The *COMPLETE* Effect And HLSL Guide

The topic of ***The Complete Effect and HLSL Guide*** is shader development and management, and therefore it is written for any developer who has an interest in being efficient at using and integrating shaders within their applications. This book is written to serve as both a teaching and reference manual, making it a must-have to everybody from hobbyist programmers to professional developers.

The approach taken throughout ***The Complete Effect and HLSL Guide*** makes it the perfect book for anyone who wants to take advantage of the power of the DirectX effect framework and the HLSL shading language. The followig topics are covered:

- Introduction to both the HLSL shading language and effect files including their detailed syntax and use.
- Complete reference, along with performance considerations to every single HLSL and assembly shader instructions (including the 3.0 shader model).
- Introdution the DirectX Effect Framework and complete overview to its API.
- Optimization tips and tricks allowing you to make the best use of the rendering hardware and achieve the best looking results.
- Coverage of all the main components of the effect Fframework in addition to putting the pieces of the puzzle together allowing you to develop a complete shader management framework.

Paradoxal Press

9981 Avondale Rd. NE
Redmond, WA 98052
www.ParadoxalPress.com

Category
Game Programming / Game Development

Level
Intermediate to Expert

$34.99 USA
$49.99 CANADA

ISBN-13 978-0-9766132-1-3
ISBN-10 0-9766132-1-2

53499>

9 780976 613213

The *COMPLETE* Effect And HLSL Guide

Paradoxal Press

The *COMPLETE* Effect And HLSL Guide

Sebastien St-Laurent

Paradoxal Press

The *COMPLETE* Effect and HLSL Guide

Sebastien St-Laurent

PUBLISHED BY
Paradoxal Press
9981 Avondale Rd. NE
Redmond, WA 98052
Unites States of America
http://www.ParadoxalPress.com
info@ParadoxalPress.com

ISBN-10: 0-9766132-1-2
ISBN-13: 978-0-9766132-1-3

Library of Congress Control Number: 2005902999

Printed in the United States of America

President, Paradoxal Press:	Sebastien St-Laurent
Copy Editor:	Dawn Snyder
Technical Reviewer:	Wolfgang Engel
Interior Layout:	Sebastien St-Laurent
Cover Designer:	Sebastien St-Laurent
Indexer:	Dawn Snyder
Proofreader	
Fiction Acquisition:	Nicole St-Laurent

To my wife, Nicole, for all her love and support while writing this book.

To my friends, co-workers and family for their understanding and support.

To my mother, Lina, another victim of cancer. I will miss dearly!

Acknowledgments

First and foremost, I want to thank my wife Nicole for all of her support throughout this project. Writing a book can be a major undertaking, self-publishing it is of another order of magnitude. Without her help and love, I would never have completed this one or might have lost my sanity doing so. I love you!

Wolfgang Engel and Steve Lacey also deserve special mention for their efforts as technical editors for this book. Their help proved invaluable in making sure I was in line and ensuring this book was the best possible book it could be. I also want to send my thanks to the kind people at NVIDIA, ATI Technologies and Microsoft for their technical information, which helped greatly with this production.

About the Author

Sebastien St-Laurent has been programming games professionally for several years, working on titles for the Xbox, PlayStation 2, GameCube, and PC. He started in the video game industry while studying computer engineering at Sherbrooke University in Sherbrooke, Quebec. By interning in a small company called Future Endeavors during his college years, he got into the industry and stood out in the line of graphics engineering.

After graduating from college, he moved to California to work full-time with Z-Axis as lead Xbox engineer, where he worked on several titles including the Dave Mirra Freestyle BMX series. Currently, he is a graphics engineer at the Microsoft Corporation, where he is working within the Microsoft Game Studios.

Having already published a book "*Shaders for Game Programmers and Artists*"; he is now a renowned authority in computer graphics and video game development.

Introduction

After completing my first book, *Shaders for Game Programmers and Artists*, in the spring of 2004 I immediately started thinking about what my next book would be about. Although I did take the summer off to relax and send quality time with my wife, I still had several ideas going through my head. Being a graphics engineer, it is obvious that topics such as shaders are at the core of my every day work but wanted to write something a little different from my previous book while remaining in the scope of things I know best.

After some serious thinking and drawing from my experience as a software developer, I started looking at some of the issues that developers are faced when writing 3D graphic applications today. When developing games on the Xbox platform, shader development was straightforward since the platform was fixed and everybody was running the same hardware. However, I came to new realizations when I start working on a larger scale PC game. When developing a game for a platform such as the PC, you want to take advantage of the newest rendering technologies while ensuring support on a wide range of hardware configurations. It makes sense, not everybody can just go out and spend 500$ every six month for the latest and greatest video card.

This wide range of possible configuration itself leads to the multiplication of shaders. It does make sense when you think about it, sure you want to render nice reflective water on the latest hardware, but you still need to be able to function properly on less advanced video card. This multiplication of shaders leads to nightmares for the developers, having to juggle a multitude of shaders. In addition, how can you determine which shader to use based on the user's hardware configuration. To make matters worse, the latest shader technologies are complex and don't lend themselves well to being written by hand in assembly instructions.

With all these difficulties in mind, Microsoft has introduced the High-Level Shader Language (HLSL) and the effect framework. Both aimed at facilitating the task of shader development and management in addition to creating a standardized art pipeline making it easier for artists to model and take advantage of shaders at every step of the artistic process.

However, at this point in time, it seems like many developers are not aware of the power of this technology and this is why I've set out to write this book. My goal is to not only write a book teaching the basics of writing shaders in HLSL and using the effect framework but also offer a complete reference manual covering this small component of DirectX from A to Z.

Who Should Read This Book

The topic of *The Complete Effect and HLSL Guide* is shader development and management, and therefore it is written for any developers who have some interest in being efficient at using shaders within their applications and focus their time and efforts on more important tasks. Because this book is written as both a teaching and reference manual, it is bound to be of interest to everybody from hobbyist programmers to professional developers.

Finally, with the approach taken throughout this book, *The Complete Effect and HLSL Guide* can also be a valuable asset in the classroom where real-time graphics have taken an even more important place in the computer science curriculum.

What We Will and Won't Cover

The topic of *The Complete Effect and HLSL Guide* is shader development and management, and it is all I will focus on. Since the intent of this book is being both a teaching guide as well as a reference manual, I will be thorough into covering most topics of the HLSL language and the effect framework but will not focus on specific shaders and rendering techniques. The following list summarizes some of the topics covered throughout this book:

- Introduction to both the HLSL shading language as well as the effect framework which are both part of the latest DirectX SDK.
- In depth coverage of the HLSL language syntax, grammar and its use.
- Coverage of all the main components of the effect framework in addition to putting all the pieces of the puzzle together allowing you to develop a shader management framework.

Support

A website is maintained at **http://www.ParadoxalPress.com** that will be used to provide support for this book. This site will be updated regularly to post corrections and updated information as needed. Be sure to check it regularly.

And if you have any questions of feedback or questions in regards to this book, feel free to contact me, Sebastien St-Laurent at **sebastien.st.laurent@gmail.com**.

Table Of Contents

Chapter 3
Functions,Nothing but Functions

Chapter 4
Shader Examples

Chapter 5
The Effect File Format

Chapter 6
Semantics and Annotations

Chapter 7
Shader Optimizations and Shortcuts

Chapter 8
Effect Framework Overview

Chapter 9
The Effect

Chapter 10
Sharing Parameters

Chapter 11
Effect State Manager

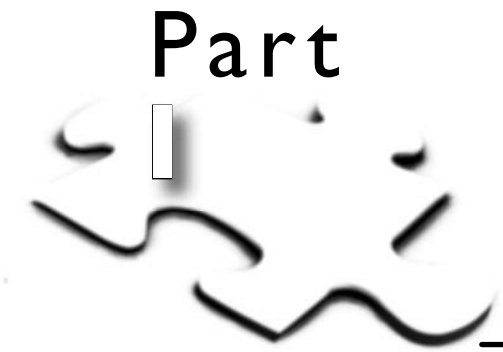
Chapter 12
Effect Compiler and Include Manager

Appendix A
Shader Assembly Instruction Reference

Appendix B
HLSL Intrinsic Function Reference

Appendix C
Standard Semantics and Annotations

Appendix D
Effect File and HLSL Grammar



Part I

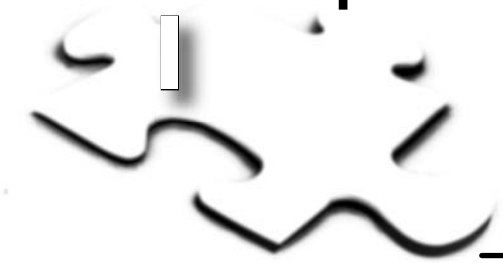
The HLSL Shading Language

Welcome to *The Complete Effect and HLSL Guide*. The title of the book is very much self-explanatory; together we will explore the use of the DirectX effect framework and the HLSL (High-Level Shading Language) shading language throughout the chapters of this book. You will learn how to use and be efficient with the HLSL shader language. In addition, I will teach you how the effect framework, which comes as a part of the DirectX SDK, can be used to facilitate the integration of shaders within your applications. Whether you are developing 3D applications or video games, the increase in shader complexity and the need for backward compatibility make the effect framework a prime choice for the integration and management of shaders.

The goal behind this book is dual. First, it serves as a learning book, teaching you the insights of HLSL and the effect framework. More importantly, however, this book serves as a reference manual, containing all the necessary information not included in the DirectX documentation. So, let's not waste time and get to it!

The first part of the book will focus on the HLSL shader language, teaching you its syntax and use. The second part of this book will focus on the effect framework, explaining how it can simplify the management of shaders for your application. Let's take an in-depth look at the HLSL shading language.

Chapter 1



Shaders and the HLSL Language

Shader technology has made huge leaps over the last few years. In this chapter, I will go over some of the basic syntax and use of the HLSL shading language. To make the understanding of this technology easier, I have broken up the description of the HLSL syntax over several chapters, each focusing on specific components of the shading language. This chapter will go over the general syntax and overview a few of the fundamentals. The following chapters will go into more details of specific components.

Before I dive into the syntax of HLSL, I will discuss some of the prerequisites for this book and go over a little bit of the history of the HLSL shader language and the effect framework.

Prerequisites

Although this book intends to teach you about HLSL and the effect framework, it has no intention of teaching you the basics of DirectX and Direct3D. Because of this, one of the important prerequisites for this book is that you have a basic understanding of the DirectX API and a basic background in 3D rendering and technologies. All of this is easy to learn even if you are new to 3D graphics!

Beyond the intellectual requirements, here is a basic list of software and hardware needs to use the contents of this book:

- DirectX 9.0 Summer 2004 Update SDK (included on the CD).
- Windows 2000 (with service pack 2) or Windows XP (Home or Professional) operating system.
- Pentium 3 class or better processor.
- At least 256 MB of RAM.
- A high-end 3D graphics card. Although any 3D capable video card would do, I recommend a 2.0 or 3.0 shader model compatible card if you want to try out all the aspects of shader programming.
- And of course, the latest drivers for your video card.

With those prerequisites in mind, you will be able to start exploring and developing the shaders and taking advantage of the effect framework. Now you know what you need to take advantage of this book. Let's go over a little bit of history about shaders and the effect framework.

A Little Bit of History...

Computer graphics and its associated hardware have made significant technological leaps since the introduction of the first consumer level 3D hardware accelerated graphics card, the 3Dfx, in 1995. This card had limited rendering capabilities but finally allowed developers break new grounds and move away from software only solutions. This finally made real-time 3D graphics and games a true reality.

Since then, the following generations of hardware improved significantly on their performance and features. They were still bound, however, by a limited fixed-pipeline architecture, which restricted developers to a constrained set of states that were combined to produce the final output.

The limited functionality of the fixed-pipeline architecture restricted developers in what they could create. This generally resulted in synthetic-looking graphics. At the other end of the spectrum, high-end software-rendering architectures used for movie CG had something that allowed them to go much farther. RenderMan is a shading language developed by Pixar Animation Studios. The purpose was to allow artists and graphic programmers to fully control the rendering result by using a simple, yet powerful programming language. RenderMan allowed creating high quality, photorealistic and non-photorealistic graphics used in many of today's movies, including Toy Story and A Bug's Life.

With the evolution of processor chip making and the increased processing power it brought along came the natural extension of the RenderMan idea to the consumer level graphics hardware. With the release of DirectX 8.0 came the introduction of vertex and pixel shader version 1.0 and 1.1. Although the standard came with limited flexibility and omitted some features such as flow control, itwas the first step in giving developers and artists the flexibility needed to produce the stunning and realistic graphics they had always dreamed. Consumer video cards could finally produce graphics that could compete with the renderings produced by Hollywood's movie studios.

During the following few years, graphics hardware and 3D APIs made giant leaps forward in functionality and performance, even shattering Moore's law with respect to technological advancement rate. With the introduction of the DirectX 9.0 SDK and the latest generations of graphics hardware such as the GeForce FX series from NVIDIA and Radeon 9800 series from ATI Technologies, came Vertex and Pixel shader version 2.0 and 2.x. The shader version 3.x was soon to follow.

Note

The term Moore's Law came from an observation made in 1965 by Gordon Moore, co-founder of Intel, which the number of transistors per square inch had doubled every year since the introduction of the integrated circuit. He also predicted that this trend would continue for at least a few decades, which turned out true so far. Also, since the transistor density relates with the performance of integrated circuits, Moore's Law is often cited as a prediction of future hardware performance increases.

This new shader model brings flexibility never before available to real-time graphic application developers. At the same time, however, most shaders were developed in a low-level language similar to assembly language on a computer. This also meant that as a shader developer, you had to manage registers, variable allocations and optimization in the same way as developers had to do it when programming assembly language on computers. The complexity of the 2.0 and 3.0 shader models added to the headache of developers since different video cards can have different amounts of registers and variables, and might even perform differently with similar instructions.

In an effort to simplify the task and give more optimization freedom to the hardware developers, the Microsoft Corporation introduced the High-Level Shading Language (HLSL) to DirectX version 9.0. This language, similar to a high-level language such as C or C++, allows developers to focus on the task the shader wishes to perform rather than logistics such as determining which registers to use and which combination of instructions is best for a specific card or shader version.

Before I go into what the HLSL language can do for you and how you can use it, let's go over some of the functionality that is given by the different shader versions. Let's face it; before you can write shader, you need to know what the hardware is capable of in the first place. Having a high-level language doesn't make the restrictions of a specific piece of hardware go away but rather hides it from you.

Vertex and Pixel Shader Pipelines and Capabilities

Vertex and pixel shader model 2.0 bring many new significant improvements to the language since the first introduction of version 1.0 and 1.1 with DirectX 8.0. Because of the recent release of DirectX 9.0 and the release of vertex and pixel shader 2.0 compliant video cards, this book will focus mostly on using shaders based on this technology.

Note

Although that at the time of this writing, 3.0 shader model cards are only starting to surface, their use is still limited. Although I will explain some of the specifics of 3.0 hardware, I have decided to stick with 2.0 shaders and below for most examples within this book.

Since I assume you already have a basic knowledge of 3D and basic shaders, let's start by going over the significant changes introduced by the second generation of shader languages over their legacy counterparts.

Vertex Shader 2.0 and 2.x includes the following improvements over their 1.x counterpart:

- Support for integer and boolean data types and proper setup instructions.
- Increased number of temporary and constant registers.
- Maximum instruction count allowable for a program has increased. Giving developers more flexibility (the minimum required by the standard has gone from 128 to 256 but each hardware implementation can support more).
- Many new macro instructions allowing complex operations such as sine/cosine, absolute and power.
- Support for flow control instruction such as loops and conditionals.

The following list outlines Pixel Shader 2.0 and 2.x improvements from the 1.x model:

- Support for extended 32-bit precision floating-point calculations.
- Support for arbitrary swizzling and masking of register components.
- Increase in the number of available constant and temporary registers.
- Significant increase of the minimum instruction card allowed by the standard from 8 to 64 arithmetic and 32 texture instructions. Pixel shader 2.x allows even more instructions by default and allows the hardware to go beyond the standard's minimum requirements.
- Support for integer and boolean constants, loop counters and predicate registers.
- Support for dynamic flow control including looping and branching.
- Gradient instructions allowing a shader to discover the derivate of any input register.

With this rich set of improvements, developers are now free to set their creativity loose and create stunning effects. At this point, it is probably good to do an overview of their architecture to give you a better understanding of how the information flows throughout the graphics hardware.

When rendering 3D graphics, geometric information is passed to the graphics hardware through the use of a rendering API such as Direct3D. Once this information is received by the hardware, it invokes the vertex shader for every vertex in your mesh. Figure 1.1 includes the functional diagram for a vertex shader 2.0 implementation as dictated by the specifications.

As you can see from Figure 1.1, vertices come in from a stream that is supplied by the developer through the 3D rendering API. The stream contains all the information needed to properly render the geometry such as positions, colors and texture coordinates. As the information comes in, it is put into the proper input registers, `v0` to `v15`, for use by the vertex shader program. The vertex shader program then has access to many other registers to complete its task. Constant registers are read-only registers which must be set ahead of time and are used to carry static information to the shader. Under the vertex shader 2.0 standard, constant registers are vectors and can be floating-point numbers, integer values or boolean values. Take note that registers within the vertex shader, are all stored as a 4 component vector where you can process each component in parallel or individually by using swizzling and masking.

On the right side of Figure 1.1, are the *Temporary Registers*, which are used to store intermediate results generated by the vertex shader. Obviously, because of their temporary nature, you can both write and read from those registers. Take note of the registers named `a0` and `aL`, which are counter registers for indexed addressing and for keeping track of loops. Also keep in mind that because HLSL is a high-level shading language, you will not need to take care of register allocation. It will happen transparently as the shader is compiled to its final form.

With access to the *Input Registers*, *Temporary Registers* and *Constant Registers*, the vertex shader program is now free to process the incoming vertices and manipulate them in whichever way the developer sees fit. Once the processing is complete, it must pass the results to the final *Output Registers*. The most important one is `oPos`, which needs to contain the final screen space projected position for the vertex. The other registers carry information such as colors and the final texture coordinates.

Once the vertex shader has done its job, the information is then passed along to the rasterizer. This part of the hardware takes care of deciding the screen pixel coverage of each polygon. It also takes care of other rendering tasks such vertex information interpolation and occlusion which helps reduce the overall work needed by the hardware. Once the rasterizer has determined the pixel coverage, the pixel shader is invoked for each screen pixel drawn. Figure 1.2 includes the functional diagram for the pixel shader architecture.

As you can see from the diagram in Figure 1.2, the hardware sends the pixels it calculates through the input *Color* and *Texture Registers*. Those values are based on the perspective interpolation of the values defined through the vertex shader. Registers `v0` and `v1` are meant to be the interpolated diffuse and specular lighting components. The registers `t0` to `tN` carry interpolated texture lookup coordinates. Finally, the registers `s0` to `sN` point to the textures which the pixel shader will sample

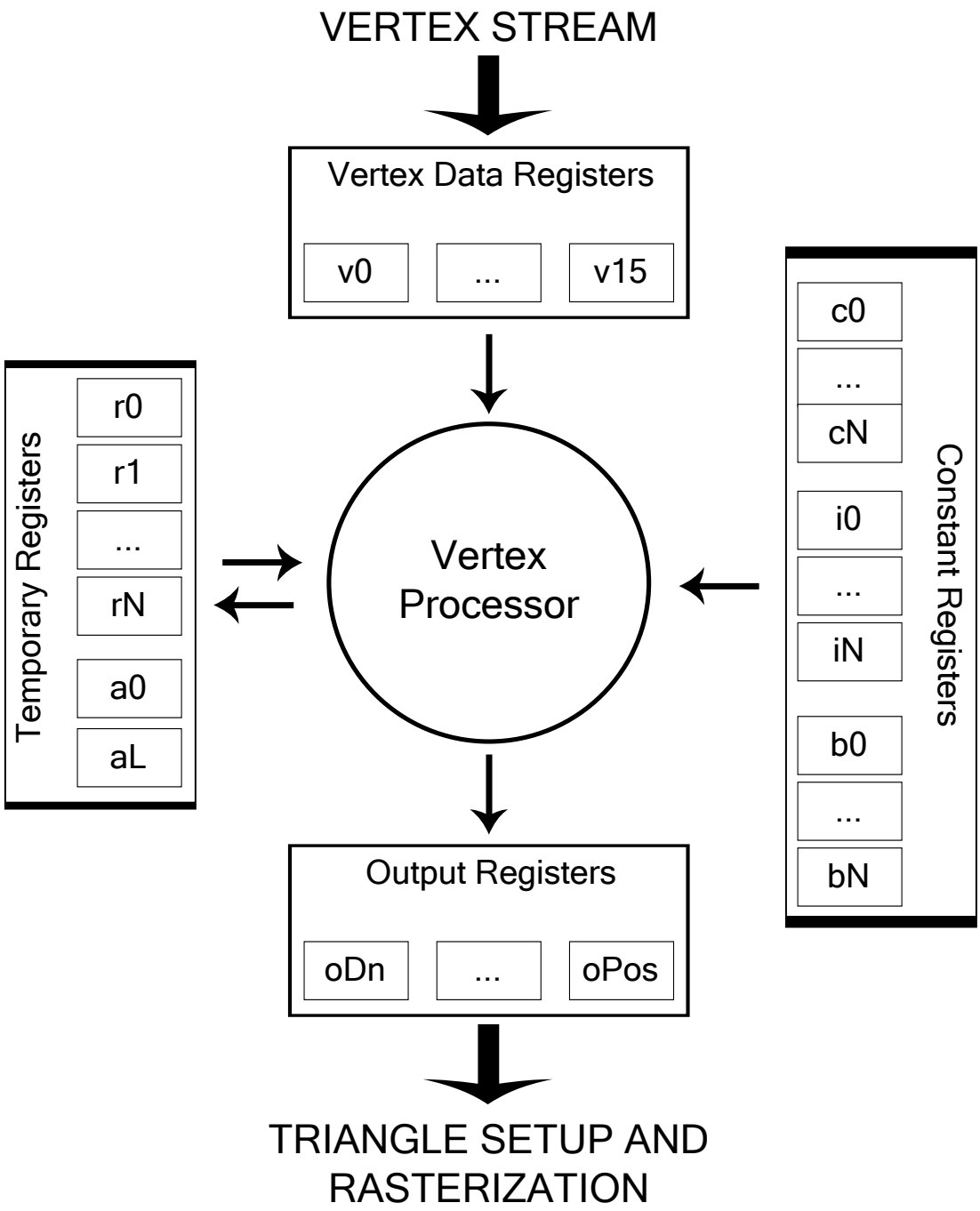


Figure 1.1: Functional diagram for the vertex shader hardware architecture.

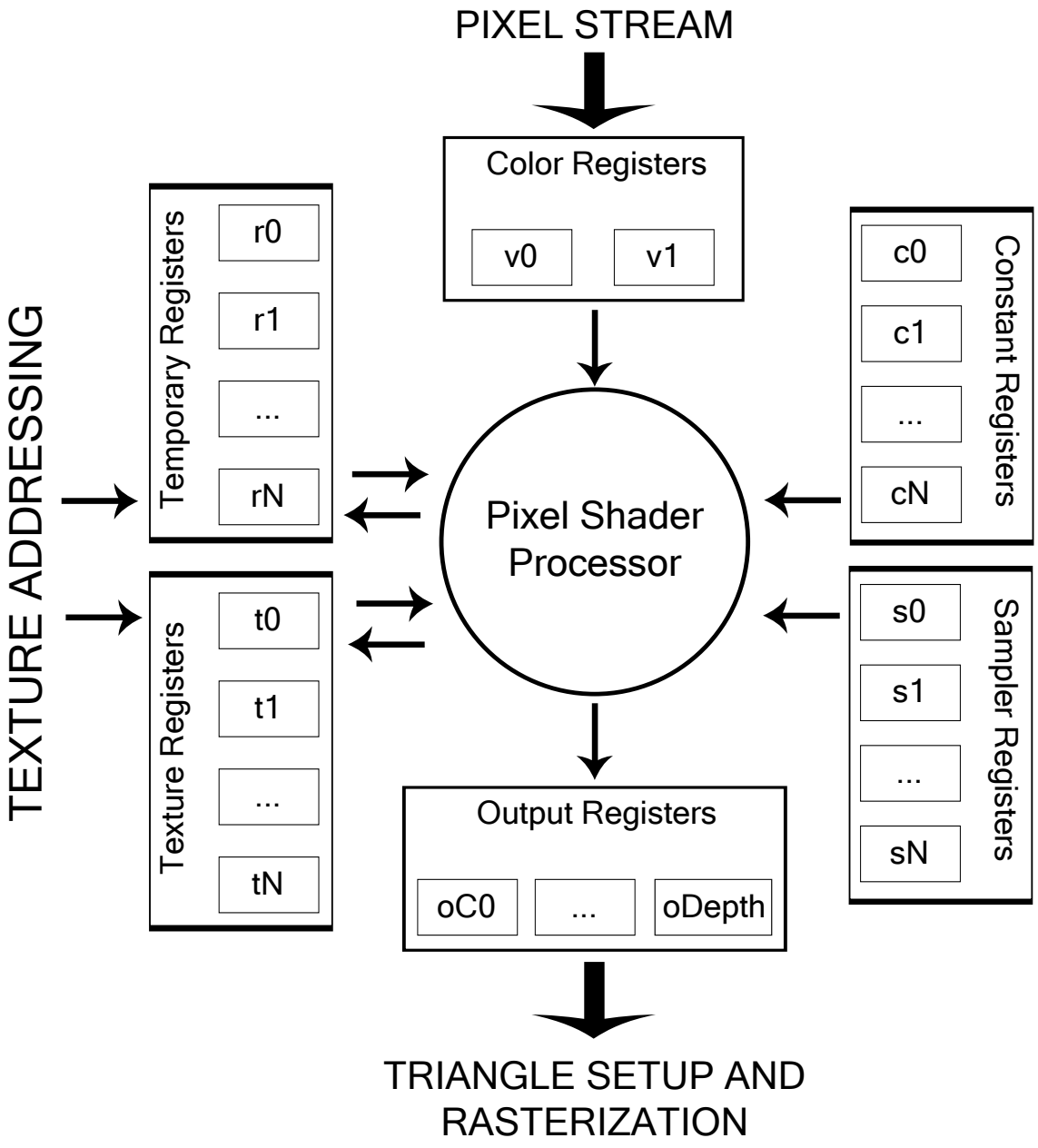


Figure 1.2: Functional diagram for the pixel shader hardware architecture.

during the processing of the pixel. Although those registers have clear semantics, they can be used to carry any information from the vertex shader onto the pixel shader.

The *Constant Registers*, `c0` to `cN`, can only be read and are setup ahead of time by the developer with values useful to the shader. Finally, the *Temporary Registers*, `r0` to `rN`, are read/write and keep track of intermediate results during the processing of the pixel. When using HLSL, all register usage and allocation is done automatically and is transparent to the user.

Most registers in pixel shaders, with exception to some addressing registers and loop counters, are vectors comprised of four floating-point values. The advantage of the vector architecture is that it allows processing of multiple values at the same time. By default, all floating-point values are processed as 32-bit precision floating-point numbers. The pixel shader specification allows processing of 16-bit precision values, which can be enabled by special instructions. On certain hardware implementations, the use of 16-bit floating-point arithmetic can be significantly faster.

Once the pixel has gone through the pixel shader, the output information is then used to blend the final result onto your frame buffer which is then presented on your computer screen.

Keep in mind that this is a simplified survey of the rendering architecture and much more happens behind the scenes. This architecture may also vary slightly from one hardware implementation to another; however, the standard does guarantee that for two different implementations with the same capabilities, the final output must be the same.

The High-Level Shading Language

Because of the increased complexity of the 2.0 and 3.0 shader models, it became increasingly tedious for developers to be efficient at developing shaders. The shader instruction set becoming more closely related to instruction sets found on general processors, it made sense to consider developing a high-level language to simplify and abstract the development of shader. This is where the High-Level Shading Language (HLSL) comes in to play. This language was developed by the Microsoft Corporation as a way to abstract shaders and let developers focus on the more important task of shader design rather than the fine grain details of register allocation and instruction optimization.

As with any high-level language, HLSL is defined by a set of syntax rules, reserved words and operands. In this section, I will go over its general syntax briefly. Do not worry, as I will go into the specific details of some of the components and give specific examples in later chapters.

Reserved Words

As with any programming language, it is comprised of identifiers and keywords. HLSL is no exception to this rule and this also implies that there is a set of keywords, defined for the purpose of the language with special meaning and thus cannot be used by your code. The first important

category language elements are the keywords. They are defined as reserved identifiers used to represent constant values to simply control the behavior of your code. Table 1-1 below summarizes all the HLSL keywords along with a brief description of its meaning.

Table 1-1 HLSL Keywords

Keyword	Definition
asm	This keyword is case insensitive and is used to allow the manual insertion of shader assembly instruction inside a shader. The syntax for this instruction is as follow: `asm{ /* assembly instructions */ }`
asm_fragment	Reserved for future use.
bool	This keyword is used to identify data of boolean type. See Chapter 2 for more information on data types.
column_major	This keyword is used in conjunction with matrices to indicate that the data is presented in a column major form.
compile	This keyword is used when declaring vertex or pixel fragments (or shaders), indicating with which profile to compile the fragment.. This keyword is to be used when assigning a shader directly to the vertex or pixel shader effect state.
compile_frgment	This keyword is used when declaring vertex or pixel fragments (or shaders), indicating with which profile to compile the fragment.
const	This keyword is used to define a constant variable.
discard	This keyword is used within a fragment shader to cancel rendering of the current pixel.
decl	This keyword is case insensitive and is used in addition to the `asm` keyword to define constant register values for assembly defined shaders.
do	This keyword is in combination with the `while` keyword to define conditional do-while loops.
double	This keyword is to identify data of the double precision floating-point type. See Chapter 2 for more information on data types.
else	This keyword is used in combination with the `if` keyword to define conditional if-else statements.
extern	This keyword is used within variable declarations to indicate that this variable can be accessed from outside the effect. See Chapter 2 for more information on data types.
false	This keyword is the false constant value for the boolean data type. See Chapter 2 for more information on data types.

Table 1-1 HLSL Keywords

Keyword	Definition
float	This keyword is to identify data of the single precision floating-point type. See Chapter 2 for more information on data types.
for	This keyword is used to define loop statements.
half	This keyword is to identify data of the half precision floating-point type. See Chapter 2 for more information on data types.
if	This keyword is used to define if-else conditional statements.
in	This keyword is used to specify that a function parameter is for input only. See Chapter 3 for more information on functions.
inline	This keyword is used to hint to the compiler that a function is to be inlined. See Chapter 3 for more information on functions.
inout	This keyword is used to specify that a function parameter is for both input and output. See Chapter 3 for more information on functions.
int	This keyword is used to define data of the integer type. See Chapter 2 for more information on data types.
matrix	This keyword is used to identify data of the matrix type. See Chapter 2 for more information on data types.
out	This keyword is used to specify that a function parameter is for output only. See Chapter 3 for more information on functions.
pass	This keyword is case insensitive and is used to define multiple passes within multipass effect. Note that this keyword relates more closely to effect files and will be discussed in Part II of this book.
pixelfragment	This keyword is used to define a pixel fragment (or shader).
return	This keyword is used to return values from a function. See Chapter 3 for more information on functions.
register	This keyword is used to pre-reserve registers for use as constants.
row_major	This keyword is used in conjunction with matrices to indicate that the data is presented in a row major form.
sampler	This keyword is to represent the sampler data type. Samplers are used to represent a combination of texture and texturing attributes (such as filtering). See Chapter 2 for more information on data types.
sampler1D	This keyword is similar to the sampler keyword but is used to represent a sampler to a 1D texture. See Chapter 2 for more information on data types.

Table 1-1 HLSL Keywords

Keyword	Definition
sampler2D	This keyword is similar to the sampler keyword but is used to represent a sampler to a 2D texture. See Chapter 2 for more information on data types.
sampler3D	This keyword is similar to the sampler keyword but is used to represent a sampler to a 3D texture. See Chapter 2 for more information on data types.
samplerCUBE	This keyword is similar to the sampler keyword but is used to represent a sampler to a cubemap texture. See Chapter 2 for more information on data types.
sampler_state	This keyword is used to define a sampler by defining a block of information representing the state of a sampler.
shared	This keyword is used to indicate that a global variable can be shared across multiple effects. See Chapter 2 for more information on data types.
stateblock	This keyword is used to declare variables of the type `stateblock`, used to contain a set of effect states. Note that this keyword is generally used for effect files. See Part II of this book for more information on effect files
stateblock_state	This keyword is used to define a state block containing a set of effect states.
static	This keyword is used to define static variables. See Chapter 2 for more information on data types and variables.
string	This keyword is used to define data of string type. See Chapter 2 for more information on data types.
struct	This keyword is used to define structures. See Chapter 2 for more information on data types.
technique	This keyword is case insensitive and is used to define different techniques to accomplish an effect. Note that this keyword relates more closely to effect files and will be discussed in Part II of this book.
texture	This keyword is to represent the texture data type. See Chapter 2 for more information on data types.
texture1D	This keyword is similar to the `texture` keyword but is used to represent a 1D texture. See Chapter 2 for more information on data types.
texture2D	This keyword is similar to the `texture` keyword but is used to represent a 2D texture. See Chapter 2 for more information on data types.
texture3D	This keyword is similar to the `texture` keyword but is used to represent a 3D texture. See Chapter 2 for more information on data types.

Table 1-1 HLSL Keywords

Keyword	Definition
textureCUBE	This keyword is similar to the `texture` keyword but is used to represent a cubemap texture. See Chapter 2 for more information on data types.
true	This keyword is the true constant value for the boolean data type. See Chapter 2 for more information on data types.
typedef	This keyword is used to declare a new data type. See Chapter 2 for more information on data types.
uniform	This keyword is used to declare a variable as uniform, meaning all shader runs will see the same initial value to the variable. See Chapter 2 for more information on data types.
vector	This keyword is to represent the vector data type. See Chapter 2 for more information on data types.
vertexfragment	This keyword is used to define a vertex fragment (or shader).
void	This keyword is used to represent a void (or empty) data type. See Chapter 2 for more information on data types.
volatile	This keyword is used as a hint to the compiler to indicate that a variable will change often. See Chapter 2 for more information on data types.
while	This keyword is used to define a conditional do-while loop.

This may seem like a lot to absorb, and it may seem confusing, especially if you are new to HLSL. Do not worry; this book is a learning book as well as a reference book. The information in Table 1-1 is mostly for reference, and you will see how it can be used as weexplore the syntax of the HLSL language as well as give examples. At this point in time, I want to outline all the major syntax and grammar components of the language. I will go into more details as we move forward.

In addition to the keywords in Table 1-1, there is a set of reserved keywords in HLSL that have no use currently. These keywords are essentially reserved for future use and have been listed in Table 1-2.

Table 1-2 HLSL Reserved Words

auto	break	case	catch
default	delete dynamic_cast	dynamic_cast	enum
explicit	end	goto	long
mutable	namespace	new	operator
private	protected	public	reinterpret_cast
short	signed	sizeof	static_cast
switch	template	this	throw
try	typename	union	unsigned
using	virtual		

Pre-Processor directives

In addition to reserved keywords, the HLSL language also defined a set of preprocessor directives to control the compilation of a program. The list of basic preprocessor directives supported includes the following: `#define`, `#endif`, `#else`, `#endif`, `#error`, `#if`, `#ifdef`, `#ifndef`, `#include`, `#line`, `#pragma` and `#undef`. A summary description of these directives is included in Table 1-3 below:

Table 1-3 Pre-Processor Directives

Directive	Definition
#define	This directive is used to declare a new compiler macro.
#if, #elseif, #endif, #ifdef, #ifndef	This set of directives is used to define a copiler conditional directive.
#error	This directive is used to force the compiler to emit an error and is generally used in conjunction with the conditional directives.
#include	This directive is used to include an external file into the compilation process.
#line	This directive is substituted with the current line number at which the directive is included within the source file.
#pragma	This directive is used to enable and control certain compiler behaviors. They will be discussed later in more details.

Take note that the `#include` directive will only function when the HLSL compiler is supplied with an `ID3DXInclude` interface. See Chapter 12 for more information on the include manager interface.

Note

Take note that the `#include` directive will only function when the HLSL compiler is supplied with an `ID3DXInclude` interface. See Chapter 12 for more information on the include manager interface.

The `#pragma` directive defines a set of specific sub-directives specific to the HLSL language. The `pack_matrix` directive can be used as follows:

```
#pragma pack_matrix (row_major) // or column_major
```

This directive tells the compiler how matrices defined within the HLSL file should be interpreted (either row or column major). I will discuss this in more detail in Chapter 2 when I will discuss the different data types defined by HLSL.

The second `#pragma` subdirective of interest is the `warning` directive. It helps control the output of warning messages by the HLSL compiler. The syntax of this directive is as follows:

```
#pragma warning( type : warning-number )
```

The *type* parameter defines how to threat the warnings specified in *warning-number*. The second parameter is a space-separated list of warning numbers that you want affected. Below is the list of the possible *type* parameters:

- **once**: Will only display the specified warnings once and ignore from that point on.
- **default**: Restored the treatment of the specified warnings to the default behavior.
- **disable**: Ignores all occurrences of the specified warnings.
- **error**: Treat the specified warnings as if they were compilation errors.

The final subdirective to the `#pragma` preprocessor directive is `def`. This instruction serves to give the compiler some hints as to what some of the registers should contain on a specific compilation profile. Keep in mind that this is an optimization hint to the compiler, and the final decision as to how registers are used is always up to the compiler.

Note

Currently, only the constant registers (`c#`) are supported by the `#pragma def` instruction.

A compilation profile defines a combination of vertex or pixel shader along with a hardware version to target. A complete list of the current profiles is available in Table 1-4.

Table 1-4 HLSL Compilation Profiles

Profile	Definition
vs_1_1	Vertex Shader version 1.1
vs_2_0	Vertex Shader version 2.0
vs_2_x	This profile represents the extended Vertex Shader version 2.0, which has extra capabilities which include predication, dynamic flow control and a number of temporary registers greater than 12.
vs_3_0	Vertex Shader version 3.0
ps_1_1	Pixel Shader version 1.1
ps_1_2	Pixel Shader version 1.2
ps_1_3	Pixel Shader version 1.3
ps_1_4	Pixel Shader version 1.4
ps_2_0	Pixel Shader version 2.0
ps_2_x	This profile represents the extended Pixel Shader version 2.0, which has extra capabilities which include predication, dynamic flow control and a number of temporary registers greater than 12.
ps_3_0	Pixel Shader version 3.0

Note

Although not exposed within the HLSL language, Direct3D exposes two variants to the pixel 2.0 profile. The *ps_2_a* and *ps_2_b* profiles are similar to the *ps_2_0* profile but offers some extra functionality as follows:

•Number of temporary registers is greater or equal to 22 (32 for *ps_2_b*).
•Arbitrary source swizzle. (only *ps_2_a*)
•Gradient instructions: `dsx`, `dsy`. (only *ps_2_a*)
•Predication. (only *ps_2_a*)
•No dependent texture read limit. (only *ps_2_a*)
•No limit for the number of texture instructions.

Also keep in mind that the 2.a and 2.b profiles do not offer backward compatibility between each other and were offered as an intermediate way of exposing new functionality until the 3.0 shader model was available.

Understanding the HLSL Syntax

Now that we have gone over what the reserved words and some of the preprocessor directives for the HLSL syntax are, we can start going over the syntax itself. Syntaxes for computer languages are generally presented in Backus-Naur Form (BNF) which is an easy to understand form.

Grammars defined as BNF are really easy to follow as they have a form which is straightforward and easy to decipher. I will go over it right away since I will use it a few times later on in the following chapters. Generally, each line of the grammar represents a rule. Each rule defines a result and of what it is comprised. For example:

```
If-statement: KW_IF KW_LPAREN expression KW_RPAREN statement
```

In this grammar rule, we define an *if* conditional statement. This statement is defined as the *if* keyword followed by an expression enclosed within parenthesis followed by a statement. The definition of *expression* and *statement* is being defined in following rules. In addition to the basic rules, a few important operators can be used within the rules that you should know.

- **a | b**: The pipe character can used to define an *or* condition within a rule.
- **(a)**: Parenthesis can be used to enclose segments of the rule into a block.
- **[a]**: A set of rules within square braces means the segment is optional.
- **a +**: The use of the plus character indicates one or more repetitions of the segment.
- **a ***: The use of the star character indicates any number of repetitions of the segment.

With the information above, you should be able to read through grammar rules pretty easily. Before I show the HLSL grammar syntax, a few lexical conventions will beexplored.

Lexical Conventions

Although the syntax of a language defines how everything ties together and, for example, what defines a function or a statement, it only defines that an expression may be a combination of operators and identifiers. This means that the language syntax does not define the grammar of the language (i.e., what is an identifier). The next few paragraphs detailsome of the basic lexical conventions used by the HLSL compiler.

Whitespaces

White spaces within the HLSL language are defined as being any one of the following:

- Space.
- Tab.

- End of line.
- C style comments (/* */).
- C++ style comments (//).
- Assembly style comments (;) within an *asm* block.

Numbers

Numbers within HLSL can either be floating-point or integer numbers. Floating-point numbers are represented as follows:

```
Float: (fractional-constant [exponent-part] [floating-suffix]) |
       (digit-sequence exponent-part [floating-suffix])

Fractional-constant: ( [digit-sequence] . digit-sequence ) |
                     ( digit-sequence . )

Exponent-part: ( e [sign] digit-sequence ) | ( E [sign] digit-sequence )

Sign: + | -

Digit-sequence: digit | ( digit-sequence digit )

Floating-suffix: h | H | f | F
```

Integer numbers follow a similar syntax:

```
Integer: integer-constant [integer-suffix]

Integer-constant: digit-sequence | ( 0 digit-sequence ) |
                  (0 x digit-sequence)

Digit-sequence: digit | ( digit-sequence digit )

Integer-suffix: u | U | l | L
```

Characters

The HLSL language allows strings and characters to be defined. Strings being composed of characters, let's start by looking at the definition of character:

- 'c' (character)
- '\t', '\n', … (escape characters)
- '\###' (octal escape sequence)

- '\x##' (hexadecimal escape sequence)

Note

Take note that escape characters cannot be used within preprocessor directives.

Strings themselves are enclosed within quotes and can contain any valid combination of characters as previously defined.

Identifiers

Identifiers represent language elements such as function names or variables. With exception to reserved keywords outlined earlier, identifiers are defined as any combination of letters and digits as long as the first character is a letter.

Operators

The HLSL language defines a set of operators used within expressions. Table 1-5 enumerates all the standard operators along with their meaning. If you are familiar with C and C++, most of the operators should seem straightforward to you.

Table 1-5 HLSL Operators

Operator	Definition
++	Unary increment
--	Unary decrement
&&	And
||	Or
==	Equal
::	Scope (for structures and classes)
<<	Left binary shift
<<=	Self assigning left binary shift. This means that *a* <<= *b* is equivalent to *a* = *a* << *b*.
>>	Right binary shift
>>=	Self assigning right binary shift. This means that *a* >>= *b* is equivalent to *a* = *a* >> *b*.

Table 1-5 HLSL Operators

Operator	Definition
…	Ellipsis operator (used for variable parameter functions)
<=	Lesser than or equal to
>=	Greater than or equal to
!=	Not equal
*=	Self assigning multiplication. This means that *a* *= *b* is equivalent to *a* = *a* * *b*.
/=	Self assigning division. This means that *a* /= *b* is equivalent to *a* = *a* / *b*.
+=	Self assigning addition. This means that *a* += *b* is equivalent to *a* = *a* + *b*.
-=	Self assigning subtraction. This means that *a* -= *b* is equivalent to *a* = *a* - *b*.
%=	Self assigning modulo. This means that *a* %= *b* is equivalent to *a* = *a* % *b*.
&=	Self assigning and. This means that *a* &= *b* is equivalent to *a* = *a* & *b*.
|=	Self assigning or. This means that *a* |= *b* is equivalent to *a* = *a* | *b*.
^=	Self assigning power of. This means that *a* ^= *b* is equivalent to *a* = *a* ^ *b*.
->	Indirection operator, used to access structure members.

Language Syntax

The syntax for the HLSL language is pretty straightforward. Although it may look complex, you will get the hang of it as you use the language. In addition, you should not worry too much about the syntax itself at this point in time as we'll explore specific components of the language in the next few chapters. Since the actual syntax is quite long, I have decided to include it in Appendix D. You can also refer to the DirectX SDK documentation for more complete details.

Looking at the top of the listing, you can see that an HLSL file is defined as a `program`. Each program is either empty or is composed of a set of `decls` (declarations). Declarations are a little more complicated. The first two lines serve to define that a declaration is a combination of one of many `decl` (declaration). From there, a declaration is defined either as an empty statement, a type declaration, a variable declaration, a structure declaration, a function declaration or a technique declaration. The grammar defines the different types of declarations and so on.

Summary and what's next?

In this chapter, I have briefly overviewed the history of DirectX and how the shader technology has progressed over the past few years. With the increased complexity of the new shader models 2.0 and 3.0, developers needed to be able to take advantage of the new capabilities of the language but at the same time able to be efficient at the task. Since the capabilities of the new shader pipelines resembled more and more of the instruction sets of modern processors, it made sense to develop a high-level language, allowing developers to focus on the task of creating shaders without having to worry about little details such as register allocations.

Because of this need, Microsoft has developed and released the HLSL shading language as part of the DirectX SDK, finally allowing developers to leverage the latest shader innovations and bring even more realism to their graphics. In this chapter, I have gone over some of the basic concepts behind the grammar of the HLSL language. Although this may have seemed like a lot to absorb in a single chapter, you need not to worry, as I'll go into more details on some of the specifics over the next few chapters.

In the next chapter, I will start discussing the concepts of variables, expressions and data types exposed by the HLSL shading language. Let's get going…