outward facing, as shown in Figure 9. It must be noted that rendering closed shadow volumes is somewhat more expensive than using depth-pass without shadow volume capping. Besides a larger primitive count for the shadow volume due to the capping geometries, additional resources are needed to compute and store the front and back capping. We go into the details of capping shadow volumes, including some possible optimization techniques, in the "Shadow Volume Capping" section.

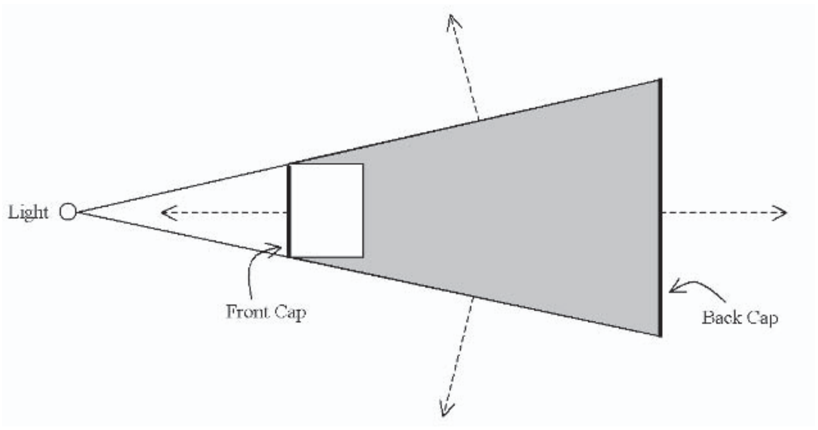


Figure 9: The capped shadow volume must be outward facing.

Problems and Solutions

Before we go into the implementation details of stencil shadow volumes, it is beneficial to take a look at the many problems and limitations of both the depth-pass and depth-fail techniques in order to avoid common pitfalls during implementation. Potential solutions to these problems are discussed, while practical implementation is outlined in the following sections.

Finite Shadow Cover

The idea of an infinite shadow volume where the shadow volume extends to infinity was presented in Figure 1. The need for an infinite shadow volume may seem obscure at first, but it really helps alleviate a common problem known as *finite shadow cover* as shown in Figure 10.

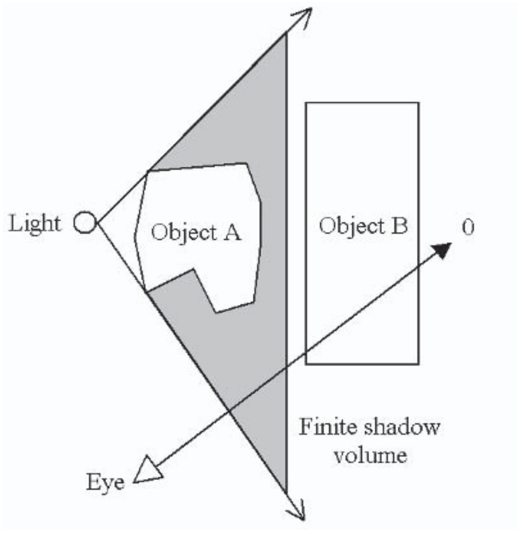


Figure 10: A finite shadow volume may fail to cover all objects adequately.

Finite shadow volume affects both depth-pass and depth-fail implementations, but let's assume the case of a depth-fail implementation in Figure 10. With the light close to object A, a finite shadow volume may not be extended far enough to cover object B properly. The ray from the eye toward object B ends up with a fragment stencil value of 0 when in fact it should be non-zero! An infinite shadow volume would ensure that no matter how close the object is to an occluder, the resultant shadow volume would cover all the objects in the scene. We discuss how to create an infinite shadow volume by extruding silhouette vertices to infinity using homogenous coordinates in the "Forming the Shadow Volume" and "Vertex Shader Implementation (InfiniteGPU)" sections.

Ghost Shadow

While extruding geometries by a huge distance or to infinity helps avoid the problem of finite shadow volume cover, it also generates another problem: Imagine two players in a dungeon first-person-shooter (FPS) game, roaming in adjacent rooms separated by a solid brick wall. A table lamp in one of the rooms causes one of the players to cast a shadow onto the brick wall separating the rooms.

The player in the other room would see this shadow since the shadow volume extrudes out to infinity. The solid brick wall suddenly feels like a thin piece of paper with a “ghost” shadow on it. Fortunately, by utilizing occlusion information and other culling techniques, we can restrict shadow volume rendering to individual rooms and avoid this kind of situation. Figure 11 shows a more awkward situation, whereby the camera sees both the occluder and its shadow on one side and the ghost shadow on the other side of the terrain. Handling such a situation is tricky because the shadow volume must not be extruded beyond the terrain. Determination of the correct extrusion distance is not trivial, especially if the light and occluder are free to move around the scene. This scenario is very possible, especially for flight simulations or aerial combat games.

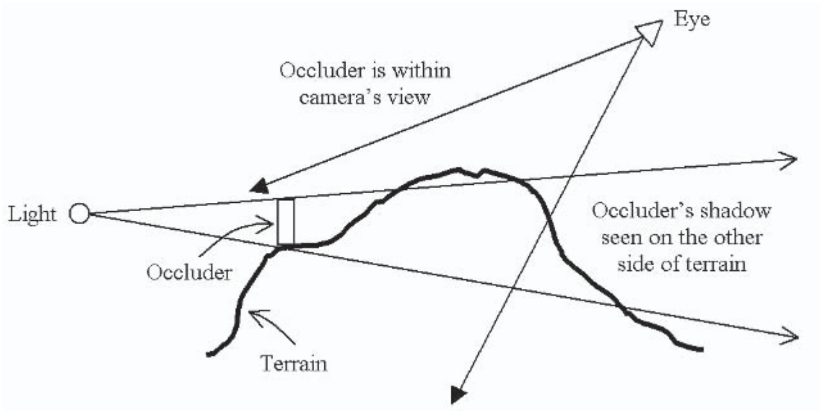

Eye
Occluder is within camera's view
Light
Occluder's shadow seen on the other side of terrain
Occluder
Terrain

Figure 11: Ghost shadow effect due to large extrusion distance

The only feasible solution to avoid both the finite shadow volume cover (Figure 10) and ghost shadow (Figure 11) is to impose limitations on the placement of light sources and occluders in a scene. If we can be sure that an occluder can never get closer than a certain distance to a shadow-casting light source, we can safely estimate the largest distance we would need to extrude the shadow volume in order to provide adequate shadow cover while not causing ghost shadows. It is thus an added responsibility of level designers to ensure that the occluder and light source placement in a scene do not compromise or break the underlying stencil shadow volume implementation. We discuss the importance of scene management in more detail in the "Efficiency and Robustness" section.

View Frustum Clipping

View frustum clipping is perhaps one of the most annoying problems that plague shadow volume implementations. Clipping is a potential problem with any 3D rendering technique, since we rely on perspective projection to view our 3D worlds. The view frustum that provides this perspective projection requires a near clip distance and a far clip distance for the creation of a near clip plane and a far clip plane. The near and far clip planes define the viewable limits of the perspective projection. Since shadow volumes are made up of polygons, they run the risk of being clipped by the near or far clip planes, just like any other polygons. However, the implication for shadow volumes is far more damaging, as the clipping would disrupt the counting of the shadow volumes by the stencil buffer and hence lead to erroneous shadows being rendered. Both the depth-pass and depth-fail techniques suffer from view frustum clipping in different ways. The depth-pass technique suffers from errors when the shadow volume gets clipped after intersecting the near clip plane, as shown in Figure 12. The ray from the eye represents just one case where the stencil value for the associated pixel will be wrong due to the clipping of the shadow volume's front face. The clipping of the front faces, which the depth-pass algorithm depends on to increment the stencil values, means that the entire region with clipped front faces will display incorrect shadows.

On the other hand, the depth-fail technique suffers from errors arising due to the clipping of the shadow volume by the far clip plane. Since the far clip plane is at a finite distance from the eye position, there is a possibility of the shadow volume's back faces being clipped by the far plane. When this happens, the shadow counting by the depth-fail technique is thrown off balance, since depth-fail requires the rendering of the back faces with

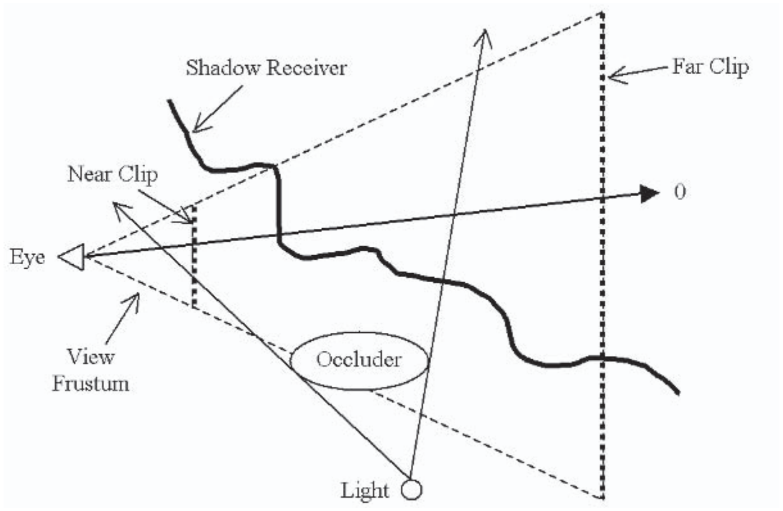


Figure 12: Shadow volume clipped at near clip plane causing depth-pass errors

proper depth testing. The ray from the eye in Figure 13 represents a case whereby the depth-fail technique generates errors since the far plane had clipped the back face of the shadow volume. The clipping of the back faces destroys the chance for the depth-fail algorithm to increment the stencil buffer and thus results in incorrect shadowing.

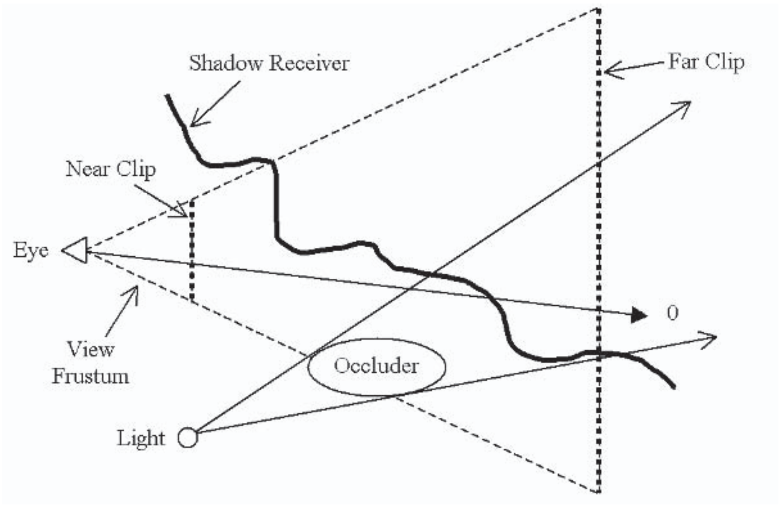


Figure 13: Shadow volume clipped at far clip plane, causing depth-fail errors

A simple solution to these problems is to move the clipping planes to avoid clipping the shadow volume. Adjusting the near plane to avoid the depth-pass problem is not feasible because doing so will greatly affect the depth precision range and may have negative impacts on other operations that are dependent on the depth buffer values. On the other hand, however, shifting the far plane by an infinite distance will actually solve the far plane clipping problems for depth-fail.

Let's first take a look at attempts to solve the depth-pass near clip plane problem, which happens to be one of the trickiest issues that one could encounter in real-time graphics. Mark Kilgard [7] presented interesting ideas on how to handle the two possible scenarios when shadow volumes intersect the near clip plane. The idea was to "cap" the shadow volume at the near clip plane so that the previously clipped front-facing geometries could now be rendered at the near clip plane instead. The first scenario is when all the vertices of the occluder's silhouette project to the near clip plane. In this case, a quad strip loop is generated from all front-facing vertices within the silhouette of the occluder. The quad strip loop is then projected onto the near clip plane, thus forming a capping for the shadow volume.

The second scenario occurs when only part of the shadow volume projects onto the near clip plane. This proved to be much more difficult to handle than the previous scenario. To his credit, Kilgard devised an elaborate system to filter out the vertices of triangles (facing away from the light) that should be projected onto the near clip plane in order to cap the shadow volume. The capping of shadow volumes at the near clip plane gave rise to another problem: depth precision. Rendering geometries at the near clip plane is analogous to rolling a coin; the coin can drop down both sides easily and unpredictably. This means that the near plane may still clip the vertices that were meant to cap the shadow volume. To overcome this, Kilgard devised yet another method that builds a depth range "ledge" from the eye point to the near plane. The idea is to render the shadow volume from a depth range of [0.0, 1.0], while normal scene rendering occurs within a depth range of [0.1, 1.0]. The ledge could be built into the view frustum by manipulating the perspective projection matrix. Once in place, the near clip plane capping of shadow volumes is done at a depth value of 0.05, which is half of the ledge. This idea is indeed original, but it does not totally solve the problem. Cracks or "holes" in the near plane shadow capping occur very frequently, resulting in erroneous results. The conclusion with the near clip plane problem is that there are really no trivial solutions. At least, there is no known foolproof solution to the problem at the time of publication. This makes the depth-pass technique less robust and confines its spectrum of application to those situations where near plane clipping of the shadow volume is not possible (e.g., real-time strategy (RTS) games).

Fortunately, there is an elegant solution to the far plane clipping problem that plagues the depth-fail technique. The antidote to the problem is simply an infinite perspective view projection or simply an infinite view frustum. By projecting the far plane all the way to infinity, there is no mathematical chance of the shadow volume being clipped by the far plane. Even if the shadow volume were extruded to infinity, the far plane at infinity would still not clip it after some projection matrix alteration. The derivation for a left-handed Direct3D perspective projection matrix is presented here. For the derivation of such a matrix applicable to OpenGL, please refer to Eric Lengyel [8]. Let's start by looking at a standard left-handed perspective projection matrix in Direct3D:

$$P = \begin{bmatrix} \cot\left(\frac{fov_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov_h}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-fn}{f-n} & 0 \end{bmatrix} \quad (1)$$

Variables:	n:	near plane distance
	f:	far plane distance
	fov_w:	horizontal field of view in radians
	fov_h:	vertical field of view in radians

A far plane at infinity means that the far plane distance needs to approach ∞. Hence, we get the following perspective projection matrix when the far plane distance goes toward the infinity limit:

$$P_\infty = \lim_{f\to\infty} P = \begin{bmatrix} \cot\left(\frac{fov_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov_h}{2}\right) & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -n & 0 \end{bmatrix} \tag{2}$$

Equation (2) defines a perspective projection view that extends from the near plane to a far plane at infinity. But, are we absolutely sure that the vertices that we extruded to infinity using the 4D homogeneous vector do not get clipped at infinity? Sadly, we cannot be 100 percent sure of this due to limited hardware precision. In reality, graphics hardware sometimes produces points with a normalized z-coordinate marginally greater than 1.0, which happens to be the limit at the far plane. These values are then converted into integers for use in the depth buffer. This is going to wreak havoc, since our stencil operations depend wholly on the depth value testing. (As a side note, the DirectX 9.0 Direct3D API features floating-point z-buffer format, which may alleviate this situation. However, it is applicable only to hardware that supports depth buffer using floating-point.)

Fortunately, there is a workaround for this problem. The solution is to map the z-coordinate values of our normalized device coordinates from a range of [0, 1] to [0, 1–ε], where ε is a small positive constant. This means that we are trying to map the z-coordinate of a point at infinity to a value that is slightly less than 1.0 in normalized device coordinates. (OpenGL has normalized device coordinates of –1.0 to 1.0.) Let D_z be the original z-coordinate value and D'_z be the mapped z-coordinate. The mapping can be achieved using the equation shown below:

$$D'_z = D_z(1-\varepsilon) \tag{3}$$

Now, let's make use of equation (2) to transform a point A from camera space (A_{cam}) to clip space (A_{clip}). Note that camera space is also commonly referred to as eye space.

$$A_{clip} = A_{cam}P_{\infty} = [A_x\ A_y\ A_z\ A_w]\begin{bmatrix} \cot\left(\frac{fov_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov_h}{2}\right) & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -n & 0 \end{bmatrix}$$

...which would give us:

$$A_{clip} = \begin{bmatrix} A_x \cot\left(\frac{fov_w}{2}\right) \\ A_y \cot\left(\frac{fov_h}{2}\right) \\ A_z - nA_w \\ A_z \end{bmatrix} \qquad (4)$$

Let's factor the desired range mapping into equation (3) by replacing D_z with $\frac{(A_{clip})_z}{(A_{clip})_w}$ and D'_z with $\frac{(A'_{clip})_z}{(A_{clip})_w}$:

$$\frac{(A'_{clip})_z}{(A_{clip})_w} = \frac{(A_{clip})_z}{(A_{clip})_w}(1-\varepsilon) \qquad (5)$$

Simplifying equation (5) by using the values given by equation (4), we get:

$$(A'_{clip})_z = A_z(1-\varepsilon) + nA_w(\varepsilon - 1) \qquad (6)$$

Using equation (6), we can enforce our range mapping into the projection matrix P_{∞} given by equation (2) to get the following:

$$P'_{\infty} = \begin{bmatrix} \cot\left(\frac{fov_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov_h}{2}\right) & 0 & 0 \\ 0 & 0 & (1-\varepsilon) & 1 \\ 0 & 0 & n(\varepsilon - 1) & 0 \end{bmatrix} \quad (7)$$

Thus, we can use the perspective projection matrix given in equation (7) without fear of far plane clipping of shadow volumes occurring at infinity! You might wonder whether stretching the view frustum volume all the way to infinity would impact depth buffer precision. The answer is yes, it does affect precision, but the loss of precision is really negligible. The amount of numerical range lost when extending the far plane out to infinity is only $\frac{n}{f}$. Say our original near clip plane is at 0.1 meters, and the far clip plane is at 100 meters. This range corresponds to a depth range of [0, 1.0]. We then extend the far plane distance to infinity. The range from 0.1 meters to 100 meters would now correspond to a depth range of [0, 0.999]. The range from 100 meters to infinity would correspond to a depth range of [0.999, 1.0]. The loss in depth buffer precision is really not a big impact at all. The larger the difference between the *n* and *f* values, the smaller the loss in depth buffer precision. You can find the above derivations and many other related mathematical derivations in Eric Lengyel's book [9]. It should be noted that using an infinite view frustum means that we have to draw more geometries. This may pose a potential performance problem.

The infinite view frustum projection is really just a software solution to the far plane clipping problem. Mark Kilgard and Cass Everitt [10] presented a hardware solution to the problem instead of using an infinite view frustum. Newer graphics hardware now supports a rendering technique called *depth clamping*. In fact, the depth-clamping extension, `NV_depth_clamp`, was specifically added to nVidia's GeForce3 and above graphics cards to solve the far plane clipping problem for shadow volume implementations.

When active, depth clamping forces all the objects beyond the far clip plane to be drawn at the far clip plane with the maximum depth value. This means that we can project the closed shadow volume to any arbitrary distance without fear of it being clipped by the far plane, as the hardware will handle the drawing properly. With such automatic support from graphics hardware, depth-fail shadow volumes become easier to implement. We can extend the shadow volume to infinity while rendering with our finite view frustum and still get correct depth-fail stencil values! However, the trade-off is hardware dependence, unless hardware vendors and graphics APIs such as Direct3D and OpenGL commonly support depth clamping in the future. If we want the depth-fail shadow volume to work for any graphics card (with stenciling support at least), we have to use the infinite view frustum projection instead of the depth-clamping extension.

With a good background on the stencil shadow volume algorithms and their associated problems, it is time to plunge into their implementations. The following sections present two different approaches to implementing stencil shadow volumes. The first approach is the common way of determining the occluder's silhouette on the CPU and uploading the new shadow volume vertices onto the hardware. The second approach makes use of the programmable pipeline (vertex shader) to construct the shadow volume on the hardware itself, thereby saving the cost of uploading new geometries every frame. It should be noted here that a one-off preprocessing of the occluder's geometries is necessary for the vertex shaders (GPU) implementation. The preprocessing adds new vertices into the source data set in order to facilitate the construction of the shadow volume on the hardware. With optimization in place, preprocessed data sets typically contain around two times more vertices.

Implementation on CPU

For this section on CPU implementation, the reader should refer to both the DepthPassCPU and DepthFailCPU samples, which can be found on the companion CD. Note that both samples are based on DirectX 8.1. A list of general steps for implementing shadow volumes on the CPU is presented shortly. The subsequent discussion of the two CPU-based samples will closely follow these steps.

How It Is Done

Let's collate what we have learned and try to come up with the steps to do both depth-pass and depth-fail stencil shadow volumes on the CPU. A general list of steps to implement stencil shadow volumes is:

1. Render the scene to fill the depth buffer with the correct z values.
2. Select a light source. Clear the stencil buffer if this is the first light. Calculate the silhouette of all the occluders with respect to the light source.
3. Extrude the silhouette away from the light source to a finite or infinite distance to form the shadow volumes and generate the capping if the depth-fail technique was used.
4. Set up the stencil operations and render the shadow volumes using the depth-pass or depth-fail technique.
5. Repeat steps 2 to 4 for all selected lights in the scene.
6. Using the updated stencil buffer, do a lighting pass to shade (or make it a tone darker) the pixels that correspond to non-zero stencil values.

The above list of steps is just one way to achieve a shadowed scene using the stencil buffer values. Many other workable approaches to creating a shadowed scene exist. For example, per-pixel attenuation techniques can be combined with the stencil shadow volume algorithm so that instead of "darkening" the pixels in shadows, the pixels are not drawn at all. We go through the implementation issues of the steps described above in the following sections.

Silhouette Determination

As described in step 2 of the previous section, once a light source is selected, the first step to constructing a shadow volume is to determine the silhouette of the occluder. The stencil shadow algorithm requires that the occluders be made up of closed triangle meshes. This means that every edge in the model must only be shared by two triangles, thus disallowing any holes that would expose the interior of the model.

The reasons for this requirement are obvious, as any seams or holes (formed for example by t-junctions) would greatly complicate the silhouette determination algorithm. In cases where the original occluder's geometries are used for forming the front capping, non-closed meshes will also throw the stencil counting off-balance. However, there are ways to circumvent a few special cases of non-closed triangle meshes for use in shadow volume implementations, but these are beyond the scope of this article.

In silhouette calculations, we are only interested in the edges shared by a triangle that faces the light source and another triangle that faces away from the light source. Let's assume that we are working with an indexed triangle mesh.

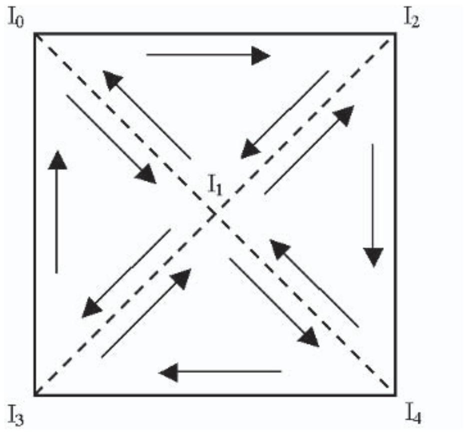


Figure 14: Edge elimination for silhouette determination

Figure 14 shows one side of a box that is made up of four triangles with consistent clockwise winding. The broken lines indicate the redundant internal edges, since we are only interested in the solid line that forms the outline of the box. The redundant internal edges are indexed twice, as they are shared by two triangles. We can take advantage of this property to come up with a simple method to determine the silhouette edges.

1. Loop through all the model's triangles.
2. If a triangle faces the light source (dot product of light's direction vector and triangle face normal is greater than zero):
    a. Insert the three edges (pair of vertices) of the triangle into an edge stack.
    b. Check for previous occurrence of each edge or its reverse in the stack.
    c. If an edge or its reverse is found in the stack, remove both edges.
3. Edges left in the stack form the silhouette.

The above algorithm ensures that all the internal edges will eventually be removed from the stack, since they are indexed by more than one triangle. This silhouette determination method is implemented in both the DepthPassCPU and DepthFailCPU samples, as the function `InsertEdge()` called from `BuildShadowVolume()`. The following code snippet is taken from the `BuildShadowVolume()` function in the DepthPassCPU sample.

```
MESHVERTEX* pVertices;
WORD*       pIndices;

// Lock the geometry buffers
pMesh->LockVertexBuffer( 0L, (BYTE**)&pVertices );
pMesh->LockIndexBuffer( 0L, (BYTE**)&pIndices );
DWORD dwNumVertices = pMesh->GetNumVertices();
DWORD dwNumFaces    = pMesh->GetNumFaces();

// Allocate a temporary edge list
WORD* pEdges = new WORD[dwNumFaces*6];
```

```
12 DWORD dwNumEdges = 0;
13
14 // For each face, check all 3 edges
15 for( DWORD i=0; i<dwNumFaces; i++ )
16 {
17   WORD wIndex0 = pIndices[3*i+0];
18   WORD wIndex1 = pIndices[3*i+1];
19   WORD wIndex2 = pIndices[3*i+2];
20
21   D3DXVECTOR3 v0 = pVertices[wIndex0].p;
22   D3DXVECTOR3 v1 = pVertices[wIndex1].p;
23   D3DXVECTOR3 v2 = pVertices[wIndex2].p;
24
25   // Note that vLight has already been transformed to object space. This
        saves some computation work
26   // Cosine value larger than 0.0 means light-facing since angle between
27   // light vector vLight and the face normal is within -90 to 90 degrees
28   // Face normal is computed in order to use welded models
29   D3DXVECTOR3 vCrossValue1(v2-v1);
30   D3DXVECTOR3 vCrossValue2(v1-v0);
31   D3DXVECTOR3 vFaceNormal;
32   D3DXVec3Cross( &vFaceNormal, &vCrossValue1, &vCrossValue2 );
33
34   // Take note that we are doing a recalculation of vLightDir, or direction
35   // vector of incoming light ray by using the first vertex of a face
        (3 vertices) to represent that face.
36   // The dot product test is also only done once per face.
37   D3DXVECTOR3 vLightDir = vLight - v0;  // Direction vector of incoming
                                              light rays
38   if( D3DXVec3Dot( &vFaceNormal, &vLightDir ) >= 0.0f )
39   {
40     InsertEdge( pEdges, dwNumEdges, wIndex0, wIndex1 );
41     InsertEdge( pEdges, dwNumEdges, wIndex1, wIndex2 );
42     InsertEdge( pEdges, dwNumEdges, wIndex2, wIndex0 );
43   }
44 }
45
```

Note that we have to compute the face normal for every face in the code from line 29 through 32. The calculation of face normals coupled with the use of indices instead of positions for comparison later in the `InsertEdge()` function will allow us to make use of welded models. Welded models result in better performance due to reduced polygon counts for the shadow volume generated. We discuss the advantages of welded models in the "Efficiency and Robustness" section. The above code will also work for non-welded models. Both the DepthPassCPU and DepthFailCPU samples make use of welded models. On line 37, the vector `vLightDir` is calculated from the light position and the first vertex of the current face. Hence, we do only one dot product test for each face, as we are using the face normal, which is the same for all three vertices. The dot product at line 38 will insert all three edges of the face into an edge stack through the `InsertEdge()` function if it is light facing. The following is the code for the `InsertEdge()` function:

```
VOID CShadow::InsertEdge( WORD* pEdges, DWORD& dwNumEdges, WORD v0,
      WORD v1 )
{
  for (DWORD i=0; i < dwNumEdges; i++)
  {
    if( ( pEdges[2*i+0] == v0 && pEdges[2*i+1] == v1 )||( pEdges[2*i+0]
          == v1 && pEdges[2*i+1] == v0 ) )
    {
      if( dwNumEdges > 1 )
      {
        pEdges[2*i+0] = pEdges[2*( dwNumEdges-1 )+0];
        pEdges[2*i+1] = pEdges[2*( dwNumEdges-1 )+1];
      }
      dwNumEdges--;
      return;
    }
  }

  pEdges[2*dwNumEdges+0] = v0;
  pEdges[2*dwNumEdges+1] = v1;
  dwNumEdges++;
}
```

The `InsertEdge()` function tests for recurrences of new edges and eliminates those that are duplicated. After running through the entire model, the edges left over in the stack represent the silhouette edges that we need.

Eric Lengyel [8] presented another silhouette determination algorithm that makes use of the consistent winding (counterclockwise for OpenGL) of vertices. The method requires two passes on all the triangles of the model to filter in all the edges shared by pairs of triangles. The resultant edge list then undergoes the dot product operations to get the edges that are shared by a light-facing triangle and a non-light-facing triangle.

It is important to note that silhouette determination is one of the two most expensive operations in stencil shadow volume implementation. The other is the shadow volume rendering passes to update the stencil buffer. These two areas are prime candidates for aggressive optimizations, which we discuss in detail in the concluding sections. Now let's get on to the business of forming the shadow volume using the silhouette edges that we have obtained.

Forming the Shadow Volume

After we have determined the silhouette edges, it's time to start forming the shadow volume, as described in step 3 of the "How It Is Done" section. There are three steps to forming the shadow volume:

1. Extrusion of silhouette edges
2. Forming the sides of the shadow volume
3. Capping the shadow volume at both ends (only applicable to depth-fail)

Note that for the depth-pass algorithm, capping at both ends is not required.

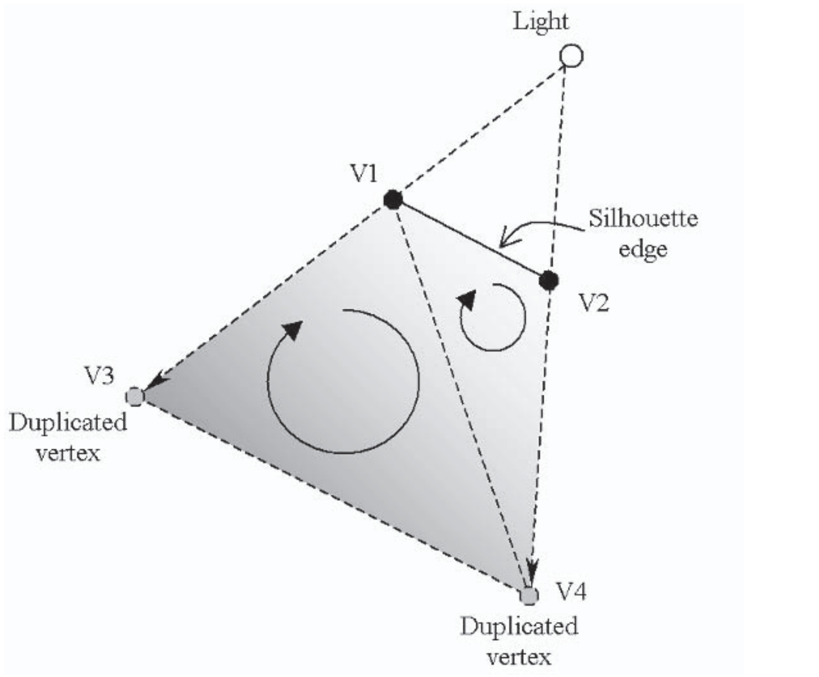


Figure 15: Extrusion and the forming of shadow volume for a point light source

As shown in Figure 15 above, the silhouette edge defined by vertices v1 and v2 is used to create two more vertices, v3 and v4. The four vertices are then used to create a quad to form the side of the shadow volume. The arrows within the quad show the clockwise ordering of the vertices that is needed to make the side face outward. This is implemented in the function `BuildShadowVolume()` for both the DepthPassCPU and DepthFailCPU samples. With regard to distance needed to extrude vertices v1 and v2 to form v3 and v4, both the DepthPassCPU and DepthFailCPU samples employ a finite extrusion distance. We discuss infinite shadow volume extrusion shortly.

In the "Finite Shadow Cover" and "Ghost Shadow" sections, we discussed the two scenarios whereby infinite or finite shadow volume extrusion might be desirable for different reasons. The implementation for finite extrusion is trivial. Referring to Figure 15 again, a light vector is formed by making use of the light position and the selected vertex. The light vector defines the direction vector of the incoming light ray at that vertex. The extruded vertex can then be computed by extending the selected vertex by a finite distance in the direction of the light vector. Take note that it is not advisable to extrude the vertex by a multiple of the magnitude of the light vector. This is because the light vector is unique for all vertices (assuming point light sources), and the magnitude can differ wildly. If the magnitude of the light vector is too small (e.g., the light is very close to the vertex), the vertex may not be extruded far enough to provide adequate shadow cover. Hence, extruding the vertices by an absolute distance is recommended. This is easily done by normalizing the light vector and multiplying individual components by the absolute distance to be extruded. Lastly, we insert two triangles using the original and extruded vertices to form the sides of the shadow volume. The following code snippet from the `BuildShadowVolume()` function in the DepthPassCPU sample accomplishes what we have just discussed:

```
// For each silhouette edge, duplicate it,
for( i=0; i<dwNumEdges; i++ )
{
  D3DXVECTOR3 v1 = pVertices[pEdges[2*i+0]].p;
  D3DXVECTOR3 v2 = pVertices[pEdges[2*i+1]].p;

  D3DXVECTOR3 v3;
  D3DXVECTOR3 v4;
  D3DXVECTOR3 vExtrusionDir;    // Direction vector from light to vertex, or rather the direction to extrude

  // Extrusion can be tricky. It is not advisable to extrude vertices by a multiple of the magnitude of
  // vExtrusionDir. This is because the magnitude may be so small that even a large multiple would not extrude
  // the vertices far enough. Results will be unpredictable if either light source of occluders are dynamic
  // objects. Hence, we normalize the vExtrusionDir vector before multiply by the ABSOLUTE distance
  // that we want to extrude the vertex to.

  vExtrusionDir = v1 - vLight;  // Compute a new extrusion direction for new vertex
  D3DXVec3Normalize( &vExtrusionDir, &vExtrusionDir );
```

```
    v3.x = v1.x + vExtrusionDir.x * g_fExt;
    v3.y = v1.y + vExtrusionDir.y * g_fExt;
    v3.z = v1.z + vExtrusionDir.z * g_fExt;

    vExtrusionDir = v2 - vLight;  // Compute a new extrusion direction for
                                     new vertex
    D3DXVec3Normalize( &vExtrusionDir, &vExtrusionDir );
    v4.x = v2.x + vExtrusionDir.x * g_fExt;
    v4.y = v2.y + vExtrusionDir.y * g_fExt;
    v4.z = v2.z + vExtrusionDir.z * g_fExt;

    // Add a quad (two triangles) to the vertex list
    m_pVertices[m_dwNumOfVertices++] = v1;
    m_pVertices[m_dwNumOfVertices++] = v4;
    m_pVertices[m_dwNumOfVertices++] = v2;

    m_pVertices[m_dwNumOfVertices++] = v1;
    m_pVertices[m_dwNumOfVertices++] = v3;
    m_pVertices[m_dwNumOfVertices++] = v4;
  }
```

As discussed previously, we may need to extrude the silhouette edges to infinity to avoid the situation shown in Figure 10, where a finite shadow volume extrusion fails to cover all the shadow receivers in a scene. However, it is not compulsory to extrude the silhouette edges to infinity if we can ensure that the situation in Figure 10 will never happen in our scene. In practical cases, a large value would normally be more than adequate.

Mark Kilgard [7] introduced the trick of using the w value of homogenous coordinates to render semi-infinite vertices. In a homogenous coordinates system, we represent a point or vector as (x, y, z, w), with w being the fourth coordinate. For points, w is equal to 1.0. For vectors, w is equal to 0.0. The homogeneous notation is extremely useful for transforming both points and vectors. Since translation is only meaningful to points and not vectors, the value of w plays an important role in transforming only points and not vectors. This can be easily deduced, since the translation values of a transformation matrix are on either the fourth column or the fourth row, depending on the matrix convention. By setting the w value of the infinity-bound vertices to 0.0, we change the homogenous representation from that of a 3D point to a 3D vector. The rendering of a vector (w = 0.0) in clip space would be semi-infinite. It is important to note that we should only set the w values to 0.0 before transformation to clip space. Technically, this implies that we want to render the vertex as a 3D vector of the form (x, y, z, 0).

Rendering such a vertex is possible in Direct3D's fixed-function pipeline by using the flexible vertex format `D3DFVF_XYZRHW`. This is because when we set the flexible vertex format to `D3DFVF_XYZRHW`, we are bypassing Direct3D's transformation and lighting pipeline. Our program becomes responsible for transforming and lighting the vertices, as Direct3D would merely pass the input to the hardware for rasterization. From the DirectX 8.1 documentation:

> "If you include the D3DFVF_XYZRHW flag in your vertex format description, you are telling the system that your application is using transformed and lit vertices. This means that Microsoft® Direct3D® doesn't transform your vertices with the world, view, or projection matrices, nor does it perform any lighting calculations. It assumes that your application has taken care of these steps. This fact makes transformed and lit vertices common when porting existing 3D applications to Direct3D. In short, Direct3D does not modify transformed and lit vertices at all. It passes them directly to the driver to be rasterized.
>
> "The system requires that the vertex position that you specify be already transformed. The x and y values must be in-screen coordinates, and z must be the depth value of the pixel to be used in the z-buffer. Z values can range from 0.0 to 1.0, where 0.0 is the closest possible position to the user and 1.0 is the farthest position still visible within the viewing area. Immediately following the position, transformed and lit vertices must include a reciprocal of homogeneous W (RHW) value. RHW is the reciprocal of the W coordinate from the homogeneous point (x,y,z,w) at which the vertex exists in projection space. This value often works out to be the distance from the eyepoint to the vertex, taken along the z-axis."

From the above explanation, RHW means reciprocal homogeneous w value (1/w), which is the result of normalizing the w component. However, we cannot explicitly represent our vertices as infinite. We can get around this by setting the w component of the vertex to 0.0 before applying the clip space transformation (world*view*projection). Originally, the homogenization process during perspective projection transformation would divide all four components by the w component in order to normalize the w component to 1.0 (Moller and Haines [11]). However, by using the `D3DFVF_XYZRHW` vertex format, we must implement the homogenization process ourselves in order to complete the transformation to clip space. Next, we need to manually map the x and y values from clip space to screen coordinates. A rectangular clipping volume defines the clip space with an x-coordinate range of [–1.0...1.0] and a y range of [–1.0...1.0]. We need to map to screen coordinates that range from (0.0, 0.0) at the top-left corner to (screen horizontal resolution, screen vertical resolution) at the bottom right corner. The screen coordinates can finally be passed on to Direct3D for rasterization.

As described above, rendering vectors (w=0) using the fixed-function pipeline can be both error-prone and inefficient. A lot of geometry transformation and mapping needs to be done, and the computation load shifts toward the CPU while the graphics hardware's powerful arithmetic processors lay wasted. Ideally, the infinite extrusion of geometries can be done more easily in a vertex program, since we are already transforming the vertices in a vertex shader. In fact, this is done in the InfiniteGPU sample using a simple vertex program that is discussed in the "Implementation on GPU (Shaders)" section. Note that we do not need to light the vertices, since surface color values have no meaning for the shadow volume polygons. The w-coordinate demo at nVidia [28] is a simple program for visualizing the rendering of vertices with different w-coordinate values.

Shadow Volume Capping

Remember that shadow volume capping is only necessary for the depth-fail technique, and hence the DepthFailCPU sample's `BuildShadowVolume()` function implements additional code to form the capping. The purpose of doing shadow volume capping is to ensure that our shadow volume is closed, and it must be closed even at infinity. As discussed previously, the extrusion of geometries for point light sources and infinite directional light sources is different. Point light sources would extrude the silhouette edges exactly point for point while infinite directional light sources would extrude all silhouette edges to a single point at infinity. This would mean that the shadow volume's back capping would be redundant for infinite directional light sources, as it is already closed at the back by the point at infinity. We shall tackle the more complicated case of point light sources, which require both front and back capping, regardless of the distance of extrusion.

The ideal time to generate the front and back capping is during silhouette generation, since we are already generating the angles between the incident light ray vector and the face normal. The creation of the capping is the main difference between the `BuildShadowVolume()` functions in the DepthPassCPU and DepthFailCPU samples. For the front capping, we just need to make use of all the light-facing geometries as capping. For the back capping, a straightforward method would be to extrude the geometries facing away from the light and use these extruded geometries to form the capping. We could also duplicate the front capping, extrude it, and reverse the vertex ordering to form the back capping. The `BuildShadowVolume()` function in DepthFailCPU forms the back capping by creating a triangle-fan from the extruded silhouette edges. This method results in less geometry and helps improve the rendering of the shadow volume.

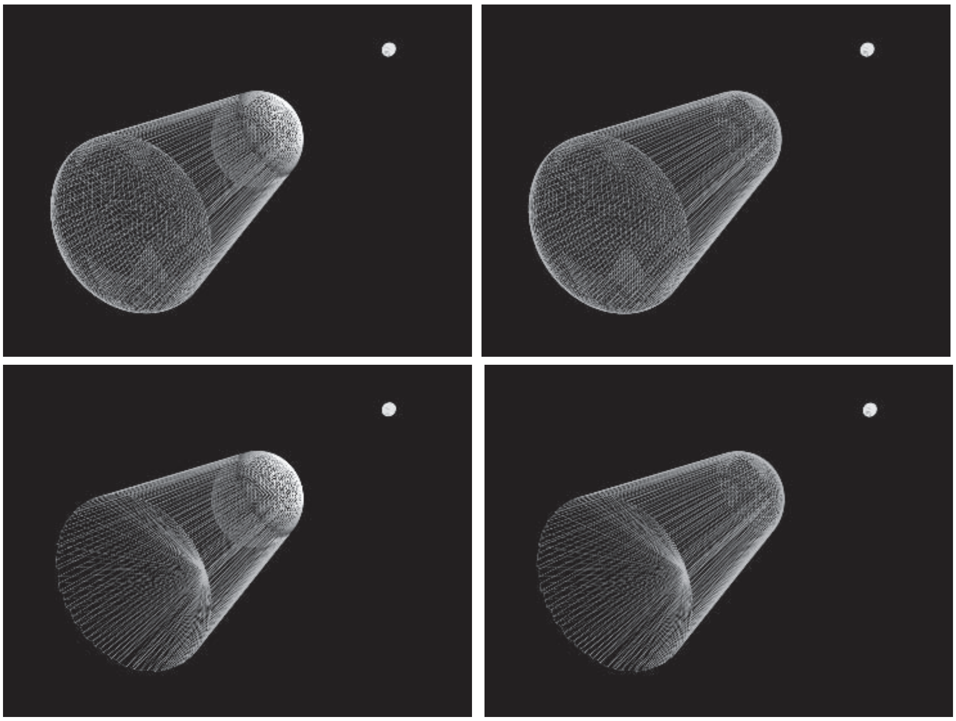


Figure 16: Front and back capping to create closed shadow volumes

Figure 16 shows two sets of images employing different geometries to close the shadow volume. The first row depicts a closed shadow volume formed by a front and back capping that reuses the occluder's light-facing geometries. The second row shows a closed shadow volume with a front capping that reuses light-facing geometries of the occluder and a triangle-fan back capping constructed from extruded silhouette edges. The triangle-fan back capping results in less geometry and hence requires less memory and rendering resources.

While optimizing the back capping with a triangle-fan is trivial, the same cannot be said for the front capping. This is due to the fact that the occluder's self-shadowing totally depends on the accuracy of the front capping. To be precise, the most accurate front capping is one that is created from the actual front-facing geometries of the occluder. Such a front capping would ensure that any grooves or knobs on the occluder's surface would be correctly self-shadowed. When the occluder is too small for any self-shadowing to be noticeable (Diablo and RTS-style games) or when the light-facing side of a static occluder is generally flat, we can make use of triangle strips formed using the silhouette edges to cut down on the amount of front-capping geometry.

Depth-pass Stenciling Operations (DepthPassCPU)

We discuss all the stenciling operations and the corresponding sample code for the depth-pass technique in this section. The reader should refer to the DepthPassCPU sample for the discussion. Going back to step 4 of the "How It Is Done" section, we have to set up the stencil operations before proceeding to render the shadow volume in order to fill the stencil buffer with the correct shadow counts. Let's examine the `RenderShadowVolume()` function in the DepthPassCPU sample.

```
HRESULT CDepthPass::RenderShadowVolume()
{
  // Disable z-buffer writes (note: z-testing still occurs) and enable
  // stencil buffering
  m_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, TRUE );

  // Don't bother with interpolating color
  m_pd3dDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_FLAT );

  // Set up stencil compare fuction, reference value, and masks.
  // Stencil test passes if ((ref & mask) cmpfn (stencil & mask)) is true.
  // Note: since we set up the stencil test to always pass, the STENCILFAIL
  // renderstate is really not needed.
  m_pd3dDevice->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_ALWAYS );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_KEEP );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP );

  // Sets up stencil reference value and stencil masks
  m_pd3dDevice->SetRenderState( D3DRS_STENCILREF, 0x1 );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILMASK, 0xffffffff );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILWRITEMASK, 0xffffffff );

  // Increment stencil buffer value if depth test passes
```

```
m_pd3dDevice->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_INCR );

// Show shadow volume front faces?
if ( m_bShowShadowVolFrontFace )
{
  m_pd3dDevice->SetMaterial( &m_ShadowVolFrontFaceMaterial );
  m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
  m_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR );
  m_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_DESTALPHA );
}
else
  m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x00000000 );

// Draw front side of shadow volume in stencil/z only
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matObject);
m_pShadow->RenderShadowVolume( m_pd3dDevice );
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matObject2);
m_pShadow2->RenderShadowVolume( m_pd3dDevice );

// Now reverse cull order so back sides of shadow volume are written.
m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CW );

// Decrement stencil buffer value if depth test passes
m_pd3dDevice->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_DECR );

// Show shadow volume back faces?
if ( m_bShowShadowVolBackFace )
{
  m_pd3dDevice->SetMaterial( &m_ShadowVolBackFaceMaterial );
  m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x0000000F );
  m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
  m_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR );
  m_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_DESTALPHA  );
}
else
 m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x00000000 );

// Draw back side of shadow volume in stencil/z only
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matObject);
m_pShadow->RenderShadowVolume( m_pd3dDevice );
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matObject2);
m_pShadow2->RenderShadowVolume( m_pd3dDevice );

```

```
68    // Restore render states
69    m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x0000000F );
70    m_pd3dDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
71    m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
72    m_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
73    m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, FALSE );
74    m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
75
76    return S_OK;
77  }
```

Note that prior to the calling of the `RenderShadowVolume()` function, the depth buffer had already been filled with the appropriate depth values during the rendering pass of step 1, as discussed in the "How It Is Done" section.

Lines 5 and 6 disable writing to the depth buffer and enable stencil testing. The code within lines 15 to 25 sets up the stencil operations prior to rendering the shadow volume. Line 15 forces the stencil test to always pass, while lines 16 and 17 instruct Direct3D to retain the stencil values in case of depth fail or stencil test fail. Line 20 sets the stencil reference value to 1. Lines 21 and 22 set the stencil comparison mask and write mask to include every bit. The following is the complete test function employed by the Direct3D API during stencil tests:

```
(StencilRef & StencilMask) CompFunc (StencilBufferValue & StencilMask)
```

For more information on the other uses of stencil buffers, please refer to [3].

Line 25 tells Direct3D to increment the stencil value if both stencil and depth tests pass. The stencil test has already been set to always pass in line 15, so it is really only the depth test in question here. Lines 28 to 36 either disable the color writes to frame buffer or set up alpha blending to reveal the front faces of the shadow volumes. Next, we proceed to render the shadow volumes of our occluders in lines 39 through 42. This is in agreement with the first step of the depth-pass algorithm presented in the "Depth-pass (z-pass)" section.

Following the second step of the depth-pass algorithm, line 45 reverses the culling mode so that we can start drawing the back faces of the shadow volume. Line 48 sets the stencil operation to decrement the stencil values if the stenciling and depth tests pass. Again, the stenciling test always passes, and it is only the depth test that we are really testing against. Lines 51 to 60 either disable the color writes to frame buffer or set up alpha blending to reveal the back faces of the shadow volumes. Lines 63 to 66 draw the shadow volumes again with the culling reversed. Lines 69 through 74 restore the render states to their original settings. That completes the rendering of the shadow volumes for the depth-pass algorithm in the DepthPassCPU sample.

We should note that the sequence of the depth-pass algorithm that we are employing is really inconsequential. This is because at lines 25 and 48, we set the stencil increment and decrement operation as wrapping, which has been available since DirectX 6. Thus, we can start with either incrementing or decrementing the stencil values. This is because the stencil buffer can only contain values from 0 to 2^n–1, where n is the stencil bit depth. When the maximum stencil value is reached, the stencil value is wrapped to 0 with the next increment operation. Similarly, when the minimum stencil value of 0 is reached, the next decrement operation wraps the stencil to 2^n–1. This ensures that the shadow volume counting will not be thrown off balance due to saturation at maximum or minimum stencil value. This guarantees that we leave behind non-zero stencil values for pixels with unbalanced shadow volume entry and exit counts. It also means that the bit depth of the stencil buffer is not important to us, as a 2-bit stencil buffer (if one exists) will work as well as an 8-bit stencil buffer. If we opt for stencil value clamping (e.g., setting `D3DRS_STENCILPASS` to `D3DSTENCILOP_INCRSAT` to clamp to the maximum value), we will lose track of the correct shadow volume count if the stencil value gets saturated at 2^n–1, and the stencil values will be incorrect. Let's move on to adding the shadows into the scene now!

The only thing we need to do now is make use of the stencil values and shade the appropriate pixels in the scene, as described by step 6 in the "How It Is Done" section. This is done with the `DrawShadow()` function in the DepthPassCPU sample.

```
01  HRESULT CDepthPass::DrawShadow()
02  {
03    // Set renderstates: disable z-buffering, enable stencil, and turn on
04    // alpha blending
05    m_pd3dDevice->SetRenderState( D3DRS_ZENABLE, FALSE );
06    m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, TRUE );
07    m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
08    m_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCALPHA );
09    m_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
10
11    // Only write where stencil val >= 1
12    m_pd3dDevice->SetRenderState( D3DRS_STENCILREF, 0x1 );
13    m_pd3dDevice->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_LESSEQUAL );
14    m_pd3dDevice->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_KEEP );
15
16    // Draw a big, gray square
17    m_pd3dDevice->SetVertexShader( D3DFVF_BIGSQUAREVERTEX );
18    m_pd3dDevice->SetStreamSource( 0, m_pBigSquareVB, sizeof(BIGSQUAREVERTEX) );
19    m_pd3dDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 );
20
21    // Restore render states
22    m_pd3dDevice->SetRenderState( D3DRS_ZENABLE, TRUE );
23    m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, FALSE );
24    m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
25
26    return S_OK;
27  }
```

To shade the pixels with non-zero stencil values, we first disable depth testing in line 5 and enable stencil testing in line 6. Alpha blending with the blending parameters is set up in lines 7 through 9. Next comes the critical stenciling operations set up in lines 12 through 14. We use a reference stencil value of 1 and do a "less than or equal" comparison with the value in the stencil buffer for the pixel in question. This means that for the stencil test to pass, the value from the stencil buffer must be at least equal to or greater than the reference value of 1, which is in agreement with the depth-pass algorithm.

Lines 17 through 19 draw the quad that covers the entire screen, and the alpha blending will kick in to shade a pixel on-screen that passes the stencil test. Lines 22 through 24 would restore the original render states. This concludes the DepthPass-CPU sample.

In the next section, we look at the stenciling operations of the depth-fail technique, which are slightly different from that of the depth-pass technique discussed here.

Depth-fail Stenciling Operations (DepthFailCPU)

The reader should refer to the DepthFailCPU sample for this section. For a recap of the depth-fail algorithm, please refer to the two-step depth-fail algorithm in the "Depth-fail (z-fail)" section. Let's look into the `RenderShadowVolume()` function in the DepthFailCPU sample to see how the stencil operations are set up.

```
HRESULT CDepthFail::RenderShadowVolume()
{
  // Disable z-buffer writes, z-testing still occurs, enable stencil-buffer
  m_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, TRUE );

  // Don't bother with interpolating color
  m_pd3dDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_FLAT );

  // (StencilRef & StencilMask) CompFunc (StencilBufferValue & StencilMask)
  m_pd3dDevice->SetRenderState( D3DRS_STENCILREF, 0x1 );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILMASK, 0xffffffff );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILWRITEMASK, 0xffffffff );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_ALWAYS );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_KEEP );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP );

  // Back face depth test fail -> Incr
  m_pd3dDevice->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_INCR );

  // Set lower z-bias for shadow volumes
  m_pd3dDevice->SetRenderState( D3DRS_ZBIAS, 0 );

  // Now reverse cull order so back sides of shadow volume are written.
  m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CW );

  // Show shadow volume back faces?
  if ( m_bShowShadowVolBackFace )
```

```
{
  m_pd3dDevice->SetMaterial( &m_ShadowVolBackFaceMaterial );
  m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
  m_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR );
  m_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_DESTALPHA  );
}
else
  m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x00000000 );

// Draw back side of shadow volume
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matObject);
m_pShadow->RenderShadowVolume( m_pd3dDevice );
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matObject2);
m_pShadow2->RenderShadowVolume( m_pd3dDevice );

// Now reverse cull order so front sides of shadow volume are written.
m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );
// Reverse the stencil op for back face
m_pd3dDevice->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_DECR );

// Show shadow volume front faces?
if ( m_bShowShadowVolFrontFace )
{
  m_pd3dDevice->SetMaterial( &m_ShadowVolFrontFaceMaterial );
  m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x0000000F );
  m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
  m_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR );
  m_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_DESTALPHA );
}
else
  m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x00000000 );

// Draw front side of shadow volume
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matObject );
m_pShadow->RenderShadowVolume( m_pd3dDevice );
m_pd3dDevice->SetTransform( D3DTS_WORLD, &m_matObject2 );
m_pShadow2->RenderShadowVolume( m_pd3dDevice );

// Restore render states
m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x0000000F );
m_pd3dDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
m_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, FALSE );
```

```
72    m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
73
74    return S_OK;
75  }
```

Lines 4 through 16 basically do the same setting up as the `RenderShadowVolume()` function in the DepthPassCPU sample. Line 19 sets the stencil operation to increment the stencil count if the stencil test passes while the depth test fails. Note that the stencil test has been set to always pass in line 14, and hence only the depth test matters here. Incrementing the stencil values with the failure of the depth test is in agreement with the depth-fail algorithm presented in the "Depth-fail (z-fail)" section.

Line 22 sets the z-bias level for the rendering of the shadow volume to a level of 0 to force it to render behind the actual occluder geometries. Let's ignore this for the time being; we shall return to the z-bias issue soon in the "Rendering Shadow Volume Capping" section.

In accordance with the depth-fail algorithm, we reverse the culling mode in line 25 to render the back faces of the shadow volume. The code from lines 28 through 36 either set up alpha blending to expose the shadow volume or disable color writes, depending on whether the program is showing the shadow volume. We then draw the back faces with the code from lines 39 to 42. The first step of the depth-fail algorithm is complete.

Next, we reverse the culling mode to draw the front faces at line 45 and set the stencil operation to decrement the stencil values with depth test failure at line 47. Lines 50 through 59 do the necessary settings, depending on whether the program is exposing the front faces of the shadow volume to the viewer. Lines 62 through 65 draw the occluder's front faces. The render states are restored with the code from lines 68 through 72. Note that the same logic of applying wrapping, instead of clamping, and stencil incrementing and decrementing operations applies for the depth-fail `RenderShadowVolume()` function described above. The `DrawShadow()` function that shades the pixels in shadows are similar for both the DepthPassCPU and DepthFailCPU samples.

The stencil operations described in this section are one-sided in nature and hence require two passes to render the shadow volume. Newer graphics cards that support DirectX 9 provide new two-sided stencil operations that allow the rendering of shadow volumes in a single pass. All the appropriate front- and back-face stencil operations fill up the stencil buffers in a single rendering pass. For more details on the two-sided stencil mode, please refer to the section titled "DirectX 9 HLSL Samples." We now continue with the DepthFailCPU sample by tackling the last tricky issue of rendering the shadow volume capping for the depth-fail technique (remember the z-biasing at line 22?).

Rendering Shadow Volume Capping

In this section, we discuss the business of rendering the shadow volume, which includes capping for the depth-fail technique. The rendering of the shadow volume for depth-pass is trivial, and there is no need for any special setup. However, the same cannot be said for the depth-fail technique due to capping, which is often coplanar to the occluder's geometries. The reader should refer to the DepthFailCPU sample for this section.

Referring back to the "Shadow Volume Capping" section, when reusing the front-facing geometries of the occluder, we should be extremely careful with regard to rendering the shadow volume, since the shadow volume's front-capping geometries are actually coplanar with the occluder's front-facing geometries. More often than not, precision problems will cause the front-capping geometries of the shadow volume to be rendered in front of the occluder's front-facing geometries, causing the entire occluder to be engulfed in its own shadow volume. We cannot reverse the order of rendering, hoping that the shadow volume would render behind the occluder's geometries, as we need the depth buffer to be filled with the correct depth values for testing, as presented by the steps in the "How It Is Done" section.

To achieve our goal of rendering the front capping behind the occluder, we can either tweak the projected depth values of the front-capping geometry or make use of polygon offsetting support from graphics APIs. Eric Lengyel [26] described how a separate projection matrix could be computed to render polygons at different depth values without altering its projected screen coordinates of texture mapping perspective. Tweaking the projected depth values on a per-object basis can provide fine control and sometimes better performance. But the implementation is also comparatively more involved. Choosing the appropriate camera space offset can also be messy due to the non-uniform nature of depth buffer precision for perspective viewing [11]. Depth precision can become horrendously poor with increasing distance from the camera and cause polygons that are close together, in terms of depth values, to be rendered incorrectly. For example, a piece of tapestry on the wall may get rendered behind the wall in several places due to poor depth precision as the viewer moves farther away. Depth precision errors are usually accompanied by the flickering of polygons, which is a problem commonly known as z-fighting. The camera space offset used for tweaking projected depth values needs to be adjusted accordingly to account for this non-linear behavior.

Alternatively, we can simply make use of Direct3D's depth bias capability to render the front capping properly without worrying about anything else. In Direct3D, depth values of fragments generated while rasterizing a primitive can be biased to help mitigate z-fighting issues when drawing coplanar polygons. The `D3DRS_ZBIAS` flag in Direct3D's `D3DRENDERSTATETYPE` can be used to bias the occluder's front-facing geometries so that they are more likely to be rendered in front of its shadow volume front capping.

```
// Set higher z-bias for occluders
m_pd3dDevice->SetRenderState(D3DRS_ZBIAS, 1);
// Render occluder here
    .
    .
    .
// Set lower z-bias for shadow volumes
m_pd3dDevice->SetRenderState(D3DRS_ZBIAS, 0);
// Render shadow volume here
```

Simply setting the D3DRS_ZBIAS values before rendering the two groups of coplanar geometries, as shown in the code above, would achieve the desired effect. We set the z-bias flag value to a higher value for the occluder's geometries and a lower value for its shadow volume. This ensures that the front capping of the shadow volume is rendered behind the occluder's front-facing geometries. This completes the entire depth-fail algorithm, and the stencil buffer would now be filled with the correct stencil values that are needed for comparison in order to shade the pixels in shadow. The pixel shading is done by the DrawShadow() function, which is similar to the one used in the DepthPassCPU sample. With that, we conclude the DepthFailCPU sample.

As a side note, DirectX 9 [12] is able to distinguish between legacy devices that expose the D3DRS_ZBIAS and those that can perform true slope-scale-based depth bias. Two new floating-point values, D3DRS_DEPTHBIAS and D3DRS_SLOPESCALEDEPTHBIAS, are used to compute the offset. The offset is added to the fragment's interpolated depth value to produce the final depth value that is used for depth testing. The new caps for these two values are D3DPRASTERCAPS_DEPTHBIAS and D3DPRASTERCAPS_SLOPESCALEDEPTHBIAS. D3DRS_DEPTHBIAS is used in the "DirectX 9 HLSL Samples" section.

This ends our discussion of the implementation of both the depth-pass (DepthPassCPU sample) and depth-fail (DepthFailCPU sample) algorithms on the CPU. Next up, we dive straight into the methodology and implementation of the depth-fail algorithm using the programmable graphics pipeline!

Implementation on GPU (Shaders)

The programmable graphics pipeline (commonly known as shaders) is fast becoming a standard capability of newer graphics hardware. In fact, you would be hard-pressed to find a new graphics card without minimal vertex and pixel shader support after the introduction of the ATI Radeon 8000 series and the nVidia GeForce3 series. The programmable graphics pipeline promises great potential and flexibility for graphics programmers to achieve effects at a level of realism never before dreamed possible. Different lighting methods, texturing, and geometry manipulation are now possible with the use of vertex and pixel shaders. Rendering engines are no longer bound by the limitations imposed by fixed-function pipelines.

With this explosion of graphics shaders, we need to look at the effects that we have achieved with the fixed-function pipeline in the past and see if it is possible to do it more efficiently and faster with the programmable graphics pipeline. That is exactly what we are going to do — by implementing the depth-fail stencil shadow volume algorithm using vertex shaders. The reader should note that implementing shadow volume in shaders may or may not improve shadow volume performance. We discuss the pros and cons of using shaders for shadow volumes in the “Better with Shaders?” section after we have gone through its implementation.

How It Is Done

For stencil shadow volume implementation using shaders, the general steps presented in the previous “How It Is Done” section for implementation on the CPU still applies. The main difference lies in the execution of the silhouette calculation. When we talk about implementing stencil shadow volume using shaders, we are actually referring mainly to the offloading of the silhouette computation from the CPU to the GPU. This means that we do not compute the silhouette of the occluder in our program; instead, this is done by a vertex program running on the GPU that is fed with the appropriate preprocessed occluder geometry and vertex shader constants. Let’s list the general steps for implementing shadow volumes using vertex shaders:

1. Preprocessing of occluder geometry. Insert degenerate quads into edges shared by exactly two triangles.
2. Render the scene to fill the depth buffer with the correct z-values.

3. Select a light source. Clear the stencil buffer if this is the first light.
4. Set up the stencil operations, update vertex shader constants, and render the shadow volumes using the vertex shader.
5. Repeat steps 3 to 4 for all the selected lights in the scene.
6. Using the stencil buffer, do a lighting pass (or make it a tone darker) to shade the pixels that correspond to non-zero stencil values.

As far as the scene rendering pass (step 2), stencil operations (step 4), and lighting pass (step 6) are concerned, there is little difference from the CPU implementations. The main difference lies in the preprocessing of the occluder geometry in step 1 and the setting of the vertex shader constants and rendering in step 4. In a nutshell, we preprocess the occluder's mesh in such a way that when it is fed into the graphics pipeline, the vertex shader deforms it into the shadow volume that we desire. In the following sections, we go through the steps and peruse the code that comes with the samples FiniteGPU and InfiniteGPU. As the name implies, the FiniteGPU sample demonstrates finite shadow volume extrusion using vertex shaders, while the InfiniteGPU sample implements infinite shadow volume extrusion through homogenous coordinates discussed in the "Forming the Shadow Volume" section. Both samples are based on DirectX 8.1. The section titled "DirectX 9 HLSL Samples" discusses two similar samples that are based on DirectX 9. Note that both the FiniteGPU and InfiniteGPU samples implement the depth-fail stencil shadow volume algorithm for good reasons, which we find out about soon.

Preprocessing of Data

The very first step to implementing shadow volumes in shaders is to preprocess the original geometry into a form usable by the vertex shader. Remember that during the creation of the shadow volume, we need to create new geometry data such as the extruded vertices and the faces that form the sides and capping of the volume. With vertex shaders, this is not possible, as the current generation of programmable graphics pipeline does not allow for the creation of new vertex data. It is strictly a one vertex in and one vertex out pipeline. This limitation is probably not going to go away in the foreseeable future. Hence, we need to overcome it by preprocessing the source geometry data in such a way that makes it possible to form a shadow volume in any direction without creating new geometries. Note that both FiniteGPU and InfiniteGPU use the same preprocessing function.

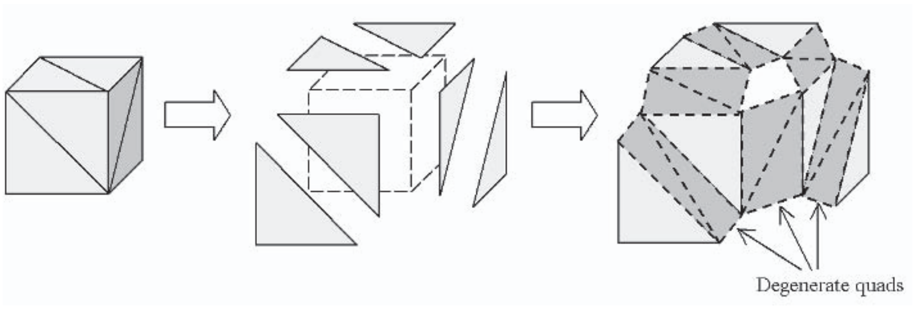


Figure 17: Insertion of degenerate quads during preprocessing

Figure 17 depicts the preprocessing of the source geometry that forms a cube; it only shows the front faces for simplicity. The shared edges of the faces are filtered out, and a degenerate quad is inserted to replace each shared edge. Degenerate quads are formed by triangles with zero area. The two edges that form the opposing sides of each degenerate quad have the same positional values (same x, y, and z coordinates) but different face normals. In the FiniteGPU and InfiniteGPU sample, preprocessing is done by the member function `Create()` of the `CShadow` class. Let's briefly run through the preprocessing algorithm:

1. Step through all the faces in the source mesh.
2. Compute face normal for each face.
3. Step through the three edges of each face.
    a. Insert edge into a list for checking.
    b. If edge already exists in the list (shared edge found):

        i. If normals of faces sharing the edge are *not* parallel, insert degenerate quad into the output list.
        ii. Else, only insert the shared edge into the output list.
    c. Remove the current edge and any shared edge from the checklist.
4. Create index and vertex buffers with only position and normal information from the output list.
5. If there are any vertices left in the checklist, the source mesh is not a closed volume since all edges should be shared in a closed volume mesh.

Note that the above algorithm also requires the source mesh to be a closed volume, which is the same requirement imposed in the CPU determination of silhouette edges presented earlier. The code in the `Create()` function follows the above steps in preprocessing the source mesh data. The reader should study the code to get a better understanding of the preprocessing algorithm. The algorithm implemented emphasizes clarity over efficiency. The general implementation in `Create()` does not handle welded meshes and is similar to the preprocessing algorithm used in the ATI demos [18, 19]. Many other more efficient algorithms do exist.

A major problem with preprocessing geometries for shader implementation of shader volume is the large number of vertices that it generates. Typical final preprocessed meshes contain around three times more vertices compared with the source meshes. This is a major problem for shader implementations of shadow volume, as we are stretching the vertex throughput of the GPU during the rendering of the shadow volume. We discuss this problem in more detail in the "Better with Shaders?" section. For now, let's implement a simple optimization to try to cut down the number of vertices generated during preprocessing.

Notice that we do not indiscriminately insert degenerate quads into every shared edge in the preprocessing algorithm. Doing so would be very inefficient, and the final preprocessed polygon count would explode. A simple optimization would be to test whether a shared edge would have a good chance of becoming a silhouette edge. If a shared edge has almost zero chances of becoming a silhouette edge, then there is really no need for the insertion of a degenerate quad to replace that edge. A simple way to determine the chances of an edge forming part of a silhouette is to test the parallelism of the normals of the faces that share it. If the two faces have normals that are almost parallel in the same direction, the shared edge lies in a flat surface and would have little chance of becoming part of a silhouette. In fact, if the face normals are exactly parallel, it is not possible for the shared edge to be part of any silhouettes. Thus, a simple dot product of the two face normals will suffice for such a test. If the dot product result is 1.0, the edge is left untouched, as it cannot possibly become a silhouette edge. In actual implementation, we can further cut down the number of vertices generated by testing the dot product result against values such as 0.9 or 0.8, which would then include surfaces that are quite flat. Figure 18 shows that this simple optimization halves the number of degenerate triangles needed for the front faces of our simple cube from 12 (Figure 17) to 6.

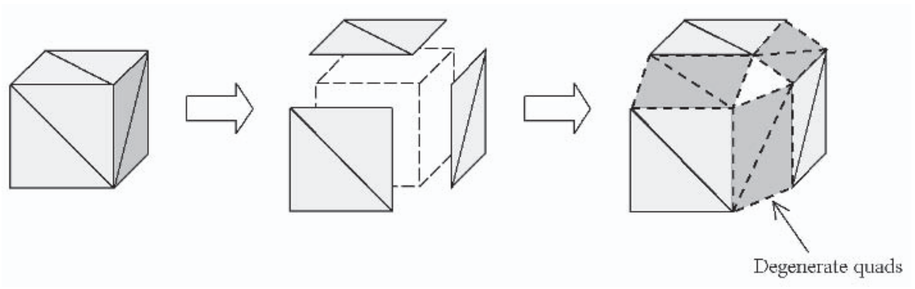


Figure 18: Shared edges on flat surfaces need not be replaced by degenerate quads.

An obvious point to note here is that such preprocessing should focus on minimizing the final geometry count instead of processing speed. In fact, the preprocessing should be done entirely offline. In the next section, we look at how these degenerate quads help form shadow volumes on the hardware without the need to create new vertices during silhouette computation.

Forming Shadow Volume in Shaders

Once we have finished processing the source mesh into the shadow volume mesh with degenerate quads inserted, we can render the geometries as they are with our vertex shader code running. The creation and rendering of the shadow volume are actually merged into one step when the shadow volume is being created as we render it! Let's consider the simplified case of a single edge shared by two faces in the following figure.

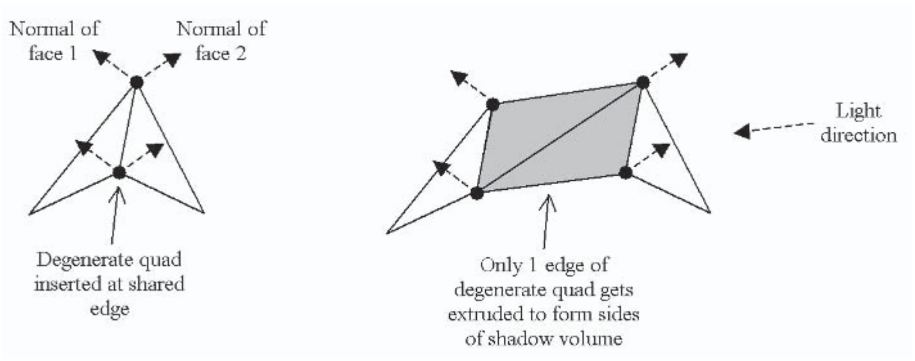

Normal of face 1
Normal of face 2
Degenerate quad inserted at shared edge
Only 1 edge of degenerate quad gets extruded to form sides of shadow volume
Light direction

Figure 19: Sides of shadow volume formed using degenerate quads

On the left side of Figure 19, we can see two faces with a common shared edge that has been replaced by a degenerate quad. The two opposing edges of the degenerate quad contain the face normal of the face to which it belongs. Next, assume that the direction of a light source is as shown on the right. Face 1 is back facing the light source, while face 2 is front facing the light source. Hence, the shared edge becomes part of the silhouette, as seen from the position of the light source. Vertices that are facing away from the light source would then be extruded in the direction of the light's ray, as shown on the right side of Figure 19. This means that the opposing edges of the degenerate quad are stretched out to form a normal quad with a non-zero area. This is exactly how the sides of the shadow volume are formed!

Also note that the extruded face 1 now becomes the back capping, while the untouched face 2 automatically acts as the front capping. Hence, for shader implementation in the FiniteGPU and

InfiniteGPU samples, it only makes sense to implement the stencil operations according to the depth-fail stencil algorithm, as the shadow volume capping already exists!

From this point onward, we are going into the implementations of the FiniteGPU and InfiniteGPU samples. The two samples are differentiated by the vertex shader constants setup and the vertex shader code they execute. This means that of the six steps presented in the last "How It Is Done" section, only step 4 is different between the two samples.

Vertex Shader Implementation (FiniteGPU)

After we have preprocessed our occluder's meshes, rendered the scene, and selected a light source as dictated by steps 1 through 3 presented earlier in the "How It Is Done" section, it is time to render the shadow volume geometry (step 4). However, before we can do that, we need to set up the vertex shader in Direct3D and also update the vertex shader constants.

```
HRESULT CShadow::InitDeviceObjects( LPDIRECT3DDEVICE8 pd3dDevice )
{
  // vertex shader declaration
  DWORD dwDecl[] =
  {
    D3DVSD_STREAM(0),
    D3DVSD_REG(0, D3DVSDT_FLOAT3 ),  // Vertex position in input reg 0
    D3DVSD_REG(1, D3DVSDT_FLOAT3 ),  // Face normal in input reg 1
    D3DVSD_END()
  };

  // loads a *.vso binary file, already compiled with NVASM and creates a
  // vertex shader
  if ( FAILED( CreateVSFromCompiledFile( pd3dDevice, dwDecl,
        "Shaders/VertexExtrusion.vso", &m_dwVertexShader ) ) )
    return E_FAIL;

  return S_OK;
}
```

We declare the vertex shader with the vertex position and face normal lined up as input registers 0 and 1. The VisualStudio.NET project files for the FiniteGPU sample have been set to compile the vertex shader code using nVidia's NVASM [17] into the .vso binary, which is fed into the `CreateVSFromCompiledFile()` function taken from Wolfgang F. Engel [16]. Next, we shall take a look at the `RenderShadowVolume()` function before going into the vertex shader constants setting.

```
HRESULT CDepthFail::RenderShadowVolume()
{
  // Disable z-buffer writes, z-testing still occurs, enable stencil buffer
  m_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, TRUE );

  // Don't bother with interpolating color
  m_pd3dDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_FLAT );

  // (StencilRef & StencilMask) CompFunc (StencilBufferValue & StencilMask)
  m_pd3dDevice->SetRenderState( D3DRS_STENCILREF, 0x1 );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILMASK, 0xffffffff );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILWRITEMASK, 0xffffffff );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_ALWAYS );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_KEEP );
  m_pd3dDevice->SetRenderState( D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP );

  // Back face depth test fail -> Incr
  m_pd3dDevice->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_INCR );

  // Set lower z-bias for shadow volumes
  m_pd3dDevice->SetRenderState(D3DRS_ZBIAS, 0);

  // Now reverse cull order so back sides of shadow volume are written.
  m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CW );

  // Show shadow volume back faces?
  if ( m_bShowShadowVolBackFace )
  {
    m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
    m_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR );
    m_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_DESTALPHA  );
  }
  else
    m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x00000000 );
```

```

// Set up shader constants and render the shadow for object 1
m_pShadow->SetShaderConstants( &m_pLight, &m_matObject, &m_matView,
                                &m_matProject );
m_pShadow->RenderShadow();

// Set up shader constants and render the shadow for object 2
m_pShadow2->SetShaderConstants( &m_pLight, &m_matObject2, &m_matView,
                                &m_matProject );
m_pShadow2->RenderShadow();

// Now reverse cull order so front sides of shadow volume are written.
m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_CCW );

// Reverse the stencil op for front face
m_pd3dDevice->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_DECR );

// Show shadow volume front faces?
if ( m_bShowShadowVolFrontFace )
{
  m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x0000000F );
  m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, TRUE );
  m_pd3dDevice->SetRenderState( D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR );
  m_pd3dDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_DESTALPHA );
}
else
  m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x00000000 );

// Set up shader constants and render the shadow for object 1
m_pShadow->SetShaderConstants( &m_pLight, &m_matObject, &m_matView,
                                &m_matProject );
m_pShadow->RenderShadow();

// Set up shader constants and render the shadow for object 2
m_pShadow2->SetShaderConstants( &m_pLight, &m_matObject2, &m_matView,
                                &m_matProject );
m_pShadow2->RenderShadow();

m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, 0x0000000F );
m_pd3dDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_GOURAUD );
m_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, TRUE );
m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, FALSE );
m_pd3dDevice->SetRenderState( D3DRS_ALPHABLENDENABLE, FALSE );
```

```
79
80   return S_OK;
81 }
```

The setup code from lines 4 to 35 is similar to that in the DepthFailCPU sample discussed earlier. The difference comes in lines 38 through 45 where we are required to set the shader constants prior to rendering the preprocessed shadow volume geometry using our vertex program. The same goes for the second shadow volume rendering pass in lines 65 through 72. Note that two-sided stenciling, as presented in the "DirectX 9 HLSL Samples" section, would work as well here to render the shadow volume in a single pass. In fact, the DirectX 9 samples in that section implement both stencil modes for comparison. Takashi Imagire [14] presented a depth-pass shadow volume implementation that utilizes two-sided stenciling and vertex shaders. Let's take a look at the code of the `SetShaderConstants()` function before going into the vertex shader code.

```
01 void CShadow::SetShaderConstants( const D3DLIGHT8* pLight,
02                                   const D3DXMATRIX* matWorld,
03                                   const D3DXMATRIX* matView,
04                                   const D3DXMATRIX* matProj )
05 {
06   D3DXMATRIX matClip, matInvWorld;
07   D3DXMatrixMultiply( &matClip, matWorld, matView );
08   D3DXMatrixMultiply( &matClip, &matClip, matProj );
09   D3DXMatrixInverse( &matInvWorld, NULL, matWorld );
10
11   D3DXVECTOR4 vConst( 0.0f, 0.0f, 0.0f, m_fExtrusionLen );
12
13   // Yellowish-green hue for drawing shadow volume if needed
14   D3DXVECTOR4 vColor( 0.3f, 0.4f, 0.2f, 0.0f );
15
16   // Light pos in world space
17   D3DXVECTOR4 objectLightPos = D3DXVECTOR4( pLight->Position.x,
18         pLight->Position.y, pLight->Position.z, 1.0f );
19
20   // Transform light pos to object space
21   D3DXVec4Transform( &objectLightPos, &objectLightPos, &matInvWorld );
22
23   // Set the shader constants
24   m_pd3dDevice->SetVertexShaderConstant( 0, &vConst, 1 );
25   m_pd3dDevice->SetVertexShaderConstant( 1, &objectLightPos, 1 );
26   m_pd3dDevice->SetVertexShaderConstant( 2, &matClip, 4 );
27   m_pd3dDevice->SetVertexShaderConstant( 6, &vColor, 1 );
28 }
```

The transformation matrix for clipping space is the first thing to be computed in lines 7 and 8. At line 11, we set up a vector with the w component as the member variable `m_fExtrusionLen` that defines the absolute extrusion distance. We define a vector at line 14 to hold an RGBA color value in case the program needs to expose the rendering of the shadow volume to the viewer.

The light source position is transformed from world space to object space at line 21. The reason for doing this is to allow us to compute the light ray vector in object space without the need to transform the face normal. It is obviously far more efficient to incur a one-time transformation cost for the light position, as opposed to transforming every single face normal. The vectors are lined up in the constants registers, as shown in lines 24 through 27. It is time to dive into the vertex shader code.

```
01 // c0    : 0, 0, 0, m_fExtrusionLen
02 // c1    : Light pos in object space
03 // c2-c5 : World*View*Proj matrix
04 // c6    : Color for exposing the shadow volume
05
06 vs.1.1
07 // Output diffuse color to expose shadow volume to viewer if needed
08   mov oD0, c6
09
10 // Ray from light to pt in object space
11   sub r1, v0, c1
12
13 // Normalize ray
14   dp3 r1.w, r1, r1
15   rsq r1.w, r1.w
16   mul r1, r1, r1.w
17
18 // Dot ray and normal
19   dp3 r10.w, v1, r1
```

```
20
21 // Normal faces away from light if dot result < 0.0
22   slt r10.x, r10.w, c0.x
23
24 // Extrude along ray
25   mul r10, r10.x, c0.w
26   mad r0, r1, r10.x, v0
27
28 // Transform to clip space and output pt
29   mul r4, r0.x, c[2]
30   mad r4, r0.y, c[3], r4
31   mad r4, r0.z, c[4], r4
32   add oPos, c[5], r4
```

We immediately output the diffuse color at line 8 using constant register `c6`, which was set with the RGBA color values. This can be skipped entirely if we do not want to expose the shadow volume to the viewer. Next, we compute the vector of the incident light ray at line 11 and normalize the result. The dot product of the light ray and the face normal is done at line 19, and the result is stored in the w component of `r10`. At line 22, we compare the result of the dot product with 0.0 and form a masking value using the result of this comparison. If the dot product result is less than 0.0, this means that the angle between the vectors is larger than 90 degrees (or you can also say smaller than –90 degrees), and the vertex has a face normal pointing away from the light source. For this case, the masking value is stored as 1.0 in the x component of `r10`. For the other case, whereby the dot product result is not less than 0.0, the masking value is set as 0.0.

The extrusion (or rather, the displacement) of the vertex is done in lines 25 and 26. We multiply the masking value with the extrusion distance to compute the final extrusion distance. Since the masking value can only be 0.0 or 1.0, the result of the multiplication can be either a zero or non-zero extrusion distance. Line 26 multiplies the normalized light ray vector with the extrusion distance and adds it into the vertex's position, effectively extruding the vertex in the direction of the light ray. If the masking value is 0.0, then the extrusion distance will be 0.0 and the vertex stays unchanged. Lines 29 to 32 simply transform the final vertex to clip space and send the result to the vertex position output register.

This concludes the entire implementation of the FiniteGPU sample. Next up, we look at the InfiniteGPU sample that makes use of the homogeneous coordinate system, discussed previously in the "Forming the Shadow Volume" section, to extrude shadow volumes to infinity.

Vertex Shader Implementation (InfiniteGPU)

In this section, we look into the implementation of the Infinite-GPU sample, which is very similar to the FiniteGPU sample discussed in the previous section. The difference this time is that InfiniteGPU extrudes the shadow volume to infinity. All rendering steps and stenciling operations are similar to the FiniteGPU sample. Hence we only look at the vertex shader constants setup and the vertex shader code.

As discussed in the "Forming the Shadow Volume" section, we can make use of homogeneous coordinates to render vertices at a semi-infinite distance by setting the w component of the vertices to 0 before transforming to clip space. Remember that 3D points with w=0 are effectively vectors or simply just directions. Rendering a vector is thus analogous to rendering a vertex at the position in infinity pointed to by the direction of the vector. Therefore, to ensure the correct direction of vertex extrusion using homogenous coordinates, we need to apply transformations to reposition the vertices with respect to the light source (and at the same time centering the light's position at the origin). With the light centered at the origin and the shadow volume vertices positioned accordingly, any vertex whose w component is set to 0 will automatically represent the incident light ray vector. Rendering this vector will hence be the same as extruding it, infinitely, in the direction to which it points. The required transformation to center the light source at the origin can be achieved by using the world and inverse light matrices. We shall refer to the transformed space as the WorldLight space. Let's check out how the vertex shader constants are going to be set in the SetShaderConstants() function:

```
void CShadow::SetShaderConstants( const D3DLIGHT8* pLight,
                                  const D3DXMATRIX* matWorld,
                                  const D3DXMATRIX* matView,
                                  const D3DXMATRIX* matProj )
{
  D3DXMATRIX matInvWorld;
  D3DXMATRIX matLight, matInvLight;
  D3DXMATRIX matWorldInvLight;
  D3DXMATRIX matLightClip;

  // Considering only the point light source (hence orientation is not
     needed),
  // light space transformation matrix thus contains only translation
  D3DXMatrixTranslation( &matLight, pLight->Position.x, pLight->Position.y,
        pLight->Position.z );
  D3DXMatrixIdentity( &matLightClip );
  D3DXMatrixMultiply( &matLightClip, &matLight, matView );
  D3DXMatrixMultiply( &matLightClip, &matLightClip, matProj );

  D3DXMatrixInverse( &matInvWorld, NULL, matWorld );

  D3DXMatrixTranslation( &matInvLight, -pLight->Position.x,
        -pLight->Position.y, -pLight->Position.z );
  D3DXMatrixMultiply( &matWorldInvLight, matWorld, &matInvLight );

  D3DXVECTOR4 vConst( 1.0f, 1.0f, 1.0f, 0.0f );

  // Yellowish-green hue for drawing shadow volume if needed
  D3DXVECTOR4 vColor( 0.3f, 0.4f, 0.2f, 0.0f );

  D3DXVECTOR4 vWorldLightPos, vObjectLightPos;
  vWorldLightPos = D3DXVECTOR4( pLight->Position.x, pLight->Position.y,
      pLight->Position.z, 1.0f );

  // Transform light pos from world space to object space
  // Light ray vector is computed in object space to avoid transforming the
  // face normal
  D3DXVec4Transform( &vObjectLightPos, &vWorldLightPos, &matInvWorld );

  // Set the shader constants
40    m_pd3dDevice->SetVertexShaderConstant( 0, &vObjectLightPos, 1 );
41    m_pd3dDevice->SetVertexShaderConstant( 1, &vConst, 1 );
42    m_pd3dDevice->SetVertexShaderConstant( 2, &matLightClip, 4 );
43    m_pd3dDevice->SetVertexShaderConstant( 6, matWorldInvLight, 4 );
44    m_pd3dDevice->SetVertexShaderConstant( 10, &vColor, 1 );
45  }
```

Since we are considering an omnidirectional point light, the light transformation matrix can be created solely by translation at line 13. We create the LightClip transformation matrix (light*view* projection) at lines 15 to 17. The LightClip transformation goes from WorldLight space to clip space, similar to the normal clip space transformation matrix where we go from world space to clip space. The transformation matrix to WorldLight space is computed at lines 21 to 23.

At line 37, we transform the light position from its original world space to the occluder's object space to compute the light ray vector in object space within the shader. This avoids the need to transform the face normals to world space for every single vertex and also results in shorter vertex shader code. Finally, lines 40 through 44 define how the values will be lined up in the constant registers. Next up, let's jump right into the vertex shader code:

```
01  // c0    : Light position in object space
02  // c1    : 1, 1, 1, 0
03  // c2-c5 : Light * View * Proj = LightClip?
04  // c6-c9 : WorldInvLight matrix
05  // c10   : Color for exposing the shadow volume
06
07  vs.1.1
08  // Output diffuse color to expose shadow volume to viewer if needed
09    mov oD0, c10
10
11  // Light to vertex ray in object space
12    sub r1, v0, c0
13
14  // Transform vertex from object space to WorldLight space
15  // where the light is centered on origin
16    mul r4, v0.x, c[6]
17    mad r4, v0.y, c[7], r4
18    mad r4, v0.z, c[8], r4
```

```
19   add r9, c[9], r4
20
21 // Normalize ray computed previously
22   dp3 r1.w, r1, r1
23   rsq r1.w, r1.w
24   mul r1, r1, r1.w
25
26   mov r10, c1
27
28 // Dot ray and normal
29   dp3 r10.w, v1, r1
30
31 // If dot result < 0.0 = face away from light
32 // Form mask 1,1,1,1 for light facing, OR mask 1,1,1,0 for non light facing
33   slt r10, c1.w, r10
34
35 // Set w value to 0.0 for infinite extrusion or 1.0 for no extrusion
36   mul r9, r9, r10
37
38 // Transform final vertex to LightClip space
39   mul r4, r9.x, c[2]
40   mad r4, r9.y, c[3], r4
41   mad r4, r9.z, c[4], r4
42   mad oPos, r9.w, c[5], r4
```

At line 12, we form the vector of the light ray in object space (remember, we do not need to transform the face normals if we do this in object space). Next, we proceed to transform the vertex to WorldLight space (World*InvLight) at lines 16 to 19.

Next, we get back to object space and normalize our light ray vector at lines 22 through 24. I hate to use `mov` but was forced to do so at line 26 because we need the masking values (1,1,1,0) for the `slt` command later on. At line 29, we use the light ray vector to perform a dot product with the face normal.

Line 33 is the heart of this shader. It compares the dot product result with the mask (1,1,1,0) that we loaded earlier on and creates either a mask (1,1,1,1) for light-facing vertices or (1,1,1,0) for vertices that face away from the light. We are going to make good use of this mask at line 36 to decide whether a vertex stays put or packs up for the trip to infinity. Obviously, those that face the light will be left unscathed, but those that face away from the light will have their w component zeroed, and the homogenous representation of a point becomes a representation of a vector. Finally, we perform the all-important transformation to clip space and pass the result to the output register.

Note You can try moving toward the extruded geometries for this sample (use wireframe mode; it is easier to see), but you will find that it never gets any closer! The vertices at infinity are "fixed" at a particular point on screen. Try the same thing with the other samples, and you will fly past the extruded volume in no time.

With this, we conclude the InfiniteGPU sample. You are now armed with a good working knowledge of not just one stencil shadow volume method but four of the same things done in a rather different fashion! You are probably confused and wondering which one of these suits your needs. Read on about some efficiency issues, possible optimizations, and high-level design problems that can help you make an informed choice.

Better with Shaders?

Using degenerate quads in order to utilize the vertex shader for shadow volume generation is not without its problems. From the previous sections, we have seen that the shadow volume generated by the GPU will always be capped, and thus it is only logical to employ the depth-fail algorithm for a more robust implementation. This also means that we cannot switch between the depth-pass and depth-fail algorithms for speed-ups when the camera is not intersecting the shadow volume. Implementing shadow volumes on the CPU gives us a bit more flexibility to switch to the cheaper depth-pass algorithm whenever possible.

Another major concern is the extra vertices generated due to the insertion of degenerate quads during preprocessing. This can sometimes become too large and adversely affect the vertex throughput of graphics hardware. When that happens, the reduced memory bandwidth savings that we have received by using shaders would be completely wiped out. This is often true when the source data sets for shadow volume creation are too large, resulting in an even greater amount of wasted vertices.

We can tackle the problem by optimizing the degenerate quad insertion algorithm and also reducing the source data sets needed for shadow volume creation. Previously, in the "Preprocessing of Data" section, we discussed a simple optimization to avoid inserting degenerate quads into edges shared by faces forming a flat plane. With good optimizations, the pre- and post-polygon count ratio can usually be reduced to around 2.0 without severe visual artifacts. Another possible optimization is to reduce the source data sets used for inserting degenerate quads. This encompasses the use of simplified models with a lower polygon count or the removal of useless polygons. The gem presented by Alex Vlachos and Drew Card [20] described two such optimizations in the form of vertex removal and edge collapsing. In most general cases where low to medium (MD2 or MD3) polygon count models are used, implementations on both CPU and GPU are comparable.

Another key concern is the magnification of this inefficiency when a scene contains a large number of shadow-casting light sources. The iteration through the light sources to generate shadow volumes would inevitably strain the graphics hardware with more wasted vertices. But with the use of the shadow volume methodology for casting shadows, we have to be very careful with the selection of light sources within a scene — even when it is done on the CPU. We discuss light sources selection further in the next section.

Finally, a small incentive for using shaders to implement the shadow volume generation is that the memory requirement is very constant, as opposed to the dynamic shadow volume size in CPU implementations. This is because the silhouettes of occluders can sometimes vary drastically at different angles of view. This directly affects the total geometry count and can cause further problems if the initial allocated memory is too small and reallocation is needed. GPU implementation does not suffer from this problem, as the preprocessed shadow volume geometry is loaded up as a static vertex buffer that contains all the vertices that will be needed for shadow volume generation from any angle.

Overall, in a normal game setting, where the CPU is required for artificial intelligence, physics, network (encryption/decryption), input, scripting, and a whole host of other computations, GPU implementation of shadow volume usually edges slightly ahead in terms of performance. However, readers are encouraged to evaluate and profile both approaches vigorously within their setup in order to find the best way for implementing shadow volumes. Greg James [15] showcased the use of degenerate quads for a vertex shader-based shadow volume implementation. Similarly, Chris Brennan's article [18] regarding shadow volumes used in the ATI island demo [19] also uses the same approach with vertex shaders.

DirectX 9 HLSL Samples

The launching of Microsoft's DirectX 9 API is a major milestone for the graphics development community. DirectX 9 introduces numerous additions and changes from DirectX 8.1, including the High Level Shading Language (HLSL), which is similar to nVidia's cg language that makes writing shaders much more intuitive and stress free.

Two HLSL samples have been included to keep the reader up to date with the latest technology. These two samples are FiniteHLSL and InfiniteHLSL. The former implements a finite shadow volume extrusion similar to the FiniteGPU sample discussed previously. The latter implements an infinite shadow volume extrusion that mirrors the InfiniteGPU sample. The reader should note that the HLSL samples are based on the updated DirectX 9 common files framework and use the effects file (*.fx) for defining the HLSL code and render states. Two important device caps are needed for the HLSL samples to run: `D3DPRASTERCAPS_DEPTHBIAS` and `D3DSTENCILCAPS_TWOSIDED`. The HLSL samples utilize the new two-sided stencil operations provided by DirectX 9. We will not go through the two HLSL samples in detail, as they are very similar to their predecessors FiniteGPU and InfiniteGPU, which are based on DirectX 8.1. The HLSL samples are comparatively simpler than the previous samples that had vertex shaders coded in assembly. Let's take a look at how to make use of the new two-sided stencil mode introduced in the HLSL samples.

With DirectX 9, the Direct3D API now includes support for two-sided stencil operations. For both the depth-pass and depth-fail stenciling operations described earlier, we need to draw the shadow volume in two passes, once for the front faces and once for the back faces. This is due to the need to change the stenciling operations before the start of each pass, since a different set of stenciling operations is needed for drawing the front faces and back faces of the shadow volume. The need for two passes to render the shadow volume geometries places extra strain on the vertex throughput of the GPU. With two-sided stenciling in DirectX 9, we can specify different sets of stenciling operations for both front faces and back faces before proceeding to render the shadow volume geometries in a single stenciling pass. Two-sided stenciling mode ensures that the stencil buffer values are filled accordingly, as if we are rendering the front and back faces separately with different stenciling operations.

Whenever two-sided stenciling mode is supported, we should make use of it — and for good reason, too. First, we just need to send the shadow volume geometries to the graphics pipeline once instead of twice. With that comes the savings on transforming primitives, memory bandwidth between transfers, and driver overhead for sending the geometries to hardware. The graphics hardware would probably also avoid inefficiencies that arise when rendering multiple culled polygons, which causes the rasterizer to go idle, since there is nothing to draw. For two-sided stenciling mode, we need to render with no culling at all, and hardware rasterizers can minimize the idling time. Note that this may not be true for all hardware vendors since graphics hardware and driver designs vary wildly from vendor to vendor. We should also note that the number of pixels rasterized is exactly the same as doing two passes to render the shadow volume. This means that fillrate would be the same for both stenciling modes. Considering the potential savings in other areas beside fillrate, two-sided stenciling mode is a highly attractive new hardware support to assimilate into any stencil shadow volume implementations.

A new render state, `D3DRS_TWOSIDEDSTENCILMODE`, can be set to true to activate two-sided stenciling. It is disabled by default. When two-sided stenciling is enabled, the following render states will apply *only* to front-facing triangles:

Render States	Operations
D3DRS_STENCILFAIL	D3DSTENCILOP to do if stencil test fails.
D3DRS_STENCILZFAIL	D3DSTENCILOP to do if stencil test passes and z-test fails.
D3DRS_STENCILPASS	D3DSTENCILOP to do if both stencil and z-tests pass.
D3DRS_STENCILFUNC	D3DCMPFUNC function. Stencil test passes if ((ref & mask) stencilfn (stencil & mask)) is true.

The following new render states will also apply *only* to back-facing triangles:

Render States	Operations
D3DRS_CCW_STENCILFAIL	D3DSTENCILOP to do if stencil test fails.
D3DRS_CCW_STENCILZFAIL	D3DSTENCILOP to do if stencil test passes and z-test fails.
D3DRS_CCW_STENCILPASS	D3DSTENCILOP to do if both stencil and z-tests pass.
D3DRS_CCW_STENCILFUNC	D3DCMPFUNC function. Stencil test passes if ((ref & mask) stencilfn (stencil & mask)) is true.

The remaining stencil render states not listed in the two tables above will always apply to both front- and back-facing triangles. As with normal stenciling operations, the two-sided stenciling render states will be ignored for point sprites and lines. Let's look at the actual code needed to set up two-sided depth-fail stenciling operations in DirectX 9.

```
// Disable z write, color write, use flat shade, and set to cull none
m_pd3dDevice->SetRenderState( D3DRS_ZWRITEENABLE, FALSE );
m_pd3dDevice->SetRenderState( D3DRS_COLORWRITEENABLE, FALSE );
m_pd3dDevice->SetRenderState( D3DRS_SHADEMODE, D3DSHADE_FLAT );
m_pd3dDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );
```

```
06
07 // Enable stencil operations and two-sided stencil mode
08 m_pd3dDevice->SetRenderState( D3DRS_STENCILENABLE, TRUE );
09 m_pd3dDevice->SetRenderState( D3DRS_TWOSIDEDSTENCILMODE, TRUE );
10
11 // Set front-facing stencil function to always pass
12 m_pd3dDevice->SetRenderState( D3DRS_STENCILFUNC, D3DCMP_ALWAYS );
13
14 // Set back-facing stencil function to always pass
15 m_pd3dDevice->SetRenderState( D3DRS_CCW_STENCILFUNC, D3DCMP_ALWAYS );
16
17 // Set stencil ref. value to 1 with full mask
18 m_pd3dDevice->SetRenderState( D3DRS_STENCILREF, 0x1 );
19 m_pd3dDevice->SetRenderState( D3DRS_STENCILMASK, 0xffffffff );
20 m_pd3dDevice->SetRenderState( D3DRS_STENCILWRITEMASK, 0xffffffff );
21
22 // Set up stencil operations for depth-fail algorithm
23 // Set up stencil to increment when z-fail occurs for back faces
24 m_pd3dDevice->SetRenderState( D3DRS_CCW_STENCILPASS, D3DSTENCILOP_KEEP );
25 m_pd3dDevice->SetRenderState( D3DRS_CCW_STENCILFAIL, D3DSTENCILOP_KEEP );
26 m_pd3dDevice->SetRenderState( D3DRS_CCW_STENCILZFAIL, D3DSTENCILOP_INCR );
27
28 // Set up stencil to decrement when z-fail occurs for front faces
29 m_pd3dDevice->SetRenderState( D3DRS_STENCILPASS, D3DSTENCILOP_KEEP );
30 m_pd3dDevice->SetRenderState( D3DRS_STENCILFAIL, D3DSTENCILOP_KEEP );
31 m_pd3dDevice->SetRenderState( D3DRS_STENCILZFAIL, D3DSTENCILOP_DECR );
```

The code above shows all the render state setup needed to make use of two-sided stenciling operations for the depth-fail algorithm. Most of the setup code is similar to that used in the DepthFail-CPU sample, with the exception of a few that involve the new render states. The HLSL samples implement the same set of render states within the effects file HLSL_ShadowVolume.fx using the effects file syntax.

Note that line 5 sets the culling mode to none, since we would want to render both front- and back-facing triangles at the same time in a single pass. In fact, the ability to draw with no culling is a requirement for graphics drivers before two-sided stenciling support is possible. Lines 8 and 9 separately enable stenciling operations and two-sided stenciling mode. Note that two-sided stenciling is disabled by default for compatibility with DirectX 8 behavior.

As before, the stenciling function is set to always pass at line 12. However, since we are now working in two-sided stenciling mode, the render state in line 12 only affects the rendering of front-facing triangles. The new render state `D3DRS_CCW_STENCILFUNC` has to be set in line 15 to force the stenciling function for back-facing triangles to always pass as well. Lines 18 through 20 set up the stencil reference value and the masks, which affects both the front- and back-facing geometries.

Finally, the code at lines 24 through 31 set up the stenciling operations for both front-facing and back-facing triangles, according to the requirements of the depth-fail algorithm. Be aware again that two different sets of render states are needed for both front-facing and back-facing geometries. Once the stencil operations have been set up, as shown in the above code, we can start rendering the shadow volume geometries, and the stencil buffers will be filled with the correct stencil values needed for drawing depth-fail shadows.

At the time of publication, only the Radeon 9700/9800 and GeForceFX consumer graphics cards fully support DirectX 9. Therefore the reader should note that two-sided stenciling is not a standard capability of most graphics hardware (even for that once-pricey GeForce4 Ti 4600 card). As such, implementations utilizing two-sided stenciling should always be backed up with hardware capability checks during program startup. A new capability bit, `D3DSTENCILCAPS_TWOSIDED`, was introduced in DirectX 9 for detecting devices that support this new stenciling mode. With this in mind, the HLSL samples implement both the old one-sided stencil mode and the new two-sided stencil mode, and on-the-fly switching between the two modes is possible.

Efficiency and Robustness

Realistic and accurate shadows in games are fast becoming a requirement as the complexity of games has skyrocketed over the past ten years. We need to provide robust, yet efficient implementations of stencil shadow volumes to satisfy the increasing expectations of the average consumer. In the case of robustness, using the depth-fail technique should suffice for almost any situation imaginable. However, hardware limitations and poor frame rates sometimes push the depth-fail technique beyond our computation budget. There are many ways to optimize shadow volume implementation to create nice-looking shadows, yet hold the frame rate at acceptable levels. Hence, to conclude the topic, let's look into some issues surrounding shadow volume implementations that are pertinent to achieving efficiency and robustness. We also look briefly into other general considerations, such as model design workflow and scene management, that are important complements to a successful integration of shadow volumes capability into an existing work process that includes developers, designers, and planners.

Due to the sheer scope of the related topics, only brief introductions are given here. Readers are strongly encouraged to do more research using the references provided.

Use Less for More

From the discussions in the previous sections, it is clear that the stencil shadow volume implementation suffers from two major performance bottlenecks: heavy silhouette computation and a costly invisible fillrate. Many performance-enhancing measures strive to minimize the impact of these two bottlenecks.

Reducing the source data set used in calculating the shadow volume is beneficial to both CPU and GPU implementations. In most cases, it is sufficient to make use of a low-polygon model of the occluder to compute the shadow volume. For Direct3D implementations, it is also advisable to use "welded" meshes as source data sets. A welded mesh simply means that there are no duplicated vertices representing exactly the same point. To see an example of an "unwelded" mesh, open the mesh viewer tool that is part of the DirectX utilities and create a cube. Look at the vertices information of the cube, and you can see that there are 24 vertices instead of just eight. This is really unavoidable, since Direct3D's version of a vertex structure contains color and normal information that cannot be shared by different faces referring to the same point due to differing lighting properties. Hence, extra vertices are generated for different faces with different color and lighting properties. The extra vertices are redundant as far as shadow volumes are concerned, but cannot be removed during the silhouette calculation without a considerable amount of comparison work. It is, therefore, wiser to use welded meshes for silhouette determination. The Direct3D mesh viewer utility provides a nifty option to do just that. Click MeshOps, then click Weld Vertices, and check Remove Back To Back Triangles, Regenerate Adjacency, and Weld All Vertices before welding. Alternatively, we can also make use of the mesh function `D3DXWeldVertices` to weld the mesh ourselves during data initialization.

Alex Vlachos and Drew Card [20] also described a method to process complex source data sets into simpler, non-overlapping shadow volume geometries for static light sources. The method described involves computing a list of all the light-facing polygons, which is the brute-force way that we have been doing it in the "Implementation on CPU" and "Implementation on GPU (Shaders)" sections. Next, the list is sorted in a back-to-front order. Going through the list polygons, a small frustum is created for each face by using the light position and the edges of the face. The face itself is used as the fourth clip plane. This frustum is used to test for obscuring polygons, which is discarded. Doing so recursively creates an unobstructed front capping that eliminates overlapping polygons. Collapsing edges and removing excessive vertices could further optimize the front capping. This is indeed a good way to speed up shadow volume implementations for static light sources. Our shadow volumes implementation should have a flexible computation path that changes according to different situational requirements. Instead of the generic brute-force method discussed in the samples, there are many other specific derivative methods that speed up our shadow volume implementation for different situations. This active selection of different methods is part of scene management in general, which we discuss shortly.

Cheat Whenever You Can

The next area to optimize is the algorithm used to compute the silhouette in real time. For CPU-based implementations, this means achieving a faster silhouette computation turnaround time through improvement to the silhouette determination process and algorithm used. For GPU-based implementations, this means achieving a smaller preprocessed data set (after inserting degenerate quads) and a faster vertex shader algorithm. There are many ways to achieve faster on-the-fly computations, with approximation undoubtedly the most preferred way. The many operations that we can speed up through approximation include trigonometry calculations, distance computation, possible silhouette interpolation, and sorting algorithms.

Particularly interesting approximations are the silhouette mapping [24] and the related silhouette clipping [25] techniques. Although the subject matter of the two papers provides a visual refinement to the coarse polygonal silhouette of low-polygon models used in place of detailed models, the proposed silhouette approximation techniques can be applied to shadow volume implementations. In the first technique, a silhouette map is created from a number of silhouettes sampled from a discrete set of viewpoints about the object (occluder). The silhouette of any arbitrary viewpoint can then be approximated through interpolation from three nearby viewpoints in the silhouette map. Apart from the required precomputations and some limitations, this technique is much faster than brute forcing our way through the occluder's triangles. The second technique, silhouette clipping, is an improved concept of computing silhouettes that makes use of a special n-ary tree hierarchy of a model's edges. The technique hinges on the use of open-ended anchored cones for fast hierarchical culling in order to extract the silhouette of the model from any viewpoint.

When it comes to optimizations, we should always be wary of optimizing the wrong areas that only have minute contributions to the total overhead. As a rule of thumb, always go for the most frequently called functions or calculation paths. Other ways to improve computations include harnessing special capabilities of the CPU, such as the SSE, SSE2, and 3DNow! technologies provided by Intel and AMD, respectively. SSE (Streaming SIMD Extensions), for example, works on a quad float basis much like shaders. Operations are done in parallel across the four operands, giving a huge boost to any arithmetic-intensive computation.

Fighting the Invisible

Curbing the cost of the invisible fillrate needed to render the passes for shadow volumes is another major issue. We can probably lessen the impact by setting the `D3DRS_COLORWRITEENABLE` render state in Direct3D before rendering the shadow volume. We can use it to turn off the red, green, blue, and alpha channel drawing, since color information is irrelevant here and we are only interested in filling the stencil buffer.

Another easy way to alleviate the problem is to reuse stencil buffer data across consecutive frames. We still incur the fillrate to shade pixels in the frame buffer, but the passes needed to fill the stencil buffer are saved. Depending on implementations, we can usually get away with reusing old stencil buffer data provided that the viewer, occluder, and light source's relative positions have not changed drastically from the previous frame. Besides saving the cost of the invisible fillrate, the cost of computing the shadow volume is also saved. Finally, another good way to cut down on the shadow volume fillrate cost for attenuated point light sources is to make sure that we draw only where it is necessary by using the scissor rectangle test. Eric Lengyel [8] described utilizing the OpenGL scissor rectangle support to cut down the fillrate penalty for rendering the shadow volumes and the illuminated fragments.

Scissor rectangle support is finally available with the introduction of DirectX 9. The DirectX 9 scissor test is implemented by the functions `SetScissorRect` and `GetScissorRect` of the `IDirect3D-Device9` interface. A new render state, `D3DRS_SCISSORTESTENABLE`, is also included to toggle the test.

Scene Management Inside and Out

Another area that we should take note of is the management of shadow-casting lights in our 3D scene. Good management of light sources invariably benefits the shadow volume generation process. A rule of thumb is to keep the number of shadow-casting light sources below a maximum of four at any one time. Future hardware improvements or algorithm advancement would definitely nullify the previous statement, but for now it serves as a good guideline and will probably remain so for the next few hardware iteration cycles. The important aspect of light source management is the method used for selecting which light sources should be included in the shadow volume generation process. The main parameters that should be taken into consideration could be intensity, distance from viewer, relevance to current gameplay, and (lastly) visual importance.

Let's look at some cases to understand the complexity of light source selection. Imagine your game character standing in the middle of a stadium with four gigantic batteries of floodlights shining down the field. There should be at least four shadows of your game character on the floor forming a cross due to the shadow casting from four different directions. Selecting only one light source here is going to make the scene look weird. Rule of thumb: Always select the dominant light sources in the scene. Note that using the viewing frustum to select light sources can be very dangerous. This is because you may have a nice gigantic 1000-megawatt photon-busting spotlight right behind the top of your head. It's not in your view frustum, but it's going to be responsible for the most distinct shadows you would see in the scene.

Additionally, performing occlusion culling on light sources is also helpful, but it should be done from the occluder point of view, not the viewer! The general rule is that if an occluder cannot see a light source, it cannot cast shadows related to that light source. We have to consider the occluder as a whole because it is nontrivial to handle cases whereby the occluder is partially exposed to the light source. Performing line-of-sight tests on a per-occluder basis can, however, be a big hit on performance, but doing such tests on a per-area basis would probably suffice for most situations. Distance and attenuated strength of light sources is also a good gauge of whether a light source has a big contribution to the scene makeup. Whenever the distance is beyond a certain predefined limit or when the attenuated strength of the light source is deemed too weak to create distinct shadows in an area, we should have no qualms about dropping it, even if there is a perfect line of sight between the light source and the occluders in the area.

The culling of occluders is just as important as the culling of light sources. Once we have selected a list of light sources, we should commence with occluder culling before computing the shadow volumes. For each selected light source, we identify the occluders whose shadow volumes would contribute to any visible shadows within the view frustum. This test can be done easily by using a bounding volume constructed from the light's position and the three opposing sides of the view frustum, as shown in the following figure:

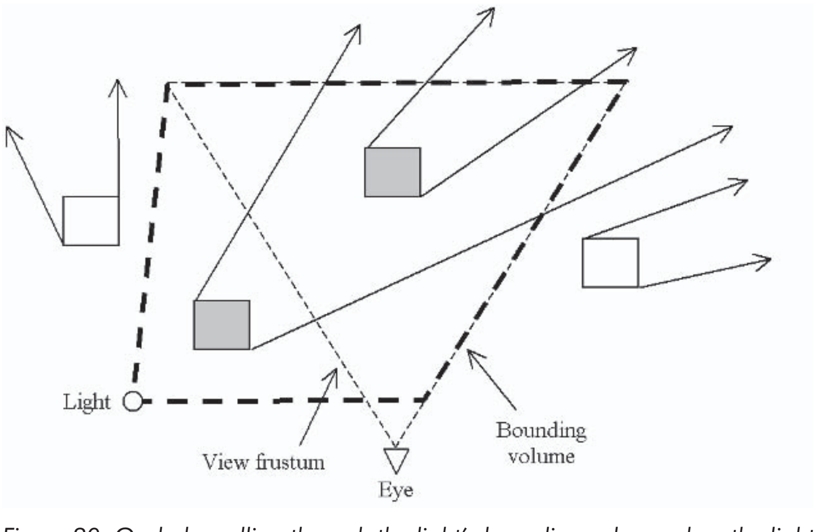


Figure 20: Occluder culling through the light's bounding volume when the light source is outside the view frustum

As shown in Figure 20, only the shadow volumes of the shaded cubes contribute to visible shadows within the view frustum. Any occluders that fell completely outside the bounding volume could be culled away (e.g., non-shaded cubes), since they would not contribute to any visible shadows. In the other case, where the light source is within the view frustum, we should use the view frustum itself as the bounding volume to perform the occluder culling as shown in Figure 21.

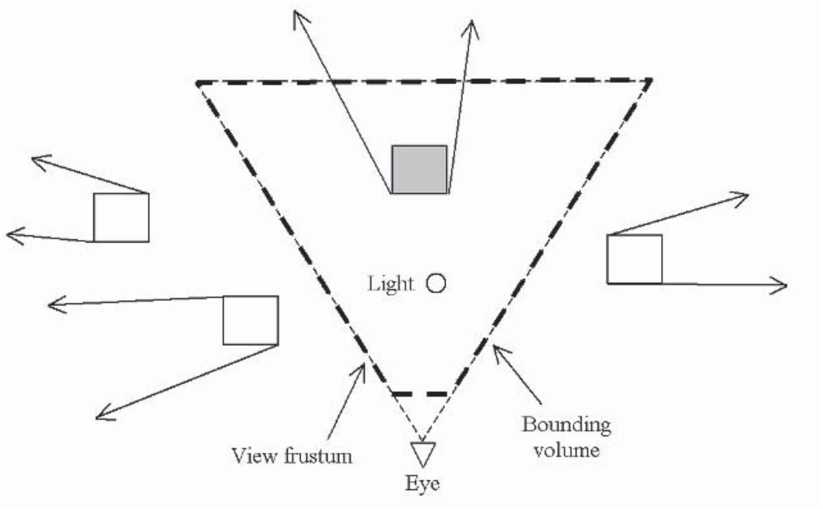


Figure 21: Occluder culling uses the entire view frustum as the bounding volume when the light source is within the frustum.

Occluder culling helps minimize the amount of work on silhouette computations and shadow volume rasterization on a per-light source basis, making each selected light pass more efficient and lean.

The whole business of selecting light sources and culling occluders boils down to good scene management. An important component of scene management is the added responsibility of level planners and designers to work out an arrangement in which the light settings and positioning in a scene would not break or compromise the underlying shadow volume implementation. Therefore, it is often imperative that level planners and designers have a thorough understanding of the underlying light source selection criteria made by the graphics engine before they set out to build the first scene. Charles Bloom [22] discussed some useful notes regarding the selection of light sources, while Cass Everitt and Mark Kilgard [27] presented several optimizations for implementing shadow volumes.

Another aspect of scene management is identifying the relationship between occluders and light sources and possibly embedding this information somewhere with the scene hierarchy. Tagging geometries according to their movement behavior and relationship to a light source is a good way to branch into faster, specific shadow volume implementation quickly. For example, let's say that we have a static light source in an oil lamp on a chandelier hanging from the ceiling of a church. The spatial relationship between the light source and the occluder (chandelier) is static because the shadow volume of the occluder will never change, even if it is swinging, since the light source would be swinging in perfect synchronization as well. Hence, for the chandelier, which can be a complicated model, we can precompute an optimized front capping that can be reused every frame. Next, a player character walks into the church. The spatial relationship between the light source and the occluder (player model) is dynamic. Hence, for the player model, we should switch back to more elaborate (slower) shadow volume estimation or calculation. Proper scene management goes a long way in cutting the cost of shadow volume implementations while retaining all the visual enhancements that comes with it.

Next, remember that one of the important requirements of shadow volumes is the need for closed volume meshes. As described before, this is needed because any gaps or holes within a mesh would potentially throw the stencil counting off-balance and thus break the shadow volume implementation. Such a requirement mandates the need for modelers and designers to alter their workflow and modeling style in order to avoid compromising the graphics engine. This is often the most daunting task for any program manager to undertake if there is a decision to turn toward stencil shadow volume support. As far as programmers are concerned, shadow volume implementations can be made more robust by adding tests to detect unclosed volumes, reduce vertices, and even remove unwanted t-junctions (Lengyel [21]) during preprocessing.

Always a Good Switch

Switching between the efficient depth-pass and the robust depth-fail algorithm on the fly can also help speed up shadow volume implementations. For a robust implementation, we usually go for the depth-fail algorithm. However, we can actually switch to the faster depth-pass technique whenever we are sure that the camera is not within any shadow volumes. This can be done easily by forming a near-clip volume and test for occluder intersection against it. The light source's position and the four sides of the near plane are used to define a pyramid of four planes. The near plane closes the pyramid and thus forms the near-clip volume. If an occluder lies completely outside this volume, we can safely employ the depth-pass technique, since the occluder's shadow volume has no chance of intersecting the near plane.

Mix and Match

Lastly, stencil shadow volume also forms a good foundation for implementing hybrid shadows that blend, attenuate, or soften the edges through a mixture of projected textures, shadow mapping, volume textures, or even clusters of shadow volume casting light sources. Even in its simplest form, shadow volume's much-maligned hard-edged shadows often stunned the average gamer. Remember those dropped jaws when Doom III screen shots first became available?

The End

This ends our discussion on stencil shadow volume implementation. I would like to take this opportunity to thank *ShaderX^2* editor Wolfgang Engel and Andre Chen for reviewing this article. My heartfelt gratitude also goes to Wordware Publishing, Inc. and my company, Silicon Illusions (www.siliconillusions.com), for their support and help. Many thanks also to James Paul Pilande who provided the models used in all the samples.

A word about the samples: There are four samples built using the common files framework provided by DirectX 8.1 (C++): DepthPassCPU, DepthFailCPU, FiniteGPU, and InfiniteGPU. There are two additional samples based on the common files framework, effects file, and the HLSL support provided by DirectX 9.0 (C++). These are FiniteHLSL and InfiniteHLSL. All source data used are standard *.x file meshes re-authored in MilkShape 3D [23] from their original *.3ds format.

Color Plates 6 and 7 provide examples of what can be done with the sample files. Plate 6 shows a scene consisting of dynamic shadow casters and light source. It showcases the increased realism with the help of accurate shadowing using the stencil shadow volume technique. This technique is fast becoming the preferred choice of shadowing in newer 3D games. Plate 7 shows the same scene re-rendered with the extruded shadow volume exposed. The "stencil counting" approach used in the technique makes accurate inter-occluders shadowing and self-shadowing possible.

References

[1] Crow, Frank, "Shadow Algorithms for Computer Graphics," *Computer Graphics*, Vol. 11:3, SIGGRAPH '77, July 1977.

[2] Heidmann, Tim, http://developer.nvidia.com/docs/IO/2585/ATT/RealShadowsRealTime.pdf.

[3] Kilgard, Mark, http://developer.nvidia.com/docs/IO/1348/ATT/stencil.pdf.

[4] Power Render X game engine, http://www.powerrender.com/prx/index.htm.

[5] Carmack, John, http://developer.nvidia.com/docs/IO/2585/ATT/CarmackOnShadowVolumes.txt.

[6] Bilodeau, Bill and Mike Songy, "Real Time Shadows," Creativity 1999, Creative Labs, Inc. Sponsored game developer conferences, Los Angeles, California, and Surrey, England, May 1999.

[7] Kilgard, Mark, http://developer.nvidia.com/docs/IO/1451/ATT/StencilShadows_CEDEC_E.pdf.

[8] Lengyel, Eric, http://www.gamasutra.com/features/20021011/lengyel_01.htm.

[9] Lengyel, Eric, *Mathematics for 3D Game Programming and Computer Graphics*, Charles River Media, 2002.

[10] Everitt, Cass, and Mark Kilgard, http://developer.nvidia.com/docs/IO/2585/ATT/GDC2002_RobustShadowVolumes.pdf.

[11] Moller, Tomas, and Eric Haines, *Real-time Rendering*, Second Edition, A K Peters Ltd., 2002, pp. 61-66, http://www.realtime-rendering.com.

[12] Microsoft DirectX MSDN, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/directx9_c/directx/graphics/programmingguide/programmingguide.asp.

[13] Watt, Alan, *3D Computer Graphics*, Second Edition, Addison-Wesley, 1993, pp. 229-237.

[14] Imagire, Takashi, http://if.dynsite.net/t-pot/program/75_shadow2Vol/index.html.

[15] James, Greg, http://developer.nvidia.com/view.asp?IO=vertexshader_shadowvolumes.

[16] Engel, Wolfgang F., *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, Wordware Publishing, Inc., 2002, pp. 51-52, http://www.shaderx.com.

[17] nVidia, NVASM vertex and pixel shader macro assembler, http://developer.nvidia.com/view.asp?IO=nvasm.

[18] Brennan, Chris, "Shadow Volume Extrusion Using a Vertex Shader," *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, pp. 188-194, http://www.shaderx.com.

[19] ATI, Treasure Chest and Island demos, http://www.ati.com/developer/demos/r8000.html.

[20] Vlachos, Alex and Drew Card, "Computing Optimized Shadow Volumes for Complex Data Sets," *Game Programming Gems 3*, Charles River Media, Inc., 2002, pp. 367-371.

[21] Lengyel, Eric, "T-Junction Elimination and Retriangulation," *Game Programming Gems 3*, pp. 338-343.

[22] Bloom, Charles, http://www.cbloom.com/3d/techdocs/shadow_issues.txt.

[23] MilkShape 3D modeler, http://www.milkshape3d.com/.

[24] Harvard University, Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, Leonard McMillan, Benedict J. Brown, and Abraham D. Stone, "Silhouette Mapping," Computer Science Technical Report: TR-1-99, http://research.microsoft.com/~hoppe/silmap_tr_text.pdf.

[25] Sander, Pedro V., Xianfeng Gu, Steven J. Gortler, Hugues Hoppe, and John Snyder, "Silhouette Clipping," ACM SIGGRAPH 2000, pp. 327-334, http://people.deas.harvard.edu/~pvs/research/silclip/.

[26] Lengyel, Eric, "Tweaking a Vertex's Projected Depth Value," *Game Programming Gems*, Charles River Media, Inc., 2000, pp. 361-365.

[27] Everitt, Cass and Mark J. Kilgard, "Optimized Stencil Shadow Volumes," http://developer.nvidia.com/docs/IO/4449/SUPP/GDC2003_ShadowVolumes.pdf.

[28] nVidia, "Understanding the w Coordinate," http://developer.nvidia.com/view.asp?IO=understanding_w.

Shader Development Using RenderMonkey

Natalya Tatarchuk
ATI Research

Introduction

Many of the current challenges facing 3D graphics application developers are centered on creating and using programmable graphics shaders. These programmable graphics shaders are at the heart of all future graphics chips. With the introduction of the Radeon 9000, shaders are now supported on the entry-level PC and will soon trickle down to all other devices.

Developers with the ability to create and use these programmable shaders are able to take advantage of all that the hardware offers and create applications that redefine the art of real-time graphics. In order to help developers unlock the creative potential of today's graphics chips and improve the shader prototyping and development process, ATI Technologies has developed the RenderMonkey Integrated Development Environment (IDE).

Although writing assembly or High Level Shading Language code is the heart of the shader development process, shaders are more than just the code. Encapsulating shader-based effects can be a complex task, since it involves capturing the entire state of the system that is involved in rendering these effects. This leads to a common problem that currently exists among shader developers — exchanging and sharing shaders is not a trivial task.

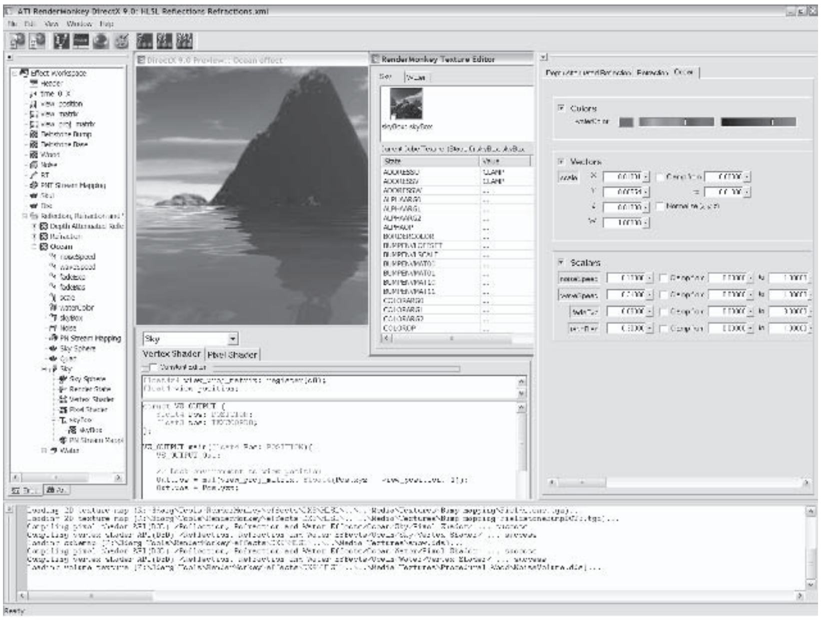
Another problem that many game developers face when starting to develop shaders is the need to closely involve artists in the process. Without tools that artists are comfortable with, it becomes difficult to collaborate on effect creation. What's needed is an environment where not just the programmers but the artists and game designers can work together to create mind-blowing special effects using shaders. RenderMonkey is designed to solve many of these problems and facilitate the shader prototyping process for your game engines. With this tool, we provide a powerful programmer's development environment for creating shaders, which can be used as a standard delivery mechanism to allow sharing of shader-based effects in the developer community. We also provide a flexible, extensible framework that supports easy integration of custom components and provides a solid basis for future tool development. RenderMonkey can be easily customized and integrated into any developer's regular workflow. The design of the RenderMonkey IDE allows easy incorporation of current and future rendering APIs. By the time this book is published, you will be able to download version 1.0 of the program from ATI's web site (http://www.ati.com/developer). That version includes support for DirectX 9 shader effects (using both assembly and HLSL), as well as support for creating OpenGL-based effects using the GL2 High Level Shading Language.

Although this chapter does not focus on the intricacies of writing shader code, there are some excellent chapters on that topic in this book and its companion book, *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9*. For those of you interested in learning the DirectX High Level Shading Language, you should read the "Introduction to the DirectX High Level Shading Language" article by Craig Peeper and Jason Mitchell, which appears in this book. There are also several articles in *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9* that I coauthored with my colleagues, which focus on the development of interesting shaders. These articles all use RenderMonkey workspaces that you can load into RenderMonkey and experiment with. Take a look at "Simulation of Iridescence and Translucency on Thin Surfaces" (N. Tatarchuk, C. Brennan), "Motion Blur Using Geometry and Shading Distortion" (N. Tatarchuk, C. Brennan, and J. Isidoro), "Layered Car Paint Shader" (C. Oat, N. Tatarchuk, and J. Isidoro), and "Real-Time Depth of Field Simulation" (G. Riguer, N. Tatarchuk, and J. Isidoro), as well as the "Advanced Image Processing with DirectX 9 Pixel Shaders" article by J. Mitchell, M. Ansari, and E. Hart. You will find a great deal of interesting material on developing spectacular visual effects in these articles.

Overview of the IDE

The RenderMonkey application interface has been designed to be intuitive for any developer who has used an IDE tool such as Microsoft Visual Studio.

Figure 1 shows a snapshot of the interface rendering an ocean effect.



ATI RenderMonkey DirectX 9.0: HLSL Reflections Refractions.xml
File Edit View Window Help
Effect Workspace
DirectX 9.0 Preview: Ocean effect
RenderMonkey Texture Editor
Sky
Vertex Shader
Pixel Shader
Colors
Vectors
Scalars
Ready

```
float4x4 view_proj_matrix: register(c0);
float4 view_position;

struct VS_OUTPUT {
   float4 Pos: POSITION;
   float3 dir: TEXCOORD0;
};

VS_OUTPUT main(float4 Pos: POSITION){
   VS_OUTPUT Out;

   // [illegible] view position
   Out.Pos = mul(view_proj_matrix, float4(Pos.xyz [illegible] view_position.xyz, 1));
   Out.dir = Pos.xyz;
```

```
Loading 2D texture map [illegible]\Media\Textures\Bump mapping\[illegible].tga)...
Loading 2D texture map [illegible]\Media\Textures\Bump mapping\[illegible].tga)...
Compiling pixel shader API(D3D) /Reflection, Refraction and Water Effects/Ocean/Sky/Pixel Shader/ ... success
Compiling vertex shader API(D3D) /Reflection, Refraction and Water Effects/Ocean/Sky/Vertex Shader/ ... success
Loading [illegible]\Media\Textures\[illegible].dds)...
Compiling pixel shader API(D3D) /Reflection, Refraction and Water Effects/Ocean/Water/Pixel Shader/ ... success
Compiling vertex shader API(D3D) /Reflection, Refraction and Water Effects/Ocean/Water/Vertex Shader/ ... success
Loading volume texture [illegible]\Media\Textures\Procedural\[illegible]NoiseVolume.dds)...
```

Figure 1: Main application interface

The main interface consists of several components:

- A *workspace view*, which shows the effect workspace being edited
- An *output* window for compilation results and text messages from the application
- A *preview* window used to preview effects being edited
- Other editor modules, such as editors for shader code and GUI editors for shader parameters. Shader parameters can be tagged as "artist-editable" and then edited in a coherent way using the artist editor module.

Creation of Basic Illumination Effect

In this section we work on creating a simple classic illumination effect; we write a shader-based effect for rendering a specular material in RenderMonkey's preview window. Although this simplified method of calculating light intensities comes from beginner's graphics books, it works wonderfully for the purpose of this tutorial to show how to quickly develop shaders using the RenderMonkey IDE. It also shows the beauty of using Microsoft's High Level Shading Language for developing shaders, as we can take these concepts for lighting equation and quickly translate them into visual effects with a few lines of code.

For this effect we implement the Phong specular-reflection lighting model. If we dig into any graphics textbook, we could find that in order to compute the illumination for that rendering model, we need to use the following equation:

$$I = I_{ambient} + I_{diffuse} + I_{specular} \tag{1}$$

…where each of the lighting contribution components can be computed as follows:

$$I_{ambient} = k_a \cdot I_a \tag{2}$$

$$I_{diffuse} = k_d \cdot I_d \cdot (\vec{N} \bullet \vec{L}) \tag{3}$$

$$I_{specular} = k_s \cdot I_s \cdot (\vec{V} \bullet \overline{R})^{n_s} \tag{4}$$

…where k_a, k_d, k_s are the coefficients for ambient, diffuse, and specular light contributions, respectively. These parameters are assigned a constant value in the range of 0 to 1, according to the reflecting properties that we want the surface to exhibit. If we want a highly reflective surface, we set the values for k_d and k_s to be near 1. This produces a bright surface with the intensity of the reflected light near that of the incident light. To simulate a surface that absorbs most of the incident light, we set the reflectivity to a value near 0. I_d is the intensity of the diffuse contribution of the point light source that we are simulating, and I_s is the intensity of the specular contribution of that light source. I_a is ambient light intensity. $\vec{N}, \vec{V}, \vec{R}$ are the normal, view, and reflection vectors, respectively. n_s is the specular-reflection parameter, proportional to the angle ϕ between the view and reflection vectors. Shiny surfaces have a narrow specular range (the angle between these two vectors is smaller), and dull surfaces have a wider reflection range. Thus, a very shiny model can be modeled with a large value for n_s (around 100, for example), and a dull surface can be modeled using an n_s value equal to 0.5.

We use the equations (2) to (4) in the pixel shader to compute the resulting color for each pixel for this illumination model. But first let's start the application and start building the workspace for the effect.

Run-Time Database Overview

Each set of visual effects in RenderMonkey is encapsulated in a single XML workspace. All of the information necessary for recreation of each effect, excluding the actual textures and model data, is stored in this single file. It is user-readable, and any game developer can create a converter from the RenderMonkey's file format into his game engine script format. We chose XML to store effect workspaces for several reasons. Most importantly, XML is an industry standard with parsers readily available (Render-Monkey uses the Microsoft XML parser; there are other alternatives freely available). It allows easy data representation, and it is user-extensible. Best of all, any user can open an XML RenderMonkey file and read the file directly in Internet Explorer; it's just another ASCII file format.

To start working on a shader-based effect, we simply launch the application, which automatically starts out with a new empty workspace. All effect-related data is stored in the effect workspace using RenderMonkey's run-time data format. Each *effect workspace* consists of these elements:

- Variable nodes
- Stream mapping
- Models
- Texture variables
- Effect group(s)

Each *effect group* is used to encapsulate a series of related effects. For example, you may want to group all effects that use a noise function to render perturbation-based effects, such as clouds or fire or plasma, in one single effect group. Another good use for this node is grouping various implementations of a single effect for fallback rendering in your engine.

Each effect group consists of one or more *effect nodes*. Each effect is used to draw a single, coherent visual effect in the viewer. You may have a single pass effect, or you may want to use several draw calls to generate the look that you want. But each draw call (or *pass*, as RenderMonkey refers to it) may consist of the following data:

- A render state block (optional)
- A vertex shader (required)
- A pixel shader (required)
- A geometry model reference (required)
- A stream mapping reference (required)
- One or more texture objects with valid texture references (optional)
- Variable nodes (optional)

All individual items in the RenderMonkey effect workspace are referred to as *nodes*.

Workspace View

The main window into the effect workspace is the *Workspace view window*. That's the dockable window usually positioned on the left of the main interface containing a tabbed tree control, which provides a high-level view of the effect database. Figure 2 shows the Workspace view window:

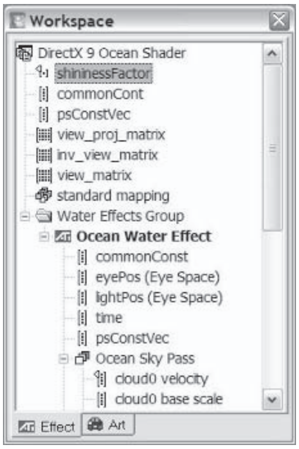


Figure 2: Workspace view window

The workspace view can be used to access all elements of the effect workspace. The intention is that individual effects will be grouped by their common attributes in an effect workspace.

There are two tabs in the workspace tree view: the Effect tab and the Art tab. The Effect tab is used to view the entire workspace — with all variables and passes visible. The Art tab is used to view only the artist-editable variables that are present in the workspace. Once an effect is developed by the programmer using the Effect tab, it can be handed over to the artists, who may want to just view the artist-editable data by simply selecting the Art tab to view the workspace.

Let's start working on our effect. If you right-click on the workspace node, you can select the Add Effect Group menu option from the context menu that appears. The context menu is shown in Figure 3:

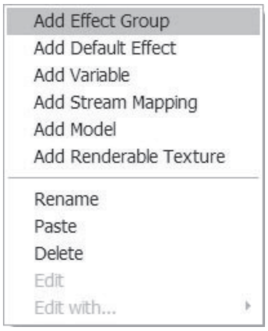


Figure 3: Context menu for adding new effect groups

When you add a new effect group to the workspace, Render-Monkey automatically populates the workspace with several nodes. It automatically adds a sample effect with one pass. The pass inside that effect contains sample vertex and pixel shaders and a sample geometry model. If you have ATI Radeon 8500 or a better type of hardware, you can see a red teapot in the preview window. If not, then you need to change the target for the pixel shader to ps_1_1 (I go over how to do that later in this article). RenderMonkey also adds a matrix variable for storing the view projection matrix called view_proj_matrix and a standard stream mapping node called *standard mapping*. A sample model node is added as well. This enables you to start right away with a fully functioning effect that you can build upon to create something more visually appealing than a red teapot.

Variable Creation and Management

Any shader-based effect that you are working on requires some parameters for the actual rendering. These parameters are specified as variable nodes in RenderMonkey. You can add a variable at any level of the workspace tree — to the effect workspace, effect group node, or effect or pass nodes.

Since we already know that we need several variables as input to our shaders, let's add them to our new workspace. To add a new variable, right-click on the node you want to add that variable to and select Add Variable from the context menu that appears (see Figure 4).

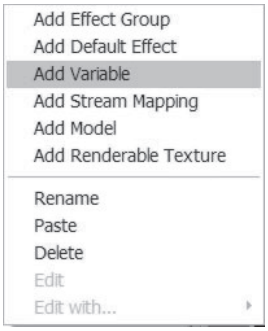


Figure 4: Example context menu for adding a new variable

You can then see the dialog in Figure 5:

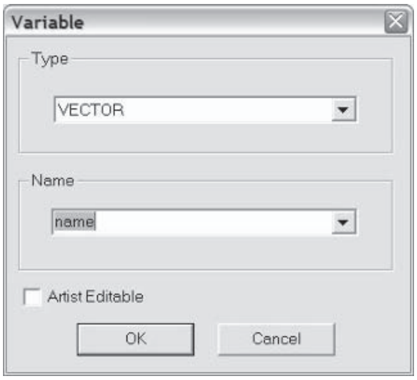


Figure 5: The Add Variable dialog

You can select one of the RenderMonkey-supported data types for your variable nodes:

- Scalar (a simple float variable)
- Vector (4D float variable)
- Matrix (4x4 float matrix)
- Color (4D float variable, RGBA color representation)

- Texture variables:
  - 2D texture map
  - Cube map
  - Volume texture

The icons on the left of each node in the Workspace view help you quickly identify their node type. For example, vectors are represented by [⋮], scalars are represented by [·], colors by ▮▮, matrices by [⁞⁞⁞⁞], etc.

By default, new scalar, vector, and matrix variables are created as not artist-editable. Color, texture, cube map, and volume texture variables are created as artist-editable. You have an option to make any new variable artist-editable by checking the Artist Editable check box in the Add Variable dialog. This is necessary to make a variable visible in the artist editor or on the Art tab in the Workspace view. If you wish to make any variable artist-editable at any point later on, you can also right-click on that variable and select the Artist Variable menu option. To remove the artist-editable property from a variable, right-click on the variable and select the Artist Variable menu option again. A check mark on that option indicates whether the variable is artist-editable or not. A small yellow flag on the variable icon indicates that the variable is artist-editable: [⋮].

Predefined RenderMonkey Variables

You probably noticed that when you added a new effect group, RenderMonkey also added a matrix called view_proj_matrix that showed up with a Predefined Variable tooltip if the mouse hovered over that variable. Predefined variables are shader constants whose values get filled in at run time by the viewer module directly. You cannot modify the values directly through the same user interface that you can use to edit other variables of similar types. RenderMonkey provides this set of predefined variables for your convenience:

- view_proj_matrix: A variable of type matrix, which contains the view projection matrix

- view_matrix: A variable of type matrix, which contains the view matrix
- inv_view_matrix: A variable of type matrix, which contains the inverse of the view projection matrix
- proj_matrix: A variable of type matrix, which contains the projection matrix
- time: A variable of type vector, which provides current time value cycled over the cycle that can be modified in the RenderMonkey Preferences dialog. By default, it is set to 120.
- cos_time: A variable of type vector, which provides the cosine of `time`
- sin_time: A variable of type vector, which provides the sine of `time`
- tan_time: A variable of type vector, which provides the tangent of `time`

The easiest way to add predefined variables to your workspace is to select the appropriate type of predefined variable that you would like to use and then choose the name from the combo box that appears in the Name area of the Add Variable dialog (see Figure 6). Note that the combo box only appears if the selected type has some predefined variables. If the user then chooses another type for a given predefined variable name, it is not appropriately initialized at run time, as RenderMonkey identifies predefined variables by both name and type.

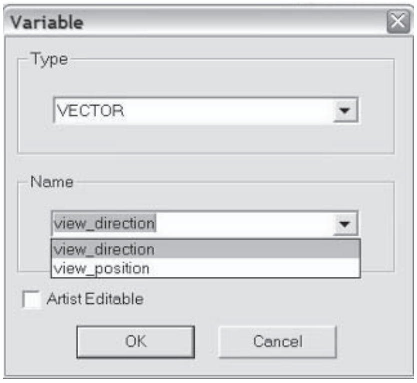


Figure 6: Selecting predefined variables

Predefined variables are easy to identify in any RenderMonkey workspace, as they will have a small green overlay on top of their usual variable type icon. For example, this is what a vector predefined variable icon would look like: .

Stream Mapping Module

Another node that was automatically added upon the new effect group addition is the stream mapping node. A stream mapping node can be created at any point in the workspace (directly under the effect workspace, directly under an effect group, within an effect group, or in an individual pass). This node is used to define streams that bind data to input registers for use by shaders. The streams get automatically generated in RenderMonkey using the data available from the model directly or computed by the application once you have defined the stream channels for that stream by using the provided user interface.

The stream mapping module is used for stream setup for the geometry model within a pass. To create a stream mapping node from scratch, you can right-click on a parent node (an effect, an effect workspace, or an effect group) and select the Add Stream Mapping menu option from the context menu (the example here is from the effect workspace context menu), as shown in Figure 7. This creates an empty stream mapping node.

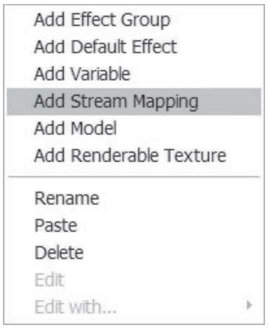
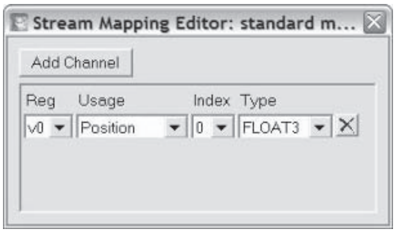


Figure 7: Adding a stream mapping node

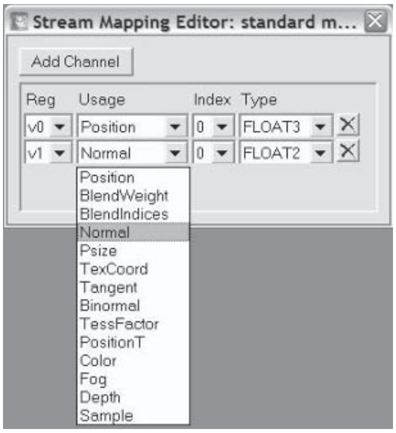
Once a stream mapping node is created, you can edit its contents by double-clicking on the node or right-clicking on the stream mapping node and selecting Edit, which brings up the stream mapping editor module shown in the following figure.



Reg	Usage	Index	Type
v0	Position	0	FLOAT3

Figure 8: Stream mapping editor

We already know that to compute correct illumination results, we need the vertex normals as well as vertex positions as inputs to the vertex shader. Let's add that channel to the stream defined in our workspace. Double-click on the standard mapping node and bring up the stream editor. To add new channels to the stream setup, you can click on the Add Channel button in the stream mapping editor. Then you can select the desired input register and name the usage for that stream, the usage index, and type. If you want to delete a specific channel, you can click on the X button to the right of the channel. In Figure 9 below, I have added a second channel to bind the normals for vertices. Don't forget to set the data type for the normals channel to FLOAT3.



Reg	Usage	Index	Type
v0	Position	0	FLOAT3
v1	Normal	0	FLOAT2

Figure 9: Adding the normals channel to the list of stream channels

To actually use the stream mapping for a specific draw call, you need to add a stream map reference to the pass in which you would like to use it. To do that, you need to first make sure that you've created a stream mapping node (like standard mapping) somewhere in the workspace tree. Then you can select the pass to which you want to add the stream mapping reference (Pass 1 in our case) and right-click on that node. Select Add Stream Mapping Reference from that context menu (as you can see from Figure 10):

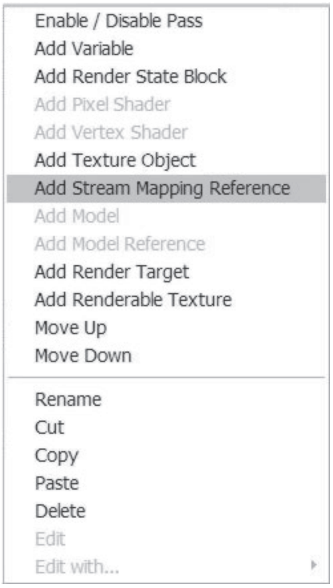


Figure 10: Adding a stream mapping reference to a pass

An empty stream mapping reference is then created. That reference is initially not linked to any stream mapping nodes. The red line on the stream mapping reference icon (standard mapping) shows you that the reference isn't correctly resolved. To link a reference to a stream mapping node, you should right-click on the stream mapping reference node and select the Reference Node menu from where you can select the name of the actual stream mapping node that you would like to reference in that pass (as shown in Figure 11). You can also double-click on the stream mapping reference node and rename the node to the name of the stream mapping node directly to link it.

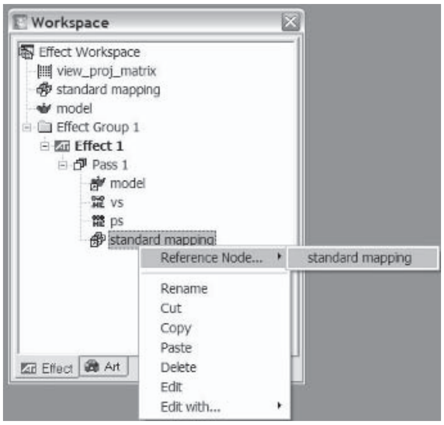


Figure 11: Linking a stream mapping reference to a stream mapping node

To resolve scope for the stream mapping for a particular pass, RenderMonkey first checks the pass tree for a stream mapping instance. If neither a stream mapping instance nor a stream mapping reference is found, the application "walks" up the workspace tree to find the first stream mapping node or reference. Note that placing stream mapping nodes and references should be done with consideration since incorrect use of stream mapping nodes results in bad rendering results.

If the stream mapping node name is found and resolved correctly, the stream mapping reference node will have this icon: Stream Mapping. Note the small arrow in the icon that denotes that it is a reference rather than the actual stream mapping node. That convention is for all reference nodes in RenderMonkey, so you can easily spot references in the workspace.

Model Management

An important aspect of every visual effect is the actual geometry that gets rendered on the screen. RenderMonkey uses the model and model reference nodes to allow you, the user, to specify which geometry to render in each draw call. As you can already see by this point, the workspace contains a model node under the main workspace node and a model reference node under the Pass 1 node. You can easily spot the model nodes by their red teapot icon: . The model reference nodes follow the convention described above for references and have a small arrow next to them: . To load a new geometry model into a model node, you double-click on that node and select a file containing your geometry object from the list of supported file formats that will be shown in the file open dialog. To actually bind the data from streams to the shaders, RenderMonkey uses the pairing of a stream map with a model data node done by adding both references to each pass to make sure that the necessary data is present at run time and then binds it to stream sources.

Managing Effects

Although we won't need to add any extra effects at this time, let's talk briefly about managing effects in RenderMonkey. As was said earlier, each effect in the workspace is used to draw a single, coherent visual effect in the viewer. It can consist of one or more draw calls. To create a new effect, you can right-click on the effect group to which you want to add the new effect. Select Add Effect from the context menu that appears (see Figure 12) to create a new effect at the bottom of this group:

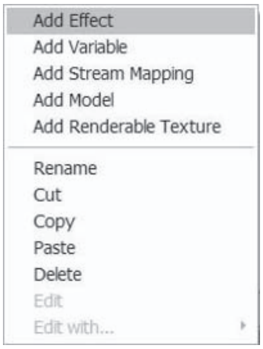


Figure 12: Adding new effects to the workspace from the context menu

You can change the effect name at any point by simply renaming it. By default, when RenderMonkey adds a new effect, it adds a single pass with HLSL vertex and pixel shaders in it. The main thing you want to do with the effects is view them. To do that, you should set the effect that you wish to render as an *active* effect. That means that this is the effect that will be rendered by the viewer module. To do that, you should right-click on the desired effect and select the Set as Active Effect menu option. You can easily check which effect is active in the workspace because it will appear in **bold** typeface.

Pixel and Vertex Shaders

Now we are getting closer to the heart of the programmable pipeline — the shaders themselves. RenderMonkey supports both assembly and the HLSL shader in its IDE. To create new pixel or vertex shaders, you need to select an effect to which to add the shader. Then you can right-click on the effect node and select Add Vertex Shader or Add Pixel Shader, depending on which type of shader you want to add (see Figure 13 for an example context menu).

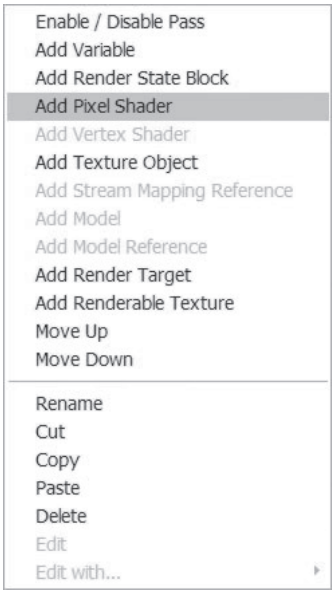
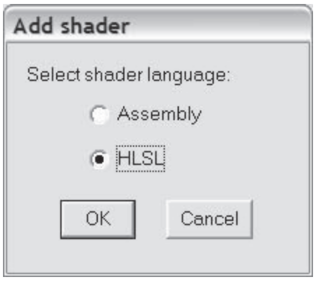


Figure 13: Adding shaders from the pass context menu

At this point you need to select what type of shader you want to add to that effect. You have a choice of adding an assembly or HLSL shader to the pass. Figure 14 shows the dialog box that appears for that purpose:


Add shader

Select shader language:

( ) Assembly
[✓ HLSL]

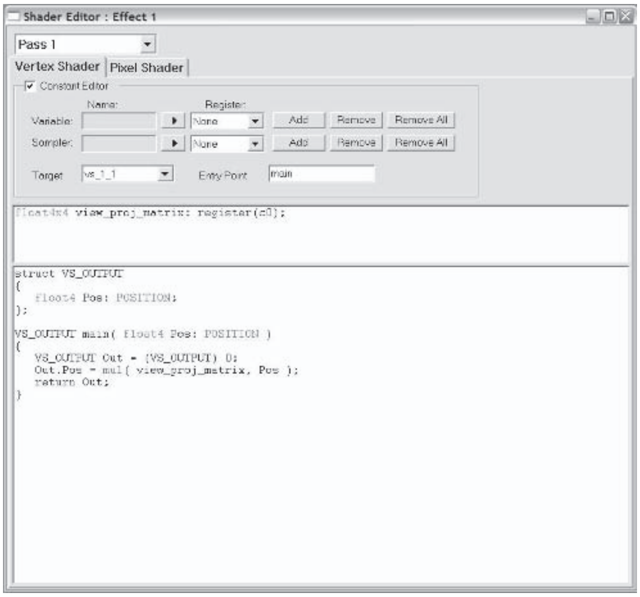
OK Cancel

Figure 14: Adding new shaders

Clicking OK will add a new shader to the selected effect. You can easily spot what type of shaders the effect has; DirectX assembly shaders will have the icon for the vertex shaders and for the pixel shader, and DirectX HLSL shaders will have the icon for the vertex shaders and for the pixel shaders. RenderMonkey will automatically choose the shader editor for each shader, depending on its version. Note that you can only have one of each vertex and pixel shader in an individual pass. If you wish to change shader types (for example, replace an assembly shader with an HLSL shader), you need to first delete the old shader and add a new one in its place.

Editing Shaders

Since we already have a pass with a pair of shaders, let's start working on the actual shader code at this point. To edit each shader, you should double-click on that shader node. RenderMonkey will open the shader editor for your shader. There is a single shader editor window for all the passes in a single effect. Figure 15 shows the shader editor user interface containing the HLSL vertex shader.



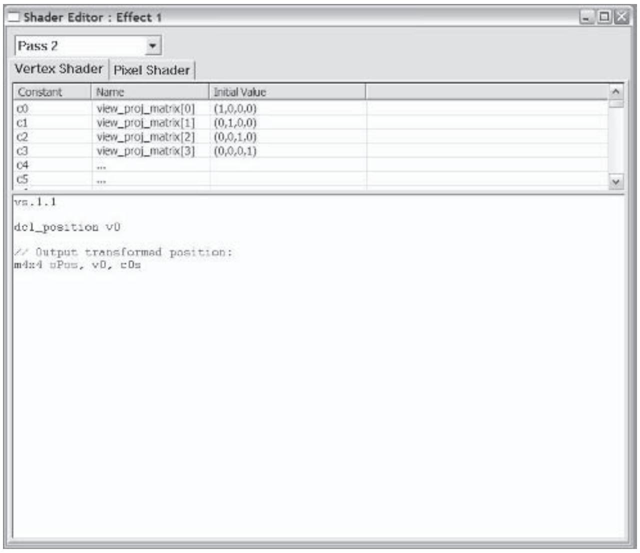
Target: vs_1_1 Entry Point: main

```
float4x4 view_proj_matrix: register(c0);
```

```
struct VS_OUTPUT
{
   float4 Pos: POSITION;
};

VS_OUTPUT main( float4 Pos: POSITION )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;
   Out.Pos = mul( view_proj_matrix, Pos );
   return Out;
}
```

Figure 15: Shader Editor window (HLSL shader)

As you can see from the UI above, the shader editor has two tabs for a vertex and a pixel shader for each pass. The UI for the actual shader editing is selected according to the shader type; see Figure 16 for a snapshot of the assembly shader editor UI.



Constant	Name	Initial Value
c0	view_proj_matrix[0]	(1,0,0,0)
c1	view_proj_matrix[1]	(0,1,0,0)
c2	view_proj_matrix[2]	(0,0,1,0)
c3	view_proj_matrix[3]	(0,0,0,1)
c4	...	
c5	...	

```
vs.1.1

dcl_position v0

// Output transformed position:
m4x4 oPos, v0, c0s
```

Figure 16: Shader Editor window (Assembly shader)

To edit shaders in a different pass, you simply need to select the pass from the top-left combo box in the main Shader Editor window. The tabs for vertex and pixel shaders will be updated to show the shaders in the new pass. You can use Ctrl+Tab to quickly switch between the vertex and pixel shader tabs.

Vertex Shader Setup and Editing

Let's add some code to the vertex shader in Pass 1. Below is the code listing that we will be adding. It is a vertex shader for computing Phong illumination.

```
float4x4 view_matrix;
float4x4 view_proj_matrix;
float4   lightDir;

struct VS_OUTPUT
{
   float4 Pos    : POSITION;
   float3 Norm   : TEXCOORD0;
   float3 View   : TEXCOORD1;
   float3 Light  : TEXCOORD2;
};

VS_OUTPUT main(
   float4 inPos  : POSITION,
   float3 inNorm : NORMAL )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;

   // Output transformed position:
   Out.Pos = mul( view_proj_matrix, inPos );

   // Output light vector:
   Out.Light = -lightDir;

   // Compute position in view space:
   float3 Pview = mul( view_matrix, inPos );

   // Transform the input normal to view space:
   Out.Norm = normalize( mul( view_matrix, inNorm ) );
   // Compute the view direction in view space:
   Out.View = - normalize( Pview );

   return Out;
}
```

This vertex shader transforms the vertex position and outputs it from the vertex shader. Then it computes the light vector using a shader parameter named lightDir (we will be adding all shader constants after we're done creating our shaders). It also computes the vertex position in view space using another RenderMonkey predefined variable, view_matrix, and computes the view vector and the normal vector in view space and outputs those to the pixel shader.

Before we add this code to the shader itself, let's go over the user interface for editing HLSL shaders in RenderMonkey first.

The High Level Shading Language (HLSL) editor consists of three sections. The UI widgets at the top of the editor are used to manage shader parameters for HLSL shaders. The text editor control in the middle portion of the editor is used to view the declaration block of an HLSL shader that contains parameter declarations. This editor pane is not editable; the declaration block is solely controlled through the UI widgets in the top portion of the editor. This is necessary to ensure that the RenderMonkey variable nodes and texture objects get properly mapped to HLSL parameters. The bottom pane is the editor widget to edit the actual shader text (take a look at Figure 15 again). Note that once you're done mapping your constants and samplers, you can simply minimize the Constant Editor block by selecting the check box on top of it:

.

To map a RenderMonkey variable node (a vector, a color, a matrix, or a scalar node), you can left-click on the arrow button next to the variable's Name label:

. This action opens up a pop-up menu containing a list of all variable nodes within the scope of the shader being edited. You can then select a variable node from that pop-up menu:

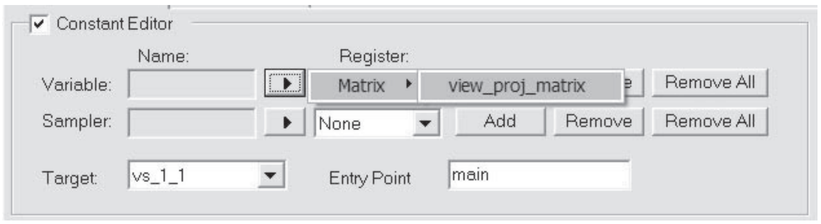


Figure 17: Adding variables to HLSL shaders

At that point, the label under the Name column will change to the name of the node that you selected. Next you should click on the Add button to add that variable node to the declaration block and map it internally as a shader constant. You will then see the actual text declaring that variable appears in the declaration block of the shader.

Let's add the light direction vector to our workspace and map it to a constant in the vertex shader that we are writing. Right-click on the effect workspace node and select Add Variable. Then select Vector as the variable type and type lightDir in the name field. You can leave the Artist Editable check box empty if you wish. Clicking OK will add a new light direction vector to the workspace tree, and you'll see a node like this in it: lightDir. Go to the vertex shader editor that we already opened, and follow the steps for mapping the light direction vector to a constant in the shader editor. After you click Add, you will see the text `float4 lightDir;` appear in the shader's declaration block. We've just added our first constant to the shader!

The next parameter that this shader uses is a view matrix. Since it's a predefined RenderMonkey variable, you won't be able to modify its values explicitly. Let's add it to the workspace first. Right-click on the effect workspace node and select Add Variable. Select Matrix as the variable type. You will see that the Name edit field changed to a combo box. Expand that combo box and select view_matrix from the list of variables that appear. After clicking OK, you will add the predefined view matrix to your workspace. You should see this node appear in the workspace tree now: view_matrix. The little green "p" icon at the bottom-left corner always lets you know that it is a RenderMonkey predefined variable. Follow the same steps described above to add it to the vertex shader declaration block; you will now see the full declaration block appear (though not necessarily in that order):

```
float4x4 view_proj_matrix: register(c0);
float4 lightDir;
float4x4 view_matrix;
```

Now you can simply type the rest of the shader code (the actual main function and vertex shader output structure declaration) into the shader text editor window.

Readers should note that for High Level Shading Language parameter definitions, the RenderMonkey nodes they desire to map must be named within the constraints of the High Level Shading Language; otherwise, improper naming will result in compilation errors. Please refer to the HLSL language manual for more information on naming conventions.

By default, an HLSL shader entry point is set to main, which is actually what we want for both shaders. If you wish to change the entry point for your shader, you can do that by typing a different name in the entry point edit field: Entry point: main .

Since every HLSL shader must provide a compilation target, we need to specify that as well. By default, RenderMonkey's HLSL added shaders have vs_1_1 and ps_1_4 shader targets. To change the version of the shader to which you are compiling, you should select from a list of available targets from the Target combo box: Target: ps_2_0 . The target sets are separate for pixel and vertex shaders — please refer to High Level Shading Language documentation for an explanation of each target value.

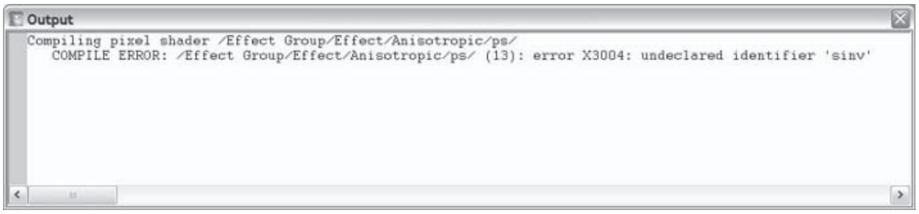
The bottom pane is used to enter the actual text of the shader. The shader text must contain at least one function with the same name as the specified entry point for the shader to compile. The shader text editor has High Level Shading Language customizable syntax coloring.

Compiling Your Shaders

Now that we have entered our vertex shader, the very next thing that any shader developer wants to do is compile it and make sure that the shader is actually correct. To do that, you should click on the Commit Changes button (with this icon:) on the main toolbar. You can also use the accelerator key (F7) to start shader compilation. Commit Changes compiles all modified shaders in the workspace and outputs the compilation results into the Output window.

Output Window

The Output module is a docked window typically located on the bottom of the main application interface (see Figure 18). That window is used to output the results of shader compilation and other application text messages. The Output window is linked with the shader editor for compilation error highlighting.

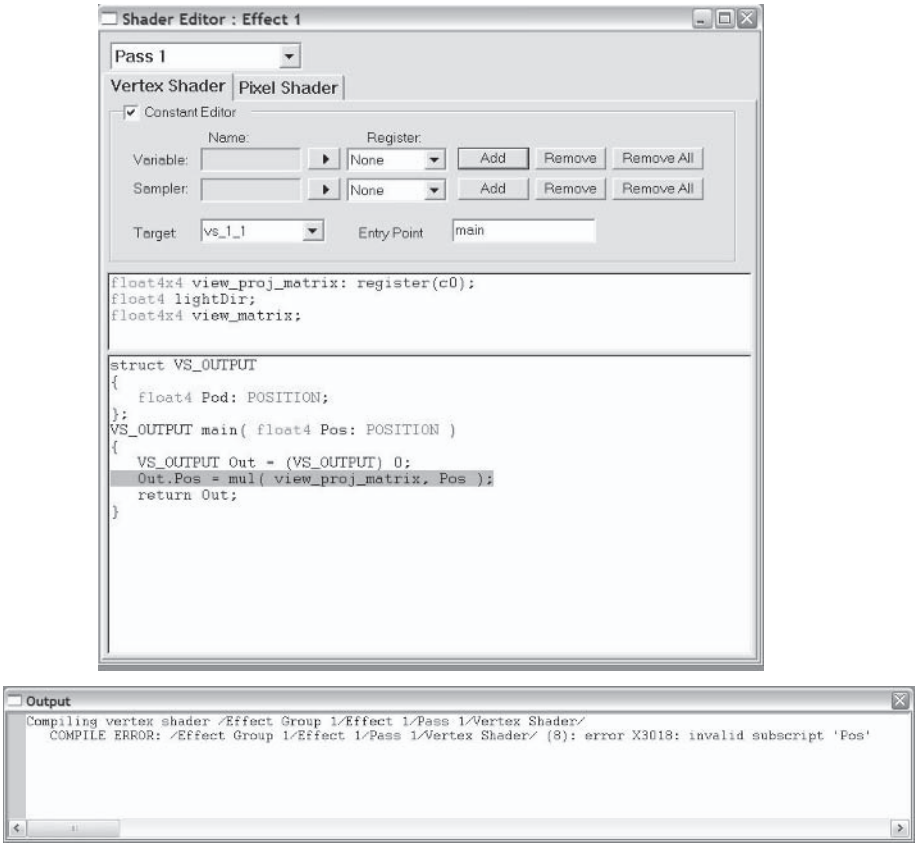


```
Compiling pixel shader /Effect Group/Effect/Anisotropic/ps/
   COMPILE ERROR: /Effect Group/Effect/Anisotropic/ps/ (13): error X3004: undeclared identifier 'sinv'
```

Figure 18: Output window

Shader Assembly or Compilation Errors

Once you press the Commit Changes button, any errors in your shader will appear in the Output window. You will see all errors appear grouped by the full shader name using its path in the effect workspace that you are editing. Pressing that button not only compiles the current shader, but it also internally saves the changes to the code of the shader. The Commit Changes action applies to all open shaders that were modified. If you have errors in multiple files, you will see errors linking to correct files. Double-clicking on shader errors will open the correct window for the shader and highlight the line containing an error (see an example in Figure 19). If you modified the shader text and then closed the editor without committing the changes, RenderMonkey will ask whether you would like to commit the changes first.



Variable: Register: None
Sampler: Register: None
Target: vs_1_1
Entry Point: main

```
float4x4 view_proj_matrix: register(c0);
float4 lightDir;
float4x4 view_matrix;
```

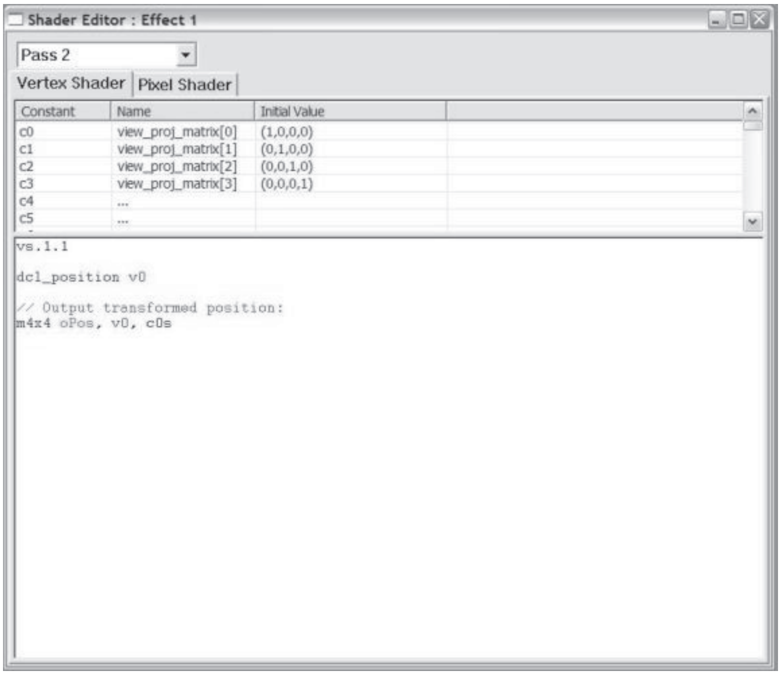
```
struct VS_OUTPUT
{
   float4 Pod: POSITION;
};
VS_OUTPUT main( float4 Pos: POSITION )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;
   Out.Pos = mul( view_proj_matrix, Pos );
   return Out;
}
```

```
Compiling vertex shader /Effect Group 1/Effect 1/Pass 1/Vertex Shader/
   COMPILE ERROR: /Effect Group 1/Effect 1/Pass 1/Vertex Shader/ (8): error X3018: invalid subscript 'Pos'
```

Figure 19: Compilation error for an HLSL shader

Editing Assembly

Although we do not edit assembly shaders in this particular example, this section describes how to edit assembly shaders. The assembly Shader Editor window consists of two panes; the top pane is used to bind RenderMonkey variable nodes to shader constant registers, and the bottom pane is used to edit the shader text. You can see a snapshot of the assembly Shader Editor window in Figure 20.



Constant	Name	Initial Value
c0	view_proj_matrix[0]	(1,0,0,0)
c1	view_proj_matrix[1]	(0,1,0,0)
c2	view_proj_matrix[2]	(0,0,1,0)
c3	view_proj_matrix[3]	(0,0,0,1)
c4	...	
c5	...	

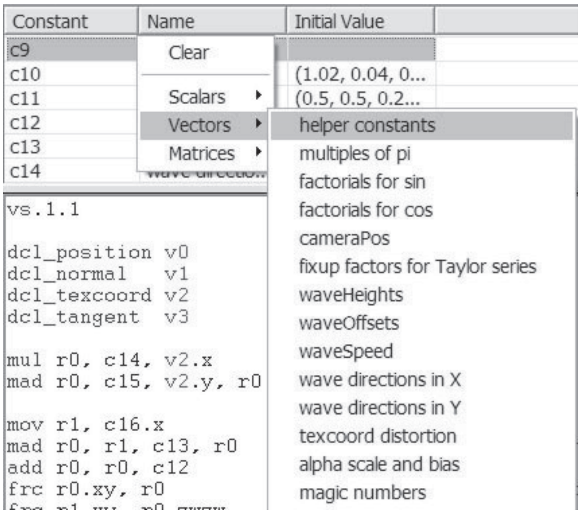
```
vs.1.1

dcl_position v0

// Output transformed position:
m4x4 oPos, v0, c0s
```

Figure 20: Assembly Shader Editor window

The constant store editor is a list view with three columns. Each row represents values for one particular register. The first column (Constant) can be used to specify the index of the register for that constant. The second column (Name) shows the name of the node that is linked to that register (or "…" if there isn't a variable linked to that register). The third column shows the initial value of the variable node linked to the register.

Binding a RenderMonkey variable node to a constant store register means that the software will actually bind the internal values of the nodes directly to the register values. Within the RenderMonkey IDE, vector and color nodes are represented by four different floats, scalars are mapped to four floats having the same value, and matrices are represented by 16 floats.

To bind a RenderMonkey node to a register, you should right-click on the field in the Name column for the constant and select a variable node from the pop-up menu (see Figure 21). The pop-up menu contains all variables that are within the scope of the shader being edited. Once a node is selected, its name will appear in the Name column for the selected register, and the current values of the node will be displayed in the Initial Value column.



Constant	Name	Initial Value
c9		
c10		(1.02, 0.04, 0...
c11		(0.5, 0.5, 0.2...
c12		
c13		
c14	wave directio...	

```
vs.1.1

dcl_position v0
dcl_normal   v1
dcl_texcoord v2
dcl_tangent  v3

mul r0, c14, v2.x
mad r0, c15, v2.y, r0

mov r1, c16.x
mad r0, r1, c13, r0
add r0, r0, c12
frc r0.xy, r0
```

Figure 21: Mapping a RenderMonkey node to an assembly shader constant register

To clear a constant store register, you can select the Clear menu option from the pop-up menu for the register. The name of the variable previously linked to that node is replaced by "…", and the Initial Value column will be cleared.

Please note that if you bind a matrix to a particular constant, the three constants below that constant are overwritten with the rows of that matrix.

The source editor has support for customizable syntax coloring for pixel and vertex shader assembly code. There is also full clipboard support for standard editing operations.

Pixel Shader Setup and Editing

Now that we have a full vertex shader in place, let's fill in the pixel shader. To start editing the pixel shader, select the Pixel Shader tab in the Shader Editor window. But before we can start editing the text of the shader, we need to go back to our lighting equation and figure out what parameters we need to use. If we look at the equations (2) to (4) we can see that they use these parameters for illumination result:

- k_a, k_d, k_s: The coefficients for ambient, diffuse, and specular light contributions, respectively
- I_a: The ambient light intensity
- I_d: The intensity of the diffuse contribution of the point light source that we are simulating
- I_s: The intensity of the specular contribution of the point light source that we are simulating
- n_s: The specular-reflection parameter, proportional to the angle ϕ between the view and reflection vectors

All of the parameters above need to be added as constants to the pixel shader, where we will be directly computing the result of the illumination equation. k_a, k_d, k_s, and n_s can be added as scalar variables to the workspace, and I_a, I_d, and I_s can be added as colors. You should add variable nodes with the following names and types to the main effect workspace node:

- k_a: Scalar variable named *Ka*
- k_d: Scalar variable named *Kd*
- k_s: Scalar variable named *Ks*
- n_s: Scalar variable named *Ns*
- I_a: Color variable named *Ia*
- I_d: Color variable named *Id*
- I_s: Color variable named *Is*

Here's a snapshot of the workspace tree view that you will have once you've completed this operation:

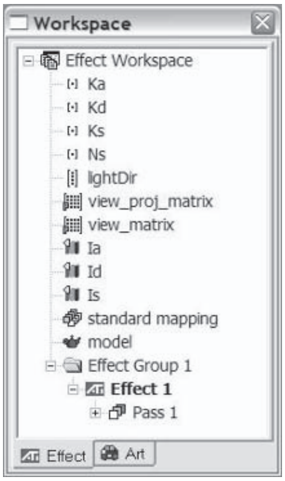


Figure 22: Workspace with all parameters for Phong specular illumination model

Let's add these parameters to the pixel shader's declaration block. Go through each node that we just added to the workspace (*Ka, Kd, Ks, Ns, Ia, Id,* and *Is*) and add them to the pixel shader declaration using the steps described in the vertex shader editing section. Once you've finished adding the last node, you should see the following pixel shader declaration block appear:

```
float Ka;
float Kd;
float Ks;
float Ns;
float4 Ia;
float4 Id;
float4 Is;
```

At this point we are ready to start writing the code for our pixel shader. This is where we can really appreciate the simplicity and elegance of writing shaders using a High Level Shading Language (the Microsoft DirectX 9.0 HLSL in our example). If you have ever tried to write assembly shaders, you can certainly appreciate the difference. The code for the complete pixel shader (without the previous declaration block) follows:

```
float4 main( float4 Diff   : COLOR0,
             float3 Normal : TEXCOORD0,
             float3 View   : TEXCOORD1,
             float3 Light  : TEXCOORD2 ) : COLOR
{
   // Compute the reflection vector:
   float3 vReflect =
            normalize( 2 * dot( Normal, Light) * Normal - Light );

   // Compute ambient term:
   float4 AmbientColor = Ia * Ka;

   // Compute diffuse term:
   float4 DiffuseColor = Id * Kd * max( 0, dot( Normal, Light ));

   // Compute specular term:
   float4 SpecularColor =
                Is * Ks * pow( max( 0, dot(vReflect, View)), Ns );

   float4 FinalColor = AmbientColor + DiffuseColor + SpecularColor;

   return FinalColor;
}
```

You can simply type that code into the pixel shader text editor and hit Commit Changes. Remember to set the target field for this pixel shader to ps_2_0 since we are using the pow instruction.

Preview Window

At this point we are done editing our shaders. But to actually see the effect of the code, we need to see the results in some sort of a viewer. In RenderMonkey, the preview window is used to interactively preview effects. All changes to a shader or its parameters update the rendered image in real time, thus truly enabling interactive shader development. Figure 23 shows the DirectX 9.0 preview window for an ocean water effect.

Simple trackball navigation is provided in the standard RenderMonkey preview module:

- To pan across the window, use the up/down/left/right arrow keys.

- To move the camera forward and backward, use the Z and X keys.
- To rotate the scene, use the mouse.

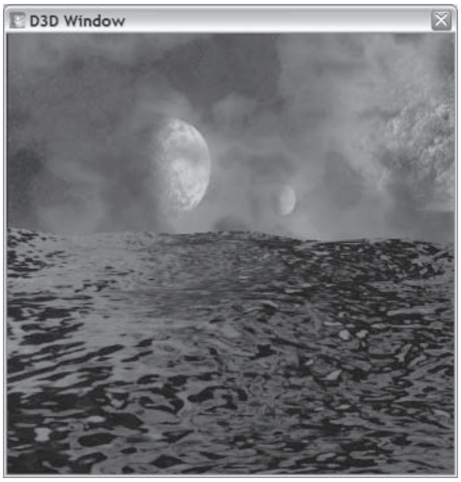


Figure 23: Preview window

Note that the model is rotated about the z-axis in the preview window.

The output of each render pass can be displayed in an arrayed viewport by the use of the P key (as shown in Figure 24 below):

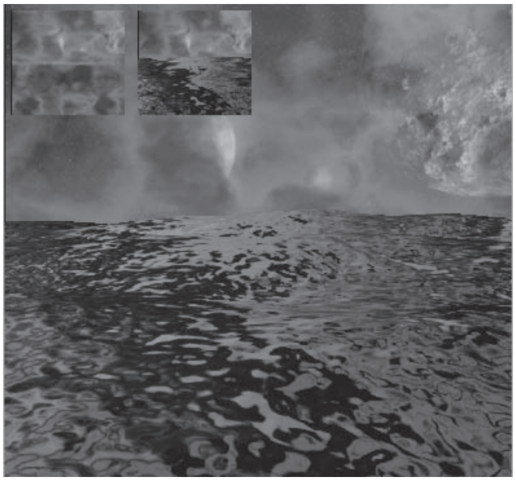


Figure 24: Multipass viewports in the preview window

You can also select from a set of predefined views for your model. To access that, right-click in the preview window and select a view from the list that will appear in that menu (Front/Back/Top/Bottom/Left/Right). You can also modify the properties of the standard preview module by selecting Properties from the right-click menu in the preview window. That action brings up a dialog that allows you to:

- Modify the clear color of the preview window
- Modify the clear color used for the pass array
- Modify the field of view
- Modify the near and far clip plane values

For the currently selected effect, the preview module has the ability to display each pass within a multipass effect in arrayed viewports.

Editing Variables

At this point, the preview window shows a white teapot in constant color. The reason for this look lies in the values for your variables. We need to set meaningful values for all of the parameters to our shaders. But before that, let's talk about how to edit variable nodes in RenderMonkey. To edit a variable, you can either double-click on the variable node or select Edit from the right-click menu for that node. That action will bring up an automatically selected editor for that node type.

Scalar Variables

Each scalar can be edited via the scalar editor module shown in the following figure. Note that you can modify the values in any way, but if you aren't happy with them at the end, you can simply click Cancel and the value set prior to opening the scalar editor will be restored. Note that at any point, the preview window will interactively show the changes.

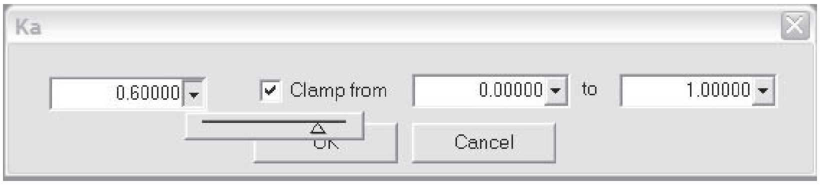


Figure 25: Scalar editor

The scalar can be edited by either directly typing the value in the main edit box or interactively using a pop-up slider, which is in the same range as the clamping bounds (regardless of whether or not the user chooses to clamp the vector).

Let's set the values for the scalars in our workspace to the following values:

- $Ka = 0.8$
- $Kd = 0.8$
- $Ks = 1.0$
- $Ns = 100.0$

Note that right after you do that, you see a white teapot in the preview window. We've turned on our illumination!

Vector Variables

Each vector can be edited via the vector editor module:

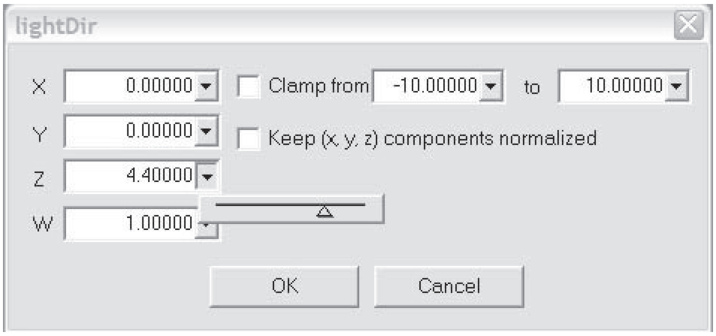

lightDir
X 0.00000
Clamp from -10.00000 to 10.00000
Y 0.00000
Keep (x, y, z) components normalized
Z 4.40000
W 1.00000
OK
Cancel

Figure 26: Vector editor

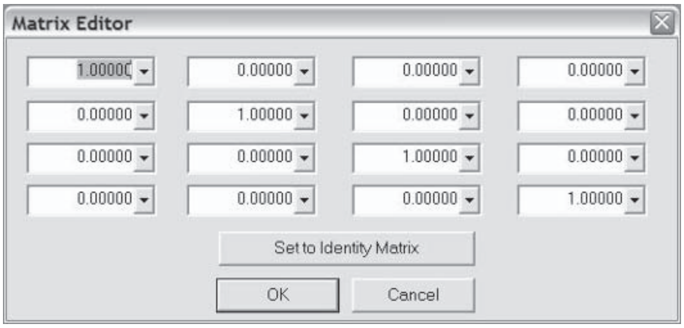
Each vector component can be edited by either directly typing the value in the component edit box or interactively using a pop-up slider for each component. The sliders' ranges will be the same as the clamping bounds for the vector (regardless of whether or not the user chooses to clamp the vector). The user may also select to keep the vector normalized by selecting the Keep (x, y, z) components normalized check box. You can revert your changes in the same way as you could in the scalar editor by pressing the Cancel button.

Let's set the values for the light direction vector for our vertex shader. Double-click on the lightDir variable and enter the following values as its components:

- X = –0.4
- Y = 0.3
- Z = 0.8
- W = 0.0

Matrix Variables

Although we aren't going to modify any matrix variables in this example, to edit a matrix variable you can use the matrix editor module shown below:



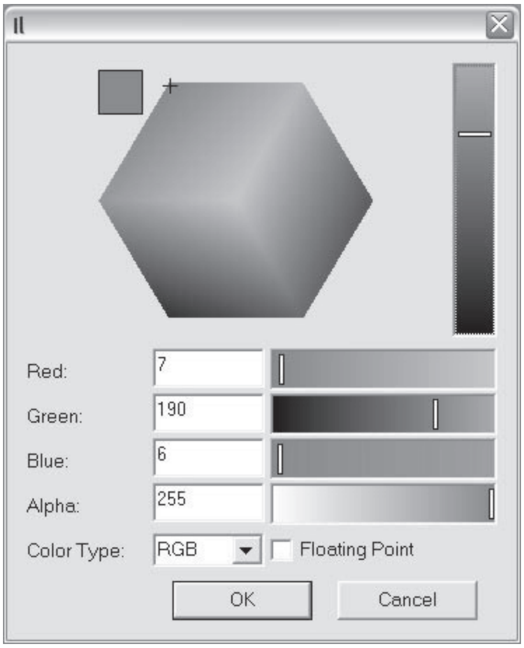
1.0000C	0.00000	0.00000	0.00000
0.00000	1.00000	0.00000	0.00000
0.00000	0.00000	1.00000	0.00000
0.00000	0.00000	0.00000	1.00000

Figure 27: Matrix editor

Each matrix component can be edited by either directly typing the value in the component edit box or interactively using a pop-up slider for each component. The slider range is preset to be in the range [–100.0, 100.0]; however, typing a value outside of that range expands the range to that value. The user can also set the matrix to an identity matrix by clicking the appropriately named button. You can revert your changes in the same way as you could in the scalar editor by pressing the Cancel button.

Color Variables

Each color variable can be edited via the color picker module:



Red: 7
Green: 190
Blue: 6
Alpha: 255
Color Type: RGB
Floating Point

Figure 28: Color editor

The user can edit color using either RGB or HSV mode by directly typing the values in the appropriate edit boxes for each component (R, G, B, A or H, S, V, A), interactively selecting color from the color wheel or color sliders for each component, or modifying the intensity of the color being edited by using the vertical intensity slider. The value of the color is shown in the color swatch at the top-left corner of the color picker. You can also choose to edit color values directly in floating-point format by checking the Floating Point check box and typing the values in the range [–1.0, 1.0] directly into the Red, Green, Blue, and Alpha edit fields. The negative values can be used in the shaders to subtract colors. You can also revert your changes the same way you could in the scalar editor by pressing the Cancel button.

If we set the values for the Ia parameter to R = 0, G = 112, B = 0, and A = 255, we can see the image in Figure 29: our Phong-shaded teapot!

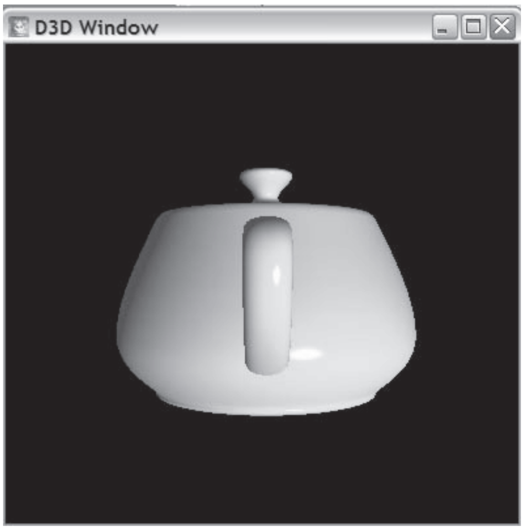


Figure 29: Phong specular illumination effect

Render State Block Management

Although we've completed our first visual effect in Render-Monkey, we should also explore how to modify render states for each draw call. Each pass may have a number of render states that it wants to either inherit from a higher-level pass or set directly. To create a render state node, you can right-click on a pass to which you would like to add the block, and select Add Render State Block from the pass context menu in Figure 30.

If no render state block is defined within a pass, the application traverses the workspace tree upward from the current pass to find a render state node and inherits the render states from the first render state block found. When you create a render state node in a pass, it inherits the values from the first higher-level

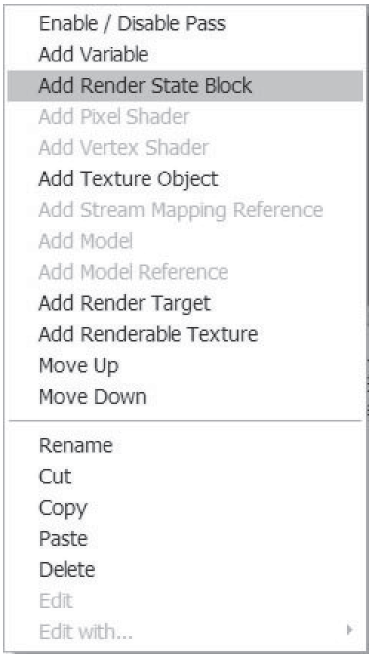
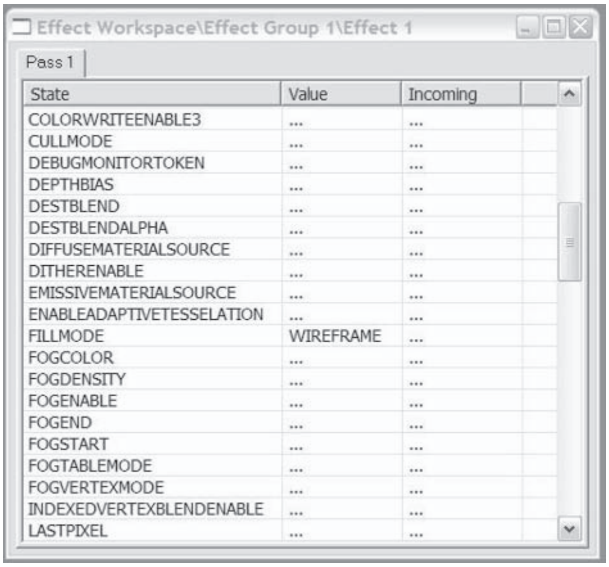


Figure 30: Adding the render state block from the pass context menu

render state block found in the workspace tree. If there are no other render state blocks found prior to the one created, it does not inherit any values. Changing the render state values in the created render state node overrides inherited values. Note that for upward traversal, the application only looks in the passes within the current effect and the default effect. The render state blocks in other effects don't propagate their values. To edit any of the render states in a render state block, you can double-click on the render state node or right-click on the node and select Edit from the node context menu. The render state editor window will appear, as shown in Figure 31 on the following page.

To edit a particular render state, click in the Value column for that render state and either select from a set of predefined values or type a value directly if none were supplied (see the above example for the blending op).



State	Value	Incoming
COLORWRITEENABLE3	...	...
CULLMODE	...	...
DEBUGMONITORTOKEN	...	...
DEPTHBIAS	...	...
DESTBLEND	...	...
DESTBLENDALPHA	...	...
DIFFUSEMATERIALSOURCE	...	...
DITHERENABLE	...	...
EMISSIVEMATERIALSOURCE	...	...
ENABLEADAPTIVETESSELATION	...	...
FILLMODE	WIREFRAME	...
FOGCOLOR	...	...
FOGDENSITY	...	...
FOGENABLE	...	...
FOGEND	...	...
FOGSTART	...	...
FOGTABLEMODE	...	...
FOGVERTEXMODE	...	...
INDEXEDVERTEXBLENDENABLE	...	...
LASTPIXEL	...	...

Figure 31: Render state editor

Let’s display our celadon teapot in wireframe. That’s very simple to do — find the Fillmode render state in the editor and set its value to WIREFRAME by right-clicking in the Value column and selecting that option from the menu. You will instantly see the teapot displayed in the preview window in wireframe:

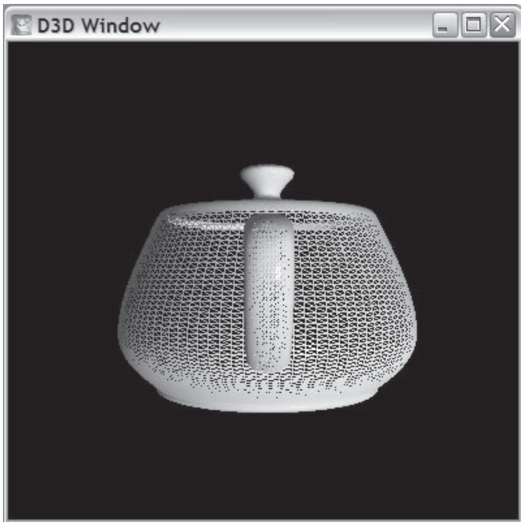


Figure 32: Wireframe display

Texturing in RenderMonkey

All games these days use various texture maps for their visual effects. Let's learn how to use texture maps in RenderMonkey. As you have learned previously, RenderMonkey has special variable types for 2D textures, cube maps, and volume textures. Let's add a 2D texture map variable to our workspace. Right-click on the effect workspace node and select Add Variable from the menu. Select Texture as type, and type baseMap into the name field. By default, all textures are added as artist-editable variables. You will see a texture variable appear in your workspace.

Next, in order to use texturing for our effect, we need to have texture coordinates stream into the vertex shader. Double-click on the stream map node named standard mapping and add the third channel for texture coordinates: Reg = v2, Usage = TexCoord, UsageIndex = 0, Type = Float2. That creates a new stream channel to feed to the vertex shader.

The next step is to add texture coordinate propagation to the vertex shader. That's very simple — open the shader editor for the vertex shader, and type the following code. The lines shown in bold are the lines that are different from the previous example's vertex shader:

```
struct VS_OUTPUT
{
   float4 Pos    : POSITION;
   float3 Norm   : TEXCOORD0;
   float3 View   : TEXCOORD1;
   float3 Light  : TEXCOORD2;
   float2 Tex    : TEXCOORD3;
};

VS_OUTPUT main(
   float4 inPos  : POSITION,
   float3 inNorm : NORMAL,
   float2 inTex  : TEXCOORD0 )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;
   // Output transformed position:
   Out.Pos = mul( view_proj_matrix, inPos );

   // Output light vector:
   Out.Light = -lightDir;

   // Compute position in view space:
   float3 Pview = mul( view_matrix, inPos );

   // Transform the input normal to view space:
   Out.Norm = normalize( mul( view_matrix, inNorm ) );

   // Compute the view direction in view space:
   Out.View = - normalize( Pview );

   // Propagate texture coordinate to the pixel shader:
   Out.Tex = inTex;

   return Out;
}
```

This forces the vertex shader to propagate texture coordinates to the pixel shader. But to actually sample textures in the pixel shader, we need to bind our texture variable to a texture object.

Texture Objects

To use texture-based variables, you have to first create a texture variable using the Add Variable dialog in the desired location of the workspace. Once that texture variable is created, you need to select a file from which to load the texture. To actually use a texture within a pass, you need to select the desired pass and select the Add Texture Object menu option, as shown in Figure 33 on the following page:

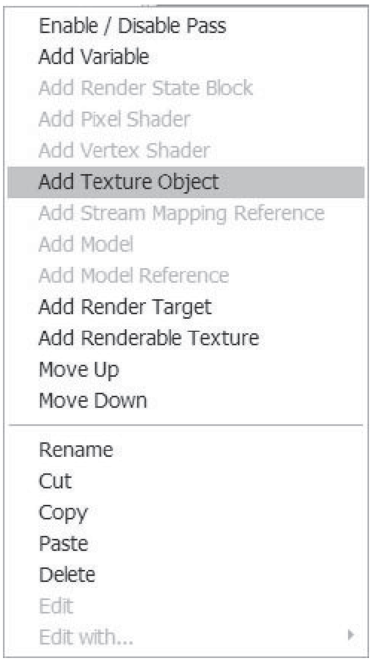


Figure 33: Adding a texture object to a pass

This creates an empty texture object. The texture object that doesn't have a valid texture reference appears with a red line through it: Texture 1. Texture objects map to texture stages used in your shaders, and they are also used to store texture stage and sampler states associated with that texture stage or a sampler. To actually use a texture object in the shader, we need to add a texture reference to it. To do that, right-click on the Pass 1 node and select the Add Texture Reference menu option from the context menu that appears (shown in Figure 34):

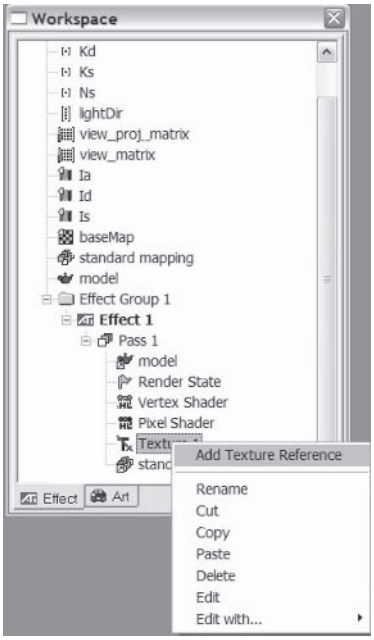
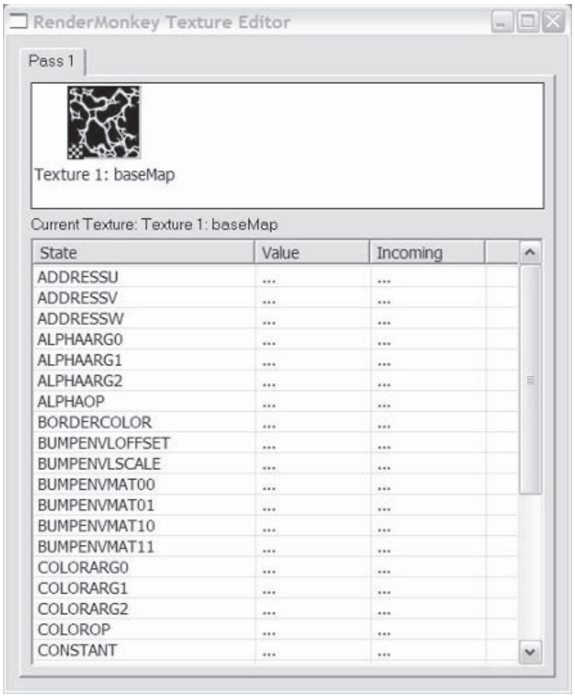


Figure 34: Adding a texture reference to a texture object

This creates an empty texture reference. To actually bind the reference to a texture variable, the user should type the name of the variable that he wants to reference. If a valid texture variable is found successfully, then the red line through the texture reference is removed. A red line across the texture reference icon denotes that the texture variable wasn't successfully referenced. By default, RenderMonkey binds the texture reference to the baseMap texture variable if one is found in the workspace, so we don't need to do anything to bind our texture reference.

If we want to specify some sampler states for our texture map, we need to specify these state values (filtering, clamping, etc.) for a particular texture reference node using the texture editor, which can be launched by double-clicking on a texture reference node. Figure 35 shows the texture editor for three texture objects:



Current Texture: Texture 1: baseMap

State	Value	Incoming
ADDRESSU	...	...
ADDRESSV	...	...
ADDRESSW	...	...
ALPHAARG0	...	...
ALPHAARG1	...	...
ALPHAARG2	...	...
ALPHAOP	...	...
BORDERCOLOR	...	...
BUMPENVLOFFSET	...	...
BUMPENVLSCALE	...	...
BUMPENVMAT00	...	...
BUMPENVMAT01	...	...
BUMPENVMAT10	...	...
BUMPENVMAT11	...	...
COLORARG0	...	...
COLORARG1	...	...
COLORARG2	...	...
COLOROP	...	...
CONSTANT	...	...

Figure 35: Texture editor

The texture editor has tabs for each individual pass within an effect. The top of the texture editor contains a list of texture references within the selected pass. By clicking on a texture icon, you can select to view and set texture states for that texture. To set a particular state, you should click on the Value field next to the state you are trying to edit and either select a value from the predefined set of values for that state or type a value if none was provided. Note that the texture editor displays thumbnails for all texture variables that have a valid file associated with them, and you can see a small icon in the bottom-left corner of each thumbnail showing what type of texture reference it is.

Also note that only the texture objects with valid texture references have the T icon or a thumbnail image. If the texture object's texture reference isn't correctly linked, then that object is displayed with the × icon. The texture editor creates thumbnails for all texture variables; however, for cube maps or volume textures, only the first image is displayed.

Using Textures with HLSL Shaders

To actually use the texture map in our pixel shader, we need to first add a sampler for it to our pixel shader. This is very simple and follows steps similar to adding a constant to the shader in HLSL. You must have a valid texture object with a texture reference to add a sampler. Once you do, you should click on the arrow next to the Sampler label (

), which opens a list of available texture objects that can be mapped as HLSL sampler objects. The name of the texture reference is used as the name for the sampler. Then you can either add or remove that sampler object in the same manner as above. The same restrictions apply, as far as managing nodes that are mapped to sampler objects.

If you wish to bind a parameter to a particular register, you should select the register by clicking on the Register combo box and selecting from the list of registers available:

. Separate register sets exist for variables and sampler mapping.

Let's map our texture object to a sampler. Click on the arrow and select baseMap from the list. Then click Add. You will see the following declaration block for the pixel shader with sampling in the declaration window:

```
float4 Ia;
float Ka;
float4 Is;
float Kd;
float4 Id;
float Ns;
float Ks;
sampler baseMap;
```

Below is the text of the pixel shader modified to use texturing (the lines in bold are updated from the previous example):

```
float4 main( float4 Diff   : COLOR0,
             float3 Normal : TEXCOORD0,
             float3 View   : TEXCOORD1,
             float3 Light  : TEXCOORD2,
             float2 Tex    : TEXCOORD3 ) : COLOR
{
   // Compute the reflection vector:
   float3 vReflect =
             normalize( 2 * dot( Normal, Light) * Normal - Light );

   // Compute ambient term:
   float4 AmbientColor = Ia * Ka;

   // Compute diffuse term:
   float4 DiffuseColor = Id * Kd * max( 0, dot( Normal, Light ));

   // Compute specular term:
   float4 SpecularColor = Is * Ks * pow( max( 0, dot(vReflect, View)), Ns );

   float4 FinalColor =
  (AmbientColor + DiffuseColor) * tex2D( baseMap, Tex ) + SpecularColor;

   return FinalColor;
}
```

Once you compile this shader, you will see a nicely textured teapot appear in the preview window:

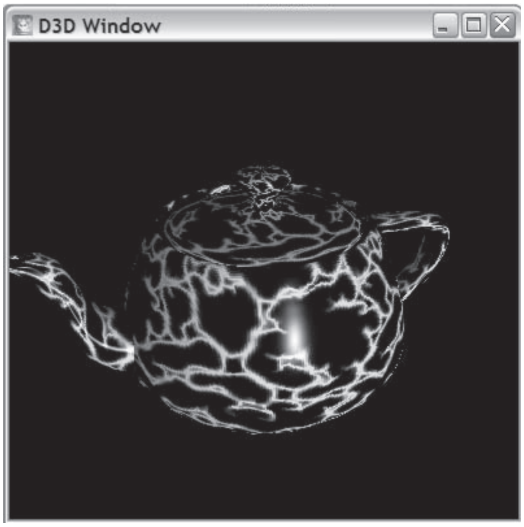


Figure 36: Textured teapot effect

Rendering to a Texture

Let's complicate our effect a little bit. Let's use the output of the first pass (the one that we just created) and funnel it as the input into the second pass. That technique is called *rendering to a texture*, and it can be used for a variety of interesting post-processing effects. (Take a look at the "Real-Time Depth of Field Simulation" (G. Riguer, N. Tatarchuk, J. Isidoro) article in *ShaderX^2: Shader Programming Tips & Tricks with DirectX 9* for an example of depth of field effects using that technique.)

Render Passes

To start working on creating the simplest rendering to a texture-based effect, we need at least two passes. Let's add a new pass to our workspace. To do that, right-click on the effect node and select Add Pass from the menu. By default, each pass is created with a sample HLSL vertex and pixel shader and geometry and stream map reference nodes; you can modify those at any time. Once you add a new pass, you can see a red teapot appear in the preview window again. That's because the passes are drawn in the order in which they are defined within their parent effect. To move a pass up or down, you can right-click on the desired pass and select Move Up or Move Down from the pass context menu shown in Figure 37. You may also use Ctrl+up arrow to move a pass up or Ctrl+down arrow to move the pass down. Try that with the two passes that we have; if you move Pass 2 to be above Pass 1, you will see the textured teapot again. Then if you move it back, the red teapot appears again.

You can also disable a particular pass to aid you in your shader debugging. To do that, you can select Enable/Disable Pass from the pass context menu (accessible by right-clicking on the desired pass). A disabled pass will have this icon on the left of its name to denote that it is disabled: . To enable the pass, just click on the same menu option again.

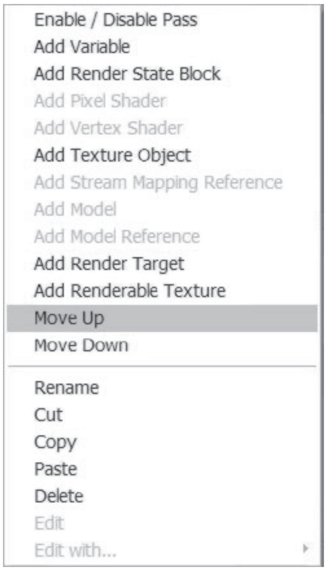


Figure 37: Modifying the pass order from the context menu

The following is an example of a workspace view with a disabled pass:

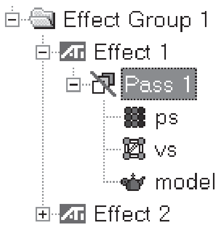


Figure 38: Disabling a pass

Restore the order of passes (Pass 1 before Pass 2) before continuing.

Renderable Texture Support

The RenderMonkey IDE supports the ability to render output of any given pass to a texture and then sample the contents of that texture in a subsequent pass. To add that functionality to your workspace, here is the step sequence you must follow:

1. Create a renderable texture at any point in the workspace. Only one pass can render output to that texture at a time.

   To add a renderable texture, click on any node that you would like to add it to and select Add Renderable Texture from the context menu that appears at that point:

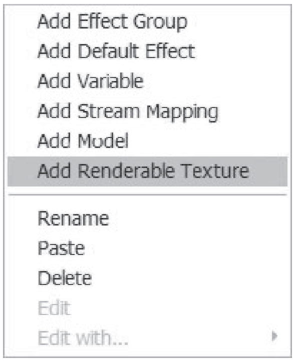
   

   Figure 39: Adding renderable texture

2. You will see a new node appear in the tree with this icon: . This node is the renderable texture node that you will link later to a render target and a texture object to sample from this renderable texture.
3. Next you need to add a render target to the pass that is going to output to the renderable texture. Select the pass node and right-click on it to select the context menu for that pass; choose Add Render Target to add a new render target (the node will have this icon next to it once it's created:).

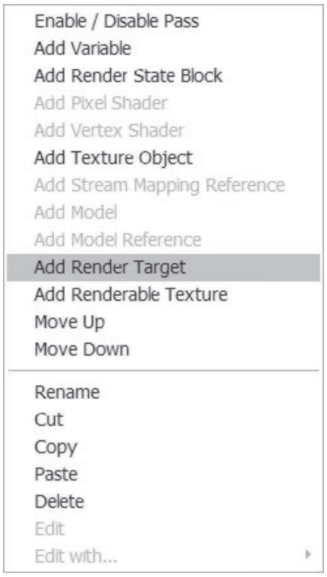


Figure 40: Adding a render target to a pass

4. Next you must link the render target node to the renderable texture that you've created. You can either rename the render target node to *exactly the same* name as the renderable texture node to which you want to link it, or you can right-click on the render target node and select a node to reference from a context menu that will appear:

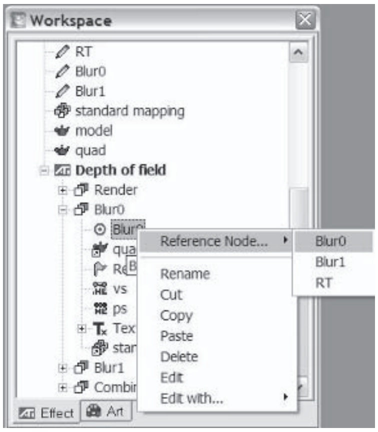


Figure 41: Linking the render target node to a renderable texture variable

5. At this point, the output of the pass that owns the render target node is drawn to the renderable texture.
6. Next, let's link the renderable texture to a pass that is going to sample from it. To do that, you must first create a texture object and a texture reference within that pass (see the section on managing textures above). Once a texture reference exists, you must link it to the renderable texture by either renaming the texture reference node to *exactly the same* name as the renderable texture or by right-clicking on the texture reference node and selecting the renderable texture you want to link it to from the Reference Node menu:

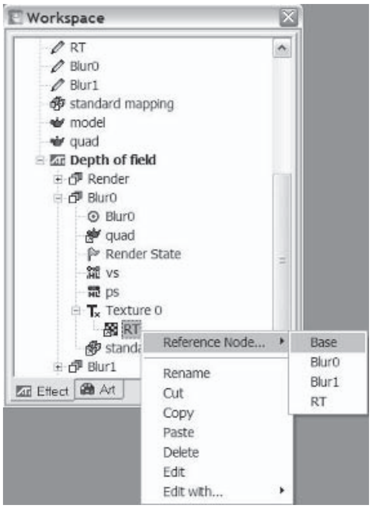


Figure 42: Linking a texture object to a renderable texture variable for sampling

7. At this point, you can use the texture object as you would normally use it in your shader (assembly or HLSL).

Let's add a renderable texture to our workspace. Right-click on the effect workspace node and select Add Renderable Texture from the context menu. Then we need to add a render target to Pass 1 — right-click on that pass and select Add Render Target. Link this render target to the renderable texture that we have created by right-clicking on the render target and selecting renderTexture from the Reference Node menu that appears. You will see that the red line across the render target node disappears and the name of the render target is now renderTexture. At this point, the output of Pass 1 is diverted to the renderable texture variable.

Next we want to add the ability to sample from that texture in our second pass. First we need to make sure that the vertex shader propagates the texture coordinates correctly. Type this text into the vertex shader:

```
struct VS_OUTPUT
{
   float4 Pos: POSITION;
   float2 Tex: TEXCOORD0;
};
VS_OUTPUT main( float4 Pos: POSITION, float2 Tex: TEXCOORD0 )
{
   VS_OUTPUT Out = (VS_OUTPUT) 0;
   Out.Pos = mul( view_proj_matrix, Pos );
   Out.Tex = Tex;
   return Out;
}
```

This ensures that we will be interpolating texture coordinates into the pixel shader. Next, let's add a texture object with a texture reference to Pass 2, following the same steps as in the earlier example. However, instead of linking the texture variable, let's link it to the renderTexture renderable texture variable. This directs the output of Pass 1 to Pass 2. Open the pixel shader for Pass 2 and add renderTexture as a sampler to that pass. Then type this text as the pixel shader code:

```
float4 main( float4 Diff : COLOR0,
             float2 Tex  : TEXCOORD0 ) : COLOR
{
     return tex2D( renderTexture, Tex );
}
```

At this point the preview window shows a green textured teapot (take a look at Figure 43). Set these sampler states for the texture objects in both passes for a nicer rendering result: Minfilter = LINEAR and Magfilter = LINEAR. (The picture below has the preview window's clear color set to a dark gray value.)

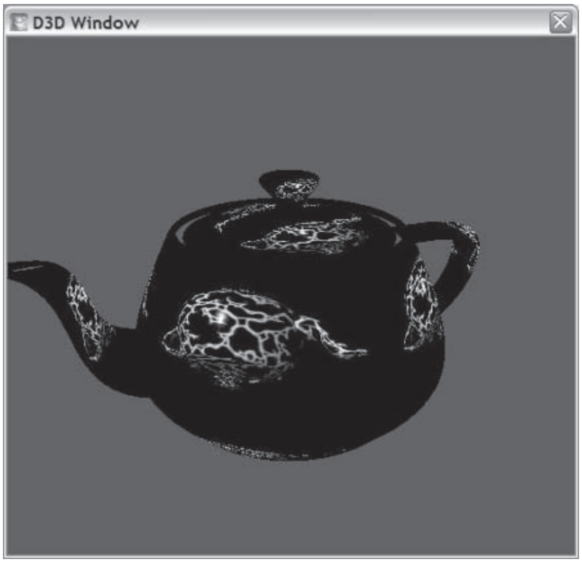


Figure 43: Render-to-texture effect

Editing a Renderable Texture

To edit a renderable texture node, double-click on the node itself (🖉) to open the renderable texture editor module (see Figure 44).

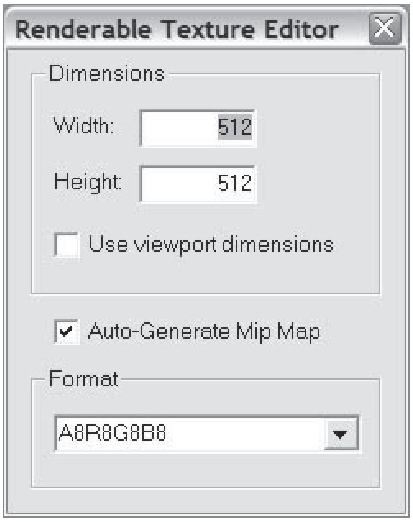
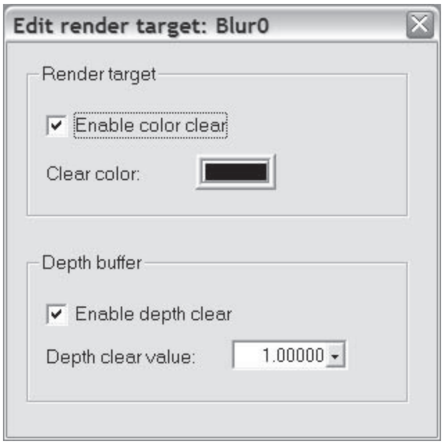


Figure 44: Renderable texture editor

In the renderable texture editor you can change the dimensions of the renderable texture. To change either the width or height of the texture, type the integer dimension into the appropriate edit box and press Enter to propagate the changes and create a new renderable texture. You may also bind the texture to use the dimensions of the current viewport by checking the Use viewport dimensions box. To change the format of the renderable texture, the user can select from a list of predefined formats by selecting them from the Format combo box control.

Editing a Render Target

To edit a render target node, the user should double-click on the node itself ( ) to open the render target editor window:


Edit render target: Blur0

Render target

☑ Enable color clear

Clear color: ■

Depth buffer

☑ Enable depth clear

Depth clear value: 1.00000

Figure 45: Render target editor

From this editor, the user can select whether to clear the renderable texture by checking or unchecking the Enable color clear box. If the user chooses to clear the texture, he can select the color he wishes to clear it to by clicking on the Clear Color button and selecting the color from the dialog that appears. The user can also select whether to enable depth clearing by checking or unchecking the Enable depth clear box. If depth clearing is enabled, the user can select the value used.

Artist Editor

One of the problems that shader developers face in production is how to present the shaders to the 3D artists to allow the artists to experiment with the shader parameters in order to achieve desired effects. RenderMonkey's solution for this problem is the artist editor module combined with the Art tab in the workspace view.

A shader developer can select certain variables in the shader effect workspace to be flagged as "artist-editable" variables. To do that, you can select Artist Editable from the right-click menu for the desired variable node, and a small yellow flag icon will be overlaid over the icon for that variable. Then you can give the Effect Workspace with your shaders to the artists you work with. The artists can select the Art tab from the workspace view to only view artist variables present in the workspace. For added convenience, artists can edit artist variables of supported types in the artist editor module. Currently, the supported types for the artist editor are vectors, scalars, and colors; however, any variable can be flagged as an artist variable and accessed from the Art workspace tab.

To open the artist editor, you can either click the button on the application toolbar or select Artist Editor from the View menu in the main application menu bar.

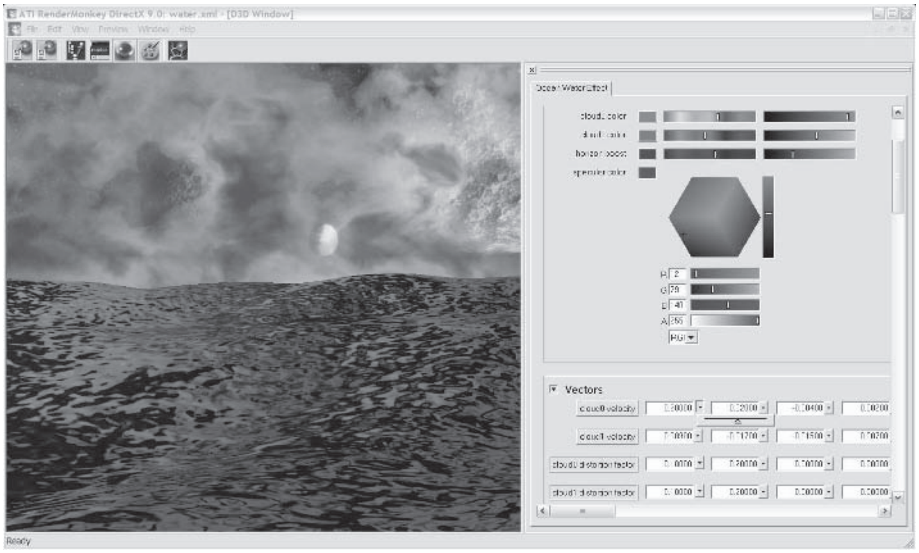
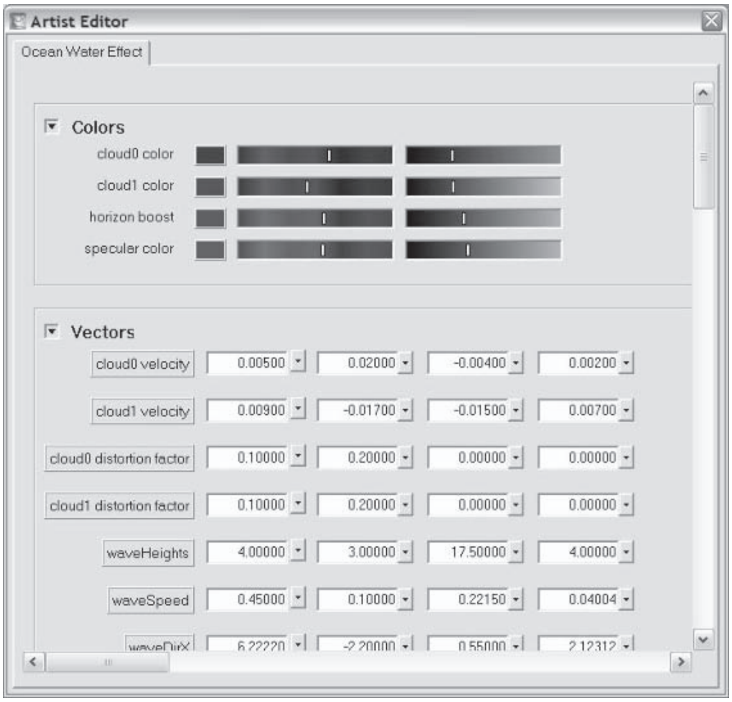


Figure 46: Artist editor interface

The Artist Editor window has tabs for each effect workspace, effect group, effect, or pass that contains artist-editable variables. If the node contains no artist-editable variables of supported types, it won't appear as a tab in the artist editor.

Artist-editable variables are arranged by their types in groups (color, vector, and scalar). Each group can be expanded or collapsed by clicking on the ▼ button within the group.


Artist Editor

Ocean Water Effect

Colors

- cloud0 color
- cloud1 color
- horizon boost
- specular color

Vectors

cloud0 velocity	0.00500	0.02000	-0.00400	0.00200
cloud1 velocity	0.00900	-0.01700	-0.01500	0.00700
cloud0 distortion factor	0.10000	0.20000	0.00000	0.00000
cloud1 distortion factor	0.10000	0.20000	0.00000	0.00000
waveHeights	4.00000	3.00000	17.50000	4.00000
waveSpeed	0.45000	0.10000	0.22150	0.04004
waveDirX	6.22220	-2.20000	0.55000	2.12312

Figure 47: Artist Editor window

Editing Variables in the Artist Editor Module

Colors

Each color variable has three related controls — a color swatch button for opening the full color picker module, a hue slider, and an intensity slider:

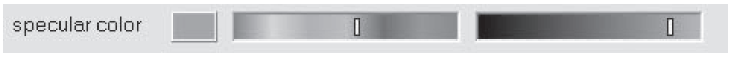
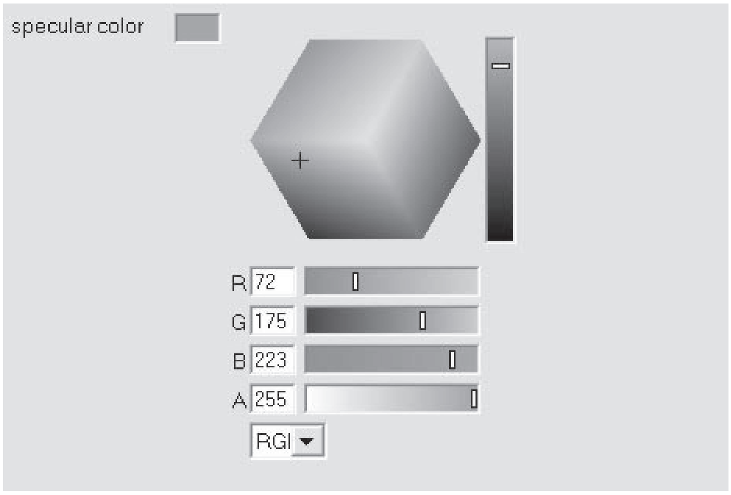


Figure 48: Individual set of controls for editing color in the artist editor

If you click on the ▭ button, you will get an expanded set of controls for editing color with more precision, as shown in Figure 49.


specular color

R 72
G 175
B 223
A 255
RGI

Figure 49: Expanded set of controls for editing color variables in the Artist Editor window

Vectors

Each vector variable has five related controls — a label button that opens up the full vector editor and four component edit boxes with pop-up slider buttons for editing each vector component interactively:

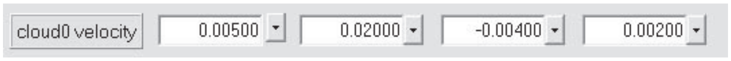
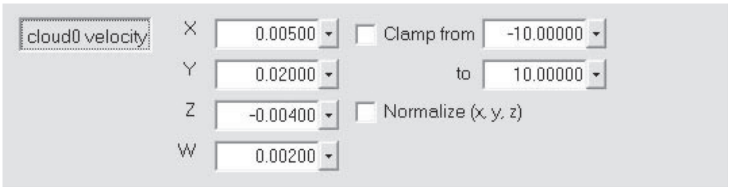

cloud0 velocity 0.00500 0.02000 -0.00400 0.00200

Figure 50: Editing vectors in the artist editor

If the user clicks on the button for a particular vector (cloud0 velocity), he will see an expanded set of controls for editing vectors with more precision and control:

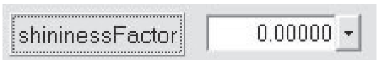


cloud0 velocity
X 0.00500 Clamp from -10.00000
Y 0.02000 to 10.00000
Z -0.00400 Normalize (x, y, z)
W 0.00200

Figure 51: Expanded set of controls for editing vectors in the artist editor

Scalars

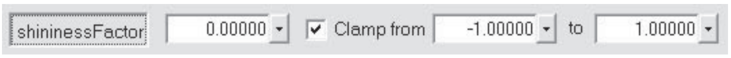
Each scalar has two related controls — a label button that opens up the full scalar editor and an edit box with a pop-up slider button for editing the slider value directly.



shininessFactor 0.00000

Figure 52: Editing scalars in the artist editor

If the user clicks on the shininessFactor button, he can see an expanded set of controls for editing scalar variables in the artist editor:



shininessFactor 0.00000 [✓ Clamp from] -1.00000 to 1.00000

Figure 53: Expanded set of controls for editing scalars in the artist editor

Summary

I hope that this article was helpful in showing you the ease of use and convenience of developing shaders with the RenderMonkey IDE. As with all the tools and samples provided by ATI, we welcome feedback from the developers who spend every day “in the trenches” solving real problems. ATI is committed to providing you with the tools that you need to make your job easier. In order to do this, we need you to tell us what works and what doesn’t. What additions or enhancements would you like to see? What additional problem areas exist that we’re not currently helping with? Please help us to help you by providing as much feedback as possible to devrel@ati.com.

Tips for Creating Shader-Friendly 3D Models

Gim Guan Chua
http://toybox.150m.com

Certain shaders, such as bump-mapping shaders, require the use of tangent space. (For more information on tangent basis and its use in bump-mapping, please refer to nVidia's "The CG Tutorial.")

Since 3D model data typically comes with only vertices, normals, and texture coordinates, a common method is to "automagically" deduce the corresponding tangents and binormals (the tangent basis consists of the normal, binormal, and tangent) based on normals and texture coordinates. This method is convenient and effective, but sometimes it can produce undesirable artifacts. This is due to the following factors:

- It requires suitable texture coordinates.
- It is influenced by "vertex weight," or the number of triangles sharing the same vertex.
- It is ideal for models with convex surfaces but presents problems for models with indentations or protrusions.

Generating Suitable Texture Coordinates

The easiest way to generate texture coordinates is to do a planar projection of the model onto a 2D plane. However, this results in some adjacent vertices having the same texture coordinates; imagine projecting a cube onto the z=0 plane. To remedy that, offset one of the texture coordinates slightly. Figure 1 shows a bevel button having the wrong specular highlight, and Figure 2 shows an "adjusted" bevel button with the correct specular highlight.

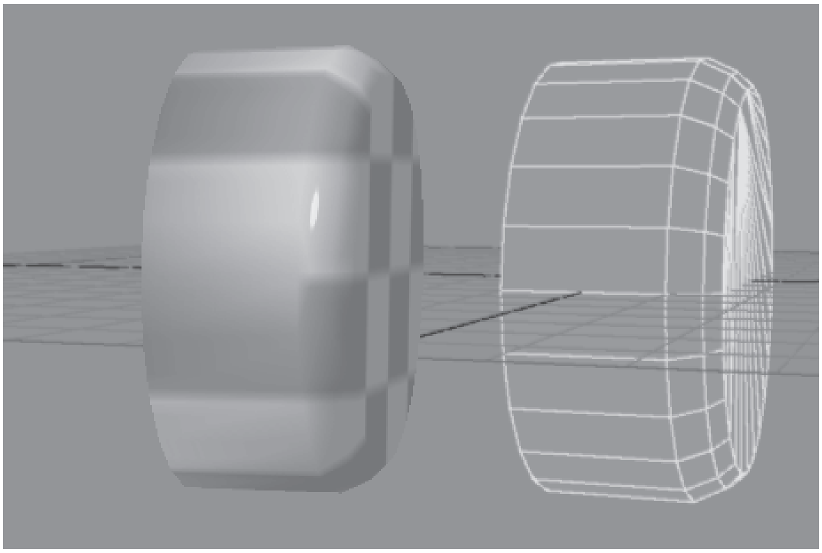


Figure 1: Wrong lighting due to similar texture coordinates

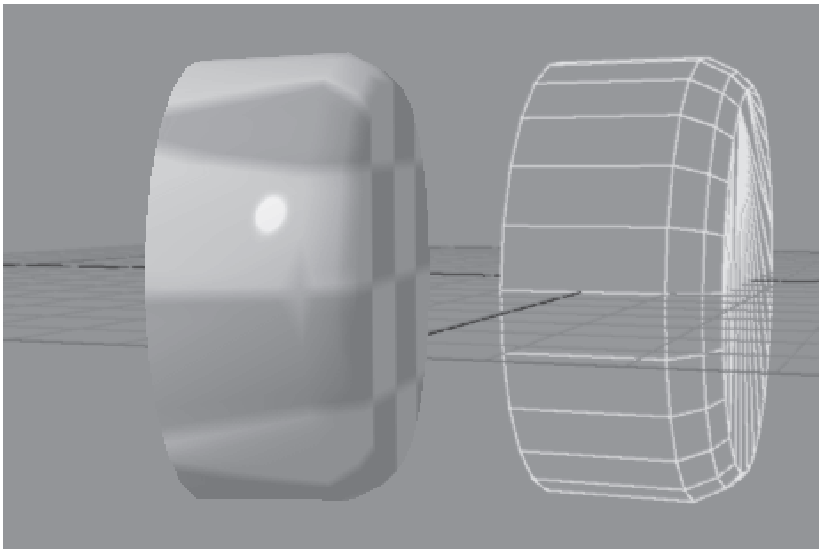


Figure 2: "Adjusted" button showing correct lighting

The Influence of "Vertex Weight"

The components of the tangent basis — vertex normal, tangent, and binormal — are averages of triangles that share the vertex. Therefore, if a vertex is shared by more triangles, its tangent basis is influenced by more triangles. An ideal mesh would be where the vertices are evenly shared, like a strip of triangles. However, meshes are often "auto-tesselated" from higher-order polygons (quads, pentagons, etc.) into triangles with no regard to evenly distributing the sharing of vertices. A way to overcome this is to selectively re-tesselate problem areas, as shown in Figures 3 and 4.

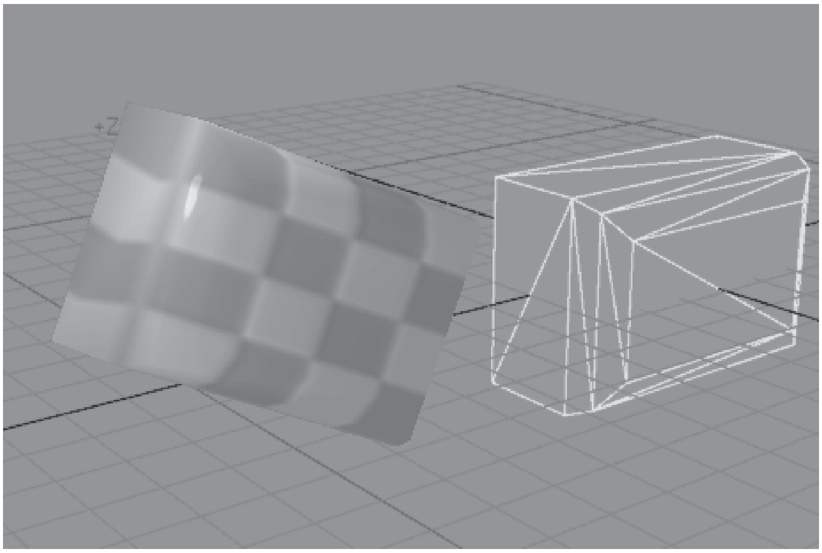


Figure 3: A rendering artifact in the left face

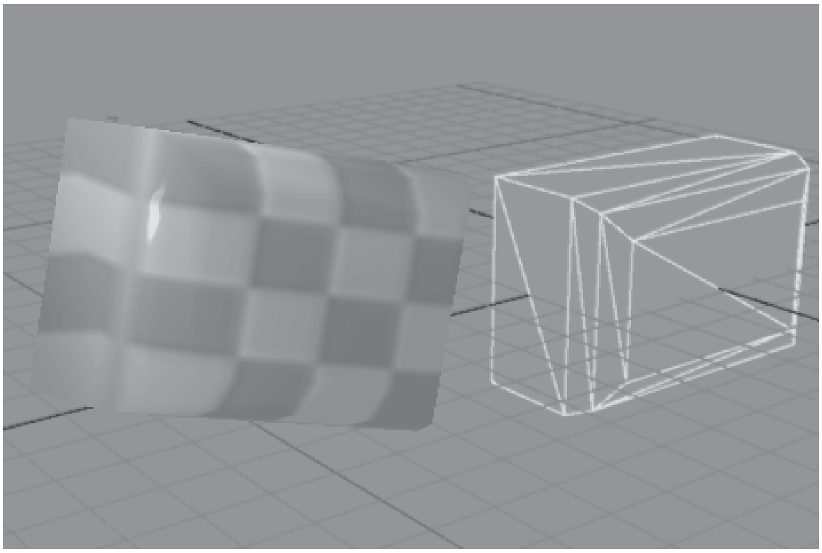


Figure 4: Remedied by changing the way the quad is tesselated

Problems with Non-Convex Surfaces

The tangent basis calculation method is ideal for smooth models with convex surfaces, such as spheres and donuts. However, complex models with indentations and extrusions often mess up the tangent basis. This is again due to the effect of vertex weights. Figure 5 shows the messiness that an indentation did to a plane.

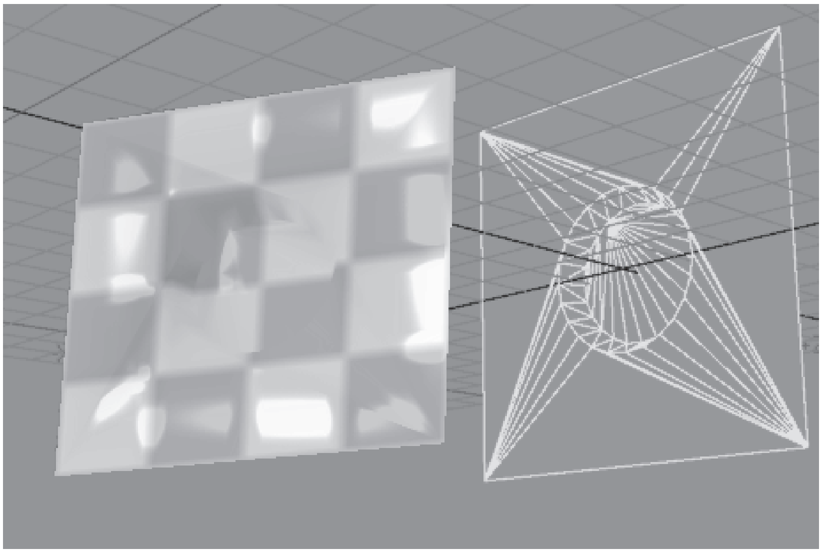


Figure 5: The tesselation has a great effect on bump-mapping and specular highlights.

There are three ways to solve this problem:

- Use a modeling tool that allows for tweaking of normals, tangents, and binormals. Some tools support normal tweaking, but tangent and binormal adjustment is rare.
- Break the model apart. Figure 6 shows such a case. The result is a total discontinuity between the two meshes.
- Additional tesselation to buffer or soften the effects of the discontinuity. Figure 7 shows the result. This actually preserves a little continuity, as seen by the highlights around the indentation compared to Figure 6.

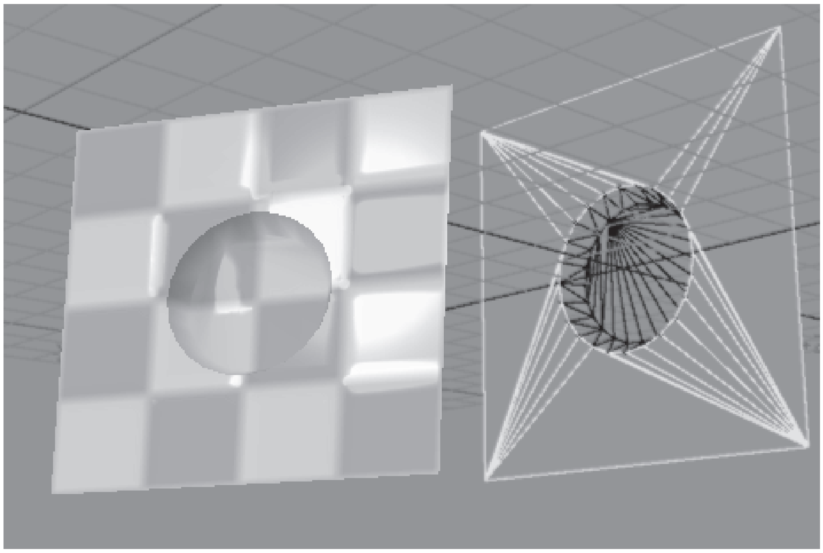


Figure 6: Breaking the mesh into two

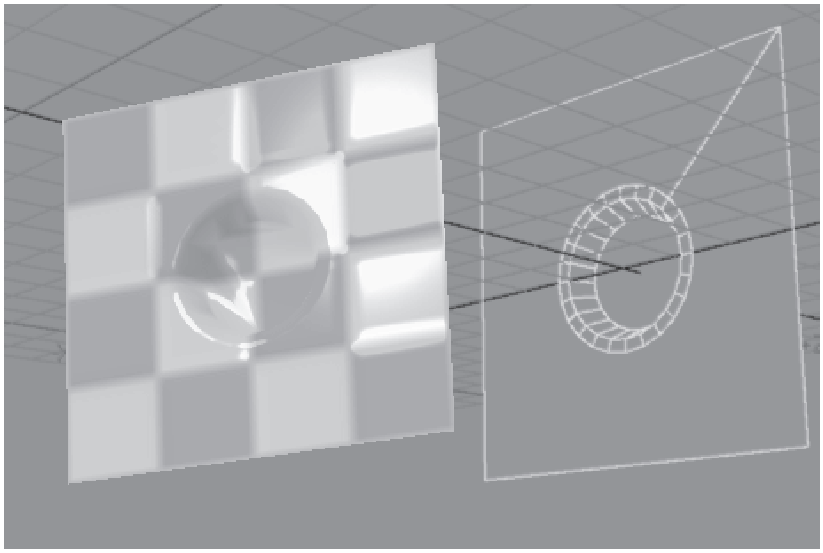


Figure 7: Buffered tesselation, with the plane before triangulation

Conclusion

The combination of the three methods (generating suitable texture coordinates, re-tesselating to distribute vertex weights more evenly, and buffered tesselation to soften the effect of discontinuity) is effective in creating complex models that would render bump maps, specular highlights, and other tangent basis-dependent effects correctly. It does not need changes to modeling tools or shaders. Instead, it only requires a little more work on the part of the modeler to tweak the model to become shader-friendly.

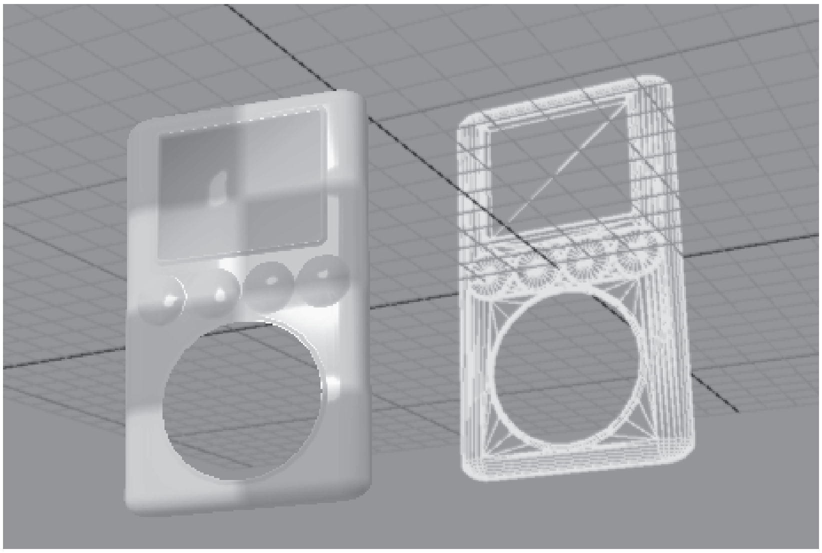
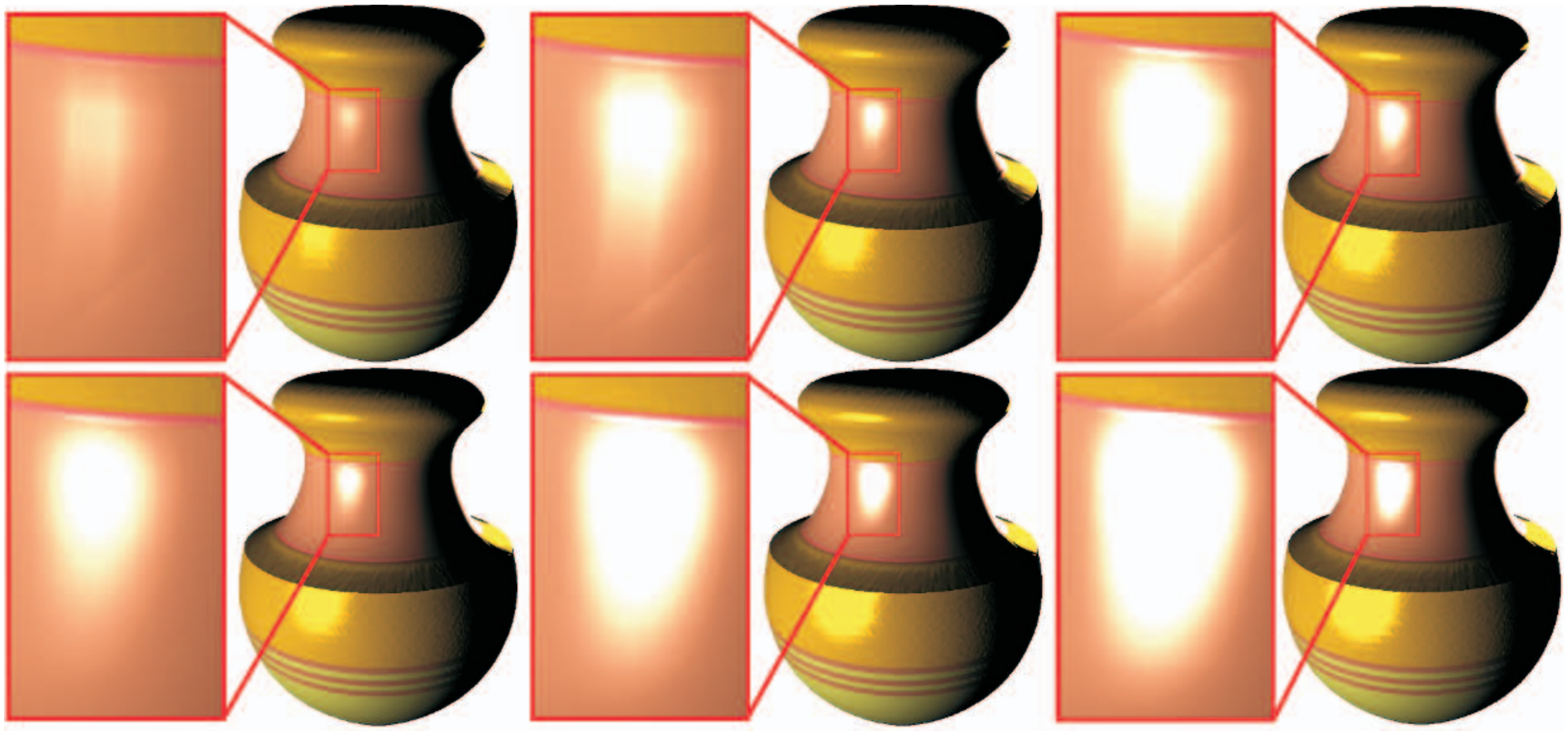
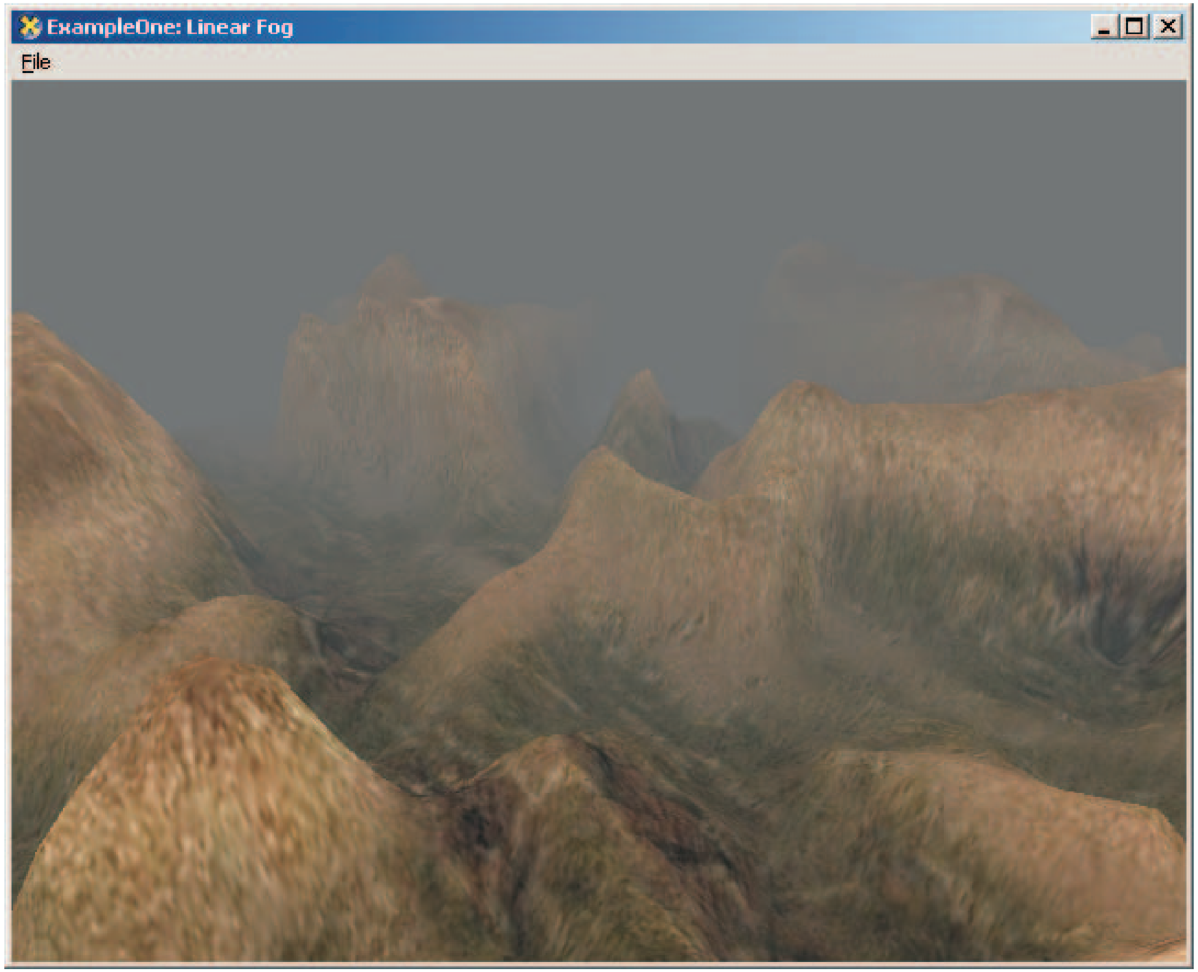
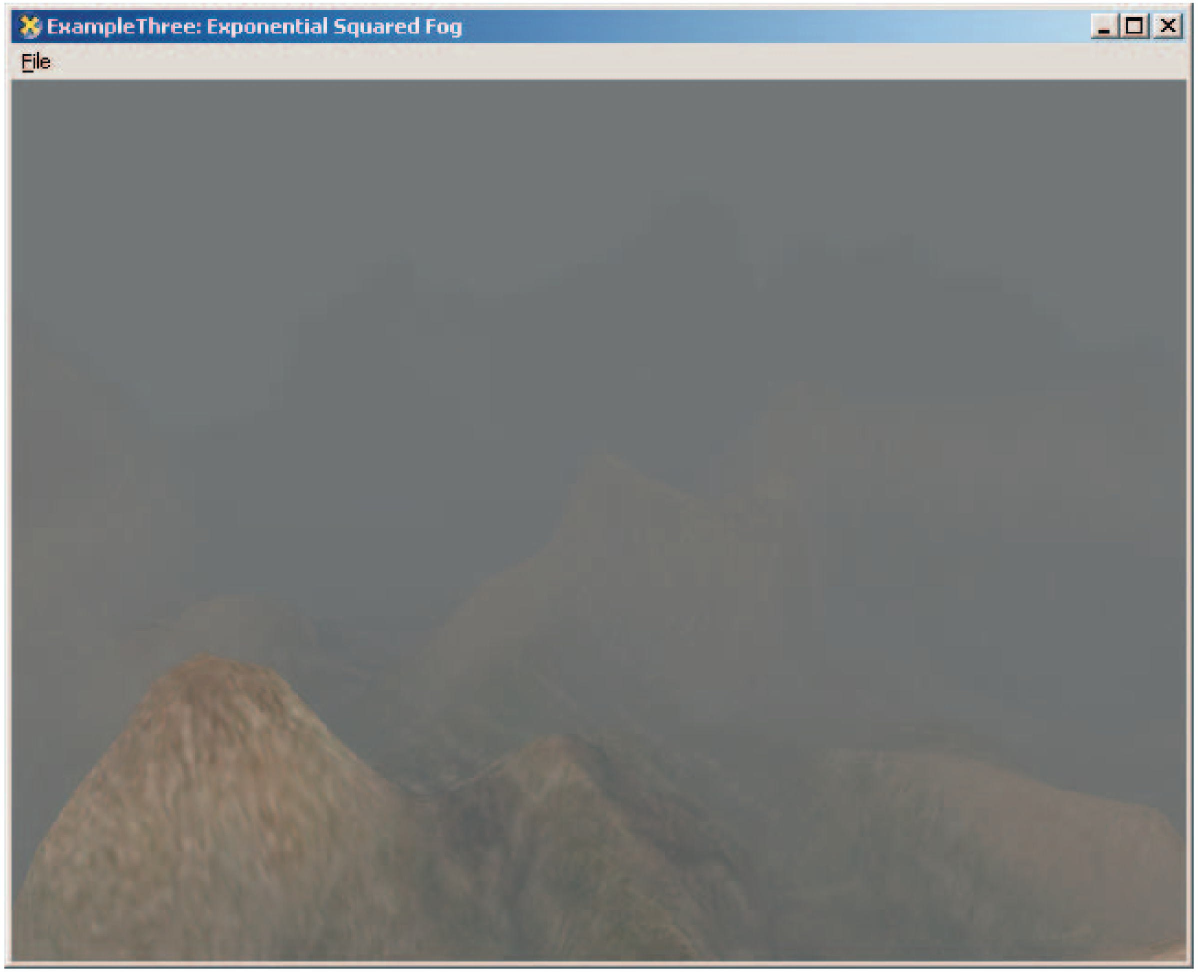


Figure 8: A complex shader-friendly model (See Color Plate 8.)

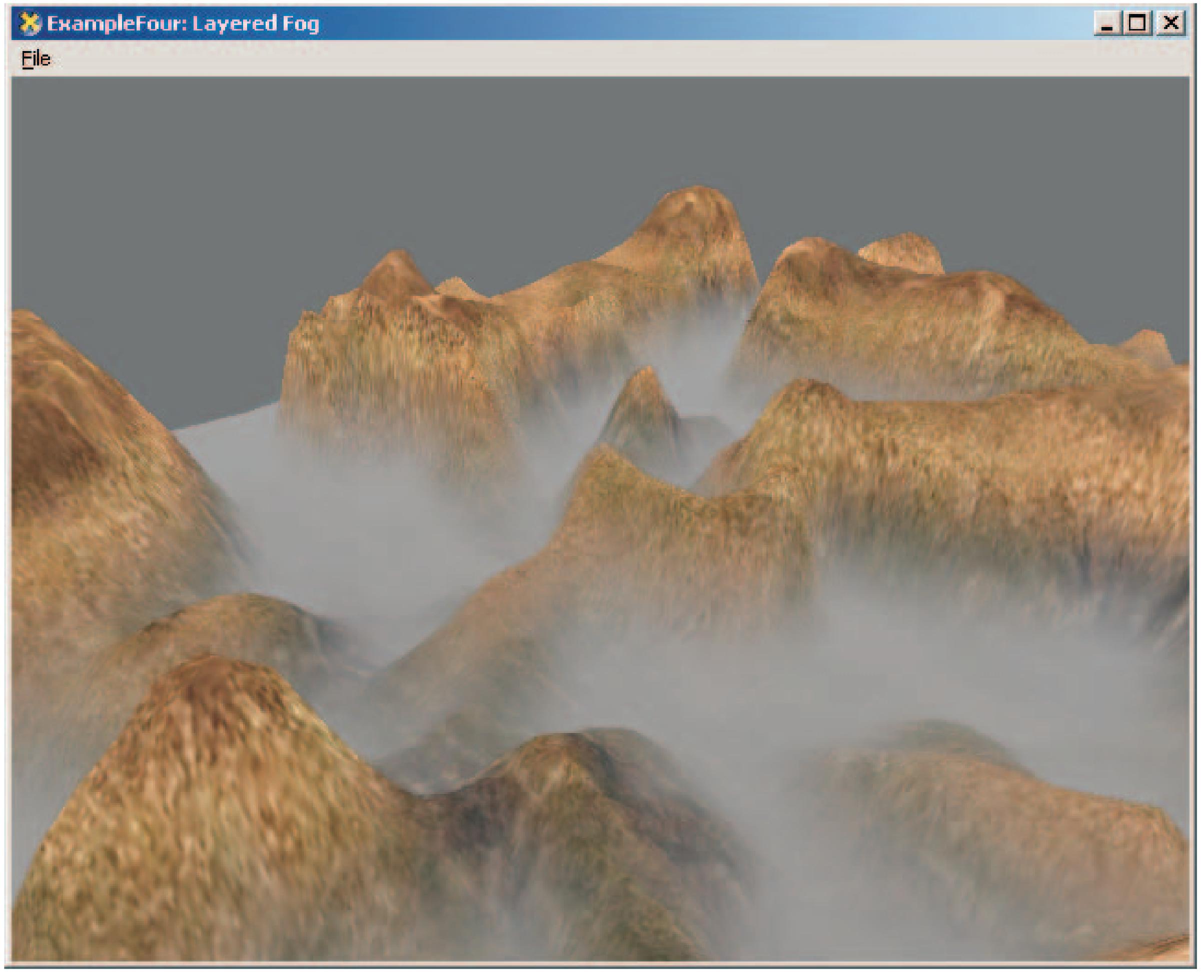

Color Plate 1. (Cook-Torrance lighting) Rendering with various refraction index values with pixel shader 1.4 (top row) and pixel shader 2.0 (bottom row). Roughness is constant at 0.15. The index of refraction is 0.15, 0.45, and 0.85 (left to right). Note the visibility of the face edges and error (crack) in the middle of the large highlights in the 1.4 version. (See page 147.)



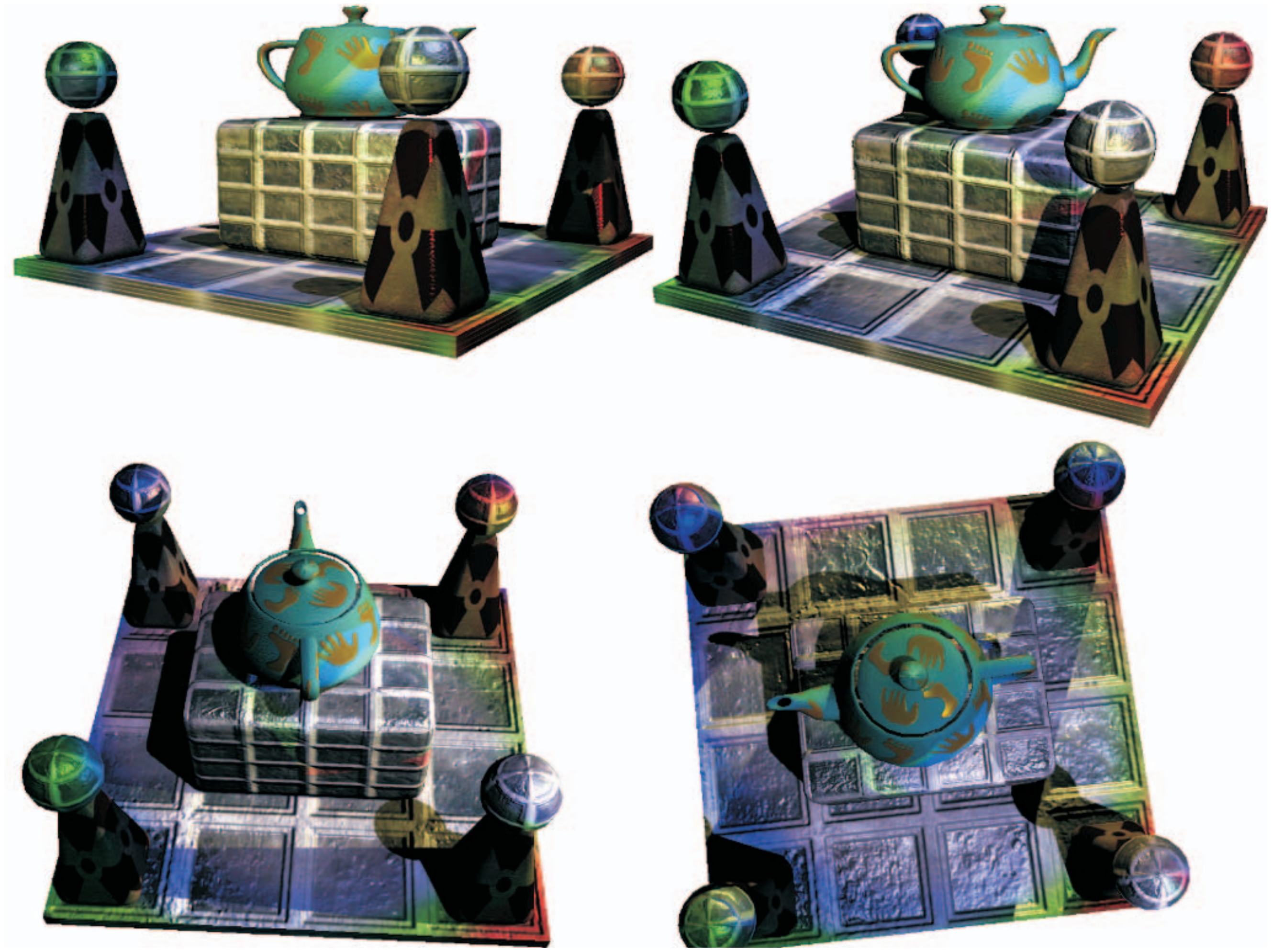
Color Plate 2. Linear fog (See page 154.)

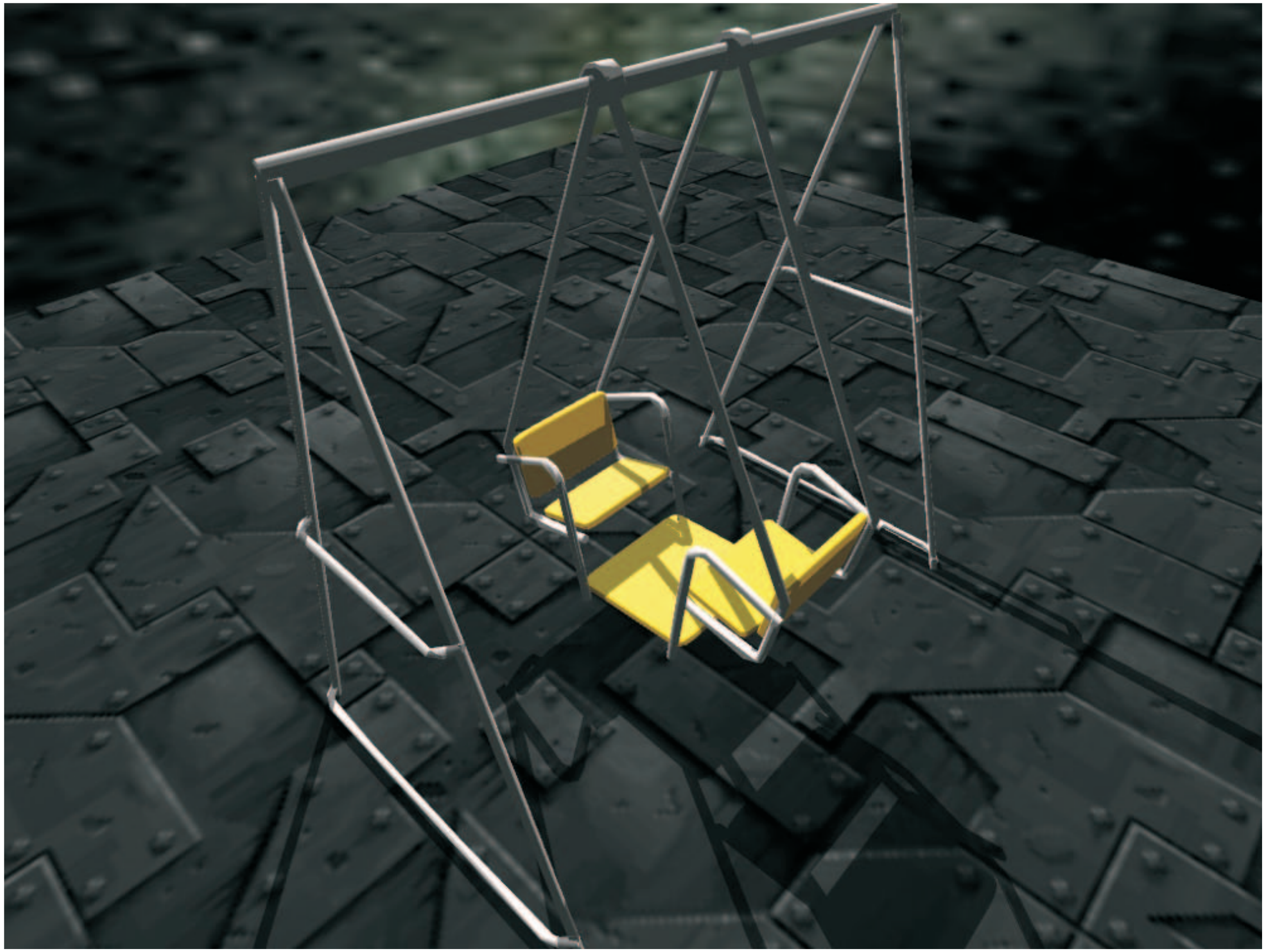


Color Plate 3. Exponential squared fog (See page 162.)

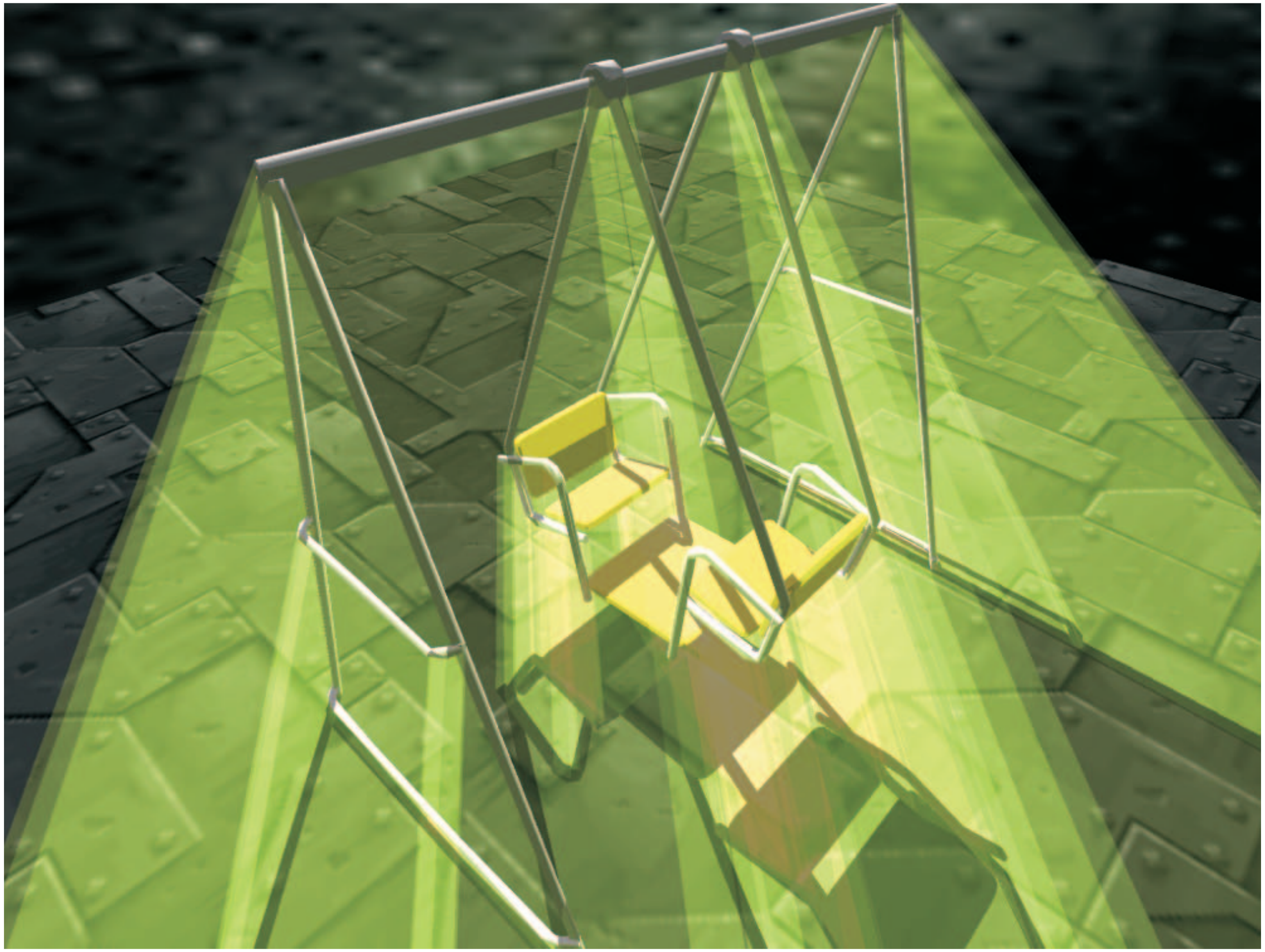


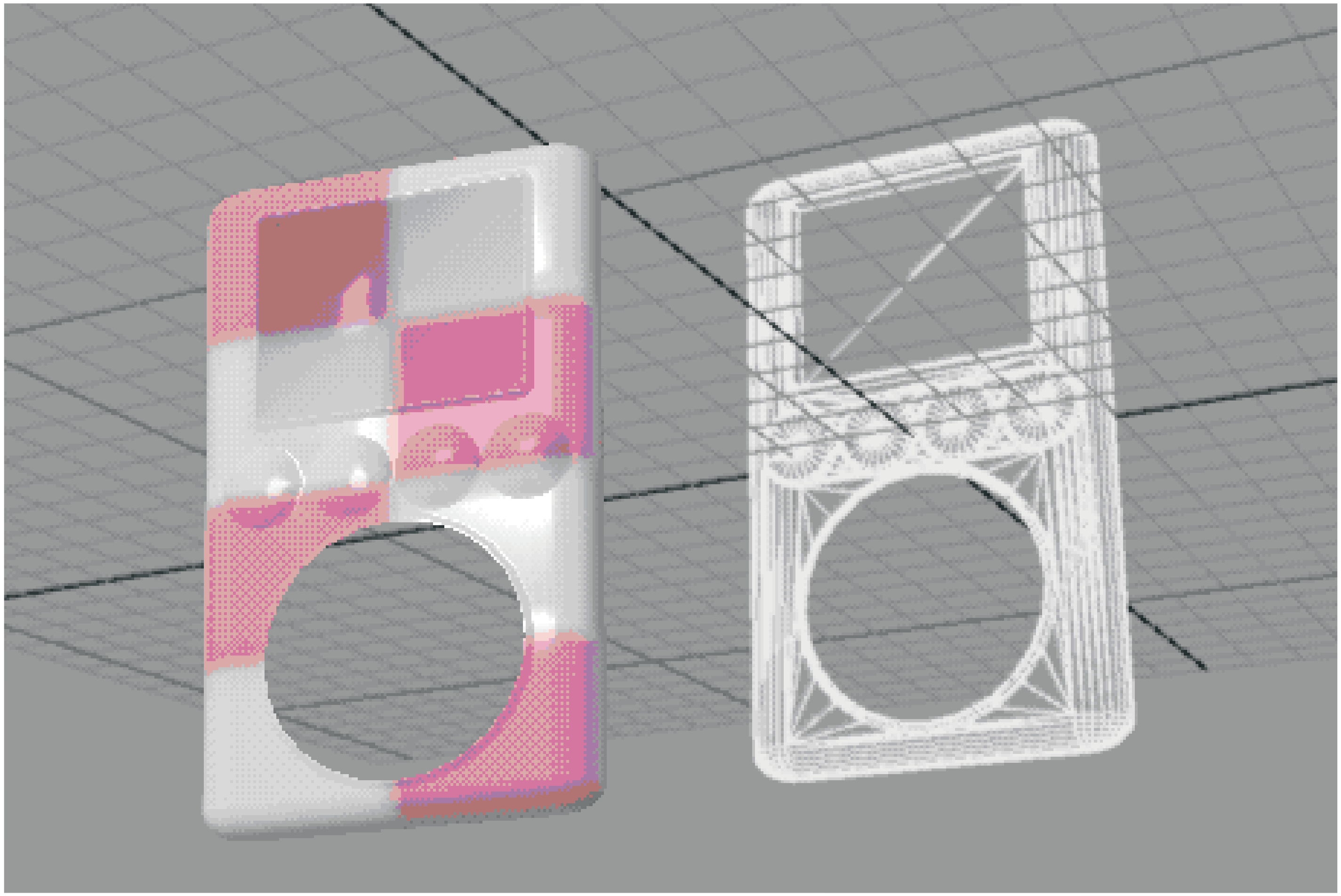
Color Plate 4. Layered fog (See page 166.)


Color Plate 5. Shadow map results (See page 194.)



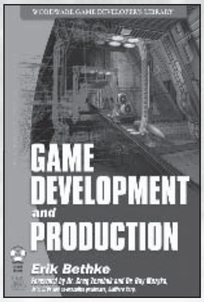
Color Plate 6. Shadowed scene (See page 276.)

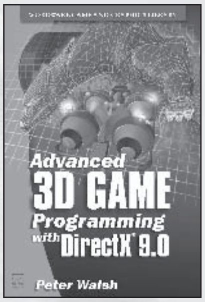

Color Plate 7. Shadowed scene with shadow volume exposed (See page 276.)

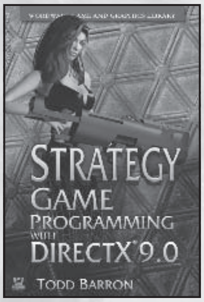


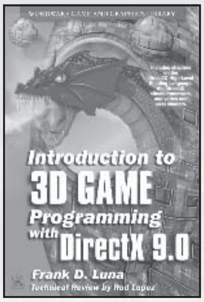
Color Plate 8. A complex shader-friendly model (See page 345.)

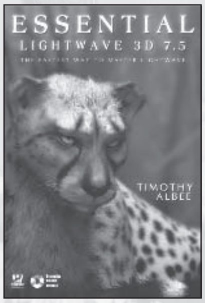
Index





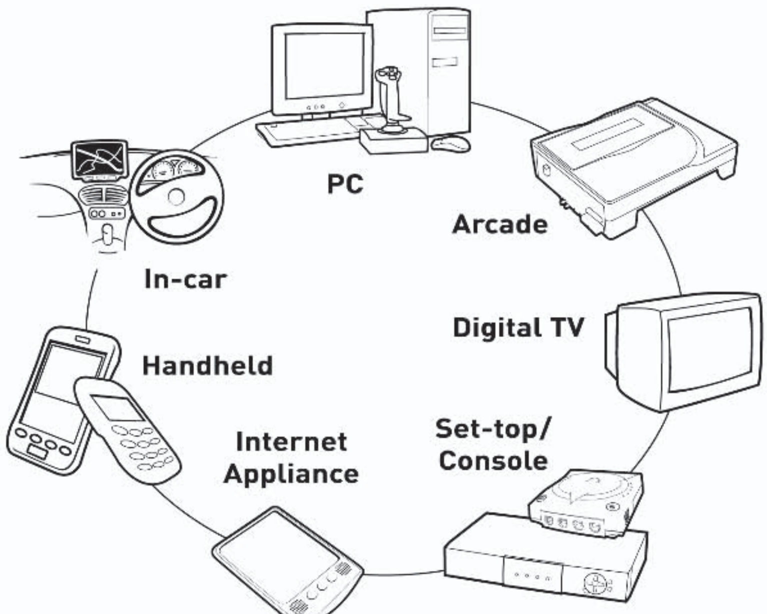






POWERVR

Visionary IP



www.powervr.com

www.pvrdev.com

Get **Deep** into 3D!

DEEP EXPLORATION.

Search, View, Translate, Animate, Render and Publish!

Deep Exploration™ provides easy navigation tools that let you search and view your 2D graphics and 3D models. It gives artists everywhere a production edge, including the ability to quickly translate 2D and 3D file formats with animation included.
Deep Exploration creates high quality fast renderings of 3D objects and scenes for use in many graphic applications. It is also a powerful web publication tool, ideal for creating interactive 3D content for web-based presentations.

DEEP PAINT 3D.

3D Painting and Texturing

Deep Paint 3D® provides artists with an intuitive, easy to use tool to paint and texture 3D models interactively in 3D! It uses textures or natural media paints which can be brushed directly or projected onto 3D models and scenes. This creative environment supports an integrated workflow with 3ds max®, Maya®, Softimage® and LightWave 3D®. Deep Paint 3D comes complete with a bi-directional interface to Photoshop® and special support for the Wacom® Intuos™ pressure sensitive tablet.

DEEP UV.

Ultimate UV Mapping

Deep UV™ is a superior set of tools for the creation and modification of UV mapping for polygonal and sub-division surface models within an interactive 2D and 3D UV mapping environment. Deep UV includes fast and easy to use tools such as soft selection, relax and 3D UV selection. Whether you're a beginner or a professional 3D artist, you'll have the most efficient UV mapping possible with both automatic and advanced features. Deep UV can be used standalone, or with Deep Paint 3D.

RIGHT HEMISPHERE.

www.righthemisphere.com

ONLY 100 WILL BE CHOSEN FOR THE QUEST *of a* LIFETIME

Twice a year the challenge is raised — and 100 of the best are admitted to the Hart eCenter at SMU Digital Games Guildhall. The Guildhall is designed to train talented students to become immediately productive digital games developers. The program is just like the industry — intense, results oriented, and only for the dedicated few.

The Guildhall grants membership to those who complete the 18-month certificate program, designed by industry professionals to train the next generation of game developers. High-profile industry leaders from top name development companies have designed the courses and take special interest in the Guildhall as teachers, mentors, and craft experts.

For more information and details on how to apply, contact David Najjab at **214-768-9903** or email **najjab@smu.edu**.

Check out the website at **guildhall.smu.edu**.

THE GUILDHALL

HART eCENTER SMU

About the CD

The companion CD contains examples and source code discussed in the articles. The files are organized into folders named for each article, although there may not be an example for every article. Each folder and/or subfolder includes a readme.txt document that explains the examples, contains instructions, and lists hardware requirements.

Simply place the CD in your CD drive and select the folder for the example you would like to see.

Warning: By opening the CD package, you accept the terms and conditions of the CD/Source Code Usage License Agreement on the following page.

Additionally, opening the CD package makes this book nonreturnable.