Chapter 3
The Graphics Processing Unit

“The display is the computer.”
—Jen-Hsun Huang

Historically, hardware graphics acceleration has started at the end of the pipeline, first performing rasterization of a triangle’s scanlines. Successive generations of hardware have then worked back up the pipeline, to the point where some higher level application-stage algorithms are being committed to the hardware accelerator. Dedicated hardware’s only advantage over software is speed, but speed is critical.

Over the past decade, graphics hardware has undergone an incredible transformation. The first consumer graphics chip to include hardware vertex processing (NVIDIA’s GeForce256) shipped in 1999. NVIDIA coined the term graphics processing unit (GPU) to differentiate the GeForce 256 from the previously available rasterization-only chips, and it stuck [898]. Over the next few years, the GPU evolved from configurable implementations of a complex fixed-function pipeline to highly programmable “blank slates” where developers could implement their own algorithms. Programmable shaders of various kinds are the primary means by which the GPU is controlled. The vertex shader enables various operations (including transformations and deformations) to be performed on each vertex. Similarly, the pixel shader processes individual pixels, allowing complex shading equations to be evaluated per pixel. The geometry shader allows the GPU to create and destroy geometric primitives (points, lines, triangles) on the fly. Computed values can be written to multiple high-precision buffers and reused as vertex or texture data. For efficiency, some parts of the pipeline remain configurable, not programmable, but the trend is towards programmability and flexibility [123].
3.1 GPU Pipeline Overview

The GPU implements the geometry and rasterization conceptual pipeline stages described in Chapter 2. These are divided into several hardware stages with varying degrees of configurability or programmability. Figure 3.1 shows the various stages color coded according to how programmable or configurable they are. Note that these physical stages are split up slightly differently than the functional stages presented in Chapter 2.

The vertex shader is a fully programmable stage that is typically used to implement the “Model and View Transform,” “Vertex Shading,” and “Projection” functional stages. The geometry shader is an optional, fully programmable stage that operates on the vertices of a primitive (point, line or triangle). It can be used to perform per-primitive shading operations, to destroy primitives, or to create new ones. The clipping, screen mapping, triangle setup, and triangle traversal stages are fixed-function stages that implement the functional stages of the same names. Like the vertex and geometry shaders, the pixel shader is fully programmable and performs the “Pixel Shading” function stage. Finally, the merger stage is somewhere between the full programmability of the shader stages and the fixed operation of the other stages. Although it is not programmable, it is highly configurable and can be set to perform a wide variety of operations. Of course, it implements the “Merging” functional stage, in charge of modifying the color, Z-buffer, blend, stencil, and other related buffers.

Over time, the GPU pipeline has evolved away from hard-coded operation and toward increasing flexibility and control. The introduction of programmable shader stages was the most important step in this evolution. The next section describes the features common to the various programmable stages.

3.2 The Programmable Shader Stage

Modern shader stages (i.e., those that support Shader Model 4.0, DirectX 10 and later, on Vista) use a common-shader core. This means that the vertex, pixel, and geometry shaders share a programming model. We differentiate in this book between the common-shader core, the functional description seen by the applications programmer, and unified shaders, a GPU architecture that maps well to this core. See Section 18.4. The common-shader core is the API; having unified shaders is a GPU feature.

Earlier GPUs had less commonality between vertex and pixel shaders and did not have geometry shaders. Nonetheless, most of the design elements for this model are shared by older hardware; for the most part, older versions’ design elements are either simpler or missing, not radically different. So, for now we will focus on Shader Model 4.0 and discuss older GPUs’ shader models in later sections.

Describing the entire programming model is well beyond the scope of this book, and there are many documents, books, and websites that already do so [261, 338, 647, 1084]. However, a few comments are in order. Shaders are programmed using C-like shading languages such as HLSL, Cg, and GLSL. These are compiled to a machine-independent assembly language, also called the intermediate language (IL). Previous shader models allowed programming directly in the assembly language, but as of DirectX 10, programs in this language are visible as debug output only [123]. This assembly language is converted to the actual machine language in a separate step, usually in the drivers. This arrangement allows compatibility across different hardware implementations. This assembly language can be seen as defining a virtual machine, which is targeted by the shading language compiler.

This virtual machine is a processor with various types of registers and data sources, programmed with a set of instructions. Since many graphics operations are done on short vectors (up to length 4), the processor has 4-way SIMD (single-instruction multiple-data) capabilities. Each register contains four independent values. 32-bit single-precision floating-point scalars and vectors are the basic data types; support for 32-bit integers has recently been added, as well. Floating-point vectors typically contain data such as positions (xyzw), normals, matrix rows, colors (rgb), or texture coordinates (uvvw). Integers are most often used to represent counters, indices, or bit masks. Aggregate data types such as structures, arrays, and matrices are also supported. To facilitate working with vectors, swizzling, the replication of any vector component, is also supported. That is, a vector’s elements can be reordered or duplicated as desired. Similarly, masking, where only the specified vector elements are used, is also supported.

A draw call invokes the graphics API to draw a group of primitives, so causing the graphics pipeline to execute. Each programmable shader stage has two types of inputs: uniform inputs, with values that remain constant throughout a draw call (but can be changed between draw calls),
and varying inputs, which are different for each vertex or pixel processed by the shader. A texture is a special kind of uniform input that once was always a color image applied to a surface, but that now can be thought of as any large array of data. It is important to note that although shaders have a wide variety of inputs, which they can address in different ways, the outputs are extremely constrained. This is the most significant way in which shaders are different from programs executing on general-purpose processors. The underlying virtual machine provides special registers for the different types of inputs and outputs. Uniform inputs are accessed via read-only constant registers or constant buffers, so called because their contents are constant across a draw call. The number of available constant registers is much larger than the number of registers available for varying inputs or outputs. This is because the varying inputs and outputs need to be stored separately for each vertex or pixel, and the uniform inputs are stored once and reused across all the vertices or pixels in the draw call. The virtual machine also has general-purpose temporary registers, which are used for scratch space. All types of registers can be array-indexed using integer values in temporary registers. The inputs and outputs of the shader virtual machine can be seen in Figure 3.2.

Operations that are common in graphics computations are efficiently executed on modern GPUs. Typically, the fastest operations are scalar and vector multiplications, additions, and their combinations, such as multiply-add and dot-product. Other operations, such as reciprocal, square root, sine, cosine, exponentiation, and logarithm, tend to be slightly more costly but still fairly speedy. Texturing operations (see Chapter 6) are efficient, but their performance may be limited by factors such as the time spent waiting to retrieve the result of an access. Shading languages expose the most common of these operations (such as additions and multiplications) via operators such as * and +. The rest are exposed through intrinsic functions, e.g., atan(), dot(), log(), and many others. Intrinsic functions also exist for more complex operations, such as vector normalization and reflection, cross products, matrix transpose and determinant, etc.

The term flow control refers to the use of branching instructions to change the flow of code execution. These instructions are used to implement high-level language constructs such as “if” and “case” statements, as well as various types of loops. Shaders support two types of flow control. Static flow control branches are based on the values of uniform inputs. This means that the flow of the code is constant over the draw call. The primary benefit of static flow control is to allow the same shader to be used in a variety of different situations (e.g., varying numbers of lights). Dynamic flow control is based on the values of varying inputs. This is much more powerful than static flow control but is more costly, especially if the code flow changes erratically between shader invocations. As discussed in Section 18.4.2, a shader is evaluated on a number of vertices or pixels at a time. If the flow selects the “if” branch for some elements and the “else” branch for others, both branches must be evaluated for all elements (and the unused branch for each element is discarded).

Shader programs can be compiled offline before program load or during run time. As with any compiler, there are options for generating different output files and for using different optimization levels. A compiled shader is stored as a string of text, which is passed to the GPU via the driver.

3.3 The Evolution of Programmable Shading

The idea of a framework for programmable shading dates back to 1984 with Cook’s shade trees [194]. A simple shader and its corresponding shade tree are shown in Figure 3.3. The RenderMan Shading Language [30, 1283] was developed from this idea in the late 80’s and is still widely used today for film production rendering. Before GPUs supported programmable shaders natively, there were several attempts to implement programmable shading operations in real time via multiple rendering passes. The Quake III: Arena scripting language was the first widespread commercial success in this area in 1999 [558, 604]. In 2000, Peercy et al. [993] described a system that translated RenderMan shaders to run in multiple passes on graphics hardware. They found that GPUs lacked two features that would make this approach
Shading Model to distinguish hardware with different shader capabilities. The GeForce 3 supported vertex shader model 1.1 and pixel shader model 1.1 (shader model 1.0 was intended for hardware that never shipped). During 2001, GPUs progressed closer to a general pixel shader programming model. DirectX 8.1 added pixel shader models 1.2 to 1.4 (each meant for different hardware), which extended the capabilities of the pixel shader further, adding additional instructions and more general support for dependent texture reads.

The year 2002 saw the release of DirectX 9.0 including Shader Model 2.0 (and its extended version 2.X), which featured truly programmable vertex and pixel shaders. Similar functionality was also exposed under OpenGL using various extensions. Support for arbitrary dependent texture reads and storage of 16-bit floating point values was added, finally completing the set of requirements identified by Peercy et al. in 2000 [993]. Limits on shader resources such as instructions, textures, and registers were increased, so shaders became capable of more complex effects. Support for flow control was also added. The growing length and complexity of shaders made the assembly programming model increasingly cumbersome. Fortunately, DirectX 9.0 also included a new shader programming language called HLSL (High Level Shading Language). HLSL was developed by Microsoft in collaboration with NVIDIA, which released a cross-platform variant called Cg [818]. Around the same time, the OpenGL ARB (Architecture Review Board) released a somewhat similar language for OpenGL, called GLSL [647, 1084] (also known as GLSLang). These languages were heavily influenced by the syntax and design philosophy of the C programming language and also included elements from the RenderMan Shading Language.

Shader Model 3.0 was introduced in 2004 and was an incremental improvement, turning optional features into requirements, further increasing resource limits and adding limited support for texture reads in vertex shaders. When a new generation of game consoles was introduced in late 2005 (Microsoft’s Xbox 360) and 2006 (Sony Computer Entertainment’s PLAYSTATION®3 system), they were equipped with Shader Model 3.0-level GPUs. The fixed-function pipeline is not entirely dead: Nintendo’s Wii console shipped in late 2006 with a fixed-function GPU [207]). However, this is almost certainly the last console of this type, as even mobile devices such as cell phones can use programmable shaders (see Section 18.4.3).

Other languages and environments for shader development are available. For example, the Sh language [837, 838] allows the generation and combination [839] of GPU shaders through a C++ library. This open-source project runs on a number of platforms. On the other end of the spectrum, several visual programming tools have been introduced to allow artists (most of whom are not comfortable programming in C-like languages) to

very general: the ability to use computation results as texture coordinates (dependent texture reads), and support for data types with extended range and precision in textures and color buffers. One of the proposed data types was a novel (at the time) 16-bit floating point representation. At this time, no commercially available GPU supported programmable shading, although most had highly configurable pipelines [898].

In early 2001, NVIDIA’s GeForce 3 was the first GPU to support programmable vertex shaders [778], exposed through DirectX 8.0 and extensions to OpenGL. These shaders were programmed in an assembly-like language that was converted by the drivers into microcode on the fly. Pixel shaders were also included in DirectX 8.0, but pixel shader SM 1.1 fell short of actual programmability—the very limited “programs” supported were converted into texture blending states by the driver, which in turn wired together hardware “register combiners.” These “programs” were not only limited in length (12 instructions or less) but also lacked the two elements (dependent texture reads and float data) that Peercy et al. had identified as crucial to true programmability.

Shaders at this time did not allow for flow control (branching), so conditionals had to be emulated by computing both terms and selecting or interpolating between the results. DirectX defined the concept of a

1 The GeForce 3 did support a dependent texture read of sorts, but only in an extremely limited fashion.
3. The Graphics Processing Unit

3.3. The Evolution of Programmable Shading

3.3.1 Comparison of Shader Models

Although this chapter focuses on Shader Model 4.0 (the newest at time of writing), often developers need to support hardware that uses older shading models. For this reason we give a brief comparison between the capabilities of several recent shading models: 2.0 (and its extended version of 2.X), 3.0 and 4.0. A listing of all the differences is beyond the scope of this book; detailed information is available from the Microsoft Developer Network (MSDN) and their DirectX SDK [261].

We focus on DirectX here, because of its distinct releases, versus OpenGL's evolving levels of extensions, some approved by the OpenGL Architecture Review Board (ARB), some vendor-specific. This extension system has the advantage that cutting-edge features from a specific independent hardware vendor (IHV) can be used immediately. DirectX 9 and earlier support IHV variations by exposing "capability bits" that can be examined to see if a GPU supports a feature. With DirectX 10, Microsoft has moved sharply away from this practice and toward a standardized model that all IHVs must support. Despite the focus here on DirectX, the following discussion also has relevance to OpenGL, in that the associated underlying GPUs of the same time periods have the same features.

Table 3.1 compares the capabilities of the various shader models. In the table, "VS" stands for "vertex shader" and "PS" for "pixel shader" (Shader Model 4.0 introduced the geometry shader, with capabilities similar to those of the vertex shader). If neither "VS" nor "PS" appears, the row applies to both vertex and pixel shaders. Since the virtual machine is 4-way SIMD, each register can store between one and four independent values. "Instruction Slots" refers to the maximum number of instructions that the shader can contain. "Max. Steps Executed" indicates the maximum number of instructions that can be executed, taking branching and looping into account. "Temp. Registers" shows the number of general-purpose registers that are available for storing intermediate results. "Constant Registers" indicates the number of constant values that can be input to the shader. "Flow Control, Predication" refers to the ability to compute conditional expressions and execute loops via branching and predication (i.e., the ability to conditionally execute or skip an instruction). "Textures" shows the number of distinct textures (see Chapter 6) that can be accessed by the shader (each texture may be accessed multiple times). "Integer Support" refers to the ability to operate on integer data types with bitwise operators and integer arithmetic. "VS Input Registers" shows the number of varying input registers that can be accessed by the vertex shader. "Interpolator Registers" are output registers for the vertex shader and input

---

2 Shader Models 1.0 through 1.4 were early, limited versions that are no longer actively used.
3.4 The Vertex Shader

The vertex shader is the first stage in the functional pipeline shown in Figure 3.2. While this is the first stage that does any graphical processing, it is worth noting that some data manipulation happens before this stage. In what DirectX calls the input assembler [123, 261], a number of streams of data can be woven together to form the sets of vertices and primitives sent down the pipeline. For example, an object could be represented by one array of positions and one array of colors. The input assembler would create this object’s triangles (or lines or points) by essentially creating vertices with positions and colors. A second object could use the same array of positions (along with a different model transform matrix) and a different array of colors for its representation. Data representation is discussed in detail in Section 12.4.5. There is also support in the input assembler to perform instancing. This allows an object to be drawn a number of times with some varying data per instance, all with a single draw call. The use of instancing is covered in Section 15.4.2. The input assembler in DirectX 10 also tags each instance, primitive, and vertex with an identifier number that can be accessed by any of the shader stages that follow. For earlier shader models, such data has to be added explicitly to the model.

A triangle mesh is represented by a set of vertices and additional information describing which vertices form each triangle. The vertex shader is the first stage to process the triangle mesh. The data describing what triangles are formed is unavailable to the vertex shader; as its name implies, it deals exclusively with the incoming vertices. In general terms, the vertex shader provides a way to modify, create, or ignore values associated with each polygon’s vertex, such as its color, normal, texture coordinates, and position. Normally the vertex shader program transforms vertices from model space to homogeneous clipping space; at a minimum, a vertex shader must always output this location.

This functionality was first introduced in 2001 with DirectX 8. Because it was the first stage on the pipeline and invoked relatively infrequently, it could be implemented on either the GPU or the CPU, which would then send on the results to the GPU for rasterization. Doing so made the transition from older to newer hardware a matter of speed, not functionality. All GPUs currently produced support vertex shading.

A vertex shader itself is very much the same as the common core virtual machine described earlier in Section 3.2. Every vertex passed in is processed by the vertex shader program, which then outputs a number of values that are interpolated across a triangle or line.3 The vertex shader can neither create nor destroy vertices, and results generated by one vertex cannot be passed on to another vertex. Since each vertex is treated independently, any number of shader processors on the GPU can be applied in parallel to the incoming stream of vertices.

Chapters that follow explain a number of vertex shader effects, such as shadow volume creation, vertex blending for animating joints, and silhouette rendering. Other uses for the vertex shader include:

- Lens effects, so that the screen appears fish-eyed, underwater, or otherwise distorted.

3 Older shader models also supported output of the size of a point sprite particle object, but sprite functionality is now part of the geometry shader.
3. The Graphics Processing Unit

Figure 3.5. On the left, a normal teapot. A simple shear operation performed by a vertex shader program produces the middle image. On the right, a noise function creates a field that distorts the model. (Images produced by FX Composer 2, courtesy of NVIDIA Corporation.)

- Object definition, by creating a mesh only once and having it be deformed by the vertex shader.
- Object twist, bend, and taper operations.
- Procedural deformations, such as the movement of flags, cloth, or water [592].
- Primitive creation, by sending degenerate meshes down the pipeline and having these be given an area as needed. This functionality is replaced by the geometry shader in newer GPUs.
- Page curls, heat haze, water ripples, and other effects can be done by using the entire frame buffer's contents as a texture on a screen-aligned mesh undergoing procedural deformation.
- Vertex texture fetch (available in SM 3.0 on up) can be used to apply textures to vertex meshes, allowing ocean surfaces and terrain height fields to be applied inexpensively [23, 703, 887].

Some deformations done using a vertex shader are shown in Figure 3.5.

The output of the vertex shader can be consumed in a number of different ways. The usual path is for each instance's triangles to then be generated and rasterized, and the individual pixel fragments produced sent to the pixel shader program for continued processing. With the introduction of Shader Model 4.0, the data can also be sent to the geometry shader, streamed out, or both. These options are the subject of the next section.

3.5 The Geometry Shader

The geometry shader was added to the hardware-accelerated graphics pipeline with the release of DirectX 10, in late 2006. It is located immediately after the vertex shader in the pipeline, and its use is optional. While a required part of Shader Model 4.0, it is not used in earlier shader models.

The input to the geometry shader is a single object and its associated vertices. The object is typically a triangle in a mesh, a line segment, or simply a point. In addition, extended primitives can be defined and processed by the geometry shader. In particular, three additional vertices outside of a triangle can be passed in, and the two adjacent vertices on a polyline can be used. See Figure 3.6.

The geometry shader processes this primitive and outputs zero or more primitives. Output is in the form of points, polylines, and triangle strips. More than one triangle strip, for example, can be output by a single invocation of the geometry shader program. As important, no output at all can be generated by the geometry shader. In this way, a mesh can be selectively modified by editing vertices, adding new primitives, and removing others.

The geometry shader program is set to input one type of object and output one type of object, and these types do not have to match. For example, triangles could be input and their centroids be output as points, one per triangle input. Even if input and output object types match, the data carried at each vertex can be omitted or expanded. As an example, the triangle's plane normal could be computed and added to each output vertex's data. Similar to the vertex shader, the geometry shader must output a homogeneous clip space location for each vertex produced.

The geometry shader is guaranteed to output results from primitives in the same order as they are input. This affects performance, because if a number of shader units run in parallel, results must be saved and ordered. As a compromise between capability and efficiency, there is a limit in Shader Model 4.0 of a total of 1024 32-bit values that can be generated per execution. So, generating a thousand bush leaves given a single leaf as input is not feasible and is not the recommended use of the geometry shader. Tessellation of simple surfaces into more elaborate triangle meshes is also not recommended [123]. This stage is more about programmatically modifying incoming data or making a limited number of copies, not about

Figure 3.6. Geometry shader input for a geometry shader program is of some single type: point, line segment, triangle. The two rightmost primitives, which include vertices adjacent to the line and triangle objects, can also be used.
massively replicating or amplifying it. For example, one use is to generate six transformed copies of data in order to simultaneously render the six faces of a cube map; see Section 8.4.3. Additional algorithms that can take advantage of the geometry shader include creating various sized particles from point data, extruding fins along silhouettes for fur rendering, and finding object edges for shadow algorithms. See Figure 3.7 for still more. These and other uses are discussed throughout the rest of the book.

3.5.1 Stream Output
The standard use of the GPU's pipeline is to send data through the vertex shader, then rasterize the resulting triangles and process these in the pixel shader. The data always passed through the pipeline and intermediate results could not be accessed. The idea of stream output was introduced in Shader Model 4.0. After vertices are processed by the vertex shader (and, optionally, the geometry shader), these can be output in a stream, i.e., an ordered array, in addition to being sent on to the rasterization stage. Rasterization could, in fact, be turned off entirely and the pipeline then used purely as a non-graphical stream processor. Data processed in this way can be sent back through the pipeline, thus allowing iterative processing. This type of operation is particularly useful for simulating flowing water or other particle effects, as discussed in Section 10.7.

3.6 The Pixel Shader
After the vertex and geometry shaders perform their operations, the primitive is clipped and set up for rasterization, as explained in the last chapter.

This section of the pipeline is relatively fixed in its processing steps, not programmable. Each triangle is traversed and the values at the vertices interpolated across the triangle's area. The pixel shader is the next programmable stage. In OpenGL this stage is known as the fragment shader, which in some ways is a better name. The idea is that a triangle covers each pixel's cell fully or partially, and the material portrayed is opaque or transparent. The rasterizer does not directly affect the pixel's stored color, but rather generates data that, to a greater or lesser extent, describes how the triangle covers the pixel cell. It is then during merging that this fragment's data is used to modify what is stored at the pixel.

The vertex shader program's outputs effectively become the pixel shader program's inputs. A total of 16 vectors (4 values each) can be passed from the vertex shader to the pixel shader in Shader Model 4.0. When the geometry shader is used, it can output 32 vectors to the pixel shader [261].

Additional inputs were added specifically for the pixel shader with the introduction of Shader Model 3.0. For example, which side of a triangle is visible was added as an input flag. This knowledge is important for rendering a different material on the front versus back of each triangle in a single pass. The screen position of the fragment is also available to the pixel shader.

The pixel shader's limitation is that it can influence only the fragment handed it. That is, when a pixel shader program executes, it cannot send its results directly to neighboring pixels. Rather, it uses the data interpolated from the vertices, along with any stored constants and texture data, to compute results that will affect only a single pixel. However, this limitation is not as severe as it sounds. Neighboring pixels can ultimately be affected by using image processing techniques, described in Section 10.9.

The one case in which the pixel shader can access information for adjacent pixels (albeit indirectly) is the computation of gradient or derivative information. The pixel shader has the ability to take any value and compute the amount by which it changes per pixel along the x and y screen axes. This is useful for various computations and texture addressing. These gradients are particularly important for operations such as filtering (see Section 6.2.2). Most GPUs implement this feature by processing pixels in groups of $2 \times 2$ or more. When the pixel shader requests a gradient value, the difference between adjacent pixels is returned. One result of this implementation is that gradient information cannot be accessed in parts of the shader affected by dynamic flow control—all the pixels in a group must be processing the same instructions. This is a fundamental limitation which exists even in offline rendering sys-

---

Footnotes:

4The notable exception is that the pixel shader program can specify what type of interpolation is used, e.g., perspective corrected or screen space (or none at all).

5In DirectX 10.1 the vertex shader will both input and output 32 vectors.
tems [31]. The ability to access gradient information is a unique capability of the pixel shader, not shared by any of the other programmable shader stages.

Pixel shader programs typically set the fragment color for merging in the final merging stage. The depth value generated in the rasterization stage can also be modified by the pixel shader. The stencil buffer value is not modifiable, but rather is passed through to the merge stage. In SM 2.0 and ou, a pixel shader can also discard incoming fragment data, i.e., generate no output. Such operations can cost performance, as optimizations normally performed by the GPU cannot then be used. See Section 18.3.7 for details. Operations such as fog computation and alpha testing have moved from being merge operations to being pixel shader computations in SM 4.0 [123].

Current pixel shaders are capable of doing a huge amount of processing. The ability to compute any number of values in a single rendering pass gave rise to the idea of multiple render targets (MRT). Instead of saving results of a pixel shader’s program to a single color buffer, multiple vectors could be generated for each fragment and saved to different buffers. These buffers must be the same dimensions, and some architectures require them each to have the same bit depth (though with different formats, as needed). The number of PS output registers in Table 3.1 refers to the number of separate buffers accessible, i.e., 4 or 8. Unlike the displayable color buffer, there are other limitations on any additional targets. For example, typically no antialiasing can be performed. Even with these limitations, MRT functionality is a powerful aid in performing rendering algorithms more efficiently. If a number of intermediate results images are to be computed from the same set of data, only a single rendering pass is needed, instead of one pass per output buffer. The other key capability associated with MRTs is the ability to read from these resulting images as textures.

3.7 The Merging Stage

As discussed in Section 2.4.4, the merging stage is where the depths and colors of the individual fragments (generated in the pixel shader) are combined with the frame buffer. This stage is where stencil-buffer and Z-buffer operations occur. Another operation that takes place in this stage is color blending, which is most commonly used for transparency and compositing operations (see Section 5.7).

The merging stage occupies an interesting middle point between the fixed-function stages, such as clipping, and the fully programmable shader stages. Although it is not programmable, its operation is highly configurable. Color blending in particular can be set up to perform a large number of different operations. The most common are combinations of multiplication, addition, and subtraction involving the color and alpha values, but other operations are possible, such as minimum and maximum, as well as bitwise logic operations. DirectX 10 added the capability to blend two colors from the pixel shader with the frame buffer color—this capability is called dual-color blending.

If MRT functionality is employed, then blending can be performed on multiple buffers. DirectX 10.1 introduced the capability to perform different blend operations on each MRT buffer. In previous versions, the same blending operation was always performed on all buffers (note that dual-color blending is incompatible with MRT).

3.8 Effects

This tour of the pipeline has focused so far on the various programmable stages. While vertex, geometry, and pixel shader programs are necessary to control these stages, they do not exist in a vacuum. First, an individual shader program is not particularly useful in isolation: A vertex shader program feeds its results to a pixel shader. Both programs must be loaded for any work to be done. The programmer must perform some matching of the outputs of the vertex shader to the inputs of the pixel shader. A particular rendering effect may be produced by any number of shader programs executed over a few passes. Beyond the shader programs themselves, state variables must sometimes be set in a particular configuration for these programs to work properly. For example, the render’s state includes whether and how the Z-buffer and stencil buffer are each used, and how a fragment affects the existing pixel value (e.g., replace, add, or blend).

For these reasons, various groups have developed effects languages, such as HLSL FX, CgFX, and COLLADA FX. An effect file attempts to encapsulate all the relevant information needed to execute a particular rendering algorithm [261, 974]. It typically defines some global arguments that can be assigned by the application. For example, a single effect file might define the vertex and pixel shaders needed to render a convincing plastic material. It would expose arguments such as the plastic color and roughness so that these could be changed for each model rendered, but using the same effect file.

To show the flavor of an effect file, we will walk through a trimmed-down example taken from NVIDIA’s FX Composer 2 effects system. This DirectX 9 HLSL effect file implements a very simplified form of Gooch shading [423]. One part of Gooch shading is to use the surface normal and compare it to the light’s location. If the normal points toward the light, a warm tone is used to color the surface; if it points away, a cool tone is used.
Angles in between interpolate between these two user-defined colors. This shading technique is a form of non-photorealistic rendering, the subject of Chapter 11. An example of this effect in action is shown in Figure 3.8.

Effect variables are defined at the beginning of the effect file. The first few variables are “untweakable,” parameters related to the camera position that are automatically tracked for the effect:

```c
float4x4 WorldXf : World;
float4x4 WorldIXx : WorldInverseTranspose;
float4x4 WvpXf : WorldViewProjection;
```

The syntax is `type id : semantic`. The type `float4x4` is used for matrices, the name is user defined, and the semantic is a built-in name. As the semantic names imply, the `WorldXf` is the model-to-world transform matrix, the `WorldIXx` is the inverse transpose of this matrix, and the `WvpXf` is the matrix that transforms from model space to the camera’s clip space. These values with recognized semantics are expected to be provided by the application and not shown in the user interface.

Next, the user-defined variables are specified:

```c
float3 LampOPos : Position <
    string Object = "PointLight0";
    string UName = "Lamp 0 Position";
    string Space = "World";
> = {0.5f, 2.0f, 1.25f};
```

```c
float3 WarmColor <
    string UName = "Gouch Warm Tone";
    string UIWidget = "Color";
> = {1.3f, 0.9f, 0.15f};
```

3.8. Effects

```c
float3 CoolColor <
    string UName = "Gouch Cool Tone";
    string UIWidget = "Color";
> = {0.05f, 0.05f, 0.6f};
```

Here some additional annotations are provided inside the angle brackets “<>” and then default values are assigned. The annotations are application-specific and have no meaning to the effect or to the shader compiler. Such annotations can be queried by the application. In this case the annotations describe how to expose these variables within the user interface.

Data structures for shader input and output are defined next:

```c
struct appdata {
    float3 Position : POSITION;
    float3 Normal : NORMAL;
};
```

```c
struct vertexOutput {
    float4 HPosition : POSITION;
    float3 LightVec : TEXCOORD1;
    float3 WorldNormal : TEXCOORD2;
};
```

The `appdata` defines what data is at each vertex in the model and so defines the input data for the vertex shader program. The `vertexOutput` is what the vertex shader produces and the pixel shader consumes. The use of `TEXCOORD` as the output names is an artifact of the evolution of the pipeline. At first, multiple textures could be attached to a surface, so these additional datafields are called texture coordinates. In practice, these fields hold any data that is passed from the vertex to the pixel shader.

Next, the various shader program code elements are defined. We have only one vertex shader program:

```c
vertexOutput std_VS(appdata IN) {
    vertexOutput OUT;
    float4 No = float4(IN.Normal,0);
    OUT.WorldNormal = mul(No,WorldIXx).xyz;
    float4 Po = float4(IN.Position,1);
    float4 Pw = mul(Po,WorldXf);
    OUT.LightVec = (LampOPos - Pw.xyz);
    OUT.HPosition = mul(Po,WvpXf);
    return OUT;
}
```
This program first computes the surface’s normal in world space by using a matrix multiplication. Transforms are the subject of the next chapter, so we will not explain why the inverse transpose is used here. The position in world space is also computed by applying the offscreen transform. This location is subtracted from the light’s position to obtain the direction vector from the surface to the light. Finally, the object’s position is transformed into clip space, for use by the rasterizer. This is the one required output from any vertex shader program.

Given the light’s direction and the surface normal in world space, the pixel shader program computes the surface color:

```c
float4 gooch_PS(vertexOutput IN) : COLOR
{
  float3 Ln = normalize(IN.LightVec);
  float3 Nn = normalize(IN.WorldNormal);
  float ldn = dot(Ln,Nn);
  float mixer = 0.5 * (ldn + 1.0);
  float4 result = lerp(CoolColor, WarmColor, mixer);
  return result;
}
```

The vector Ln is the normalized light direction and Nn the normalized surface normal. By normalizing, the dot product ldn of these two vectors then represents the cosine of the angle between them. We want to linearly interpolate between the cool and warm tones using this value. The function lerp() expects a mixer value between 0 and 1, where 0 means to use the CoolColor, 1 the WarmColor, and values in between to blend the two. Since the cosine of an angle gives a value from [-1, 1], the mixer value transforms this range to [0, 1]. This value then is used to blend the tones and produce a fragment with the proper color. These shaders are functions. An effect file can consist of any number of functions and can include commonly used functions from other effects files.

A pass typically consists of a vertex and pixel (and geometry) shader, along with any state settings needed for the pass. A technique is a set of one or more passes to produce the desired effect. This simple file has one technique, which has one pass:

```c
technique Gooch < string Script = "Pass=p0;"; > {
  pass p0 < string Script = "Draw=geometry;"; > {
    VertexShader = compile vs_2_0 std_vs();
    PixelShader = compile ps_2_a gooch_PS();
  }
}
```

---

6A pass can also have no shaders and control the fixed-function pipeline, in DirectX 9 and earlier.

---

Figure 3.9. A wide range of materials and post-processing effects are possible with programmable shaders. (Images produced by FX Composer 2, courtesy of NVIDIA Corporation.)
ZEnable = true;
ZWriteEnable = true;
ZFunc = LessEqual;
AlphaBlendEnable = false;
}

These state settings force the Z-buffer to be used in the normal way—enabled for reading and writing, and passing if the fragment’s depth is less than or equal to the stored z-depth. Alpha blending is off, as models using this technique are assumed to be opaque. These rules mean that if the fragment’s z-depth is equal to or closer than whatever was stored, the computed fragment color is used to replace the corresponding pixel’s color. In other words, standard Z-buffer usage is used.

A number of techniques can be stored in the same effect file. These techniques are usually variants of the same effect, each targeted at a different shader model (SM 2.0 versus SM 3.0, for example). A huge range of effects are possible. Figure 3.9 gives just a taste of the power of the modern programmable shader pipeline. An effect usually encapsulates related techniques. Various methods have been developed to manage sets of shaders [845, 847, 887, 974, 1271].

We are at the end of the tour of the GPU itself. There is much else the GPU can do, and many ways in which its functions can be used and combined. Relevant theory and algorithms tuned to take advantage of these capabilities are the central subjects of this book. With these basics in place, the focus will move to providing an in-depth understanding of transforms and visual appearance, key elements in the pipeline.

Further Reading and Resources

David Blythe’s paper on DirectX 10 [123] has a good overview of the modern GPU pipeline and the rationale behind its design, as well as references to related articles.

Information about programming vertex and pixel shaders alone can easily fill a book. Our best advice for jumping right in: Visit the ATI [50] and NVIDIA [944] developer websites for information on the latest techniques. Their free FX Composer 2 and RenderMonkey interactive shader design tool suites provide an excellent way to try out shaders, modify them, and see what makes them tick. Sander [1105] provides an implementation of the fixed-function pipeline in HLSL for SM 2.0 capable hardware.

To learn the formal aspects of shader programming takes some work. The OpenGL Shading Language book [1084] picks up where the Red Book [969] leaves off, describing GLSL, the OpenGL programmable shading language. For learning HLSL, the DirectX API continues to evolve with each new release; for related links and books beyond their SDK, see this book’s website (http://www.realtimerendering.com). O’Rourke’s article [974] provides a readable introduction to effects and effective ways to manage shaders. The Cg language provides a layer of abstraction, exporting to many of the major APIs and platforms, while also providing plug-in tools for the major modeling and animation packages. The Sh metaprogramming language is more abstract still, essentially acting as a C++ library that works to map relevant graphics code to the GPU.

For advanced shader techniques, read the GPU Gems and ShaderX series of books as a start. The Game Programming Gems books also have a few relevant articles. The DirectX SDK [261] has many important shader and algorithm samples.