1 Overview of Direct3D 11

The Direct3D 11 rendering API has extended Direct3D in several key areas, while retaining much of the API structure that was introduced with Direct3D 10. This chapter will provide an introduction to Direct3D 11 and will discuss some of these changes to the API. It begins with a brief description of how Direct3D is organized and how it interacts with a GPU installed in your computer. This will provide an idea of what the various software components involved in using Direct3D must do to perform some rendering.

Next is a discussion of the rendering pipeline, with a brief overview of each of the pipeline stages. Then we look at the new additions to the API that allow the GPU to be used for general computation, in addition to its normal rendering duties. Once we have constructed a clear overview of how the pipeline is structured, we will look at all of the different types of resources that can be bound to the pipeline, and the different places where they can be bound. After we understand the pipeline and how it interacts with resources, we will look at the two primary interfaces that are used to work with Direct3D 11—namely, the Device and the Device Context. These two interfaces are the bread and butter of the API and hence warrant a discussion of what they are used for. The discussion of these main interfaces then leads into a walk-through of a basic rendering application that uses Direct3D 11.

This chapter provides an overview of the topics mentioned above. It is intended to introduce a conceptual overview of Direct3D 11, and then to provide a working knowledge of how an application can use it. The first half of this chapter uses few or no actual code samples, while the second half provides a more hands-on introduction to the API. After reading this chapter, you should have a basic understanding of the architecture of Direct3D 11, and of what an application needs to do to manipulate the contents of a window. Each of the concepts discussed in the overview chapter is explored in great detail in the following chapters.

1.1 Direct3D Framework

Direct3D 11 is a native application programming interface (API) which is used to communicate with and control the video hardware of a computer to create a rendered image. The term *native* indicates that the API is designed to be accessed through C/C++, as opposed to by a managed language. So, the most direct way to use the API is from within a C/C++ application. Although there are a number of different native windowing frameworks, we will be working with the Win32 API, simply to reduce the number of additional software technologies needed to get Direct3D 11 up and running.

Even though they are transparent to the application, there are actually a number of different software layers that reside below Direct3D, which together make up the overall graphics architecture used in the Windows client environment. In this section we will discuss how the application and Direct3D fit into this overall architecture and will explore what the other components are used for, to provide a well-rounded idea of what we are actually doing when working with the API.

1.1.1 Graphics Architecture

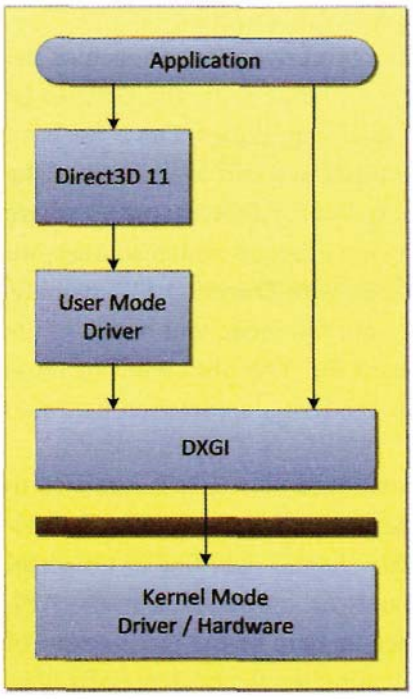

Figure 1.1. The various components of the graphics subsystems used with Direct3D

The general graphics architecture used in Windows is shown in Figure 1.1. In this diagram, we can see that the application sits at the highest level, and primarily interacts with Direct3D. In the next layer down, the Direct3D runtime interacts with the user-mode driver of the video hardware to execute the commands generated by the application. The driver then interacts with the DXGI framework, which is responsible for performing the low level communication with the kernel-mode driver, and for managing the available hardware resources. With such an array of different layers and interfaces, it can be somewhat complex to understand what each of them is responsible for. However, we will briefly walk through each of the layers to gain an insight into what operations each of them perform..

The application is ultimately in control of the content that will end up being presented to the user in her application window. Any two-dimensional or three-dimensional objects, images, text, animations,

or other high level content are provided by the application; they are usually specialized to the particular needs of each application.

Direct3D provides a set of API functions that can be used to convert the high-level content of the application into a format that can be interpreted by the user-mode driver. The content required by the application must be formatted according to the rules of Direct3D. This includes the actual data formatting, as well as the required sequencing of function calls. As long as the application follows the required Direct3D usage, it can be sure that it is operating within the graphics architecture.

Once Direct3D has received a series of function calls from the application, it interacts with the user-mode driver to produce a list of commands that are executed by the GPU. The results of these actions are relayed to the DXGI interface to manipulate the contents of hardware resources. DXGI handles the lower-level hardware interfaces, such as providing an "adapter" interface for each video card that is installed in a computer. It is DXGI that also provides access to what are called *swap chains,* which encapsulate the actual buffers that can be used to present the rendered contents to a window.

Although this is not shown in Figure 1.1, it is quite likely that multiple applications will simultaneously be using the Direct3D runtime. This is one of the benefits of the Windows display driver model (WDDM), that the video resources are virtualized and shared among all applications. Applications do not have to worry about gaining access to the GPU, since it is available to all applications simultaneously. Of course, when a user is performing a very graphically intensive operation, like playing a high end game, there are likely to be fewer applications using the GPU, and hence more of its computational power can be directed to the game. This is a departure from the Windows XP driver model (XPDM) which did not explicitly virtualize access to the GPU.

1.1.2 Benefits of Abstraction

Since an application interfaces with the Direct3D API instead of directly to a driver, it only needs to learn the rales of interfacing with one API to produce the desired output instead of trying to adapt to all of the various devices that a user might have installed. This shifts the burden of creating a uniform rendering system with which to interact to the video card manufacturers—they must adhere to the Direct3D standard, as opposed to the application trying to implement its own "standard" rendering system.

The video system of modern computers typically contains specialized hardware that is designed to perform graphics operations. However, it is possible that an entry-level computer doesn't have Direct3D-capable hardware installed, or that it only has a subset of the required functionality for a particular application. Fortunately, Direct3D 11 provides a software driver implementation that can be used in the absence of Direct3D-capable

hardware (up to the Direct3D 10.1 feature level). Whether such hardware is present or not is conveniently abstracted from the developer by Direct3D.

Both of these situations greatly benefit the developer. As long as an application can properly interact with Direct3D 11, it will have access to a standardized rendering system, as well as support for whatever devices are present in the system, regardless of if they are hardware- or software-based devices.

1.2 Pipeline

Arguably, the most important concept required for understanding Direct3D 11 is a clear understanding of its pipeline configuration. The Direct3D 11 specification defines a conceptual processing pipeline to perform rendering computations, and ultimately, to produce a rendered image. The term *pipeline* is used since data "flows" into the pipeline, is processed in some way at each stage throughout the pipeline, and then "flows" out of the pipeline. Over the past few iterations of the API, several changes have been made to the pipeline to both simplify it and add additional functionality to it. The pipeline concept itself is composed of a number of stages, which each perform distinctly different operations. Data passed into each stage is processed in some fashion before being passed on to the next stage. Each individual stage provides a unique configurable set of parameters that can be controlled by the developer to customize what processing is performed. Some of the stages are referred to *as fixed-function* stages, which offer a limited customization capability. These stages typically expose a state object, which can be bound with the desired new configuration as needed.

The other stages are referred to as *programmable* stages, which provide a significantly broader configuration capacity by allowing custom programs to be executed within them. These programs are typically referred to as *shader* programs, and the stages are referred to as *programmable shader stages.* Instead of performing a fixed set of operations on the data that is passed into it, these programmable shader stages can implement a wide range of different functions.

It is these shader programs that provide the most flexibility for implementing custom rendering and computational algorithms. Each programmable shader stage provides a common feature set, called the *common shader core,* to provide some level of uniformity in its abilities. These features can be thought of as a toolbox that is available to all of the programmable shader stages. Operations like texture sampling, arithmetic instruction sets, and so on are all common among all of the programmable stages. However, each stage also provides its own unique functionality, either in its instruction set or in its input/output semantics, to allow distinct types of operations to be performed.

From a high level, there are currently two general pipeline paradigms—one which is used for rendering, and one which is used for computation. Strictly speaking, the distinction between these two configurations is somewhat loose, since they can both be used for the other purpose. However, the distinction exists, and there is a clear difference in the two pipelines' capabilities. We will discuss both pipelines, beginning with the rendering pipeline.

1.2.1 Rendering Pipeline

The *rendering pipeline* is the origin from which the modern GPU has grown. The initial graphics accelerators provided hardware vertex transformation to speed up 3D applications. From there, each new generation of hardware provided additional capabilities to perform ever more complex rendering. Today, we have quite a complex pipeline, with significant flexibility to perform nearly any algorithm in hardware. Books are published regularly describing new techniques for using the latest hardware such as (Engel, 2009) and(Nguyen, 2008). The level of advances in real-time rendering has really been quite staggering over the last 5-10 years.

The rendering pipeline is intended to take a set of 3D object descriptions and convert it into an image format that is suitable for presentation in the output window of an application. Before diving into the details of each of the individual stages, let's take a closer look at the complete Direct3D 11 rendering pipeline. Figure 1.2 shows a block diagram of the pipeline, which we will see many times throughout the remainder of the book, in several different formats.

Here, the two different types of pipeline stages, fixed-function and programmable, are depicted with different colors. The fixed-function stages are shown with a green background, and the programmable stages with a blue background. Each stage defines its own required input data format and also defines the output data format that it will produce when executed. We will step through the pipeline and briefly discuss each stage, and its intended purpose.

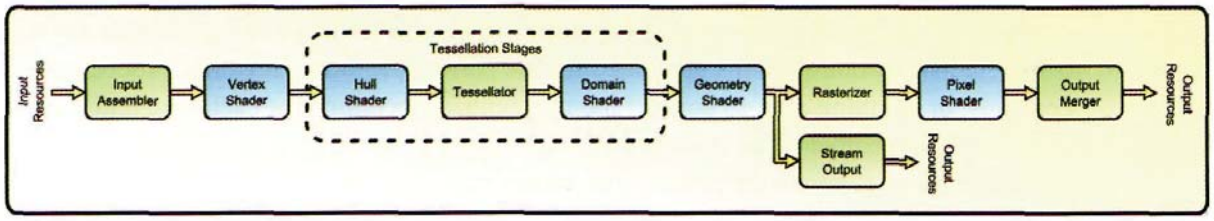

Figure 1.2. The complete Direct3D 11 rendering pipeline.

The entry point to the rendering pipeline is the input assembler stage. This stage is responsible for reading input data from resources and then "assembling" vertices for use later in the pipeline. This allows for multiple vertex buffers to be used and also provides the ability to use *instanced rendering* (described in more detail in Chapter 3). In addition, the connectivity of the vertices is also determined, based on the input resources and the desired rendering configuration. The assembled vertices and primitive connectivity information are then passed down the pipeline.

The vertex shader stage reads the assembled vertex data from the input assembler stage and processes a single vertex at a time. This is the first programmable stage in the pipeline, and it applies the current vertex shader program to each input vertex. It cannot create or destroy vertices; it can only process the vertices that are given to it. In addition, every vertex is processed in isolation—the information from one vertex shader invocation is never accessible in another invocation. The primary responsibility of the vertex shader used to be to project the vertex positions into clip space, but the addition of the tessellation stages (discussed next) has somewhat changed this. In general, any operation that must be performed on every vertex of the input model should be performed in the vertex shader.

The next three stages are recent additions to the pipeline to accommodate hardware tessellation, and they must all be used together. The hull shader stage receives primitives from the vertex shader and is responsible for two different actions. First, the hull shader provides a function that is only run once per primitive to determine a set of tessellation factors. These factors are used by the tessellator stage to know how finely to tessellate or split up the current primitive. The second action that the hull shader stage must perform is executed once for each control point in the desired output control patch configuration. In essence, it must create the control points that will be later be used by the domain shader stage to create the actual tessellated vertices that will eventually be used in rendering.

When the tessellation stage receives its data from the hull shader, it uses one of several algorithms to determine an appropriate sampling pattern for the current primitive type. Depending on the tessellation factors (from the hull shader), as well as its own configuration (which is actually specified in the hull shader as well), it will determine which points in the current primitive need to be sampled in order to tessellate the input primitive into smaller parts. The output of the tessellation stage is actually a set of barycentric coordinates that are passed along to the domain shader.

The domain shader takes these barycentric coordinates, in addition to the control points produced by the hull shader, and uses them to create new vertices. It can use the complete list of control points generated for the current primitive, textures, procedural algorithms, or anything else, to convert the barycentric "location" for each tessellated point into the output geometry that is passed on to the next stage in the pipeline. This flexibility in deciding how to generate the resulting geometry from the tessellator stages' amplified output provides significant freedom to implement many different types of tessellation algorithms.

The geometry shader stage resides in the next pipeline location. As its name implies, the geometry shader operates on a geometric level. In practice, this means that it operates

on complete geometric primitives, and also produces geometric primitives. This stage can both add and remove data elements from the pipeline, which allows for some interesting, non-traditional uses. In addition, it can take one type of geometry as input and generate a different type of geometry as output. This allows the conversion of single vertices into complete triangles, or even multiple triangles. The geometry shader is also the conduit through which processed geometry can be streamed out of the pipeline into a buffer resource. This is accomplished in the stream output stage.

After the geometry is sent out of the geometry shader, it has completed the portion of the pipeline that operates at the geometric level. From this point on, the geometry is rasterized and dealt with at the fragment level. *A fragment* is essentially a group of data that corresponds to a pixel in a render target, and that can potentially be used to update the current value of that pixel if it makes its way through the rest of the pipeline. The generation of fragment-level data begins with the fixed-function rasterizer stage. The rasterizer produces fragments from the geometric data passed to it, by determining which pixels of a render target are covered by the geometry. Each fragment receives interpolated versions of all of the per-vertex attributes, to provide the information needed for further processing later in the pipeline. In addition, the rasterizer produces a depth value for each fragment, which will later be used for visibility testing in the output merger stage.

Once a fragment has been generated, the pixel shader is invoked to process it. The pixel shader is required to generate a color value output for each of the render target outputs bound to the pipeline. To accomplish this, it may sample textures or perform computations on the incoming fragment attributes. In addition, it can also override the depth value produced by the rasterizer stage, to allow for specialized algorithms to be implemented in the pixel shader stage.

After the pixel shader has finished its work, its output is passed to the output merger stage. It must correctly "merge" the pixel shader output with the bound depth/stencil and render target resources. This includes performing depth and stencil tests, performing the blending function, and finally, performing the actual writing of the output to the appropriate resources.

This has been a brief high-level overview of each of the pipeline stages, but there are many more details to be considered when using each of them. We will dive much deeper into all of the available configurations and capabilities of each stage in the Chapter 3, "The Rendering Pipeline." Until we reach that point, keep this overview of the pipeline in mind as we continue our overview of the API.

1.2.2 Computation Pipeline

With all of the other new pipeline stages that came along with Direct3D 11, there is one additional stage that has not yet been discussed—the compute shader stage. This stage is intended to perform computation outside of the traditional rendering paradigm, and therefore

is considered to execute separately from the traditional rendering pipeline. In this way, it implements a single-stage pipeline devoted to general purpose computation. The general trend of using the GPU for purposes other than rendering has been incorporated directly into the Direct3D 11 API, and is manifested in the compute shader.

Several features have been provided to the compute shader that facilitate a flexible processing environment for implementing more general algorithms. The first new functionality is the addition of a structured threading model that gives the developer significant freedom in using the available parallelism of the GPU to implement highly parallel algorithms. Previously, the invocation of a shader program was restricted to how an input was processed by a particular stage (for example, the pixel shader program was invoked once for each fragment generated), and the developer didn't have direct control over how the threads were used. This is no longer the case with the compute shader stage.

The second new functionality that makes the computer shader stage more flexible is the ability to declare and use a "Group Shared Memory" block. This memory block is then accessible to all of the threads within a thread group. This allows communication between threads during execution, which has typically not been possible with the rendering pipeline. With communication possible, there is a potential for significant efficiency improvements by sharing loaded data or intermediate calculations.

Finally, the concept of random read *and* write access to resources has been introduced (which we will discuss further in the "Resources" section of this chapter). This represents a significant change from the shader stages we have already discussed, which only allow a resource to be bound as either an input or as an output. The ability to read and write to a complete resource provides another avenue for communication between threads.

These three new concepts, when taken together, introduce a very flexible general-purpose processing stage that can be used for both rendering calculations, as well as for general purpose computations. We will take a much closer look at the compute shader in Chapter 5, "The Computation Pipeline," and there are several sample algorithms throughout the second half of the book that make extensive use of the compute shader.

1.3 Resources

The rendering and computation pipelines that we discussed in the previous section are two of the major concepts that one must understand when learning about the Direct3D 11 API. The next major concept that we need to understand is the resources that are connected at various locations throughout the pipeline. If you consider the rendering pipeline to be an automobile assembly line, the resources would be all of the components that come into the line that are put together to create the automobile. When used without resources, the pipeline can't do anything, and vice versa.

There is a wide diversity of different resources at our disposal when considering how to structure an algorithm, and each type has different use cases. In general, resources are split into two groups—textures and buffers. *Textures* are roughly split by their dimension, and can be created to have one-dimensional, two-dimensional, and three-dimensional forms. *Buffers* are somewhat more uniform and are always considered one-dimensional (although in some cases they are actually O-dimensional, such as a single data point). Even so, there is a good variety of different buffer types, including vertex and index buffers, constant buffers, structured buffers, append and consume buffers, and byte address buffers.

Each of these resource types (both textures and buffers) provides a different usage pattern, and we will explore all of their variants in more detail in Chapter 3, "The Rendering Pipeline." However, there are some common overall concepts regarding the use of resources with the pipeline which we can discuss now. In general, there are two different domains for using resources—one is from C/C++, and the other is from HLSL. C/C++ is primarily concerned with creating resources and binding/unbinding them to the desired locations. HLSL is more concerned with actually manipulating and using the contents of the resources. We will see this distinction throughout the book.

A resource must be bound to the pipeline in order to be used. When a resource is bound to the pipeline, it can be used as a data input, an output, or both. This primarily depends on where and how it is bound to the pipeline. For example, a vertex buffer bound as an input to the input assembler is clearly an input, and a 2D texture bound to the output merger stage as a render target is clearly an output. However, what if we wanted to use the contents of the output render target in a subsequent rendering pass? It could also be bound to a pixel shader as a texture resource to be sampled during the next rendering operation.

To help the runtime (and the developer!) determine the intended use of a resource, the API provides the concept of *resource views.* When a particular resource can be bound to the pipeline in several different types of locations in the pipeline, it must be bound with a resource view that specifies how it will be used. There are four different types of resource views: a *render target view,* a *depth stencil view,* a *shader resource view,* and an *unordered access view.* The first two are used for binding render and depth targets to the pipeline, respectively. The shader resource view is somewhat different, and allows for resources to be bound for reading in any of the programmable shader stages. The unordered access view is again somewhat different and allows a resource to be bound for simultaneous random read and write access, but is only available for use in the compute shader and pixel shader stages. There are still other resource binding types that don't require a resource view for binding to the pipeline. These are typically for resources that are created for a single purpose, such as vertex buffers, index buffers, or constant buffers, and the resource usage is not ambiguous.

With so many options and configuration possibilities, it can seem somewhat daunting when you are first learning about what each resource type can do and what it can be used for. However, as you will see later in the book, this configurability of the resources provides significant freedom and power for designing and implementing new algorithms.

1.4 Interfacing with Direct3D

With a high level overview of Direct3D 11 complete, we now turn our attention to how an application can interface with and use the API. This begins with a look into the two primary interfaces used by the application—the device and the device context. After this, we will walk through a simple application to see what it must do in order to use Direct3D 11.

1.4.1 Devices

The primary interfaces that must be understood when using Direct3D 11 are the *device* and the *device context.* These two interfaces split the overall responsibility for managing the functionality available in the Direct3D 11 API. When the resources mentioned above are being created and interfaced with, the device interface is used. When the pipeline or a resource is being manipulated, the device context is used.

The *ID3DllDevice* interface provides many methods for creating shader program objects, resources, state objects, and query objects (among others). It also provides methods for checking the availability of some hardware features, along with many diagnostic- and debugging-related methods. In general, the device can be thought of as the provider of the various resources that will be used in an application. We will see later in this chapter how to initialize and configure a device before using it. Due to the size of this interface and the frequency with which it is used, we will discuss the device methods when their particular subject area is relevant. For a direct list of the available methods, the DXSDK documentation provides a complete linked list for every method.

In addition to serving as the resource provider for Direct3D 11, the device also encapsulates the concept of *feature levels.* The use of feature levels is a technique for allowing an application to use the Direct3D 11 API and runtime on hardware that implements an older set of functionality, such as the Direct3D 10 GPUs. This allows applications to utilize a single rendering API but still target several generations of hardware. We will see later in this chapter how to use this great feature of the API.

1.4.2 Contexts

While the device is used to *create* the various resources used by the pipeline, to actually *use* those resources and manipulate the pipeline itself, we will use a *device context.* Device contexts are used to bind the created resources, shader objects, and state objects to the pipeline. They are also used to control the execution of rendering and computation pipeline invocations. In addition, they provide methods for manipulating the resources created by the device. In general, the device context can be thought of as the consumer of the resources produced by the device, which serves as the interface for working with the pipeline.

The device context is implemented in the *ID3DllDeviceContext* interface. To help in the introduction of multithreaded rendering support, two different flavors of contexts are provided. The first is referred to as an *immediate context,* and the second is called a *deferred context.* While both of these types implement the same device context interface, their usage has very different semantics. We will discuss the concepts behind these two contexts below.

Immediate Contexts

The immediate context is more or less a direct link to the pipeline. When a method is called from this context, it is submitted immediately by the Direct3D 11 runtime for execution in the driver. Only a single immediate context is allowed, and it is created at the same time that the device is created. This context can be seen as the interface for directly interacting with all of the components of the pipeline. This context must be used as the main rendering thread, and serves as the primary interface to the GPU.

Deferred Contexts

The deferred context is a secondary type of context that provides a thread-safe mechanism for recording a series of commands from secondary threads other than the main rendering thread. This is used to produce a command list object which can be "played" back on the immediate context at a later time. Allowing the command list to be generated from other threads provides some potential that a performance improvement could be found on multi-core CPU systems. In addition, Direct3D 11 also allows asynchronous resource creation to be carried out on multiple threads, which provides the ability to simplify multithreaded loading situations. These are important concepts when considering the current and future PC hardware environment, in which we can expect more and more CPU cores to be available in a given PC.

The topic of multithreaded rendering is discussed in more detail in Chapter 7, "Multithreaded Rendering." The topic is itself worthy of a complete book, but in the interests of providing a well-rounded discussion of Direct3D 11, we have attempted to provide design-level information about how and when to use multithreading, rather than trying to explain all of the details of multithreaded programming in general.

1.5 Getting Started

This portion of the chapter is devoted to giving the reader a crash course in how to get a basic Direct3D 11 application running. The description provided here introduces a number of basic concepts used to interact with Direct3D 11 and introduces the basic program

flow used in a generic application. This is intended to be a basic introduction with a focus on what actions are needed for using Direct3D 11, and hence it will not explain all of the details of Win32 programming. At the end of this section, we will discuss the application framework that will be used for the sample applications provided with this book, which abstracts most of the direct Win32 interactions away from the developer.

1.5.1 Interacting with Direct3D 11

Before getting to the actual application details, we first need to cover a few basic Direct3D 11 concepts. This includes the COM based framework in which Direct3D resides, as well as understanding how to call, and evaluate the success of the various API functions that an application must use.

COM-Based Interfaces

Direct3D 11 is implemented as a series of component object model (COM) interfaces in the same way that the previous versions of Direct3D have been. If you are not familiar with COM, it may seem like an unnecessary complication to the API. However, once you understand some of the design implications of using COM, it becomes clear why it has been chosen for implementing Direct3D 11. We won't dig too deeply into COM, but we will cover a few of the most visible details that an application will likely use. COM provides an object model that defines a basic set of interface methods that all COM objects must implement. This includes methods for performing reference counting of the COM objects, as well as methods that allow an object to be inspected at runtime to discover which interfaces it implements.

Reference counting. An application will primarily be interested in the reference counting behavior of the COM objects, since it is used to manage their lifetime. When a Direct3D-based COM object is created with by a function, a reference to that object is returned, typically in a pointer provided by the application, as an argument to the function call. Instead of directly creating an instance of a class with the "new" operator, the objects are always created through a function or an interface method and returned in this way.

When the object reference is returned, it will usually have a reference count of 1. When the application is finished using the object, it will call the object's Release method, which indicates that the object's reference count should be decremented. This is performed instead of directly freeing the object with the "delete" operator. After calling the Release method, the application should erase the pointer reference and treat it as if it were directly deleted. As long as the application accurately releases all of its references to the objects that it creates, the COM framework will manage the creation and destruction of the objects.

This usage allows both the application and the Direct3D 11 runtime to share objects with multiple references. When a resource is bound to the pipeline as a render target, it is referenced by the runtime to ensure that the object is not deleted while it is being used. Similarly, if both the application and the runtime have a reference to an object, and a method call eliminates the runtime reference, the application can be sure that the object will not be destroyed until it releases its reference, as well. This reference-counting object management holds true for any Direct3D 11 object that inherits from the IUnknown interface, and it must be followed to ensure that no objects are left unreleased after an application is terminated.

Interface querying. As mentioned above, the IUnknown interface also provides some methods for querying an object to find additional interfaces that it implements. This is performed with the *IUnknown::QueryInterface()* method, which takes a universally unique identifier (UUID) that identifies the desired interface, and then a pointer to a pointer, which will be filled with an appropriate object reference if the object implements the requested interface. If such a reference is successfully returned in the method call, the reference count of the object is increased and should be managed in the same way as any other object.

In Direct3D, this mechanism is used primarily in conjunction with the ID3D1 IDevice interface. There are several different situations in which additional device interfaces are requested, such as acquiring the debug interface of the device or acquiring the DXGI device interface from the Direct3D device. We will occasionally see this method in action in this text, and the process will be noted as it arises.

Some other properties of COM objects are not used as frequently with Direct3D 11, such as the language independence of the binary interface, or the fact that objects can be instantiated and used from a remote computer. These uses are beyond the scope of this book, but are still interesting to know about. Further information about COM can be found on the MSDN website, and there are also many additional resources available online, due to the longevity of the technology.

Interpreting API Call Results

Direct3D 11 also implements a standard way of handling return codes from functions and methods. In general, the success or failure of a method is indicated by a return code, which is provided to the application as the returned value of the function call. This return code is implemented as an HRESULT value, which is really just a long integer variable.

The list of possible return values can be found in the DXSDK documentation, and additional return codes may added between releases. In some situations, these return codes do not clearly indicate the success or failure of a method, and often the interpretation of the codes depends on the context in which it is returned. To simplify testing of these return codes, a simple macro can be used to test if a method has failed or not. Listing 1.1 shows how the FAILED macro is used.

```
HRESULT hr = m_pDevice->CreateBlendState( &Config, SpState );

if ( FAILED( hr ) )
{
    Log::Get().Write( L"Failed to create blend state!" );
    return( -1 );
}
```

Listing 1.1. A sample usage of the FAILED macro.

In general, any case in which this macro returns a true value should be treated as an error and handled accordingly. If the macro returns a false value, the method has succeeded and should be considered not to have produced an error. This makes testing the result of a method invocation much simpler and can be handled in a relatively simple way.

1.5.2 Application Requirements

With the basics of COM and HRESULT values behind us, we can move on to examine what exactly a Win32 application must do to use Direct3D 11. We will begin with an overview of the complete responsibilities of the application, which is then followed by a more detailed look at each of the processes that it must implement.

Program Flow Overview

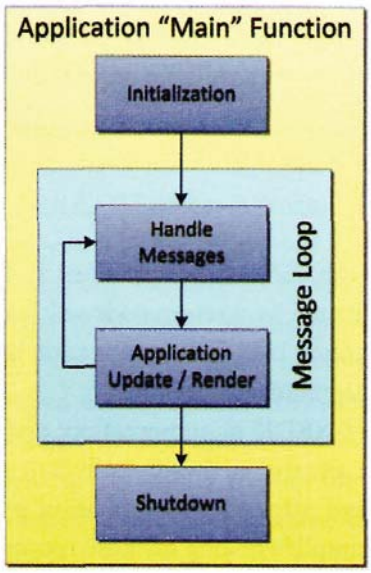

Figure 1.3. The standard operations performed in a Win32 application.

The Direct3D 11 application lifecycle follows quite a similar sequence to most other Win32 applications. The block diagram for this standard sequence is shown in Figure 1.3.

From Figure 1.3, we can see that the application begins with a one-time initialization action. This is used to obtain references to the device and device context, as well as to create any resources that will be used during the runtime of the application. The resources will vary widely from application to application, while the device and device context creation are normally the same across multiple applications.

After the application has been initialized, it enters the *message loop*. This loop consists of two phases. First, all pending Windows messages are handled with the application's message handler. After all available messages have been processed, the actual application

workload can be performed. For our rendering application, this consists of clearing the render targets, then performing the desired rendering operations, and finally presenting the results to the window's client area. Each time these two processes are repeated represents one frame rendering. This will continue until the user requests the termination of the application.

After the application exits the message loop, it performs a one-time shutdown routine. In this routine, all of the resources that have been created are released. After they have been released, the device context, and the device itself, are released as well. At this point, as long as all of the reference counts of the Direct3D 11 objects have been properly managed, the application will have completely released all of its references to all objects. After the shutdown process has completed, the application will end by exiting its main function.

Now that we have seen each of the steps that are performed, we can take a closer look at how each of these steps involves Direct3D 11 and see what the actual code looks like for using the API.

Application Initialization

The application initialization procedure begins with the application creating the Win32 window that will present the results of our rendering operations. Basic Win32 code for creating a window has been presented many, many times in numerous books and tutorials and won't be repeated here. However, the sample program rendering framework also includes this type of code, and a quick reference to this code within the source distribution can be found in the class files for the Win32RenderWindow class. For the purposes of this discussion, we will assume that the window has been properly registered and created, and that a window handle is available for it.

Device and context creation. After the window has been created, the next step is to obtain a reference to the device. This process is performed with either the D3DllCreateDevice() or the D3DllCreateDeviceAndSwapChain() functions. For this example, we will use the latter function simply because it creates a swap chain from within the same function, in addition to creating the device and device context.

The prototype for the function is shown in Listing 1.2. We will step through the parameters to this function to gain an understanding of what is needed to create a device.

```
HRESULT    D3DllCreateDeviceAndSwapChain(
        IDXGIAdapter   *pAdapter,
        D3D_DRIVER_TYPE  Driver-Type,
        HMODULE  Software,
        UINT  Flags,
        const  D3D_FEATURE_LEVEL  *pFeatureLevels,
        UINT  FeatureLevels,
        UINT  SDKVersion,
```

```
        const DXGI_SWAP_CHAIN_DESC *pSwapChainDesc,
        IDXGISwapChain **ppSwapChain,
        ID3DllDevice **ppDevice,
        D3D_FEATURE_LEVEL *pFeatureLevel,
        ID3DllDeviceContext **ppImmediateContext
);
```

Listing 1.2. The D3DllCreateDeviceAndSwapChain function prototype.

The first parameter, pAdapter, is a pointer to the graphics adapter that the device should be created for. This parameter can be passed as NULL to use the default adapter. The second parameter is vitally important to properly configure. The DriverType parameter specifies which type of driver will be used for the device created with this function call. The available options are shown in Listing 1.3.

```
enum D3D_DRIVER_TYPE {
    D3D_DRIVER_TYPE_UNKN0WN,
    D3D_DRIVER_TYPE_HARDWARE,
    D3D_DRIVER_TYPE_REFERENCE,
    D3D_DRIVER_TYPE_NULL,
    D3D_DRIVER_TYPE_S0FTWARE,
    D3D_DRIVER_TYPE_WARP
}
```

Listing 1.3. The D3D_DRIVER_TYPE enumeration, showing the types of drivers that can be created.

As you can see, it is possible to use hardware based drivers that interact with actual hardware devices. This is the configuration that will be used in the majority of cases, but there are also several other driver types that must be considered. The next driver type is the *reference driver*. This is a software driver that fully implements the complete Direct3D 11 specification, but since it runs in software, it is not very fast. This driver type is typically used to test if a particular set of rendering commands produces the same results with both the hardware driver and the reference drivers. The null driver is only used in testing and does not implement any rendering operations. The software driver type allows for a custom software driver to be used. This is not a common option, since it requires a complete third-party software driver to be provided on the target machine. The final option is perhaps one of the most interesting. The *WARP device* is a software renderer that is optimized for rendering speed and that will use any special features of a CPU, such as multiple cores or SIMD instructions. This allows an application to have a moderately fast software driver available for use on all target machines, making the deployable market significantly larger for many application types. Unfortunately, this device type only supports the functional-

ity provided up to the Direct3D 10.1 feature level,[1] which means that it can't be used to implement most of the techniques discussed in this book!

After the desired device type is chosen, the next parameter allows the application to provide a handle to the software driver DLL in situations when a software driver type is selected. If a non-software device is chosen, this parameter can be set to NULL, which will usually be the case. Next up is the Flags parameter, which allows for a number of special features to be enabled when the device is created. The available flags are shown in Listing 1.4.

```
enum D3D11_CREATE_DEVICE_FLAG {
    D3D11_CREATE_DEVICE_SINGLETHREADED,
    D3D11_CREATE_DEVICE_DEBUG,
    D3D11_CREATE_DEVICE_SWITCH_TO_REF,
    D3D11_CREATE_DEVICE_PREVENT_INTERNAL_THREADING_OPTIMIZATIONS,
    D3D11_CREATE_DEVICE_BGRA_SUPPORT
}
```

Listing 1.4. The D3D11CREATEDEVICE FLAG enumeration.

The first flag indicates that the device should be created for single-threaded use. If this flag is not present, the default behavior is to allow multithreaded use of the device. The second flag indicates if the debug layer of the device should be created. If this flag is set, the device is created such that it also implements the ID3DllDebug interface. This interface is used for various debugging operations and is retrieved using the COM query interface techniques. It also causes complete error/warning messages to be output at run-time and enables memory leak detection as well. The third flag is unsupported for use in Direct3D 11, so we won't discuss it in any great detail. The fourth flag is used to disable multithreaded *optimizations* within the device while still allowing multithreaded use. This would likely decrease performance, but it could allow for simpler debugging and/or profiling characteristics. The final flag is used to create a device that can interoperate with the Direct2D API. We don't cover Direct2D in this book, but it is still important to know that this feature is available.

The following two parameters also offer a very interesting set of capabilities for creating a device. The first is a pointer to an array of D3D_FEATURE_LEVEL values. The concept of *a feature level* is the replacement for the old CAPS bits from older versions of Direct3D. CAPS bits were a name for the myriad number of options that a GPU manufacturer could choose to support or not, and the application was responsible for checking if a particular feature was available or not before trying to use it.

[1] The concept of feature levels is discussed in the following pages.

With each new generation of hardware, the number of available CAPS bits was becoming increasingly unmanageable. Instead of requiring the application to parse through all of these options, the Direct3D 11 API groups implementations into categories called *feature levels*. These feature levels provide a more coarse representation of the available features for a given GPU and driver, but using them provides a significantly simpler process to determine what capabilities the GPU supports. The available feature levels are shown in Listing 1.5.

```
enum D3D_FEATURE_LEVEL {
    D3D_FEATURE_LEVEL_9_1,
    D3D_FEATURE_LEVEL_9_2,
    D3D_FEATURE_LEVEL_9_3,
    D3D_FEATURE_LEVEL_10_0,
    D3D_FEATURE_LEVEL_10_1,
    D3D_FEATURE_LEVEL_11_0
}
```

Listing 1.5. The D3D_FEATURE_LEVEL enumeration.

As you can see, there is one feature level for each minor revision that was released for Direct3D since version 9. Each feature level is a strict superset of the previous feature level, meaning that a higher-level GPU will support the lower feature levels if an application requests them. For example, if a particular application only requires features from the 10.0 feature level, it will be able to run on a wider range of hardware than an application that requires the 11.0 feature level. The application passes an array of feature levels that it would like to use in the pFeatureLevels parameter, with the array elements arranged in the order of preference. The number of elements in the array is passed in the FeatureLevels parameter to ensure that the function doesn't exceed the array size while processing the input array. The device creation method will try to create a device with each feature level, beginning with the first element in the array. If that feature level is supported for the requested driver type, the device is created. If not, then the process continues to the next feature level. If no supported feature levels are present in the array, the method returns a failure code.

We have two input parameters left in the device creation function. The SDKVersion parameter is simply supplied with the D3D11_SDK_VERSION macro. This is a defined value that changes with each new release of the DirectX SDK. However, since the value is automatically updated with a new SDK installation, the developer does not need to make any changes when converting to the new SDK.

The final input parameter is a pointer to a swap chain description. As described earlier in this chapter, a *swap chain* is an object created by DXGI that is used to manage the contents of a window's client area. The swap chain object defines all of the lower-level options that

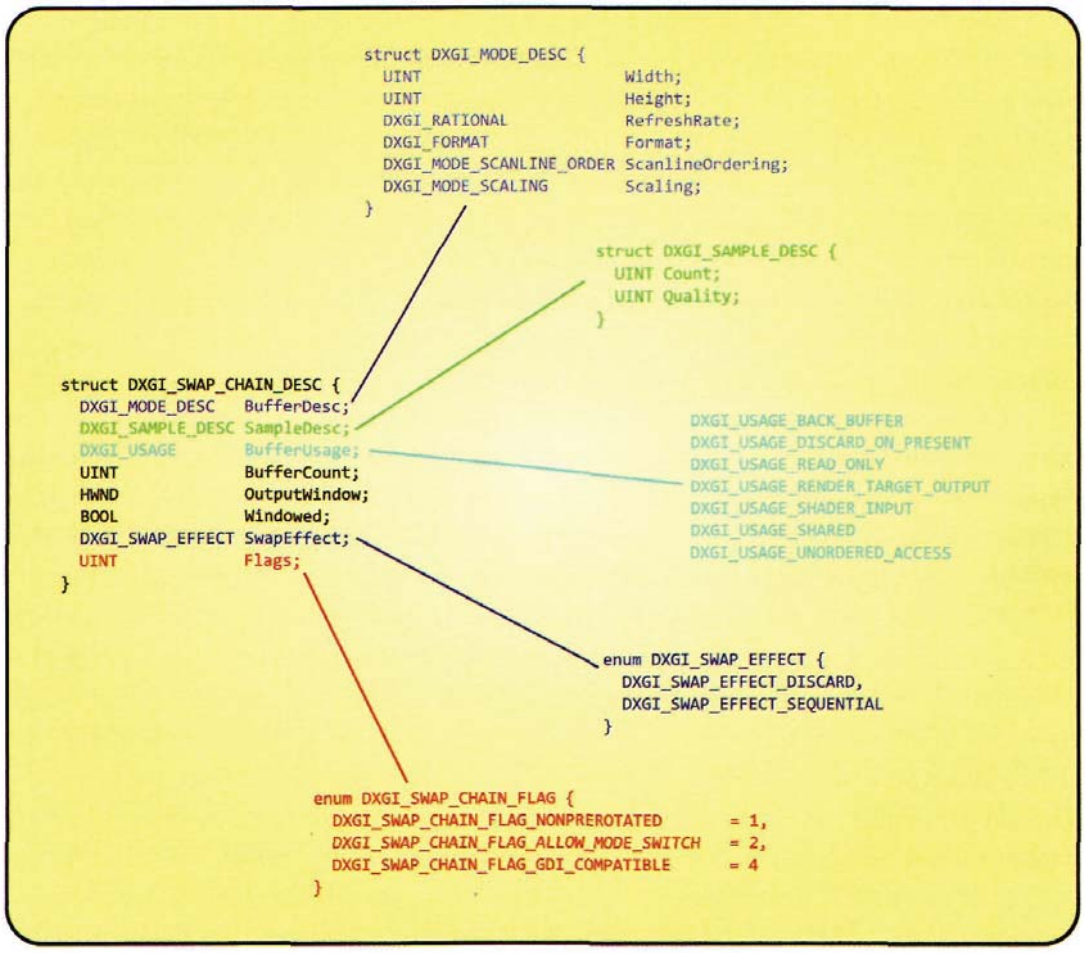

Figure 1.4. A graphical representation of the DXGI_SWAP_CHAIN_DESC structure and its members.

are needed to have DXGI properly manage a window's contents. We specify out desired swap chain options in the pSwapChainDesc parameter by filling in a DXGI_SWAP_CHAIN_DESC structure. This structure is shown in Figure 1.4, with the available options for its members.

These structure members each control various aspects of the swap chain's behavior. The BufferDesc parameter provides details about the buffer that will hold the windows contents. Its Width and Height parameters give the size of the buffer in pixels. The RefreshRate parameter is a structure that provides a numerator and denominator to define the refresh rate of the contents of the window. The Format parameter is a member of the DXGI_FORMAT enumeration, which provides all of the available formats for a texture resource to use. The ScanlineOrdering parameter defines the raster techniques for image generation. And finally, the Scaling parameter specifies how the buffer will be applied to the window's client area, allowing either scaled or centered presentation. Many of these options are chosen once and then simply reused as a default behavior.

After the buffer mode description structure, we next specify the options for multisample anti-aliasing (MSAA). This requires that we select the number of samples and the quality of their sampling pattern. This functionality is discussed in more detail throughout Chapter 2 and Chapter 3, but for now we will disable it by specifying a single sample per pixel, with a quality of zero.

Next, we indicate the intended usage of the swap chain, which will be used in our case as a render target output. The BufferCount parameter is used to indicate how many buffers will be used within the swap chain. Having multiple buffers allows DXGI to use one of the buffers to display the rendered image, while at the same time, Direct3D can be generating the next frame in a secondary buffer. This allows for smoother animation in the window. It is common to use two buffers for this purpose, but additional buffers can be used if necessary.

The next two parameters configure the connection to the application window. The OutputWindow parameter is simply the window handle of the window that will receive the swap chain's contents. The next parameter, Windowed, indicates what it sounds like—if the window will appear as a window on the desktop, or if it will occupy the complete screen. This will vary from application to application, but we will stick with windowed mode for now.

The SwapEffect parameter configures what is done to the buffer contents after they are presented to the window. This flag is normally set to discard the buffer contents after presentation, but it can be changed as needed. The final parameter needed to create a swap chain is a combination of a number of miscellaneous flags that can further configure the buffer, and how it is used by DXGI. In general, these are special usage flags that can allow special operations by the application, such as handling monitor rotation, switching between windowed and full screen modes, and allowing GDI access to the buffers. We won't be using any of these flags in our example. After all of the parameters are filled in, we also supply pointers to all of the objects that we expect to receive back. These are the swap chain, the device, the created device's feature level, and the immediate device context.

It is important to always check the return value from API calls, to ensure that the function has succeeded. All of the returned items except for the feature level are COM objects and thus must have their reference counts managed accordingly. After we have the swap chain interface, we must obtain the texture resource interface from it, for use in the rendering pipeline. This is done with the IDXGISwapChain: :GetBuffer() method, which works similarly to the COM query interface method described above. The difference is that the index of the buffer we are trying to acquire is passed as the first parameter. The acquisition of the texture interface is shown in Listing 1.6.

```
ID3D11Texture2D* pSwapChainBuffer = 0;
hr = pSwapChain->GetBuffer( 0,
                            __uuidof( ID3D11Texture2D ),
                            (void **)&pSwapChainBuffer );
```

Listing 1.6. Acquiring the texture interface from a swap chain.

In addition, if we created the device with the debug layer installed, we need to acquire the debug interface from the device itself in the same manner. This is shown in Listing 1.7.

```
m_pDebugger = 0;
hr = pDevice->QueryInterface(__uuidof( ID3DllDebug ), (void **)&pDebugger );
```

Listing 1.7. Acquiring the debug interface from the device using the query interface method.

After these interfaces have been acquired, we can consider Direct3D 11 to be initialized and can move on to the next application phase.

Resource creation. After the application has initialized and acquired the device and device context and also has a swap chain to render into, it should then create any resources, shader objects, and state objects that will be used during the rendering portion of the application. It is the best practice to acquire as many of these objects at startup time as possible, since acquiring them during rendering operations can introduce momentary delays in rendering. These glitches can be mostly alleviated with multithreaded rendering, but acquiring the resource during initialization eliminates the problem completely.

Since we haven't yet examined the available resources, or the components of the rendering pipeline, it would not be useful to describe the creation of these objects here. Both of these topics have an entire chapter devoted to them (Chapters 2 and 3), so we will not attempt to provide a glossed-over description here. However, in order to use the texture interface that we acquired from the swap chain, we will need an object called a *resource view* to bind the texture as a render target for the pipeline. The particular type of resource view that is needed is called a *render target view*. As is the case for most objects in Direct3D 11, it is created with the device interface and then used with the device context to perform some action. Listing 1.8 shows a code listing for creating a render target view when the texture resource is already available.

```
ID3DllRenderTargetView* pView = 0;
HRESULT hr = m_pDevice->CreateRenderTargetView( pSwapChainBuffer,
                                                0, &pView );
```

Listing 1.8. Creating a render target view to allow binding the swap chain texture to the pipeline for rendering.

In this case, we pass the texture resource as the first parameter, and instead of a view description, we simply pass NULL to use a default view description. Resource views are reference counted as well, so the application must be sure to properly release the reference when it is finished with it. With the resource view acquired, we are now ready to use our swap chain texture resource.

In addition to the render target, we also need to create a *depth stencil buffer* and a corresponding *depth stencil view* for binding it to the pipeline. The code required to do this is shown in Listing 1.9. Once again, this creation process is covered in detail in Chapter 2.

```
D3D11_TEXTURE2D_DESC desc;
desc.Width = width;
desc.Height = height;
desc.MipLevels = 1;
desc.ArraySize = 1;
desc.Format = DXGI_FORMAT_D32_FLOAT;
desc.SampleDesc.Count = 1;
desc.SampleDesc.Quality = 0;
desc.Usage = D3D11_USAGE_DEFAULT;
desc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
desc.CPUAccessFlags = 0;
desc.MiscFlags = 0;

ID3DllTexture2D* pDepthStencilBuffer = 0;
HRESULT hr = m_pDevice->CreateTexture2D( &desc,
                                         0, &pDepthStencilBuffer );

ID3DllDepthStencilView* pDepthview = 0;
HRESULT hr = m_pDevice->CreateDepthStencilView( pDepthStencilBuffer,
                                                pDesc, &pDepthView );
```

Listing 1.9. The process of creating a depth stencil buffer, and a depth stencil view for using it.

Even though we won't be creating additional resources or objects in this sample, we can still consider the types of items that should be created in this initialization routine. Essentially, all Direct3D resources should be created in the initialization phase, unless a particular use case absolutely requires that a resource be created and destroyed during runtime. For example, if a particular texture is procedurally generated, and its size is dependent on some runtime parameter, it would not be easy to create the texture during startup. However, it may be worth considering changing the algorithm to allow early creation. In our example, this could include just declaring one large resource that is shared by all objects that need such a procedural texture. It is also acceptable to create a resource at startup and then modify its contents during runtime, especially if the multithreading capabilities of Direct3D 11 are put to use.

In addition, all pipeline configurations should be pre-created. This includes any state objects that would be used by fixed function stages, as well as the shader program objects that would be used for the programmable shader stages. Since these items are typically planned for well in advance, there shouldn't be any reason that they couldn't be created at startup, instead of by using a dynamic loading scheme. Unless an application uses an enormous number of these objects, they should not cause any issues with memory consumption. In particular, the shader programs would need to read from the hard disk during

runtime if these objects are dynamically loaded—which would almost surely cause a brief pause in the application.

Application Loop

After initialization is complete, the application can move on to the looping portion of its execution. This phase consists of three individual sections, which will be described in the following sections.

Handle pending messages. After an application enters its application loop, it must respond to and handle the Windows messages that it receives from the operating system. This is performed by a callback function that is specified when the application window is registered and created. The handling of messages is not directly relevant to Direct3D programming, so we will skip it for now. However, all of the sample programs from this book implement message handling, so the fundamentals can be seen there. The message handler can be found in the main. cpp file in the source code distribution.

Update. Our simple test application doesn't perform any simulations or implement special physics calculations, but it does perform a simple animation to change the color of the render target. This is performed in the Update phase of the application loop. This is a typical place to perform such updates, so that any changed state can be reflected in the current frame's rendering, which is performed in the next step of the loop. For our sample application, we simply calculate a sinusoidally varying value that will be used to determine the background color of the window. This is shown in Listing 1.10.

```
float fBlue = sinf( m_pTimer->Runtime() * m_pTimer->Runtime() ) * 0.25f + 0.5f;
```

Listing 1.10. Producing a time-varying parameter for animation.

Here we see a timer class being used to acquire the amount of time that an application has been running. This class is taken from the sample framework of this book, and more detail on its inner workings can be found in the source code distribution in the Timer class files. For now, we will simply assume that it gives us the floating point number of seconds that an application has been running.

Rendering. After updating the state of our application, we next render a representation of our current scene for the user to see. This process begins by binding both the render target view and the depth stencil view to the pipeline, to receive the results of any rendering operations. Since this is a pipeline manipulation, it is performed with the device context. Listing 1.11 shows how this is done.

```
ID3DllRenderTargetView* RenderTargetViews[l] = { pView };
ID3DllDepthStencilView* DepthTargetView = pDepthView;

pContext->OMSetRenderTargets( 1, RenderTargetViews, DepthTargetView );
```

Listing 1.11. Binding the render and depth targets to the pipeline for rendering.

Both the render target and depth stencil target are bound to the pipeline with the same method call. This method for setting render targets takes an array of render target views, so even if you are only using a single render target, it is a best practice to use an array to pass its reference into the function. After the render and depth targets have been bound, we can clear them to prepare for the coming rendering pass. The body of a method that performs this process is shown in Listing 1.12. As you can see, the clearing process is also performed with the device context.

```
ID3DllRenderTargetView* pRenderTargetViews[D3Dll_SIMULTANE0US_RENDER_TARGET_
COUNT] = {NULL};
ID3DllDepthStencilView* pDepthStencilView = 0;

m_pContext->OMGetRenderTargets( D3D11_SIMULTANE0US_RENDER_TARGET_C0UNT,
                                pRenderTargetViewsj SpDepthStencilView );

for ( UINT i = 0; i < D3D11_SIMULTANE0US_RENDER_TARGET_C0UNT; ++i )
{
    if ( pRenderTargetViews[i] != NULL )
    {
        float clearColours[] = { color.x, color.y, color.z, color.w }; // RGBA
        m_pContext->ClearRenderTargetView( pRenderTargetViews[i],
                                           clearColours );
        SAFE_RELEASE( pRenderTargetViews[i] );
    }
}

if ( pDepthStencilView )
{
    m_pContext->ClearDepthStencilView( pDepthStencilView.,
                                       D3Dll_CLEAR_DEPTH, depth, stencil );
}

// Release the depth stencil view
SAFE_RELEASE( pDepthStencilView );
```

Listing 1.12. Clearing the contents of the bound render and depth targets.

Once the render target and the depth stencil target have been cleared, the application can perform whatever rendering operations it needs to. This includes rendering the

current scene, any two-dimensional screen elements such as user interfaces, and any text that might be needed for a given frame. Once again, our sample application won't perform any actual rendering. Instead, it simply uses the method shown in Listing 1.12 to clear the render target to the color specified in our update phase.

Presenting results to the window. After all of the rendering for a given frame has been completed, its contents can be presented to the window's client area. This is done quite simply, using a single method call from the swap chain interface. This is shown in Listing 1.13.

```
pSwapChain->Present( 0, 0 );
```

Listing 1.13. Presenting the contents of the swap buffer resource to its window.

When this method is called, the contents of the render target are presented to the client area of the window. The application doesn't need to perform any manual render target manipulation if multiple buffers are used within the swap chain. Instead, this is managed by the swap chain itself, which simplifies the application's responsibilities. After this method has completed, the user will be able to see the render target contents in the window.

After the rendered output has been displayed, the application loop starts over and checks for any pending Windows messages. This looping sequence repeats until the user decides to terminate the application. This is normally done by pressing the escape key or selecting a "quit" option from an application user interface.

Application Shutdown

Before an application can terminate, it needs to clean up all of the open references that it has acquired. With respect to Direct3D 11 objects, this typically means calling the Release method for each reference that the application has used. One point to also consider is that the pipeline itself will keep references to some of the objects that have been bound to it. To ensure that these pipeline references are released, the application can call the ClearState method of the device context. This will clear any references from the pipeline and put it into the default state. The shutdown code for our sample application is shown in Listing 1.14.

```
pContext->ClearState();

SAFE_RELEASE( pView );
SAFE_RELEASE( pDepthView );
SAFE_RELEASE( pDepthStencilBuffer );

SAFE_RELEASE( pSwapChainBuffer );
SAFE_RELEASE( pSwapChain );
SAFE_RELEASE( pContext );
```

```
SAFE_RELEASE( pDebugger );
SAFE_RELEASE( pDevice );
```

Listing 1.14. Releasing the various interfaces that have been used in the simple example application.

If there are still some references that have not been properly released, a debug message will be printed to the debug console, indicating which interface object types still have outstanding references that haven't been released. If all references were properly released, the application will exit, just like any other application.

1.5.3 Further Application Considerations

While the example steps that we walked through in the second half of this chapter are indeed important to understand, they are not the most exciting operations to perform for every application that you create. Many of these tasks, such as creating a window or initializing the device, can be handled with library functions that can be reused for all of the applications that you create. In the real-time rendering context, this type of a library is often called an *engine,* a term first coined by John Carmack of id Software.

This is the path that we have chosen to follow for the sample programs in this book. All of the samples are built upon an open source engine developed by the authors, called Hieroglyph 3. The engine and the samples are both freely downloadable from the Hieroglyph 3 project page, which can be found at http://hieroglyph3.codeplex.com. In addition to the samples from this book, many other example applications are included in the source code repository for learning the basics of the library.

By using such an open source library, we can devote more of this book to the concepts and use of Direct3D 11, rather than spending time explaining basic Win32 application code or giving repetitive code samples. The library takes care of the basics, so that we can focus on the more interesting portions of the subject matter and ultimately provide a better book. The Hieroglyph 3 library is provided with the MIT license, which includes a liberal set of usage guidelines, and that you can use as the basis of your own projects.