2 Direct3D 11 Resources

In many cases, discussions of modern real-time rendering techniques tend to revolve around the programmable pipeline and what can be done with it. The pipeline is indeed a critically important piece in the real-time rendering puzzle, especially since one of the primary tasks of the graphics developer is writing shader programs for use in the pipeline. However, the importance of the memory resources connected to the pipeline is often overlooked in these discussions. It is just as important to understand what resources are required for a particular algorithm, where they will be used, and how they will be used.

This chapter will inspect the topic of Direct3D 11 Memory Resources in depth. We begin by identifying how resources are organized and managed by the Direct3D 11 API and then consider the two major types of resources—buffers and textures. We will discuss in detail each subtype of resource, how it is created, how it is used, and how it is released when no longer needed. The discussion of resource use will also introduce the concept of a resource view, a type of adapter used to connect a resource to a particular place in the pipeline.

Any discussion about resources must also look at both aspects of their usage. The application is concerned with creating, populating, and connecting resources to the pipeline. However, we must also consider how resources are used inside of the pipeline as well. How they are declared and used within a programmable shader is just as important of a consideration as properly creating the resources. This topic is discussed in detail to provide a sound understanding of this usage within shader programs, which will be built upon in Chapters 3-7.

As we will see later in this chapter, it is very important to have a clear understanding of how a resource will be used before creating it, since its usage domain is predetermined by its creation input parameters. If incorrectly configured, a resource can have several negative effects on your applications, ranging from poor performance all the way to a complete inability to execute the pipeline due to errors. This makes resources a very important topic indeed!

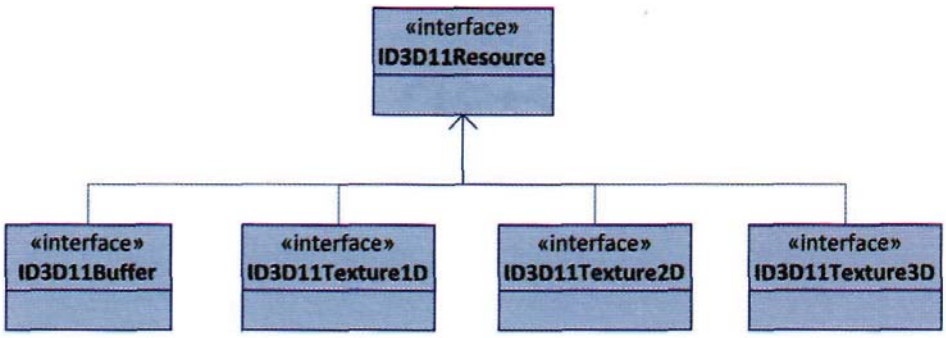

Figure 2.1. The Direct3D 11 Resource Interface Hierarchy.

2.1 Resources Overview

There are two basic categories of resources in Direct3D 11: *buffers* and *textures*. Each of these has several subtypes, and each subtype has different configuration options. The sheer number of available options for creating resources can be somewhat overwhelming at first, so our discussion of the topic will begin with how they are organized within the Direct3D 11 API. By starting at a high level, we can begin to understand the differences in the various resources and then build on this knowledge with additional details about the way that these differences determine how a resource can be used.

The Direct3D 11 resource class structure is organized as shown in Figure 2.1. Here, you can see that there are only four resource classes available for the entire API—buffers, and ID, 2D, and 3D textures.

The diagram also shows that each of these resource classes is derived from a single common base class, named *ID3D11Resource*. This makes sense and provides an indication of a common theme that we will be revisiting throughout this chapter. That is, the resources are only blocks of memory that can be attached to the pipeline and used for input or output (or in some cases, both input and output). Restated another way, resources are blocks of memory that are made available for the GPU to use and manipulate. Even though they have different names and support different concepts, the only real differences between them are the semantics with which they are bound to the pipeline and the rules about their formatting, access, and use.

2.1.1 Resource Creation

As described in Chapter 1, "Overview of Direct3D 11," the ID3D11Device interface is responsible for creating all memory resources. The created resources can then be attached

to the pipeline either directly or with a resource view, where they are then used during a pipeline execution event. The resource creation process uses a different ID3DllDevice method for each type of resource, but they all follow the same general pattern.

The creation methods all take three parameters. The first parameter is a structure that specifies all of the various options that a resource can be created with. It is referred to as a *resource description.* Each resource type uses its own description structure, since they each have a different set of available properties, but the structures all serve the same purpose—to define the desired characteristics of the created resource. These structures will be investigated in detail in this chapter, and their proper configuration comprises the bulk of the work required to make a resource do what you want it to. The second parameter in the resource creation methods is a pointer to a D3D11_SUBRESOURCE_DATA structure, which is used to provide the initial data to be loaded into a resource. For example, if a buffer resource will hold static vertex data, this structure would be used to pass a model's vertex data into the buffer. This eliminates the need to manually manipulate the buffer after it is created, if its contents will not be changing. This parameter can also just be set to null if no initialization is required. The final parameter is a pointer to a pointer to the appropriate resource interface, which is where the created resource pointer is stored after a successful resource creation event.

In each of these methods, the real configuration occurs in the resource description structure. As mentioned above, each resource type has its own structure used to define its properties. However, there are some common elements that are shared across all of the structures. These include the usage flags, bind flags, CPU access flags, and miscellaneous flags. Since these parameters are common across all the various resource types, we will discuss them here in detail. The individual parameters that are unique to certain types of resources will be discussed in their respective resource type sections later in this chapter.

Resource Usage Flags

The first resource parameter that we will inspect is the *usage specification.* Stated simply, this parameter is used to specify what the application intends to do with the resource. In general, we can consider a resource to be a block of memory somewhere within the computer. Both video memory and system memory can hold resources, and resources can be copied to and from each type of memory by the Direct3D 11 runtime. To optimize where a resource resides and how it is handled internally by the runtime/driver, the application must specify its intentions with this usage parameter. This differs from previous versions of Direct3D, which allowed the user to specify the memory pool that the resource would reside in. However, by indicating the usage intent of the resource, a similar level of control is provided. The list of available values is encapsulated in the D3D11_USAGE enumeration, which is shown in Listing 2.1.

```
enum  D3D11_USAGE  {
      D3D11_USAGE_DEFAULT,
      D3D11_USAGE_IMMUTABLE,
      D3D11_USA6E_DYNAMIC,
      D3D11JJSAGE_STAGING
}
```

Listing 2.1. The D3D11_USAGE enumeration.

Each of these flags indicates one usage pattern, which identifies the required read/write capabilities of the resource for both the CPU and the GPU. For example, a resource that is only read and written by the GPU and is never accessed by the CPU can be safely placed in video memory. It will never be transferred back to system memory, since the CPU is not able to access it. This provides an optimization by keeping the resource closer to the GPU. Table 2.1 shows the read/write permissions for each of these flags.

Resource Usage	Default	Dynamic	Immutable	Staging
GPU-Read	yes	yes	yes	yes
GPU-Write	yes			yes
CPU-Read				yes
CPU-Write		yes		yes

Table 2.1. The accessibility defined for each usage flag.

Immutable usage. The simplest usage pattern is the *immutable* usage. A resource created with this usage pattern can only be read by the GPU and is not accessible by the CPU. Since the resource can't be written to by the GPU or CPU, it can't be modified after it is created. This requires that the resource be initialized with its data at creation time. Since the resource can't be modified, the usage type's name is appropriately immutable. Common examples of this resource type are static constant, vertex, and index buffers that are created with data that won't change over the lifetime of the application. In general, these resources are used to provide data to the GPU.

Default usage. As can be seen in the table above, the *default* usage provides read and write access to the GPU, but provides no access to the CPU. This is the most optimal usage setting for any resources that will not only be used by the GPU, but that will also be modified by the GPU at some point. Some common examples of this usage pattern are render textures (which are rendered into the GPU and subsequently read from it), and stream output vertex buffers (which have data streamed into them by the GPU and are later read by the

GPU for rendering). Since resources of this type can remain in video memory, the GPU will have the fastest possible access to them, and the application's overall performance will benefit.

Dynamic usage. The *dynamic* usage is the first usage type that allows the CPU to write to a resource that can then be read by the GPU. In this usage case, the contents of the resource are provided by the CPU and are then consumed by the GPU. The most common example of this is a constant buffer, which provides rendering data to the programmable shader stages which changes in every rendered frame. This type of data could be transformation matrices, for example. An important consideration is that the CPU is not able to read back the contents of the resource—the data can only flow in one direction from the CPU to the GPU.

Staging usage. The *staging* usage is the final usage flag, and it provides a special type of usage pattern. The usage patterns that we have discussed up to this point have covered the typical resource usage scenarios when performing rendering. However, there are many situations in which we want to manipulate data with the GPU and then read it back to the CPU for storage or examination. With the introduction of the GPGPU facilities with the DirectCompute technology, along with the ability to stream vertex information out with the stream output stage, it is possible to use the GPU to process information and then read it back to the CPU. Instead of forcing the other usage patterns to allow read access to the CPU, a resource with staging usage can be used as an intermediate instead.

The basic concept is that a desired resource can be manipulated by the GPU, then copied onto a staging resource, and then subsequently read back to the CPU. This requires creating an additional staging resource to retrieve the data, but it lets the other resources used by the GPU remain as close as possible to the GPU and not be concerned with CPU access. We will see additional detail on manipulating resource contents later in this chapter.

CPU Access Flags

After the usage flag of a resource has been determined, the needed CPU access flag must be chosen. The CPU access flag exposes similar information as we have seen in the usage property, but it is restricted to defining the possible CPU access to resources. There are only two possible flags to use for the CPU access flags. These are shown in Listing 2.2.

```
enum  D3D11_CPU_ACCESS_FLA6  {
      D3D11_CPU_ACCESS_WRITE,
      D3D11_CPU_ACCESS_READ
}
```

Listing 2.2. The D3D11_CPU_ACCESS_FLAG enumerations.

These flags can be combined with a bitwise OR operation to indicate if the resource will be read, written, or read and written by the CPU. These settings must also take into account the usage flags that are specified for this resource as well. As shown in the table above, a staging resource can be both read from and written to by the CPU, allowing both of these flags to be set. In contrast, a default usage is not accessible to the CPU at all, and hence its CPU access flags must be set to 0.

Bind Flags

Our next common resource flag is the *bind flag*. This property indicates where on the pipeline a resource can be bound and includes various flags that represent each available binding location. These flags can also be combined with a bitwise OR to allow for multiple bind locations to be defined for the same resource. If a resource is created without the appropriate flag, and the application tries to bind the resource to the pipeline, then it will cause an error. All of the available binding locations are shown in Listing 2.3 in the D3D11_BIND_FLAG enumeration.

```
enum D3D11_BIND_FLAG {
      D3D11_BIND_VERTEX_BUFFER,
      D3D11_BIND_INDEX_BUFFER,
      D3D11_BIND_C0NSTANT_BUFFER,
      D3D11_BIND_SHADER_RES0URCE,
      D3D11_BIND_STREAM_OUTPUT,
      D3D11_BIND_RENDER_TARGET,
      D3D11_BIND_DEPTH_STENCIL,
      D3D11_BIND_UNORDERED_ACCESS
}
```

Listing 2.3. The D3D11_BIND_FLAG enumeration.

As you can see from this list, there are only eight types of locations where resources can be bound to the pipeline. The vertex and index buffer flags are used to declare resources that will be attached to the input assembler stage to feed the pipeline with geometry, while the render target and depth stencil flags allow resources to be connected to the output merger stage to receive the rendered output from the pipeline. The *stream output* flag also allows output from the pipeline, although it receives geometry instead of rasterized image data. These flags all represent a single binding point on the pipeline.

In contrast, the D3D11_BIND_C0NSTANT_BUFFER, D3D11_BIND_SHADER_RES0URCE, and D3D11_BIND_UN0RDERED_ACCESS access flags all indicate that the resource can be

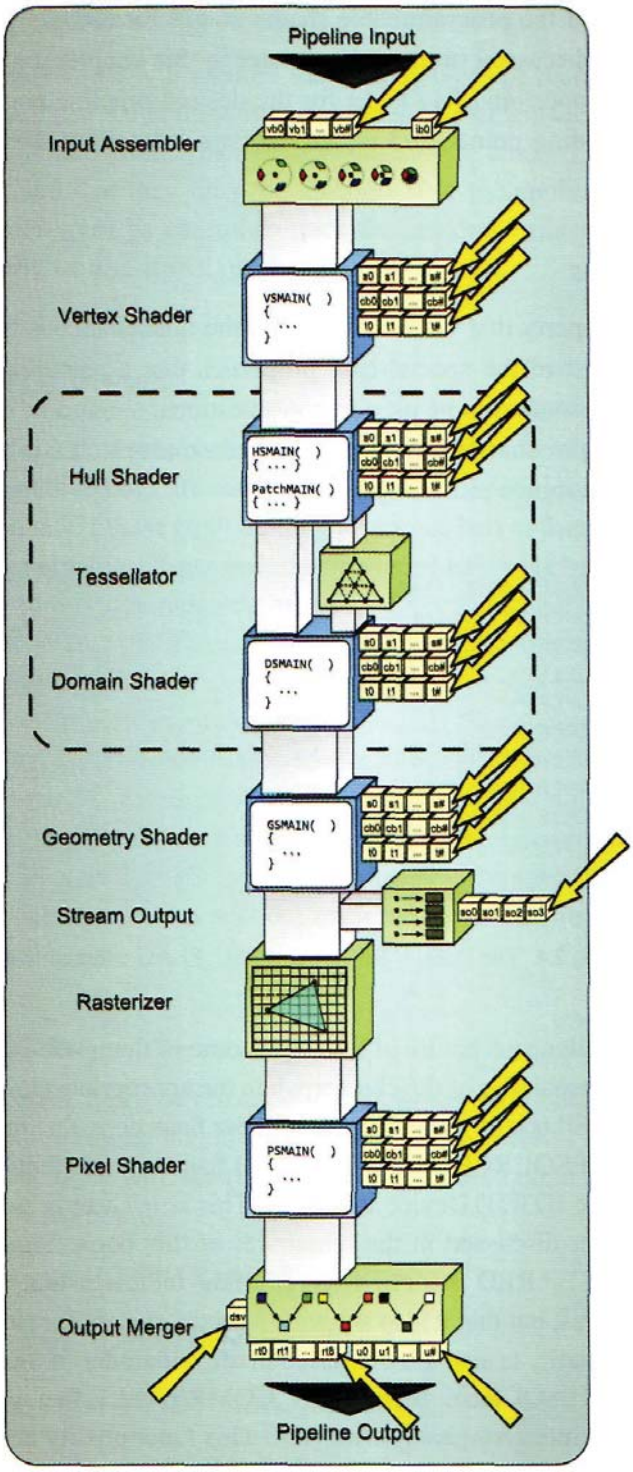

Figure 2.2. The resource binding locations of the rendering pipeline.

bound to some or all of the programmable shader stages for use in the shader programs. These usages will be discussed in more detail later in this chapter, but in general it must be ensured that the proper bind flag is set for the desired pipeline connection points. An overview of these binding points, and where they are located on the pipeline, is shown Figure 2.2.

Miscellaneous Flags

The final common property that we will look at is the miscellaneous flags. This group of flags encapsulates most of the special-case properties that a resource can use. Some of these flags allow for resources to be used in non-traditional situations, such as interoperating with GDI drawing or sharing a resource between multiple ID3D11 Device instances. The available flags are shown in Listing 2.4.

```
enum  D3D11_RES0URCE_MISC_FLAG  {
      D3D11_RESOURCE_MISC_GENERATE_MIPS,
      D3D11_RESOURCE_MISC_SHARED,
      D3D11_RESOURCE_MISC_TEXTURECUBE,
      D3D11_RESOURCE_MISC_DRAWINDIRECT_ARGS,
      D3D11_RESOURCE_MISC_BUFFER_ALO0W_RAW_VIEWS,
      D3D11_RESOURCE_MISC_BUFFER_STRUCTURED,
      D3D11_RESOURCE_MISC_RES0URCE_CLAMP,
      D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX,
      D3D11_RESOURCE_MISC_GDI_C0MPATIBLE
}
```

Listing 2.4. The D3D11_RESOUR_MISC_FLAG enumeration.

Due to the miscellaneous nature of the flags, some of them will be discussed in more detail throughout the remainder of the chapter when the appropriate content has been introduced. However, we will briefly discuss some of these flags here, due to their more general nature. The D3D11_RESOURCE_MISC_SHARED flag indicates that the resource can be shared among multiple ID3DllDevice instances. This is used only in advanced applications and is not further discussed in the remainder of this book. Similarly, the D3D11_RESOURCE_MISC_SHARED_KEYEDMUTEX flag indicates that a resource will be used by multiple devices, but that it also supports a mutex system for sharing control of the resource. This functionality is also not discussed in the remainder of this book. And finally, we have the D3D11_RESOURCE_MISC_GDI_COMPATIBLE flag, which indicates that a resource can be used interchangeably with GDI. This functionality is used in some of the demo programs to implement text rendering, but it is not further described in this book.

Releasing Resources

Throughout this chapter, we will explore the various types of memory resources that are available to the developer. All of these resource types implement COM interfaces, and they ultimately have the IUnkown interface in their inheritance chain. This means that they are reference counted, and that they must be released after the application is finished using them. The application must be careful to track the resource references that it retains and release them properly, or else it will result in a memory leak.

2.1.2 Resource Views

Resources also share a common technique for binding them to the pipeline. As we saw above, eight different bind flags are available which identify the locations that a resource is allowed to be bound at. Of these eight binding locations, half of them allow resources to be bound directly to the pipeline. These include the vertex and index buffers, constant buffers, and stream output buffers (the meaning of these buffer types will be further explained in the "Buffer Resources"section of this chapter). However, the other four binding locations all require a type of adapter called a *resource view* to be used when binding a resource to the pipeline. A resource view is an adapter object used to connect a resource to various points in the pipeline. It conceptually provides a particular "view" of a resource. Like the configuration of resources, resource views also provide some configurability. By allowing a resource view to provide some interpretations of the resource, it is possible to use multiple resource views on a single resource for use at different locations on the pipeline. In fact, as we will see later in this section, resource views can also be used to represent subsets of a resource.

Resource View Types

There are four different types of resource views, which each correspond to a particular location that a resource can be bound to on pipeline. These views determine what operations can and can't be done with the resources that they are bound to. These four types of resource views are listed below:

Render target views (ID3D11RenderTargetView)

Depth stencil views (ID3D11DepthStencilView)

Shader resource views (ID3D11ShaderResourceView)

Unordered access view (ID3 D11UnorderedAccessView)

Each of these resource views implies the particular semantics that its resource can use. In addition, each type of resource view provides a number of different configuration options that allow further specification of how a resource can be used. We will briefly discuss the semantics for each of the types of resource views, and their precise usage will be made much clearer, in Chapter 3, "The Rendering Pipeline."

Render target views. The *render target view (RTV)* is used to attach a resource to receive the output of the rendering pipeline. This means that the resource that is connected will be written to by the pipeline, and that it will also be read from in some cases to perform blending operations. Traditionally, a render target is a two-dimensional texture, but it is also possible to bind other types of resources. The various configuration options for a render target view depend on the type of resource that is being bound, but include the DXGI format of the resource, as well as various methods to select subportions of a resource to expose to the pipeline.

Depth stencil views. A *depth stencil view (DSV)* is similar to a render target view in that it is attached for receiving output from the rendering pipeline. However, it differs from the render target view in that it represents the depth stencil buffer, instead of a color render target. The depth stencil view resource is actively used to perform the depth and stencil tests. This makes the attached depth stencil resource very important for the efficiency of the pipeline. Thus, to further improve performance, the depth stencil view also exposes an additional flag to determine if the attached resource will be written to or not. This allows a resource to be attached as the depth stencil buffer and used for the depth stencil tests, but at the same time to be used in a shader program with a shader resource view (the next resource view that we will inspect). In this configuration, both views are read-only, meaning that the resource will not be modified by either one. With such a usage, the same resource can be bound to the pipeline in multiple places without read/write hazards, which allows for flexible use of the resource. If a standard use of the depth stencil buffer is needed, a second resource view can replace the read-only one to allow writing to take place.

Shader resource views. A *shader resource view (SRV)* provides read access to a resource to the programmable shader stages of the pipeline. This view corresponds to the traditional role that a texture would play in a pixel shader—data that is read from and used in the shader program, but it is not written to. A shader resource view can be used with all of the different programmable shader stages.

Unordered access views. The *unordered access view (UAV)* provides some of the most interesting new uses for resources in Direct3D 11. Like the shader resource view, a UAV can be used to read information from a resource. However, it also allows the resource to be simultaneously written to from within the same shader program. Further, the output location is not predefined, which allows the shader program to perform scattered writes to

any location within the resource. This provides a completely new class of resource access, since it allows the resource to be used in a much more flexible manner. However, it is important to note that this type of view is not available for all programmable shader stages. These resource views can only be used with the pixel shader and compute shader stages.

Resource View Creation

The process of creating a resource view is somewhat similar to the process that we have seen for creating a resource. The ID3D11 Device is also responsible for creating resource views, and it provides one creation method for each type. All creation methods follow the same pattern, with three different parameters provided by the application. As an example, Listing 2.5 shows the creation of a shader resource view.

```
ID3DllShaderResourceView* CreateShaderResourceView( ID3D11Resource* pResource.,
                                        D3D11_SHADER_RESOURCE_VIEW_DESC* pDesc )
{
    ID3DllShaderResourceView* pView = 0;
    HRESULT hr = m_pDevice->CreateShaderResourceView( pResource,
                                                      pDesc, SpView );

    return( pView );
}
```

Listing 2.5. An example technique for creating a shader resource view.

The first parameter for all of these creation methods is a pointer to the resource that the resource view will represent. The second parameter is a pointer to a description structure, with all of the available options for that particular type of resource view. The third and final parameter is a pointer to a pointer to the corresponding resource view type, which is where the newly created resource view will be stored if the creation call succeeds. The second parameter is the most interesting with respect to configuring a resource view in the way we want it to behave. Each of the four resource view types uses a unique description structure, exposing each of their various unique properties. We will take a closer look at each of these structures to gain an understanding of what options are available to the developer.

Render target view options. We begin by looking closer into the available options for creating a render target view. Figure 2.3 shows the structure that must be filled in and passed to the creation method, ID3DllDevice::CreateRenderTargetView().

As seen in Figure 2.3, this structure uses several normal member variables, followed by a union of possible structures. This allows all different resource types to use the same description structure, while still providing unique properties for each one. The format parameter specifies the data format that the resource will be cast to when it is read. In some

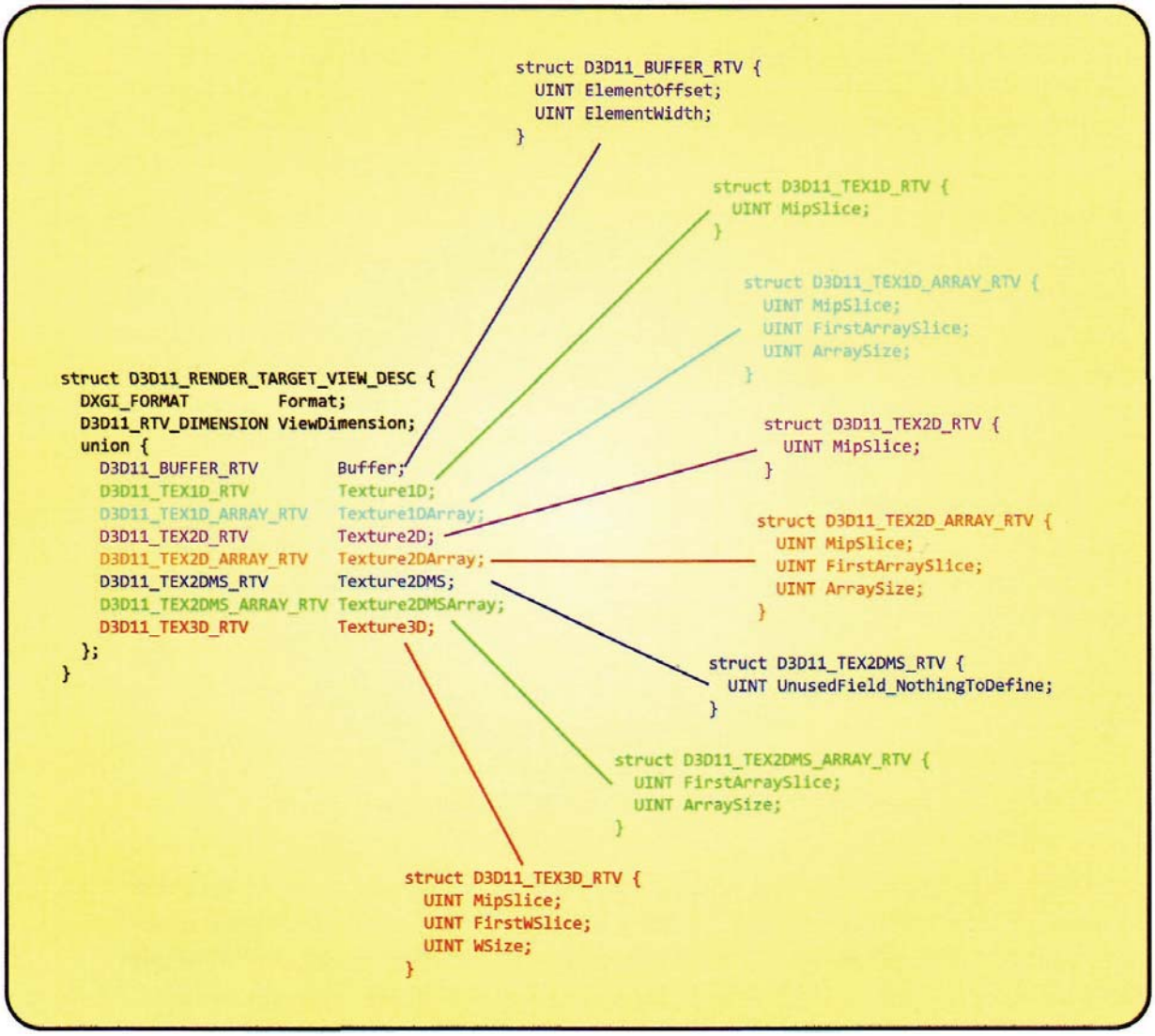

Figure 2.3. The D3D11_RENDER_TARGET_VIEW_DESC structure.

circumstances, this allows the format of the resource to be determined at runtime by the format supplied in the resource view being used. This can be helpful for certain applications, such as video format conversion. This requires that the resource be created with a format that is compatible with the desired view format specified in this structure.

The second variable, ViewDimension, indicates which type of resource will be bound with this resource view. The selection of this variable determines which of the unioned structures will be used to create the resource view. This is how the device knows which version of the unioned structures to interpret, which allows the same creation function and input data structure to be used for all of the various resource types, while still keeping the

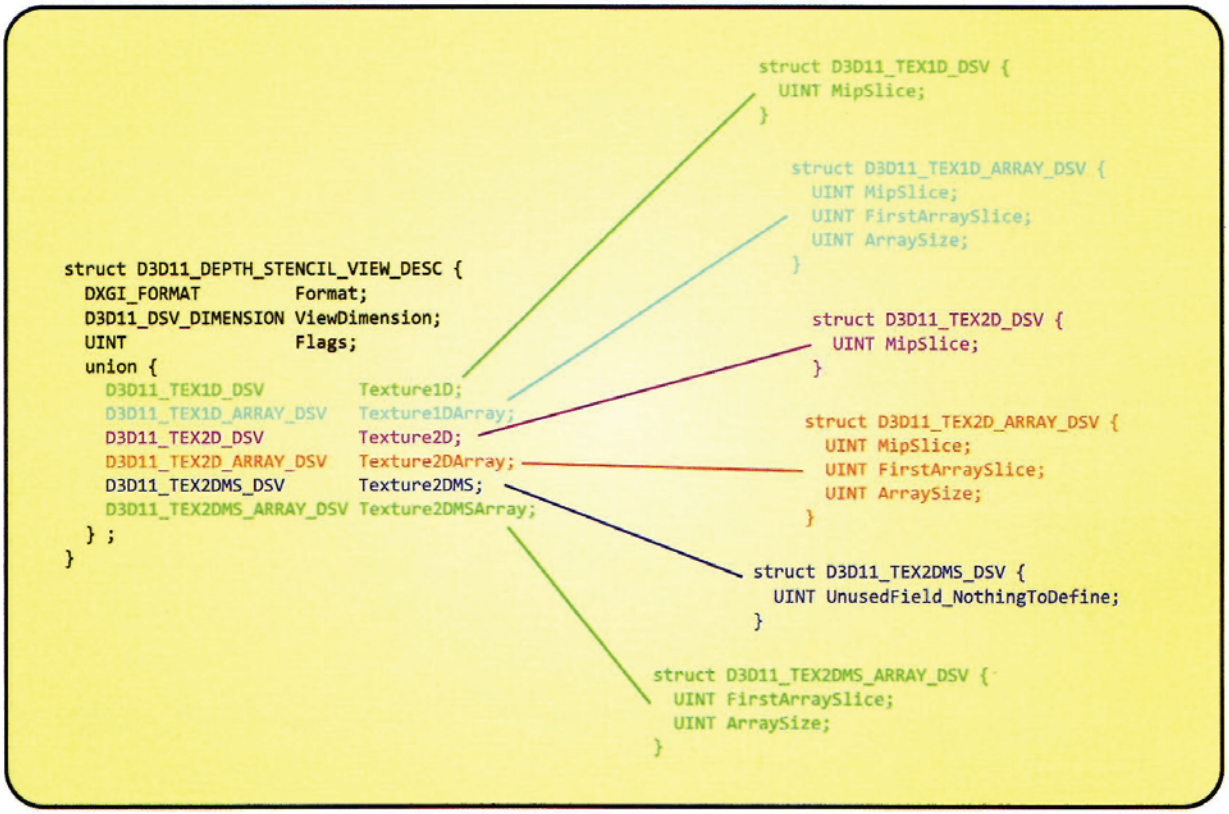

Figure 2.4. The D3D11_DEPTH_STENCIL_VIEW_DESC structure.

size of the structure to a minimum. All of the "union-ed" structures are used to specify what portions of the resource to make available in the resource view. Of course, since each type of resource uses a different memory layout and different options, it makes sense to have different structures for each resource type. Since we haven't covered the various resource types (or their various configurations) in detail, we must defer a detailed look into these properties. When we investigate the various resource types and their layout concepts later in this chapter, we will revisit these structures to see how they provide appropriate subresource selections.

Depth stencil view options. The depth stencil view option structure has a description structure that is similar to the one for the render target view. Its individual components are shown in Figure 2.4.

As can be seen above, this resource view type also requires a DXGI data format and a view dimension property to be specified. One point of interest is that the number of different resource types that can be used for a depth stencil resource view is somewhat smaller than can be used for a render target view. This is due to the specific nature of the operations that are performed on a depth stencil resource.

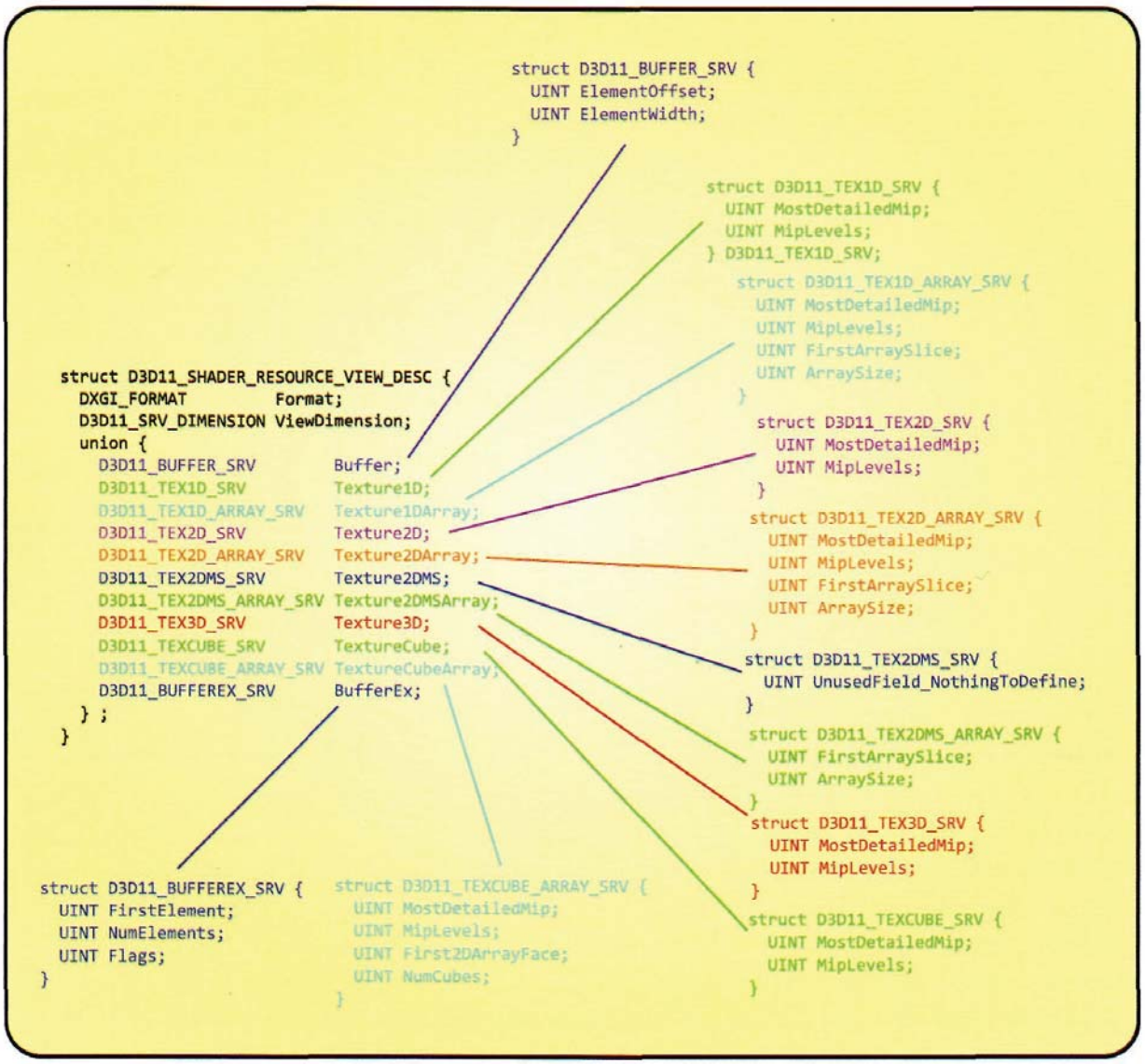

Figure 2.5. The D3D11_SHADER_RESOURCE_VIEW_DESC structure.

Another notable feature of this structure is the addition of a Flags parameter. This allows a bitwise OR combination of flags that specifies if the depth or stencil portion of the resource will be read only to the pipeline with this resource view. This allows multiple views of a depth stencil resource to be bound simultaneously to the pipeline for algorithms that are required to inspect its contents.

Shader resource view options. The *shader resource view* also follows the same basic formula that we have already seen in previous description structures. The format and view dimension operate in the same manner as the resource views we have already inspected.

However, you will notice three new resource types that are available in the unioned structure. Figure 2.5 shows this structure.

The new types of resources listed here are TextureCube, TextureCubeArray, and BufferEx. A texture cube allows reinterpretation of a Texture2D Array resource to a cube texture, which allows HLSL programs to use specialized intrinsic functions for sampling the texture. This also holds true for a TextureCubeArray, which is basically an array of the same type of cube resource interpretations. We will explore what this means in more detail later on in the "Texture Resources" section, but it provides a good example of the ability of a resource view to take a resource's data and provide a different "view" of it for a particular purpose.

The final new type of resource available for shader resource views is the Buf f erEx type. This is essentially a structure that allows a buffer to be interpreted as a raw buffer. This gives HLSL programs the freedom for structure interpretation within the shader program itself. This will also be discussed in more detail in the "Buffer Resources" section of this chapter.

Unordered access view options. The final resource view type is the *unordered access view*. Its description structure also follows the now standard format, as shown in Figure 2.6.

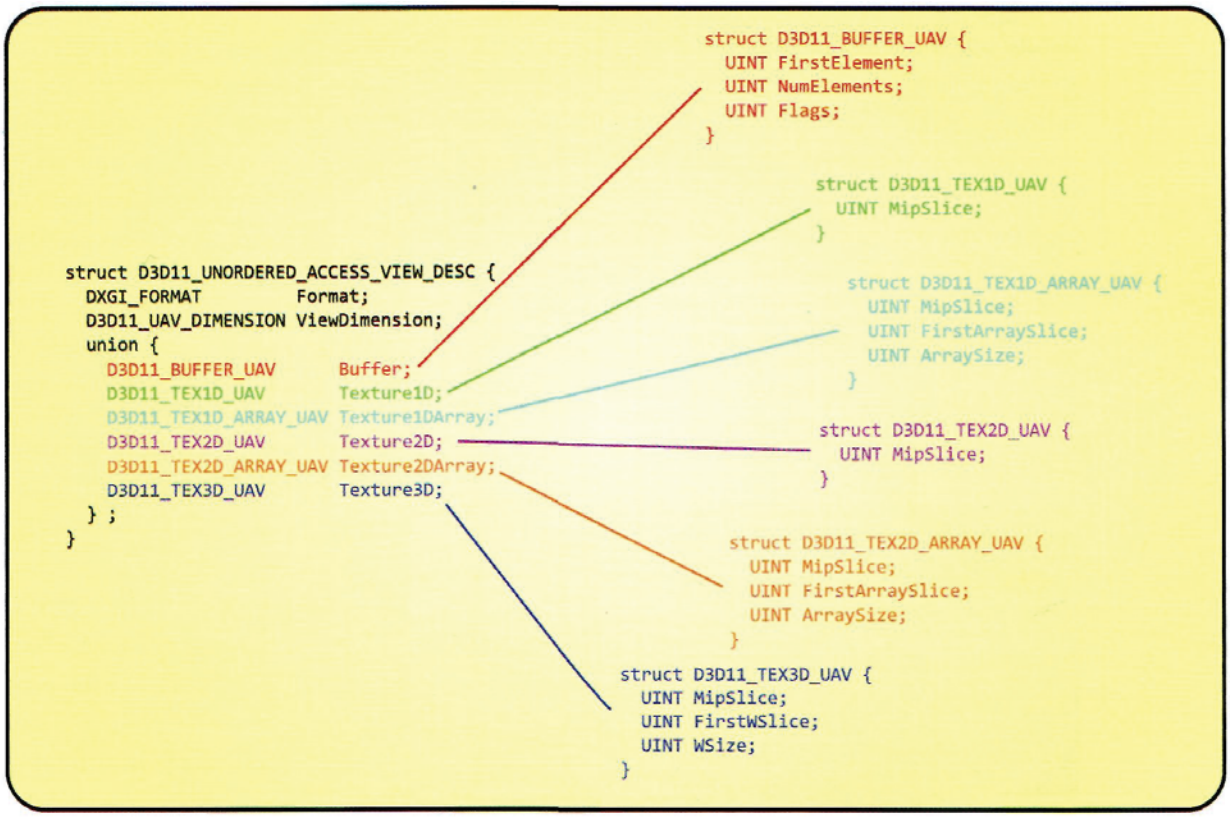

Figure 2.6. The D3D11_UNORDERED_ACCESS_VIEW_DESC structure.

The unordered access view provides a lesser number of available resource types when compared to the shader resource view. However, it does provide several new options for configuring buffer resources. As we will see later in the "Buffer Resources" section, there are some unique types of buffers that can be created for special uses. There are additional flags in this description structure for using buffers as append and consume buffers, as well as another flag to indicate that a built-in counter will be available in the buffer object.

2.2 Resources in Detail

Now that we have learned some of the basic principles of resources and resource views, it is time to begin a more detailed examination of the various types of resources to see what makes them different from one another, and what they can do. This section will introduce each type of resource and discuss each of the available subtypes they can be used to create. We begin with the buffer resource, exploring each of the individual types of buffers that can be used, and their various properties. This is followed by a general discussion about texture resources, followed by a similarly detailed discussion about the various configurations and options available for each texture type. It is also important to understand the dual nature of the resources that we are working with. They are created, released, and connected to the pipeline at a high level in C/C++, but are also used at a much finer level of granularity in the HLSL shader programs of the programmable shader stages. These two usage paradigms are of course intertwined, as certain resource types are required for certain operations within HLSL. We will explore both sides of the resources as we progress through the chapter. We will also explore both sides of a resources usage equation throughout this section.

2.2.1 Buffer Resources

The *buffer resource* type provides a one-dimensional linear block of memory for use by Direct3D 11. A number of different configurations can be used to change the behaviors of a buffer, but they all share this same basic linear layout. Figure 2.7 shows a diagram of how a buffer resource is organized.

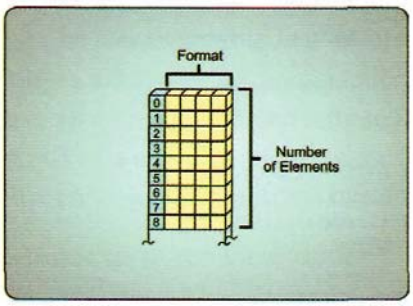

Figure2.7. The layout ofa buffer resource.

As can be seen in Figure 2.7, a buffer's size is measured in bytes. The elements that make up a buffer can have different sizes, depending on the type of the buffer, as well as on some specific configurations for each particular buffer type. Multiplying the number of elements by the size of each element gives the total size of the buffer. This simple

array-like layout structure provides a surprisingly wide variety of available buffer types. Some buffer types will be used primarily in the C++ side of an application, while others will be used primarily by HLSL shader programs after being attached to the pipeline. This section will explore each of the different types of buffers, describe what functionality they provide, and demonstrate how to create them. In addition, we will discuss common uses for buffers and provide the basic HLSL syntax to declare and use these resources in shader programs.

Vertex Buffers

The first buffer type that we will examine is the *vertex buffer*. The purpose of a vertex buffer is to house all of the data that will eventually be assembled into vertices and sent through the rendering pipeline. The simplest vertex buffer configuration is an array of vertex structures, where each vertex contains elements such as position, normal vector, and texture coordinates. These vertex elements must conform to the available format and type specifications. However, whatever generic information that is desired within the vertex can also be packed into a vertex structure to allow customized input data for a particular rendering algorithm.

In addition to the simple array-style vertex buffers described above, these buffers also allow for some other, more complex configurations. For instance, it is possible to use more than one vertex buffer at the same time. This allows vertex data to be split among multiple buffers. For example, vertex positions could be stored in one buffer, and vertex normal vectors in another buffer. This allows the application to selectively add in vertex data as it is needed, instead of using one large overall buffer for all rendering scenarios. This technique could be used to reduce the amount of bandwidth required for rendering operations.

It is also possible to perform *instanced rendering*, where one or more vertex buffers provide a model's *per-vertex* data, and an additional vertex buffer provides *per-instance* data instead of per-vertex data. The model defined in the first buffer is then rendered as a series of models, with the per-instance data from the second buffer applied to each instance. Figure 2.8 depicts these different vertex submission configurations. The per-instance data could include

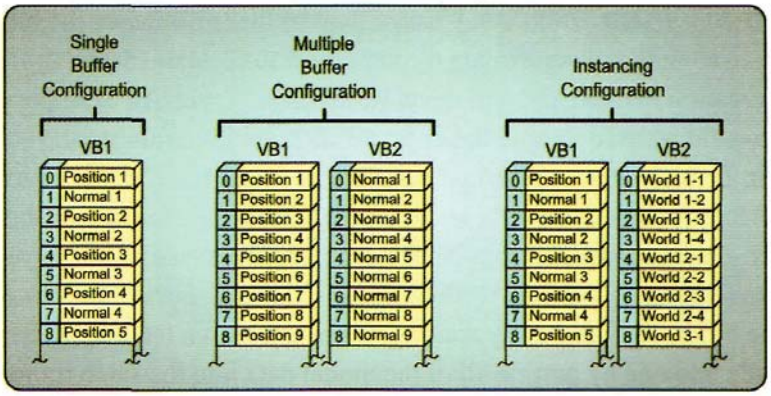

Figure 2.8. The various vertex buffer configurations that are available to an application.

a world transformation, color variations, or whatever else is used to differentiate between the various instances of the model. This setup allows for many objects to be rendered with a single *draw* call, which reduces the overall CPU overhead for rendering operations.

You can see in the diagram that each type of vertex buffer follows our general buffer layout. Each buffer is a one-dimensional array of same-sized elements. The sizes of the individual data elements are always the same as those of the other elements in the same buffer, although if multiple buffers are used, they can have different element sizes. We will see how instanced rendering is performed in more detail in Chapter 3.

Vertex buffer uses. As mentioned above, a vertex buffer's primary purpose is to provide per-vertex information to the pipeline, either directly, in multiple buffer configurations, or through instancing. To this end, the primary place to bind a vertex buffer to the pipeline is in the input assembler stage, which serves as the entry point into the pipeline. In addition to the input assembler stage, vertex buffers can also be attached to the stream output stage to allow the rendering pipeline to stream vertex data into the buffer, so that data can be used in subsequent rendering passes. Both of these stages of the pipeline will be discussed in more detail in Chapter 3, "The Rendering Pipeline," but these connections points are highlighted on the pipeline schematic in Figure 2.9 for easy reference in the future.

Creating vertex buffers. As described in the "Resource Creation" section of this chapter, for each type of location in the pipeline that a resource can be bound to, there is a corresponding bind flag that must be set at resource creation time in order to allow the resource to be bound there. With this in mind, a vertex buffer must always have the D3D11_BIND_VERTEX_BUFFER bind flag set. It can also optionally include the D3D11_BIND_STREAM_OUTPUT bind flag if it will be used for streaming vertex data out of the pipeline.

In addition to the choice of bind flags, the other major consideration for vertex buffers revolves around the usage scenario that the buffer will experience. Depending on whether or not the vertex buffer's contents will be changing frequently or not, and on if those changes will be coming from the CPU or GPU, different usage flags will be needed. For example, if the data that will be loaded into the buffer will be static, the buffer resource should be created with the D3D11_USAGE_IMMUTABLE usage flag. In this case, when the buffer is created it will be initialized with the vertex data through the D3D11_SUBRESOURCE_DATA parameter passed to the creation method and will never be modified again. An example of this type of vertex buffer would be used to hold the contents for a static terrain mesh.

However, if the buffer will be updated frequently by the CPU, the buffer should be created with D3D11_USAGE_DYNAMIC, and also with a CPU write flag for the CPU access flag parameter. An example of this type of vertex buffer usage is when vertex transformations are performed on the CPU instead of on the GPU. These updates would then be copied into the buffer resource every frame. This is a common technique used to condense many *draw* calls into one by putting all of the model data into the same frame of reference, which is typically either world space or view space. Still a third usage type would be to

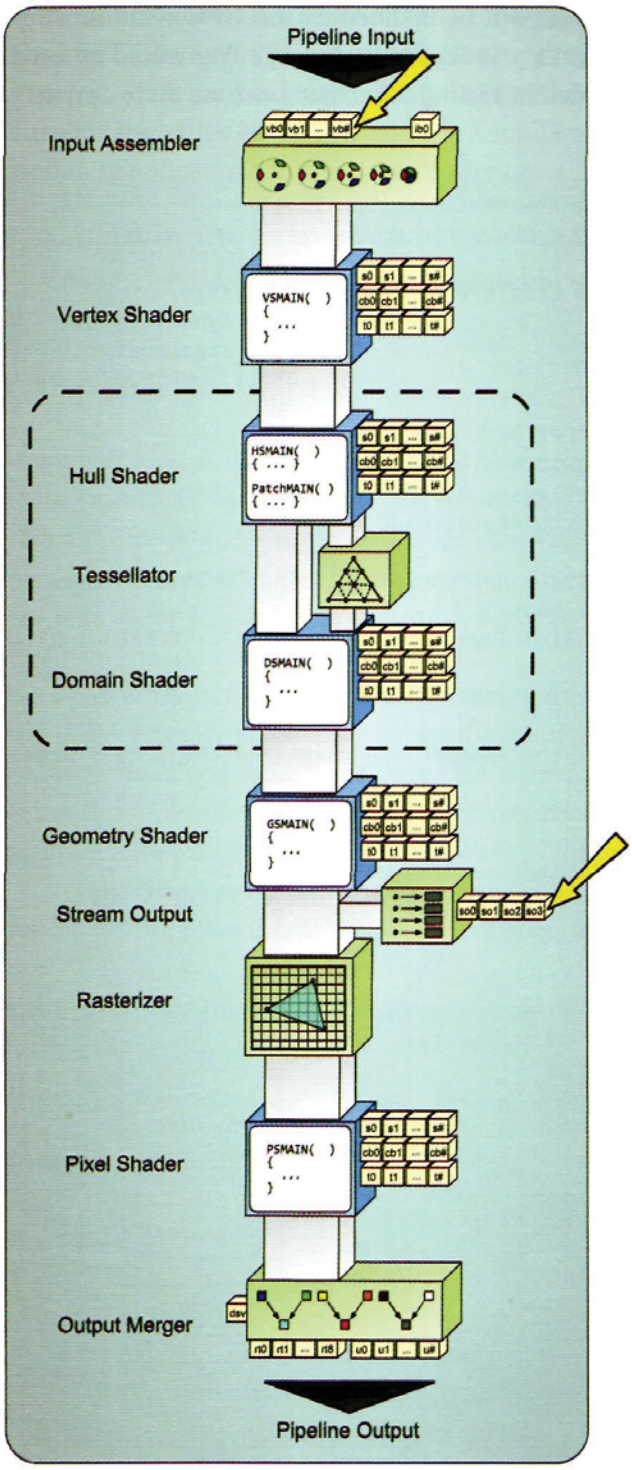

Figure 2.9. The available pipeline binding locations for vertex buffers.

create a buffer that will be updated by the GPU with the stream output functionality. In this case, the D3D11_USAGE_DEFAULT usage flag would be used. A sample buffer creation process is provided in Listing 2.6 to demonstrate these options.

```
// Initialize the device using it...
ID3DIIDevice* g_pDevice = 0;

ID3DIIBuffer* CreateVertexBuffer( UINT size,
                                  bool dynamic,
                                  bool streamout,
                                  D3D11_SUBRESOURCE_DATA* pData )
{
    D3D11_BUFFER_DESC desc;
    desc.ByteWidth = size;
    desc.MiscFlags = 0;
    desc.StructureByteStride = 0;

    // Select the appropriate binding locations based on the passed in flags
    if ( streamout )
        desc.BindFlags = D3D11_BIND_VERTEX_BUFFER | D3D11_BIND_STREAM_OUTPUT;
    else
        desc.BindFlags = D3D11_BIND_VERTEX_BUFFER;

    // Select the appropriate usage and CPU access flags based on the passed
    // in flags
    if ( dynamic )
    {
        desc.Usage = D3D11_USAGE_DYNAMIC;
        desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
    }
  else
    {
        desc.Usage = D3D11_USAGE_IMMUTABLE;
        desc.CPUAccessFlags = 0;
    }
    // Create the buffer with the specified configuration
    ID3D11Buffer* pBuffer = 0;
    HRESULT hr = g_pDevice->CreateBuffer( &desc, pData, &pBuffer );

    if ( FAILED( hr ) )
    {
        // Handle the error here...
        return( 0 );
    }

    return( pBuffer );
```

Listing 2.6. A method for creating vertex buffers with various usage considerations.

From this listing, we can see how the buffer description is configured differently for different usage scenarios. The size of the buffer elements and the overall size of the buffer remain the same in both scenarios, while the bind flags, usage flags, and CPU access flags all vary, depending on the intended use of the buffer. We will see a similar pattern throughout each of the buffer description specifications for the other buffer types.

Resource view requirements. Vertex buffers are bound directly to either the input assembler stage or the stream output stage. Because of this, there is no need to create a resource view when using it.

Index Buffers

The second buffer type that we will look at is the *index buffer*. The index buffer provides the very useful ability to define primitives by referencing the vertex data stored in vertex buffers. More or less, the index buffer provides a list of indices that point into the list of vertices. Depending on the desired type of primitive (such as points, lines, and triangles) appropriately sized groups of indices are formed to define which vertices that primitive is made up of. Figure 2.10 depicts this operation visually.

The use of index buffers can potentially provide a significant reduction in the total number of vertices that need to be defined. Since each adjacent primitive definition can reference the same vertex data as its neighboring primitives, the shared vertices do not need to be repeated in the vertex buffer. In addition, this sharing of vertices allows multiple primitives to use the same output vertex from a vertex shader. We will discuss the vertex shader in more detail in Chapter 3, "The Rendering Pipeline," but for now, we just need to understand that after a vertex has been run through the vertex shader, the result can be

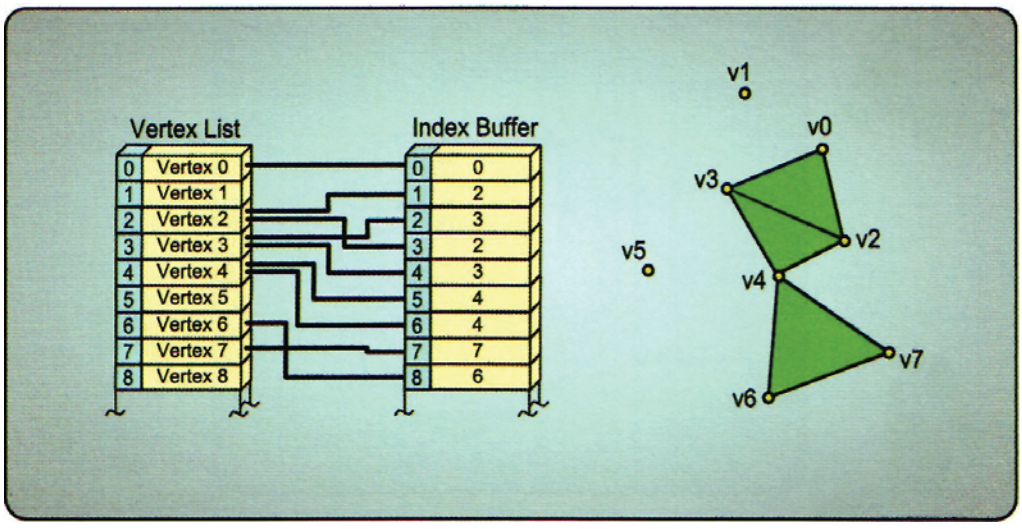

Figure 2.10. An index buffer creating triangles by referencing vertices.

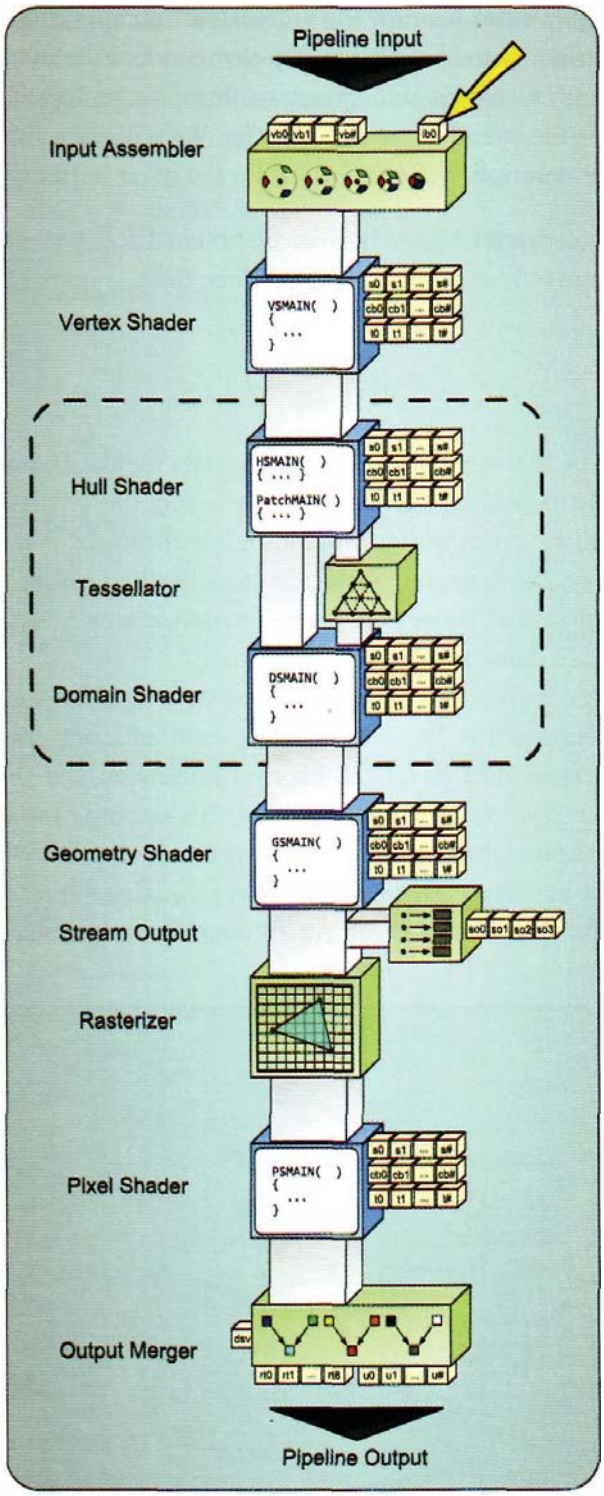

Figure 2.11. The available pipeline binding locations for index buffers.

cached and reused by multiple primitives. This can potentially reduce the amount of vertex shader processing required for a given model.

Using index buffers. Since the index buffer specifies which vertices are to be used in the primitive setup operations, you must know ahead of time what primitive topology you will be using—otherwise you wouldn't know which order to put the indices into. This is typically determined well in advance, with the selection of the rendering algorithm and the geometry loading routines. During the pipeline configuration when preparing to perform a *draw* operation, the desired index buffer is bound to the *input assembler* stage, where it will be used to generate the input primitives for the pipeline. Since these buffers serve a very specific purpose, they are normally not bound to the pipeline in other locations. This binding location is shown in Figure 2.11.

Creating index buffers. When creating an index buffer, we follow the standard buffer creation process and fill in a D3D11_BUFFER_DESC structure. The description structure of an index buffer does not change very frequently from buffer to buffer, since this type of data would normally be defined once in a content creation program and then exported as is. The index buffer indices normally would not change after the application startup phase. However, if some new algorithm does require dynamic updating of an index buffer, it can also be created with the dynamic update properties discussed in the "Vertext Buffers" section of this chapter. This could be used in the draw call reduction scheme discussed in the "Vertext Buffers" section, where multiple sets of geometry are dynamically grouped together into a single set of vertex and index buffers. A typical creation sequence is provided in Listing 2.7.

```
ID3DIIBuffer* CreateIndexBuffer( UINT size,
                                 bool dynamic,
                                 D3D11_SUBRES0URCE_DATA* pData )
{
    D3D11_BUFFER_DESC desc;
    desc.ByteWidth = size;
    desc.MiscFlags = 0;
    desc.StructureByteStride = 0;
    deSC.BindFlagS = D3D11_BIND_INDEX_BUFFER;

    // Select the appropriate usage and CPU access flags based on the passed
    // in flags
    if ( dynamic )
    {
        desc.Usage = D3D11_USAGE_DYNAMIC;
        desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
    }
    else
```

```
    {
        desc.Usage = D3D11_USAGE_IMMUTABLE;
        desc.CPUAccessFlags = 0;
    }

    // Create the buffer with the specified configuration
    ID3DllBuffer* pBuffer = 0;
    HRESULT hr = g_pDevice->CreateBuffer( &desc, pData, &pBuffer );

    if ( FAILED( hr ) )
    {
        // Handle the error here...
        return( 0 );
    }
    return( pBuffer );
}
```

Listing 2.7. A method for creating index buffers depending on their usage.

The first item to specify is the size of the buffer in bytes. As can be seen in the code listing, the index buffer is always created with the D3D11_BIND_INDEX_BUFFER bind flag. When a static index buffer is desired, the usage flag is specified as D3D11_USAGE_IMMUTABLE, without any CPU access flags set. In this case, the intended contents of the buffer must be provided to the creation call with the D3D11_SUBRESOURCE_DATA structure. If a dynamic buffer is required, we would choose the D3D11_USAGE_DYNAMIC, along with the D3D11_CPU_ACCESS_WRITE CPU access flag.

Resource view requirements. Index buffers are bound directly to the input assembler stage, without the aid of a resource view. Because of this, there is no need to create a resource view to use index buffers.

Constant Buffers

The *constant buffer* is the first resource type that we have encountered that is accessible from the programmable shader stages and is subsequently used in HLSL code. A constant buffer is used to provide constant information to the programmable shader programs being executed in the pipeline. The term *constant* refers to the fact that the data inside this buffer remains constant throughout the execution of a *draw* or *dispatch* call. Any information that only changes between pipeline invocations, such as a world transformation matrix or object color, is supplied to the shader program in a constant buffer. This mechanism is the primary means of data transfer from the host application to each of the programmable shader stages. The type and amount of information contained within a constant buffer may vary from buffer to buffer. This depends completely on the data needed for each particular shader program, and is defined by a structure declaration in the shader program. The buffer

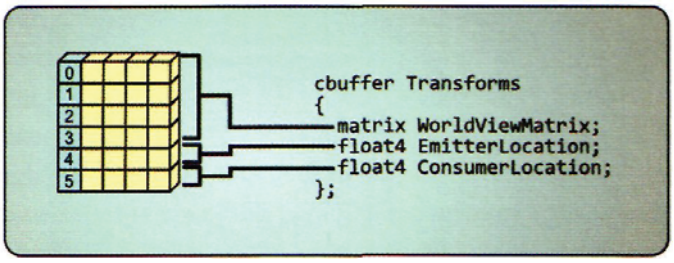

Figure 2.12. A sample constant buffer showing the contents of its structure.

can be tailored to more or less any combination of the basic HLSL types, as well as to structures composed of these basic types. We will cover the HLSL data types in more detail in Chapter 6, "High Level Shading Language," but we'll mention now that these types include scalars, vectors, matrices, arrays of these types, class instances, and combinations of each of these types within structures. A single combination of these types is depicted in Figure 2.12.

The constant buffer is somewhat different than the other buffers that we have discussed thus far. Both the vertex buffer and the index buffer define a basic data element and then repeat that element many times in an array-like fashion. The constant buffer defines a basic element, but it doesn't provide more than one instance of the element—the buffer is created large enough to fit the desired information, but not larger. This implies that the constant buffer implements a structure instead of an array.

Using constant buffers. Each of the programmable pipeline stages can accept one or more constant buffers. It then makes the information in the buffers available for use in the shader program. The data is accessed as if the structure contents were declared globally in the shader program. This means that the elements of each structure must have unique names at this pseudo-global scope. This ability provides a significant amount of flexibility for adding variation to any of the shader programs that are required for a particular rendering algorithm. It is important to note that a buffer created for binding as a constant buffer may not be bound to any other type of connection points on the pipeline. In practice, this is not a problem, since the contents of a constant buffer are already available to all of the programmable pipeline stages anyway. The locations on the pipeline that are available for binding constant buffers are highlighted in Figure 2.13.

The ability to use relatively large constant buffers does not mean that you should automatically create one large buffer for all of the variables that are needed in a shader program. Since constant buffers are updated between uses by the CPU, their contents must be uploaded to the GPU after any changes. Let's assume a hypothetical situation as an example. Suppose there are ten parameters needed by a shader program, but only two of them change between each time that an object is rendered. In this case, we would be writing all

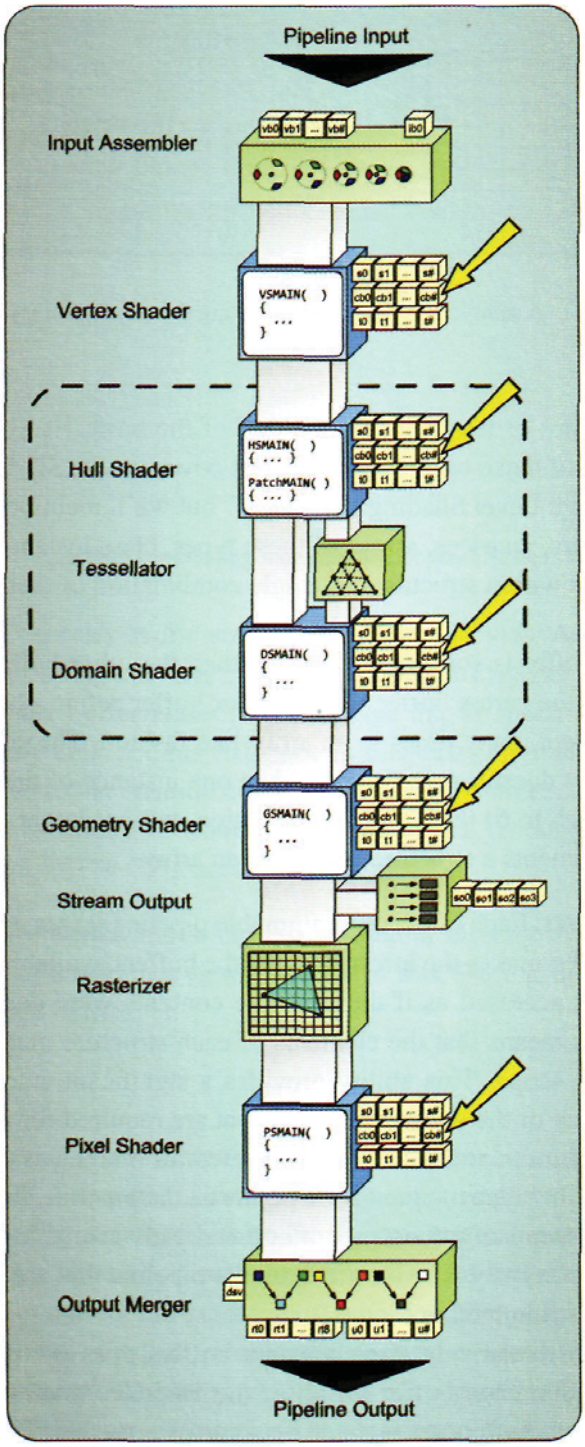

Figure 2.13. The available binding locations for constant buffers in the pipeline.

ten parameters to the buffer between every draw call, just to update the two varying parameters, since the entire buffer is always completely updated. Depending on the number of objects to render, these additional parameter updates can add up to a large amount of unnecessarily wasted bandwidth. If we instead create two constant buffers, where one of them holds the eight static parameters and the second one holds the two dynamic parameters, we can update only the changing parameters and significantly reduce the volume of the required data update for each frame. However, we would need to take care to ensure that the buffers are only updated when they really need to be!

Another potential reduction in updating constant buffers comes from the fact that these buffers only support read accesses from the shader programs. Since it is read-only, a single constant buffer can be simultaneously bound to multiple locations in the pipeline, without the possibility of causing memory access conflicts.

Creating constant buffers. Constant buffers also provide a number of different resource configurations for the best possible performance. Depending on how frequently a constant buffer will be updated by the CPU, one of two typical configurations is chosen. If constant buffers are updated many times throughout an application's lifetime, a dynamic buffer resource makes the most sense. Of course, if a constant buffer will contain some data that will not change throughout an application (such as a fixed back buffer size) it would be more efficient to create it as a completely static buffer, with immutable usage flags. Listing 2.8 demonstrates how a constant buffer is typically created.

```
ID3DIIBuffer* CreateConstantBuffer( UINT size,
                                    bool dynamic,
                                    bool CPUupdates,
                                    D3D11_SUBRESOURCE_DATA* pData )
{
    D3D11_BUFFER_DESC desc;
    desc.ByteWidth = size;
    desc.HiscFlags = 0;
    desc.StructureByteStride = 0;
    desc.BindFlags = D3Dll_BIND_CONSTANT_BUFFERj

    // Select the appropriate usage and CPU access flags based on the passed
    // in flags
    if ( dynamic && CPUupdates )
    {
        desc.Usage = D3D11_USAGE_DYNAMIC;
        desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
    }
    else if ( dynamic && !CPUupdates )
    {
        desc.Usage = D3D11_USAGE_DEFAULT;
        desc.CPUAccessFlags = 0;
    }
    else
```

```
    {
        desc.Usage = D3D11_USAGE_IMMUTABLE;
        desc.CPUAccessFlags = 0;
    }

    // Create the buffer with the specified configuration
    ID3D11Buffer* pBuffer = 0;
    HRESULT hr = g_pDevice->CreateBuffer( &desc, pData, &pBuffer );

    if ( FAILED( hr ) )
    {
        // Handle the error here...
        return( 0 );
    }

    return( pBuffer );
}
```

Listing 2.8. A method for creating a constant buffer depending on its intended usage.

As with vertex and index buffers, the creation of dynamic constant buffers uses the D3D11_USAGE_DYNAMIC usage flag, in combination with the D3D11_CPU_ACCESS_WRITE flag, to allow the CPU to update the resource at runtime. For static contents, the D3D11_USAGE_IMMUTABLE usage is used without any CPU access flags. One additional possibility is to have a buffer that is updated at runtime only by the GPU, such as when copying the number of elements in an Append/Consume buffer to a constant buffer with the ID3D11 DeviceContext: :CopyStructureCount() method. In this case, we would require a default usage flag, but we would not set any of the CPU access flags. This allows the runtime to optimize the created resource for use on the GPU only.

There are several items of interest that are not shown in Listing 2.8. The first thing to notice is that the application typically defines a C++ structure that mirrors the desired contents of the constant buffer in HLSL. This allows the application updates to be applied to a system memory instance of this structure, which can then be directly copied into the buffer. We will explore the methods of updating the contents of resources later in this chapter. The second point of interest is that a constant buffer must be created with a ByteWidth that is a multiple of 16 bytes. This requirement allows for efficient processing of the buffer with the 4-tuple register types of the GPU, and it only appears as a requirement for constant buffers. This must be accounted for in the structure declaration within C/C++. The third interesting point is that the buffer description is not allowed to contain any other bind flags than the D3D11_BIND_CONSTANT_BUFFER. Once again, in practice this is not really a big restriction, since there typically are no situations where it would be desirable to bind a constant buffer to another location in the pipeline.

Resource view requirements. Although constant buffers are bound to the programmable shader stages and are accessible through HLSL, their contents are not interpreted with

resource views in any way. They are specified exactly as they should be made available within HLSL, so there is no need to use a resource view.

HLSL constant buffer resource object. This is the first buffer type we have encountered that is directly accessible from within HLSL code in the programmable shader stages. In HLSL, constant buffers are resource objects that are declared with the cbuff er keyword. A sample declaration is provided in Listing 2.9.

```
cbuffer Transforms
{
      matrix WorldMatrixj
      matrix ViewProjMatrix;
      matrix SkinMatrices[26];
};

cbuffer LightParameters
{
      float3 LightPositionWS;
      float4 LightColor;
};

cbuffer ParticleInsertParameters
{
      float4 EmitterLocation;
      float4 RandomVector;
};
```

Listing 2.9. Various declarations of a constant buffer from within HLSL.

Each of the variables declared within this cbuffer structure can be directly used within the HLSL shader program as if it were declared at a global scope. The name of the constant buffer, as specified in the listing above, is used by the host application to identify the buffer by name, and to load the appropriate contents into it. However, the buffer name is not used within HLSL. As with the constant buffer name, the names and types of the individual elements of the constant buffer are also accessible through the shader reflection API.[1] This series of methods can be used to determine the name and type of each subparameter, to allow the application to know which information to insert into the buffer at runtime.

Buffer/Structured Buffer Resources

Our next type of buffer is referred to by two different names, depending on what type of data it holds. A *standard buffer resource* means a buffer whose elements are one of the

[1] The shader reflection API is discussed in more detail in Chapter 6: High Level Shading Language.

built-in data types. In this way, a standard buffer resource is similar to an array of values. Each value is stored at a unique array location and can be referenced by the index of its location. This allows the resource to be easily accessed by many different instances of a shader program simultaneously on the GPU. Since each element is uniquely identified, the developer can easily structure the program to avoid any memory collisions. Figure 2.14 graphically represents a buffer resource.

Very similar to the buffer resource is a structured buffer resource. The only difference between the two is that a structured buffer allows the user to define a structure as the basic element, instead of one of the built-in data types. This makes mapping the data of a particular processing problem to a resource relatively simple. If the developer can define a suitable structure in C++, a corresponding structure in HLSL can be defined, and the buffer will contain an array of these structures, which can be used by the programmable shader stages as a resource object.

The structured buffer is intended to provide a flexible memory resource for simplifying custom algorithm development. Since the data format within the structure can use any of the available types within HLSL, it can be customized to suit a particular scenario. The available structure formation is quite similar to that of the constant buffer, except that the structured buffer provides an array of the desired structures, instead of a single instance like the constant buffer. This concept is depicted graphically in Figure 2.15. In this figure, a hypothetical structure is shown that could represent the data for a particle in a particle system.

Here we see that the particle structure uses multiple variables, and that the complete particle system is contained within the structured buffer. A very interesting point to note is that these buffer types are the first resources that we have discussed that can be used for writing, as well as reading, by the programmable pipeline stages. We will explore the details of how to use this functionality throughout the remainder of this section.

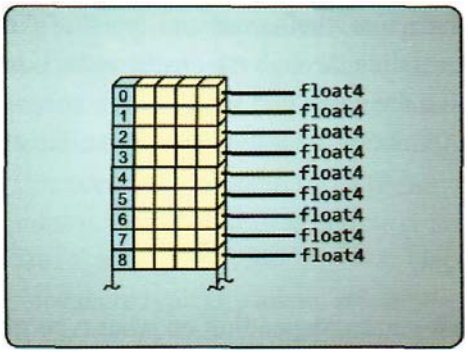

Figure 2.14. A depiction of a standard buffer resource.

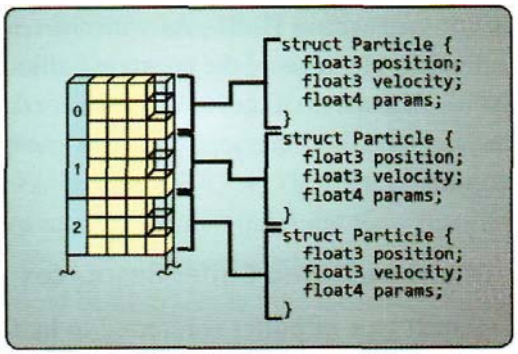

Figure 2.15. A structured buffer used to contain data from a particle system.

Using **buffers/structured buffers.** Due to this ability to provide a large amount of structured data, the buffer and structured buffer are great options for larger data structures that are to be accessed by the programmable shader stages. These buffers are the first resources that are attached to the pipeline with resource views instead of being directly bound. These buffers are available with read access to all of the programmable pipeline stages, and read/write access is also possible in the pixel and compute shader stages, depending on which resource view is used. This means that these resources can potentially provide a means of communication between the various pipeline stages, as well as between the individual processing elements within the same pipeline stage.

A pipeline stage's resource access capabilities are determined by which type of resource view is used to bind the resource to the pipeline. Unlike the constant buffer, the buffer/structured buffer must be bound to the pipeline through a resource view, which means that it must be created with an appropriate bind flag for binding with either a shader resource view, an unordered access view, or both. The shader resource view allows binding to all of the programmable pipeline stages, while the unordered access view is only allowed to be bound to the pixel and compute shader stages. Since an unordered access view is required to perform write accesses to a resource, the available binding locations for UAVs effectively determine where these write accesses can be performed. The available connection points for both shader resource views and unordered access views are shown in Figure 2.16. Like a constant buffer, a buffer/structured buffer can be bound to multiple locations when used for read-only uses. This is done by binding it to the pipeline with a shader resource view. However, when an unordered access view is used, it may only be bound to a single location. This is enforced by the runtime, which will print error messages if the resource is bound to multiple locations for writing. The use of an unordered access view also excludes simultaneous use of shader resource views, due to the read-after-write issues that it could introduce.

Creating buffers/structured buffers. As we have seen in the other buffer resource creation discussions, it must be determined which usage scenario the buffers/structured buffers will experience. There are in general three different types of usage patterns for these resources. The first case is that the data within the buffer will be static throughout the lifetime of the application. An example of this would be to use the buffer/structured buffer resource to hold precalculated data, such as precomputed radiance transfer data or some other form of a lookup table. The second case is when data needs to be loaded periodically into the buffer by the CPU. This is a likely scenario in many general purpose computation systems, where the GPU is being used as a co-processor to the CPU and is continually feeding the GPU with new data to process. In both of these cases, the GPU is only allowed to read the buffer data and is not able to write to the resource.

The final case arises when the buffer contents are updated in some way by the GPU itself. A common example of this can be found in situations where the GPU is performing

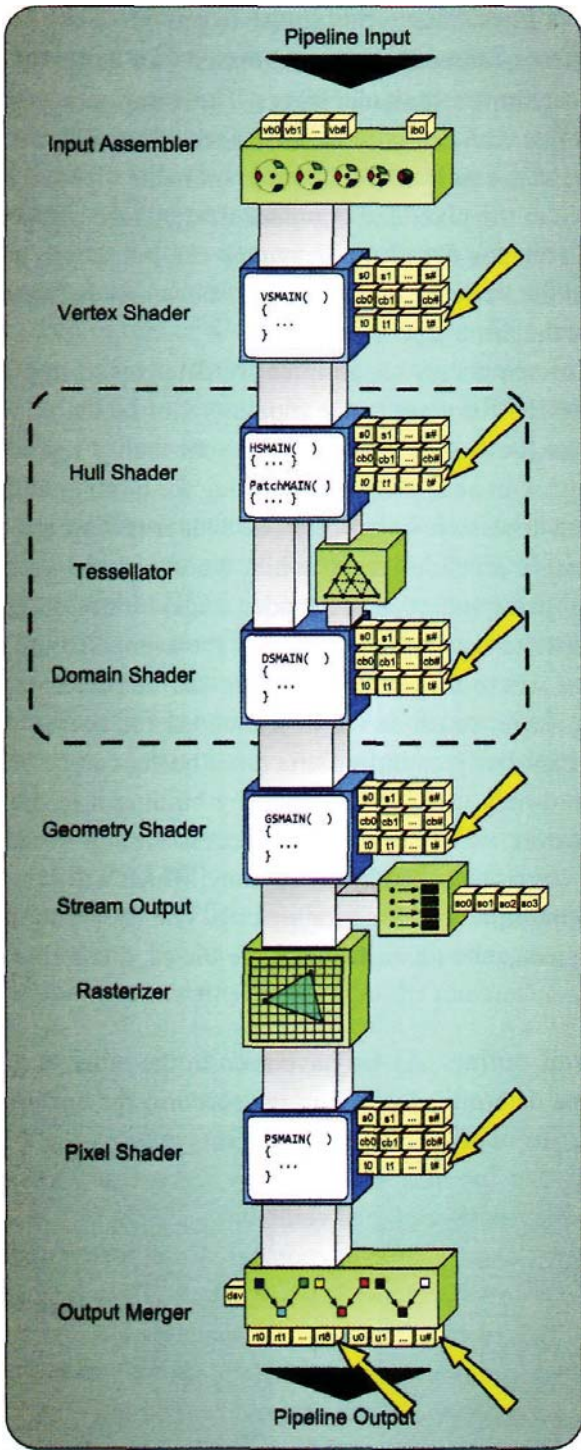

Figure 2.16. The available connection points for the buffer and structured buffer resource types.

some type of physical simulation that is later used for rendering a representation of the results of the simulation. In this case, the GPU will require read and write access to the buffer so that it can read the contents, perform some updates, and then write the results back into the buffer. Each of these three scenarios will require different settings for the usage, CPU Access, and bind flags used when creating the buffer.

In addition to these flags, we must also provide some additional size information when using structured buffers. All types of buffers have required the specification of the ByteWidth parameter, which indicates the desired size of the buffer in bytes. Structured buffers require us to also specify the size of the structure element being used. This information is used as the step size when the resource is accessed by one of the programmable pipeline stages. The other requirement for using structured buffers is to indicate that the buffer resource will indeed be used as a structured buffer. This is done by setting the D3D11_RESOURCE_MISC_BUFFER_STRUCTURED miscellaneous flag. A typical sample creation is shown in Listing 2.10.

```
ID3DllBuffer* CreateStructuredBuffer( UINT count,
                                      UINT structsize,
                                      bool CPUWritable,
                                      bool GPUWritable,
                                      D3D11_SUBRES0URCE_DATA* pData )
{
    D3D11_BUFFER_DESC desc;
    desc.ByteWidth = count * structsize;
    desc.MiscFlagS = D3D11_RES0URCE_MISC_BUFFER_STRUCTURED;
    desc.StructureByteStride = structsize;

    // Select the appropriate usage and CPU access flags based on the passed in flags
    if ( !CPUWritable && IGPUWritable )
    {
        desc.BindFlagS = D3D11_BIND_SHADER_RES0URCE;
        desc.Usage = D3D11_USAGE_IMMUTABLE;
        desc.CPUAccessFlags = 0;
    }
    else if ( CPUWritable && IGPUWritable )
    {
        desc.BindFlagS = D3D11_BIND_SHADER_RES0URCE;
        desc.Usage = D3D11_USAGE_DYNAMIC;
        desc.CPUAccessFlags = D3D11_CPU_ACCESS_WRITE;
    }
    else if ( ICPUWritable && GPUWritable )
    {
        desc.BindFlagS = D3D11_BIND_SHADER_RES0URCE |
                         D3D11_BIND_UN0RDERED_ACCESS;
        desc.Usage = D3D11_USAGE_DEFAULT;
        desc.CPUAccessFlags = 0;
    }
    else if ( CPUWritable && GPUWritable )
```

```
    {
        // Handle the error here...
        // Resources can't be writable by both CPU and GPU simultaneously!
    }
    // Create the buffer with the specified configuration
    ID3DllBuffer* pBuffer = 0;
    HRESULT hr = g_pDevice->CreateBuffer( &desc, pData, &pBuffer );

    if ( FAILED( hr ) )
    {
        // Handle the error here...
        return( 0 );
    }
    return( pBuffer );
}
```

Listing 2.10. A method for creating a structured buffer.

We split the size information into a *count* of the number of structures to include in the buffer, and a `structsize` parameter to indicate the size of each structure in bytes. These are then used to calculate the overall byte width of the buffer. In addition, each of the three usage scenarios discussed above is represented in the code listing; they differentiate themselves with the CPU access, usage, and bind flags.

Resource view requirements. We have already seen that the only resource views that can be used with a buffer or structured buffer are the shader resource view and the unordered access view. In fact, these are the only ways to bind these buffers to the pipeline. For standard buffer types, the format must be provided directly in the SRV description structure. For structured buffer resources, the format should be set to DXGI_FORMAT_UNKNOWN, since there won't be a DXGI format available to match all possible structure types. Instead, the format is derived from the HLSL structure declaration in the shader program, and the size of the elements is provided by the application when the buffer resource is created.

The shader resource view description structure for buffer resources requires specification of the beginning element, and of the number of elements to include in the view. This may select the entire range of elements in the buffer, or it can be used to select a subset of the total resource. This effectively maps the selected data to the [0, width-1] range when viewed from HLSL, with all other accesses considered out of bounds. These values are specified in the D3D11_BUFFER_SRV structure, as shown in Listing 2.11.

```
ID3DllShaderResourceView* CreateBufferSRV( ID3DllResource* pResource )
{
    D3D11_SHADER_RESOURCE_VIEW_DESC desc;
```

```
    // For structured buffers, DXGI_FORMAT_UNKNOWN must be used!
    // For standard buffers, utilize the appropriate format,
    desc.Format = DXGI_FORMAT_R32G32B32_FLOAT;

    desc.ViewDimension = D3D11_SRV_DIMENSION_BUFFER;
    desc.Buffer.ElementOffset = 0;
    desc.Buffer.ElementWidth = 100;

    ID3D11ShaderResourceView* pView = 0;
    HRESULT hr = g_pDevice->CreateShaderResourceView( pResource, &desc,
                                                      &pView );

    return( pView );
```

Listing 2.11. A method for creating a shader resource view for a buffer resource.

By adjusting the ElementOffset and ElementWidth parameters, a single large buffer resource can be used to contain a collection of smaller datasets, each with its own SRV, to provide access to them. By using the shader resource view to restrict access to a particular subset of the resource and creating an appropriate number of shader resource views, the application can effectively control which data is visible to each invocation of a shader program by using a particular shader resource view. It is also possible to use more than one view to simultaneously select different ranges of a buffer.

The unordered access view uses the same element range selection and format specification, and also provides an additional set of flags that can be specified for special-use scenarios. These flags enable the unordered access view to be used for *append/consume buffers* and *byte address buffers*, both of which are specialized ways to use buffer resources through unordered access views. The specific details of how these function are discussed in more detail in the following two sections. Listing 2.12 demonstrates how to create an unordered access view.

```
ID3D11UnorderedAccessView* CreateBufferUAV( ID3DIIResource* pResource )
{
    D3D11_UNORDERED_ACCESS_VIEW_DESC desc;

    // For structured buffers, DXGI_FORMAT_UNKNOWN must be used!
    // For standard buffers, utilize the appropriate format,
    desc.Format = DXGI_FORMAT_R32G32B32_FLOAT;

    desc.ViewDimension = D3D11_UAV_DIMENSION_BUFFER;
    desc.Buffer.FirstElement = 0;
    desc.Buffer.NumElements = 100;

    desc.Buffer.Flags = D3D11_BUFFER_UAV_FLAG_COUNTER;
    //desc.Buffer.Flags = D3D11_BUFFER_UAV_FLAG_APPEND;
    //desc.Buffer.Flags = D3D11_BUFFER_UAV_FLAG_RAW;
```

```
    ID3DllUnorderedAccessView* pView = 0;
    HRESULT hr = g_pDevice->CreateUnorderedAccessView( pResource, &desc,
                                                       &pView );

    return( pView );
}
```

Listing 2.12. A method for creating an unordered access view for a buffer or structured buffer resource.

The listing shows the setting of the D3D11_BUFFER_UAV_FLAG_COUNTER flag, with the append and raw flags commented out. This flag is used to provide a counter for the buffer resource that is accessible through HLSL. This counter is operated by using the methods of the HLSL buffer resource object, as described in the following section.

HLSL structured buffer resource object. Since this type of buffer resource is available in the programmable shader stages, it is necessary to understand how it can be accessed through HLSL. Fortunately, the resource object declaration and usage are fairly straightforward. A structured buffer requires that the corresponding structure be defined in the shader program. Once a suitable structure is available, the structured buffer is declared with a template-like syntax. A sample declaration is shown in Listing 2.13, taken from the fluid simulation demo described in Chapter 12.

```
// Declare the structure that represents one fluid column's state
struct GridPoint
{
    float Height;
    float4 Flow;
};

// Declare the input and output resources
RWStructuredBuffer<GridPoint> NewWaterState     : register( u0 );
StructuredBuffer<GridPoint>   CurrentWaterState : register( t0 );
```

Listing 2.13. A declaration in HLSL of a standard buffer, a structured buffer, and read/write standard and structured buffers.

In this code listing, we see that there are actually two different ways to declare a structured buffer—with StructuredBuffer<> and with RWStructuredBuffero. These two forms represent the differences between a structured buffer bound to the pipeline with a shader resource view (for read-only access) and with an unordered access view (for read and write access), respectively. This is also indicated by the register statements, with the read/write resource using a u# register and the read-only resource using a t# register. The choice of the declaration type will determine which type of resource view the application will have to use to bind the resource to the programmable shader stage for use in this shader program.

Once the object is declared, we can see that there is a simple way to interact with the structured buffer from within HLSL. The individual elements are accessed with an array

like syntax, using brackets to indicate which index to use. This operates very similarly to C++, and the individual members of a structure can be accessed with a dot operator. When the resource is bound with a shader resource view, the elements can only be read. However, with the unordered access view, they can also be written to, using the same syntax. When a subset of a buffer resource is selected with the resource view, its elements appear to be remapped, such that the first element in the subrange is accessed with index 0, and each subsequent element is accessed with an incremented index.

These HLSL resource objects also provide a GetDimensions() method that returns the size of a buffer, or the size of the buffer and the structure element size for structured buffers. These can be used by the shader programs to implement range checking. It is also possible to use a counter variable that is built into a RWStructuredBuffer if the UAV is created with the D3D11_BUFFER_UAV_FLAG_COUNTER flag (as described above in the "Creating Buffers/Structured Buffers" section of this chapter). If a UAV is created with this flag set, the HLSL program can use the IncrementCounter() and DecrementCounter() methods. These can be used to implement customized data structures by providing a total count of the elements that have been stored in the structured buffer. Using this functionality requires setting the UAV format to be DXGI_FORMAT_R32_UNKNOWN.

Append/Consume Buffers

The *append* and *consume* buffer type provide a special variation of a structured buffer resource. In essence, these are both buffers that require unordered access views, but they provide new access methods in HLSL to implement a stack-like behavior. They simply use special unordered access view creation flags that provide a new method for accessing their contents. More specifically, these access methods are used to push elements into the buffers (using the append() HLSL method) and pull elements out of them (using the consume() HLSL method). In this way, a resource can be used to accumulate data, as well as to distribute data. The order in which elements are added to the buffer is not important, but the number of elements added is. The unordered access view maintains an internal count of the number of items that have been added or removed from the buffer. This allows the runtime and driver to perform more efficiently in some cases, since the ordering of the elements within the buffer does not need to be synchronized between individual GPU threads of execution.

Using append/consume buffers. This functionality can best be explained with an example. In this scenario, a shader program can be used to implement a GPU-based particle system. In fact, this is one of the example algorithms provided later in this book. The particle system would use two buffer resources to hold its particle information. One buffer would hold the current state of the particles, and the other would be used to receive the updated particles. When running, a compute shader program would read one particle per thread from the current state buffer using the consume functionality, then perform the update procedure on the particle, and finally, append the resulting particle state to the output

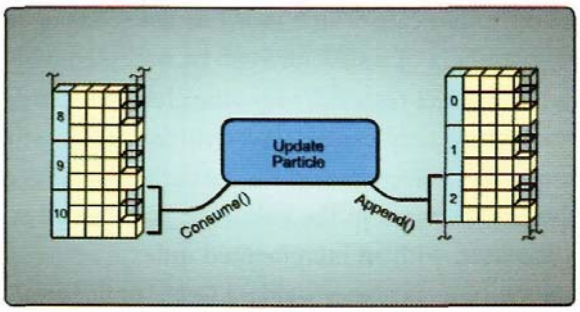

Figure 2.17. A particle system implementation using "append" and "consume" buffers.

buffer. Since each particle is updated in isolation from the others, the particle order within the buffer is completely irrelevant to the updating procedure. In addition, since the buffers maintain a count of the number of elements contained in the buffers, the update procedure can add or remove particles as needed. The buffers' internal counter tracks the total number of elements that are currently residing in the buffer. Because of this, there is no need to synchronize between GPU threads for accessing the correct number of particle elements. This particle system example is depicted in Figure 2.17.

Creating append/consume buffers. To create a buffer that can be used as an append or consume buffer, we require a slightly more restrictive creation process. Since the buffers are intended for read/write access by the programmable shader stages, we require the D3D11_USAGE_DEFAULT usage flag to be used. In addition, the bind flag must include the D3D11_BIND_UNORDERED_ACCESS flag to allow use of an unordered access view and will typically also include the D3D11_BIND_SHADER_RESOURCE flag as well. This also indicates that the CPU is not capable of directly accessing the buffer contents, due to the default usage flag. The remainder of the configurations can be selected as we have already seen in the "Buffer/Structured Buffer Resources" section. An example creation method is shown in Listing 2.14.

```
ID3D11Buffer* CreateAppendConsumeBuffer( UINT size,
                                         UINT structsize,
                                         D3D11_SUBRESOURCE_DATA* pData )
{
    D3D11_BUFFER_DESC desc;
    desc.ByteWidth = size * structsize;
    desc.MiscFlags = D3D11_RESOURCE_MISC_BUFFER_STRUCTURED;
    desc.StructureByteStride = structsize;

    // Select the appropriate usage and CPU access flags based on the passed
    // in flags
    desc.BindFlags = D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_UNORDERED_ACCESS;
    desc.Usage = D3D11_USAGE_DEFAULT;
    desc.CPUAccessFlags = 0;
```

```
    // Create the buffer with the specified configuration
    ID3D11Buffer* pBuffer = 0;
    HRESULT hr = g_pDevice->CreateBuffer( &desc, pData, SpBuffer );

    if ( FAILED( hr ) )
    {
        // Handle the error here...
        return( 0 );
    }

    return( pBuffer );
}
```

Listing 2.14. A method for creating a buffer resource to be used as an append/consume buffer.

One can see that this is more or less one of the configurations that was already available for creating a buffer/structured buffer resource, with the exception that it is mandatory to use the default usage flag. This means that the append/consume buffer usage does not require a specially created buffer resource, but instead, it is the unordered access view that must be specially configured to provide the additional functionality. The UAV used to bind the buffer resource to the pipeline as an append/consume buffer must be created with the D3D11_BUFFER_UAV_FLAG_APPEND flag specified in its description structure.

Resource view requirements. When creating the resource view to be used with the append/consume buffer, there are a few differences from the previously discussed resource views. The format parameter of the resource view must be set to DXGI_FORMAT_R32_UNKNOWN to use the append/consume functionality. In addition, the unordered access view must be created with the D3D11_BUFFER_UAV_FLAG_APPEND for both append and consume buffers. These requirements are the only additional settings for creating an unordered access view for using a buffer resource as an append/consume buffer.

As we have mentioned in previous buffer discussions, it is possible to simultaneously use multiple resource views on the same resource. Unfortunately, this is not true for the unordered access views. An entire buffer resource is considered to be composed of a single subresource. The unordered access views allow read and write access to their resources, and are not allowed to have more than one writable resource view attached to a single subresource at the same time. This means that, for example, it is not possible to use a single buffer with two unordered access views to simultaneously append and consume from different regions of the buffer. In this case, the append and consume UAVs would need to be attached to separate buffer resources

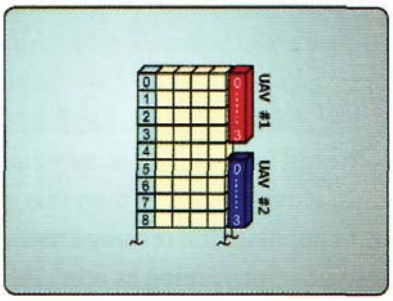

Figure 2.18. A single buffer resource being used by multiple unordered access views.

to be used at the same time. Figure 2.18 shows graphically how a buffer can be referenced simultaneously by more than one resource view.

HLSL append/consume buffer resource objects. Once the buffers have been created with the appropriate resource views, they can be bound to the pipeline and used from within a shader program. The syntax for declaring append and consume buffers is provided in Listing 2.15.

```
struct Particle
{
    float3 position;
    float3 velocity;
    float  time;
};

AppendStructuredBuffer<Particle>  NewSimulationState : register( u0 );
ConsumeStructuredBuffer<Particle> CurrentSimulationState : register( u1 );
```

Listing 2.15. The declaration of append and consume buffers in HLSL.

Probably the next most frequently used methods for append and consume buffers are the actual append() and consume() resource object methods. These operate as would be expected, with the desired value being returned in the case of consume(), and being pushed into the buffer in the case of append(). We can also see in the listing that the GetDimensions() method is also available on the append and consume buffers. This can be used from within HLSL to read the number of elements that are available in the buffer, and to subsequently ensure that data is not appended to the buffer too many times, which would result in loss of the additional data.

Byte Address Buffers

Still another variation of the buffer resource type is available. The *byte address buffer* provides a much more raw memory block for the HLSL program to manipulate. Instead of using a fixed structure size to determine where to index within the resource, a byte address buffer simply takes a byte offset from the beginning of the resource and returns the four bytes that begin at that offset as a 32-bit unsigned integer. Since the returned data is always retrieved in four-byte increments, the requested offset addresses must also be a multiple of four. However, other than this minimum size requirement, the HLSL program is allowed to manipulate the resource memory as it sees fit. The returned unsigned integer values can also be reinterpreted as other data types by some of the type-converting intrinsic functions.

It may not be immediately obvious what the utility of such a buffer would be. Since the HLSL program can interpret and manipulate memory contents as it sees fit, there is no requirement for each data record to be the same length, as we have seen in the various

types of structured buffers. With variable record lengths, an HLSL program can be written to implement almost any data structure that will fit within the bounds of the resource. It would be equally simple to create a variable-element-sized linked list as it would be to create an array-based binary tree. As long as the program implements the accessing semantics for the data structure, there is complete freedom to use memory as desired. This is a very powerful feature indeed, and it opens the doorway for a very large class of algorithms that either were impossible to implement on the GPU before, or were unwieldy to implement.

Using byte address buffers. As mentioned above, byte address buffers are intended to allow the developer to implement custom data structures in buffer resources. The use of the memory block is then, by definition, open to interpretation by the algorithm that it is being used in conjunction with. For example, if a linked-list data structure will be used to store a list of 32-bit color values, each link node will consist of a color value followed by a link to the next element. When the first element is added, the color value is written into the memory location at offset 0, and the link to the next element is initialized to -1, since the next element has not been added yet. When another element is to be added to the list, the HLSL program would start indexing the memory at location 0 and read two 32-bit elements worth of data—one for the color value and one for the link to the next element. If the link is set to -1, the program has found the tail of the list and can add another element at the end. This simple scenario is shown in Figure 2.19.

With these basics of a linked list available, the program can insert or remove nodes in the same way as in traditional CPU-based programs. This provides a completely generic way to use the resources from within the GPU. However, the parallel nature of the GPU can also introduce some complexity into the use of these resources. Since the memory access patterns are defined by the HLSL program, it is necessary to ensure that memory is accessed in a coherent and safe manner between multiple threads of execution. We will discuss the details of this type of synchronization in Chapter 5, "The Computation Pipeline," but there is a range of techniques available to the developer to ensure thread-safe access to the resources.

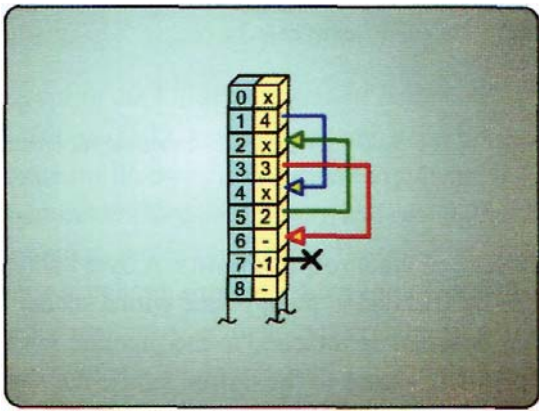

Figure 2.19. A linked-list implementation using byte address buffers.

Creating byte address buffers. Creating a byte address buffer is very similar to the process we have seen for the previous buffer types. It must also indicate the usage scenario that it will be used for, and at what locations it will be bound to the pipeline with its

bind flags. A byte address buffer must also be created with the D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS miscellaneous flag. An example method for creating such a resource is provided in Listing 2.16.

```
ID3DHBuffer*  CreateRawBuffer(  UINT  size,
                                bool  GPUWritable,
                                D3D11_SUBRESOURCE_DATA*  pData  )
{
    D3D11_BUFFER_DESC desc;
    desc.ByteWidth = size;
    desc.MiscFlagS = D3D11_RESOURCE_MISC_BUFFER_ALLOW_RAW_VIEWS;
    desc.StructureByteStride = 0;

    // Select the appropriate usage and CPU access flags based on the passed
    // in flags
    desc.BindFlags = D3D11_BIND_SHADER_RESOURCE | D3D11_BIND_UNORDERED_ACCESS;
    desc.Usage = D3D11_USAGE_DEFAULT;
    desc.CPUAccessFlags = 0;

    // Create the buffer with the specified configuration
    ID3DllBuffer* pBuffer = 0;
    HRESULT hr = g_pDevice->CreateBuffer( &desc, pData, &pBuffer );

    if ( FAILED( hr ) )
    {
        // Handle the error here...
        return( 0 );
    }

    return(  pBuffer  );
```

Listing 2.16. A method for creating byte address buffer resources.

Resource view requirements. A byte address buffer can be used in two different ways. It can be attached to the pipeline with a shader resource view for read-only access to the buffer, or it can be attached to the pipeline with an unordered access view that provides read and write access to the buffer. While the shader resource view provides read-only access, it is available to all of the programmable shader stages. The unordered access view is only available in the compute and pixel shader stages. The formats for the resource views must be the DXGI_FORMAT_R32_TYPELESS in both cases. When an unordered access view is used, it must also be created with the D3D11_BUFFER_UAV_FLAG_RAW to indicate that it would supply access to data as a byte address buffer.

HLSL byte address buffer objects. Once the buffers have been created and appropriate resource views are available for binding the resources to the pipeline, the HLSL program

must declare the appropriate resource object to interact with the buffers. There are two different objects that can be declared, each of which corresponds to the type of access that the HLSL code will have to the resource. If the buffer is bound to the pipeline with a shader resource view, the corresponding HLSL object will be declared as a *ByteAddressBuffer*. If it is bound to the pipeline with an unordered access view, the corresponding HLSL object will be declared as a *RWByteAddressBuffer*. Samples of these byte address buffer type declarations are provided in Listing 2.17.

```
ByteAddressBuffer    rawBufferI
RWByteAddressBuffer  rawBuffer2j
```

Listing 2.17. Declaring byte address buffers in HLSL.

In general, the methods of these objects require an address to specify the location within the buffer at which to perform the given operation. These addresses must be 4-byte aligned, and always work with the `uint` data type. If other data types are required, the returned data can be reinterpreted using one of the conversion intrinsic methods.[2] Each of these two objects support the various *Load* methods that can be used to retrieve data from the buffer resource, and up to four 32-bit memory locations can be read at a time. However, the read/write byte address buffer also provides a large array of methods for manipulating the contents of the buffer. There are the simple *Store* analogs to the *Load* methods, in addition to a host of atomic instructions for updating the resource simultaneously from many threads.

Indirect Argument Buffers

The final buffer configuration that we will inspect is the *indirect argument buffer*. As we will see in many cases throughout this book, Direct3D 11 has taken many steps toward making the GPU more useful and able to operate on its own. The indirect argument buffer is one of these steps. It is used to provide parameters to the rendering and computation pipeline invocation methods from within a resource, instead of having those parameters directly passed by the host program. The concept behind allowing this type of pipeline control is to allow the GPU to generate the needed geometry or input data resources in one execution pass, and to then process or render that data in a following pass, without the CPU having any knowledge of how many primitives are being processed. Even though the CPU must still initiate the pipeline execution and pass the indirect argument buffer reference, this scenario reduces the CPU's role to a simple broker between GPU passes.

Using indirect argument buffers. Indirect argument buffers can be used in several different pipeline execution methods. We haven't covered the pipeline execution in detail yet,

[2] The conversion intrinsic methods are discussed in Chapter 6, "High Level Shading Language."

but we can still discuss how these buffers operate without intimate knowledge of what each parameter means. The available indirect argument pipeline execution methods are listed below.

- DrawInstancedIndirect(ID3D11 Buffer *pBufferForArgs,
  UINT AlignedByteOffsetForArgs)

  DrawIndexedInstancedIndirect(ID3D11 Buffer *pBufferForArgs,
  UINT AlignedByteOffsetForArgs)

  DispatchIndirect(ID3D11 Buffer *pBufferForArgs, UINT AlignedByteOffsetForArgs)

Each of these methods appends the *Indirect* term at the end of one of the standard methods, and simply replaces the arguments of the method with a reference to a buffer resource and an offset into the buffer, which identifies where the parameters are located in it. For example, the standard DrawInstanced() method takes four uint parameters: VertexCountPerInstance, InstanceCount, StartVertexLocation, and StartInstanceLocation. These arguments are passed directly in the API call to instruct the runtime and driver what to draw and what options are desired. The DrawInstancedIndirect() method passes these values to the runtime and driver in a buffer resource, instead of directly passing them. For this to work, the application must ensure that there are four consecutive uint values stored within the buffer at the location indicated by the offset argument passed with the buffer reference. Including an offset argument in these methods allows a single buffer to be used for many different pipeline invocations, and makes it possible to store the parameters simultaneously in different locations. Figure 2.20 shows how this scheme operates. It is also important to note that the offset into the buffer must be 4-byte aligned to allow the individual values to stride the standard 32-bit variable sizes.

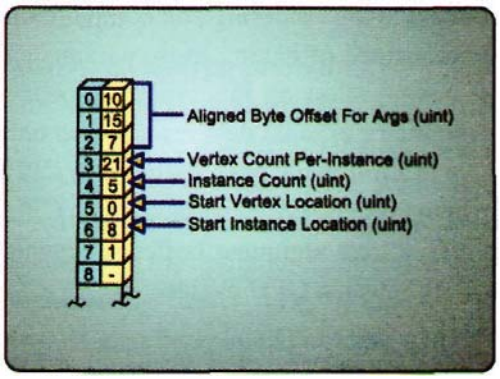

Figure 2.20. The indirect arguments buffer resource being used to execute the rendering pipeline.

There are many ways to update the data in the indirect argument buffer. Any pipeline output method or CPU manipulation technique can be used. As long as the data is available in the buffer, it can be used to execute the pipeline. For example, one possibility is to use the AppendStructuredBuffer resource type in a compute shader to fill it with some GPU-generated content. The number of items within the buffer could be copied to the indirect argument buffer with the ID3DllDeviceContext::CopyStructureCount() method, and the geometry could

finally be drawn with the ID3DllDeviceContext: :DrawInstancedIndirect() method, with a single instance provided. This would allow the GPU-generated content within the AppendStructuredBuffer to be rendered directly, without the CPU ever knowing how many items exist within the buffer itself.

Creating indirect argument buffers. Creating a indirect argument buffer is quite similar to the other creation methods we have seen before. It also follows the same usage control semantics that we have seen. In general, if we want to be able to generate the data on the GPU and then subsequently use the buffer as input to the pipeline, we must select the default usage flag to provide both read and write access to the GPU. In addition, to indicate to the runtime that this buffer will be used for indirect arguments, the D3D11_RESOURCE_MISC_DRAWINDIRECT_ARGS miscellaneous flag must be specified. The ByteWidth parameter should be a 4-byte multiple, since the input arguments to the *draw* and *dispatch* methods are all 32-bit variables. A sample method for creating these buffers is provided in Listing 2.18.

```
ID3DllBuffer* CreateIndirectArgsBuffer( UINT size, D3D11_SUBRESOURCE_DATA* pData )
{
    D3D11_BUFFER_DESC desc;
    desc.ByteWidth = size;
    desc.MiscFlagS = D3Dll_RESOURCE_MISC_DRAWINDIRECT_ARGS;
    desc.Usage = D3D11_USAGE_DEFAULT;
    desc.StructureByteStride = 0;
    desc.BindFlags = 0;
    desc.CPUAccessFlags = 0;

    // Create the buffer with the specified configuration
    ID3DllBuffer* pBuffer = 0;
    HRESULT hr = g_pDevice->CreateBuffer( &desc, pData, &pBuffer );
    if ( FAILED( hr ) )
    {
        // Handle the error here...
        return( 0 );
    }

    return( pBuffer );
}
```

Listing 2.18. A sample method for creating a buffer resource to be used as an indirect arguments buffer.

Resource view requirements. An indirect argument buffer is used in two different phases. First, it is loaded by the GPU or CPU with the data that will eventually represent the parameters to a pipeline invocation call. Second, it is passed as input to a *draw/dispatch* call by the CPU. The first case may require a resource view, while the second case does not.

When the buffer is to be filled by the GPU, it can be updated with the stream output functionality, written to as a render target, or modified with an unordered access view. If the buffer will be updated by the CPU, one of the standard methods for modifying its contents can be used, such as the ID3D11DeviceContext: :UpdateSubresource() method, or updated with the ID3D11DeviceContext: :CopyStructureCount() method to read the hidden counter value from a buffer. In general, these techniques for writing to the buffer are discussed in their respective areas of Chapter 2 and Chapter 3 and won't be repeated here. However, if it is possible to get the data into the buffer, it can subsequently be used to execute the pipeline.

HLSL indirect argument buffer objects. The indirect argument buffers are used as input to the pipeline execution methods, and are hence not directly used in HLSL. As mentioned above, there are a variety of methods for getting the parameter data into the buffer, some of which will require resource views and must be written directly from HLSL, and some which do not. The particular technique used will dictate if the data can be written to the buffer through an HLSL resource object.

2.2.2 Texture Resources

As we have seen throughout the last few sections, there are a myriad of different buffer resources, and many different ways to configure them for various uses. However, the buffers are only half of the resource story. Textures provide a different concept for resources based on the evolutionary roots of the GPU as a rendering co-processor. The term *texture* refers to a memory resource that is similar in nature to an image, or image-like. This is a very loose definition, because there is a range of different texture types, with varying dimensions, topologies, and multisampling characteristics. These make the term *image-like* seem somewhat out of place, since some textures don't resemble a traditional two-dimensional image at all. The common link between these texture resources and an image is the concept of a *pixel element,* which is the same across all texture types. The pixel (or *texel,* as it is referred to in textures) is the smallest data element that all textures are comprised of. Each pixel is represented by up to four components. The format of the components varies, depending on the function of the texture resource.

While texture resources are still blocks of memory that are made available to the GPU (as are buffer resources), additional dedicated hardware is available for certain texture operations, which can provide greater efficiency than a corresponding operation on a buffer. One example of this would be texture-filtering hardware in a GPU. It is simply faster to interpolate between two neighboring data elements with a texture resource than it is with a buffer resource, because of the additional hardware. In addition, some functions are naturally better suited to textures, such as render targets or depth stencil targets. Since

buffers and textures have different qualities, the appropriate resource type must be chosen for a particular situation. In addition, textures support the ability to use mip-maps, which are essential for achieving good performance and visual quality when rendering texture-mapped geometry.

We will explore the texture resource types in more detail throughout this chapter. We begin with a general discussion about textures, and some specific details about their uses and available operations. This is followed by a more detailed look at using, creating, and manipulating each type of texture resource, both in C++ and in HLSL.

Common Texture Properties

As mentioned above, texture resources are available in ID, 2D, and 3D versions. The specific memory layouts that they use are distinct from one another, due to their differences in dimension. These are similar to the differences you would expect to see in C/C++ in layout between arrays of different dimensions. As an indication of how these layouts are organized from a usage perspective, consider the graphical representations shown in Figure 2.21. Even with distinct layouts, the texture resources share many other properties. We will explore some of these common properties before moving on to examine each texture type.

As shown in Figure 2.21, textures are arranged in according to the familiar *X, Y,* and *Z* axes. This figure also includes additional axis descriptions, named *U, V,* and *W*. These indicate the texture addressing scheme used by Direct3D 11, in which the total size of a texture (regardless of how large or small it is) in each of the *X, Y,* and *Z* directions is mapped into a range of [0,1]. This is done to provide a convenient method for sampling a texture. If you sample a one-dimensional texture at address u=0.5, then regardless of whether the texture has 16 elements, or 1600 elements, you will receive the element that is at the halfway point of the texture. This is a very important consideration when performing sampling, and will be considered further when we discuss samplers in the "Sampler State Object" section.

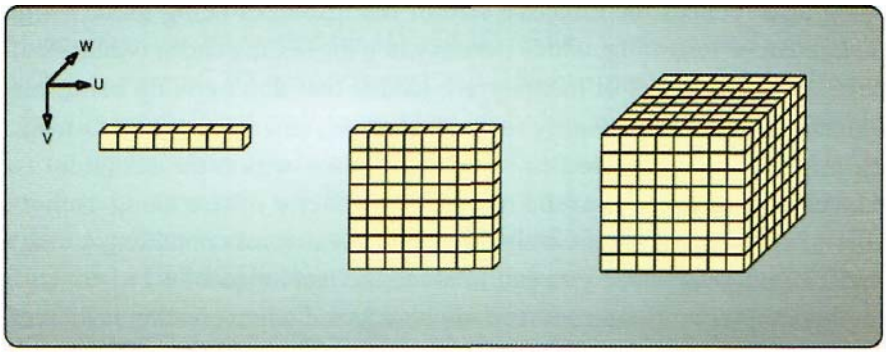

Figure 2.21. A visualization of the various texture configurations.

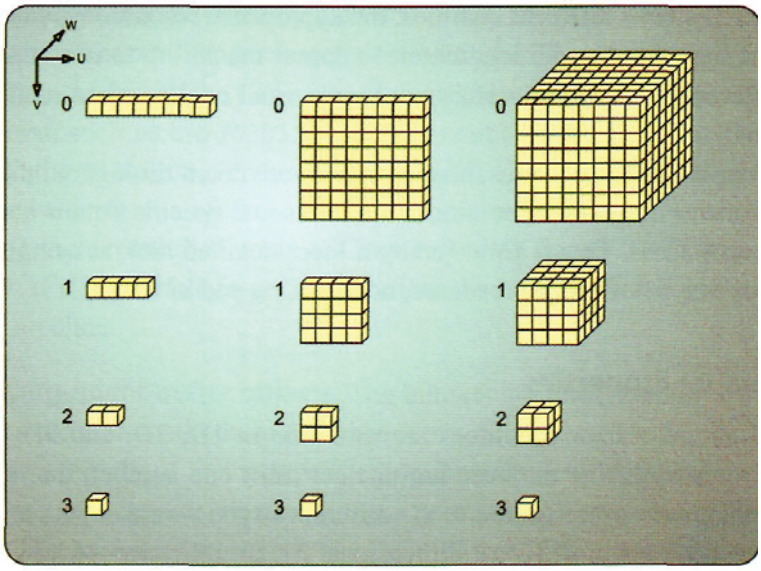

Figure 2.22. A visual representation of texture mip-maps.

Texture resource mip-maps. All of the various types of textures share the concept of *mip-maps*. A mip-map quite simply represents a lower-resolution version of a texture. It is common for a texture to contain several mip-map levels, with each successive level providing a half scale copy of the texture contents. Using mip-maps allows for lower-resolution texture lookups in situations where the highest levels of resolution will not add useful information, or could even contribute to rendering artifacts. A common example of this is when a textured model is rendered far from the camera, and a texture applied to the model appears to sparkle when the model moves. This is caused when the sampling pattern of the rendered image is at a much lower resolution than the texture content's sampling pattern. By lowering the effective texture resolution by looking up an appropriate mip-map level instead, the two sampling patterns can be brought much closer together, virtually eliminating this shimmering effect.

Figure 2.22 visualizes these mip-maps for each of the three texture types. Because mip-mapping also reduces the effective size of the resources being used, it significantly reduces *texture cache thrashing*, which occurs when the texture cache is continually forced to miss because large chunks of memory are loaded that don't end up being used, which ultimately leads to very high memory request latencies, and degrades performance.

Each mip-map is considered to be a *subresource* within the resource itself. Each mip-map level reduces the size of the resource by a factor of two along each dimension. The number of mip-map levels is limited by the lowest level containing a single texture element, which is logical, since you can't reduce the resolution of a 1x1 texture. We will see later in this chapter how subresources can be selected when creating resource views, as well as for manipulating resource contents.

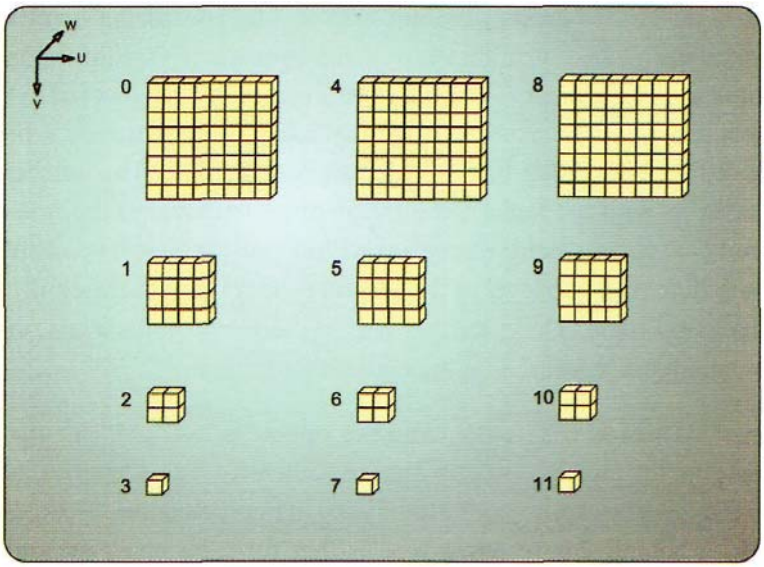

Figure 2.23. A texture resource with texture slices.

Texture resource arrays. Several types of texture resources can also be created as an array of textures. For example, a single 2D texture can be created as an array of 2D texture subresources. This allows a single resource to contain what are referred to as many individual *texture slices.* This is shown in Figure 2.23.

When a resource is created as an array, each of the individual texture slices can be selected with a resource view and used as if it were a standalone resource. This allows the texture resources to be used as a higher-level data structure that encapsulates multiple different contents in each texture slice. For example, having access to multiple texture slices can simplify some forms of *texture atlases,* which are essentially a texture resource that contains multiple individual textures. Texture atlases can then be used for multiple object renderings, which allows them to be rendered consecutively without modifying the pipeline state in between draw calls. Having access to an array-based resource significantly simplifies this type of technique.

Texture arrays can be created for 1D and 2D texture types, as well as the *texture cube* type (which is a form of 2D texture array), but which is not allowed for 3D textures. In practice, this limitation is not really a problem, since having an array of 3D textures would essentially be representing a 4D data resource. The memory consumption of such a resource would grow extremely quickly with the resource size in each dimension. However, if such a data structure is needed, the application can simply use multiple individual 3D texture resources or can simply create a 3D texture where the third dimension is specified as a multiple of basic record size and can manually be indexed by a shader program, or implemented with a series of resource views that select appropriate subregions of the resource.

Subresources. With our discussion of mip-map levels, and of the array elements of a resource, we have introduced the concept of a *subresource.* This name is used to identify complete subportions of a resource. For example, every mip-map level of every element of a texture array is a unique subresource. To aid in selecting a particular subresource, each one is given a subresource index by which it can be identified. The numbering of subresource indices begins with the highest-resolution mip-map level of the first element of an array, and increments for each mip-map level within that element. The count continues on the highest-resolution mip-map level of the second element of the array, and continues until all resources have an index. These indices will be very important when we consider the methods for manipulating resources at the end of this chapter.

Texture 2D multisampled resources. Another option is available for two-dimensional texture resources, and is aimed at improving image quality. Two-dimensional textures can be created as *multisample textures,* and can be used to implement *multisample anti-aliasing (MSAA).* The idea behind this resource type is that for each pixel, multiple subsamples are actually stored in a pattern within the pixel boundaries. When rendering is carried out, these subsamples are used to determine at a subpixel level the amount of coverage that a pixel should receive from a given primitive being rasterized. This lets each pixel be generated from a number of subsamples, which improves the quality of geometry edges by effectively increasing the sampling rate of the render target. An example pixel with a subsample pattern is shown in Figure 2.24.

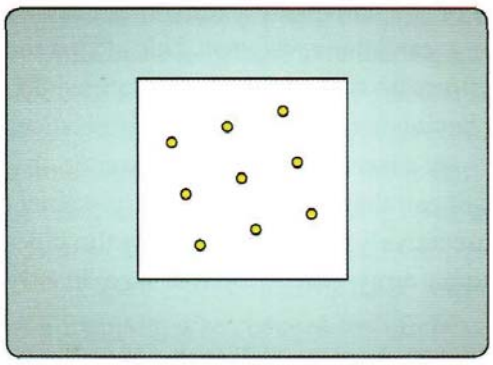

Figure 2.24. An example subsample pattern within a pixel.

Up to 32 subsamples can be selected by the application, depending on hardware support. In the "Texture 2D" section we will see how to create a texture with a valid number of samples for a given hardware configuration. Care must be taken when using MSAA, since the amount of memory consumed by the resources in question is effectively multiplied by the number of subsamples used. Further details about why MSAA is useful, as well as information on how to use these resources, is discussed in Chapter 3 in the "Rasterizer," "Pixel Shader," and "Output Merger" sections.

Texture 1D

The first texture resource type that we will examine is the 1D texture. These textures are arranged along a single axis, with each element being comprised of one of the basic DXGI formats provided in the DXGI_FORMAT enumeration. The various subresource configurations of a 1D texture are shown in Figure 2.25. These texture resources can be created

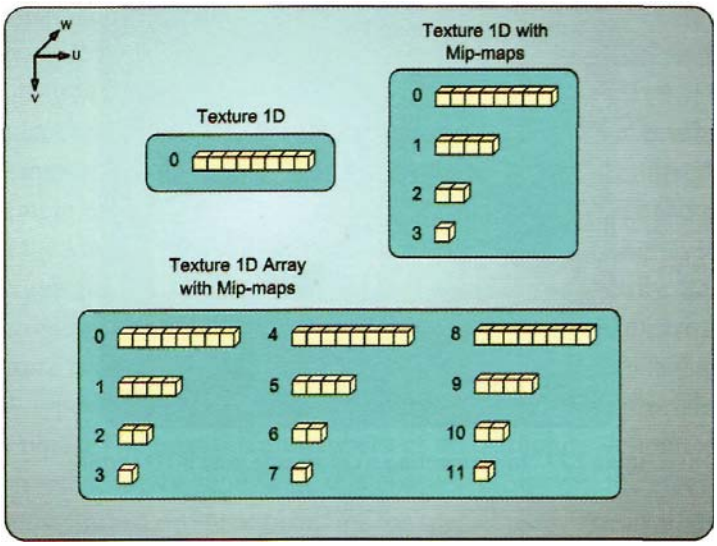

Figure 2.25. A visualization of a ID texture resource.

with or without mip-maps, or as an array of ID textures with or without mip-maps. The subresource index is also visible in Figure 2.25.

Using 1D textures. One-dimensional textures are commonly used to implement lookup tables. For example, if a very complex function can be replaced by a ID lookup texture, it is possible to reduce the number of arithmetic instructions needed by replacing the function with a texture lookup instruction instead. Depending on the workload present in a given shader program, this can be beneficial to the performance of the shader. In addition, since the GPU has dedicated hardware for filtering texture data with samplers, the filtering process is performed more or less for free, from the shader program's point of view. It doesn't need to perform any additional arithmetic to receive a filtered texture value. Figure 2.26 demonstrates a complex trigonometric function that can be replaced by a 1D texture with the values indicated in the image. In this type of usage, the texture can use a single component format, such the DXGI_FORMAT_R32_FLOAT format.

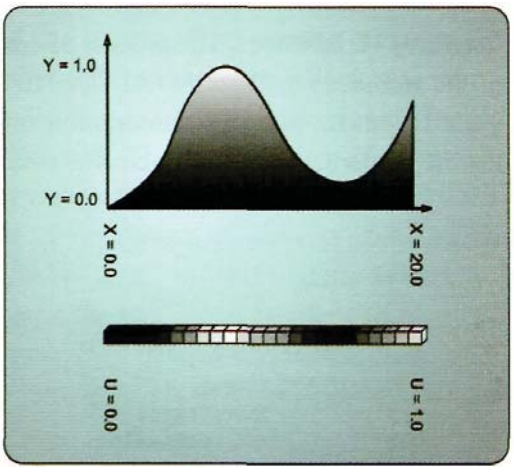

Figure 2.26. A visualization of replacing a complex function with a simple lookup-table, which is implemented as a ID texture.

Another common use of 1D textures is to implement a color visualization scale. In this scenario, a 1D texture resource is

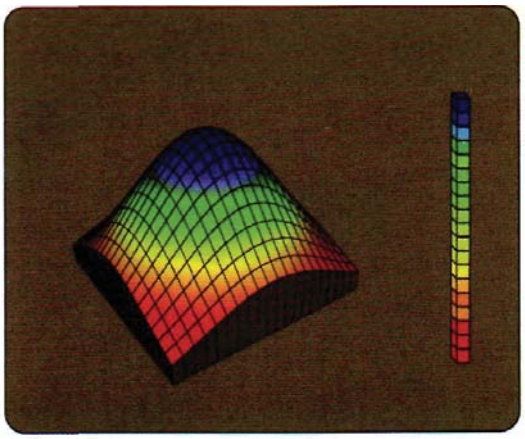

Figure 2.27. Implementing a color scale with a ID texture.

populated with color values, where each color represents a scalar data value range. For example, when the value is in a normal range, it can be colored blue and progress to "hotter" colors as the input value approaches a more critical range. To implement such a color scale within a shader program, the particular scalar property value that is being visualized is mapped to the [0,1] range and is then used to sample the appropriate color in the 1D texture. This process is depicted in Figure 2.27.

A ID texture resource can also be used in a computational context within the compute shader. With the general purpose computation capabilities of the compute shader, any 1D data array can be stored in a texture. If the data can be stored as a texture—that is, if it doesn't require a structure to represent its basic element—it can take advantage of the sampling abilities of the GPU, as described above, with little or no additional computational cost.

Creating 1D textures. The process of creating a 1D texture follows the same process as all of the resources in Direct3D 11. The ID3DllDevice: :CreateTexturelD() method takes a D3D11_TEXTURE1D_DESC description structure pointer, which specifies all of the desired texture properties of the resource. The structure and its members are provided in Listing 2.19.

```
Struct D3D11_TEXTURE1D_DESC {
    UINT        Width;
    UINT        MipLevels;
    UINT        ArraySize;
    DXGI_FORMAT Format;
    D3D11_USAGE Usage;
    UINT        BindFlags;
    UINT        CPUAccessFlags;
    UINT        MiscFlags;
}
```

Listing 2.19. The D3D11_TEXTURE1D_DESC structure and its members.

The Width parameter alone specifies the size of the texture, because this is a one-dimensional texture. It represents the number of texture elements that will be included in the texture. The desired number of mip-map levels is specified in the MipLevels member. The most important consideration for the number of mip-maps is that a value of 1 provides just the top level mip-map, while a value of 0 will create a complete mip-map chain all the way down to a single element. Any value greater than 1 will specify the number of mip-map levels, until the single element mip-map level is reached. The ArraySize member of this method indicates how many texture elements to create in the array. Each texture slice in the array will contain the number of mip-maps specified in the MipLevels member. The format of the texture is logically specified in the Format member. This must specify one of the DXGI_FORMAT types, which provide a large variety of different precisions, component counts, and data types. The next three members of this structure define the usage characteristics, as described at the beginning of this chapter. However, the MiscFlags member allows for some customized resource behavior for special uses. The available flags that are usable with a Texture1D resource are listed below.

- D3D11_RESOURCE_MISC_GENERATE_MIPS
- D3D11_RESOURCE_MISC_RESOURCE_CLAMP

The generate mips flag allows the Direct3D runtime to populate the available mip-map levels with data based on the current values within the top level. To use this functionality, the resource must be used as a render target (and thus created with the D3D11_BIND_RENDER_TARGET bind flag) to fill the top mip-map level, and then the lower mip-map levels can be filled with a call to the ID3D11DeviceContext::GenerateMips() method. This method takes a shader resource view of the texture resource and will fill in the mip-map levels indicated in the view. Of course, to be able to use the resource in a shader stage, it must also be created with the shader resource bind flag (D3D11_BIND_SHADER_RESOURCE).

The resource clamp flag is used to indicate that a texture resource will use the minimum LOD functionality. This is a way for the application to use the device context to set the maximum detail level of a texture that will be used. The specification of the maximum detail level is performed with the ID3D11DeviceContext: :SetResourceMinLOD() method, which takes a pointer to the resource and a float parameter that can have a value between 0 and 1. This functionality allows the GPU to page out unused portions of a resource so that more memory can be devoted to resources that are more likely to be used. This setting could be used to implement a resource level of detail that varies as a function of the distance from the camera, where the top mip-map levels will not be used when an object is very far from the camera.

Resource view requirements. A 1D texture resource can use all four types of resource views and cannot be bound to the pipeline without a resource view. The usage semantics

associated with each of these resource views that we have seen for buffers also hold true for textures. However, depending on the resource dimensions and the type of resource view being used, a subset of the resource can be specified to be visible to the resource view. This provides the ability to manipulate only portions of a resource at a time and also allows multiple views to reference different subresource regions of a resource. The subresource selections are described below for each of these cases.

Texture1D shader resource view. For standard, non-array-based 1D texture resources, a shader resource view can be used to select a range of mip-map levels and make them accessible to the programmable shader stages. This gives the developer the option of either using a complete resource, or only using a subset of it. In general, the subset range of mip-map levels will be treated the same as a full resource by shader programs, since they are not aware of what the resource behind the resource view actually looks like. The parameters used to determine the subset are specified in the D3D11_TEX1D_SRV structure, which is used as the union parameter from the D3D11_SHADER_RESOURCE_VIEW_DESC structure (which we have seen in the "Resource Views" section earlier in this chapter). These parameters are shown in Listing 2.20.

```
struct D3D11_TEX1D_SRV {
    UINT MostDetailedMip;
    UINT MipLevels;
}
```

Listing 2.20. The contents of the D3D11_TEX1D_SRV structure.

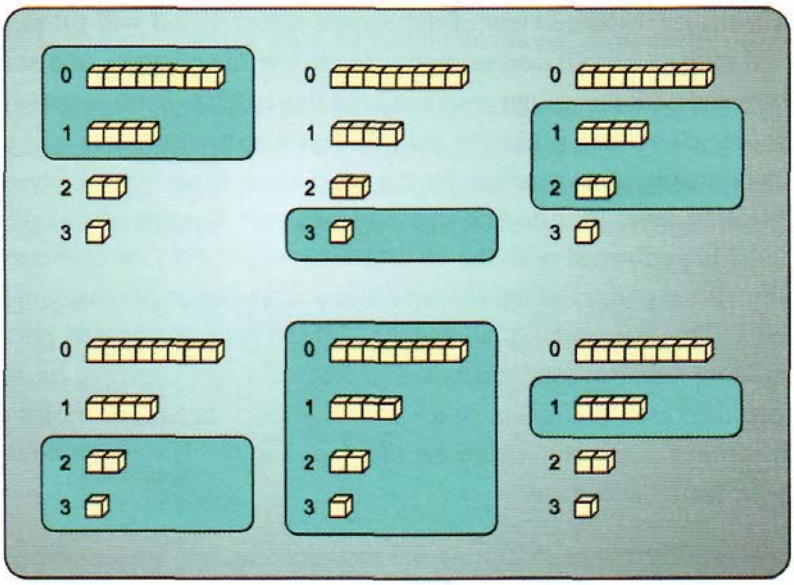

Figure 2.28. Several examples of mip-map ranges that can be selected.

The starting mip-map level is specified by the MostDetailedMip parameter, and the number of levels that follow it is specified in the MipLevels parameter. A few possibilities for mip-map level selection are graphically demonstrated in Figure 2.28.

For array-based 1D texture resources, a shader resource view references a subset of mip-map levels in each texture slice in the same way as mentioned above for non-array resources, but it can also select a subrange of texture slices to read those mip-map levels from. The range selections are made with the D3D11_TEX1D_ARRAY_SRV structure in the D3D11_SHADER_RESOURCE_VIEW_DESC structure. These parameters are shown in Listing 2.21.

```
struct D3D11_TEX1D_ARRAY_SRV {
    UINT MostDetailedMip;
    UINT MipLevels;
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.21. The members of the D3D11_TEX1D_ARRAY_SRV structure.

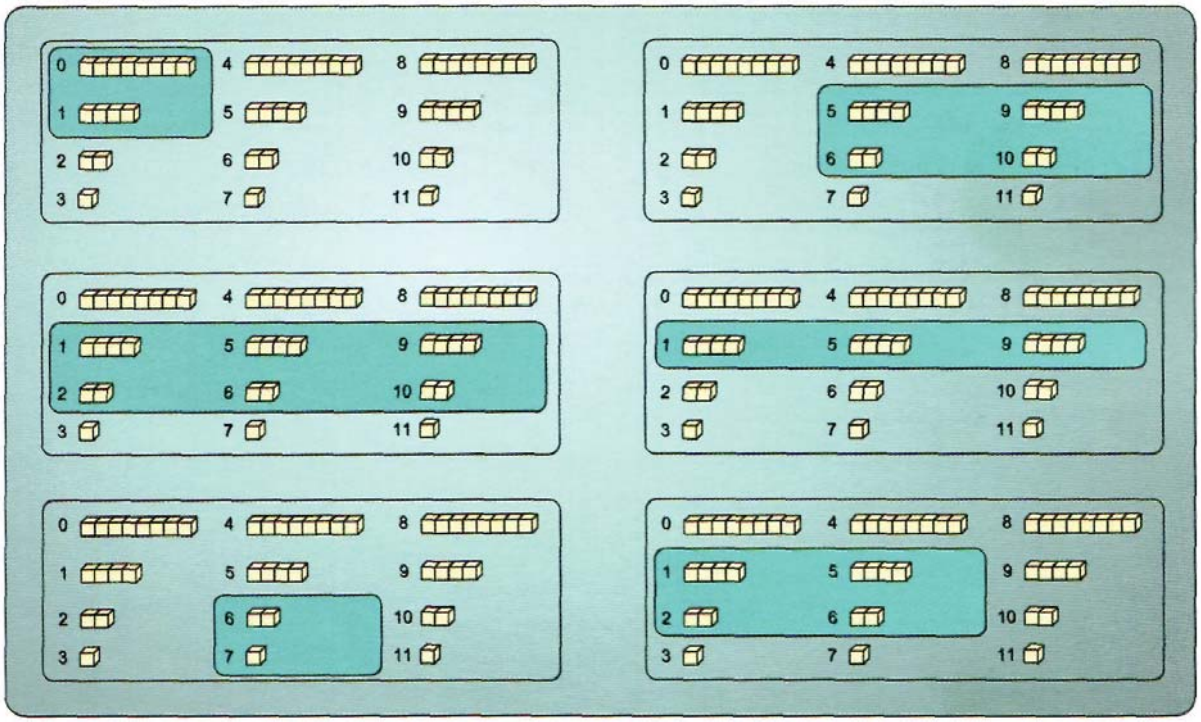

Figure 2.29. Selecting subresources in a 1D texture array resource.

As seen in Listing 2.21, the MostDetailedMip and MipLevels parameters are the same as they were for non-array resources. However, the texture slices are selected with the additional FirstArraySlice and the ArraySize parameters, which specify the starting slice and how many subsequent slices will be included in the view. This is depicted graphically in Figure 2.29.

Texture1D unordered access view. The unordered access view for non-array 1D texture resources specifies a single mip-map level. In practice, the restriction to a single mip-map level is not really a hindrance, since it is possible to create additional unordered access views to access other mip-map levels. This solution only partially overcomes the issue, since there are a limited number of UAV slots, but an increased number of mip-maps could be processed in multiple passes to overcome this limit. The desired mip-map level is specified in the D3D11_TEX1D_UAV structure within the D3D11_UNORDERED_ACCESS_VIEW_DESC structure.

```
struct D3D11_TEX1D_UAV {
    UINT MipSlice;
}
```

Listing 2.22. The members of the D3D11_TEX1D_UAV structure.

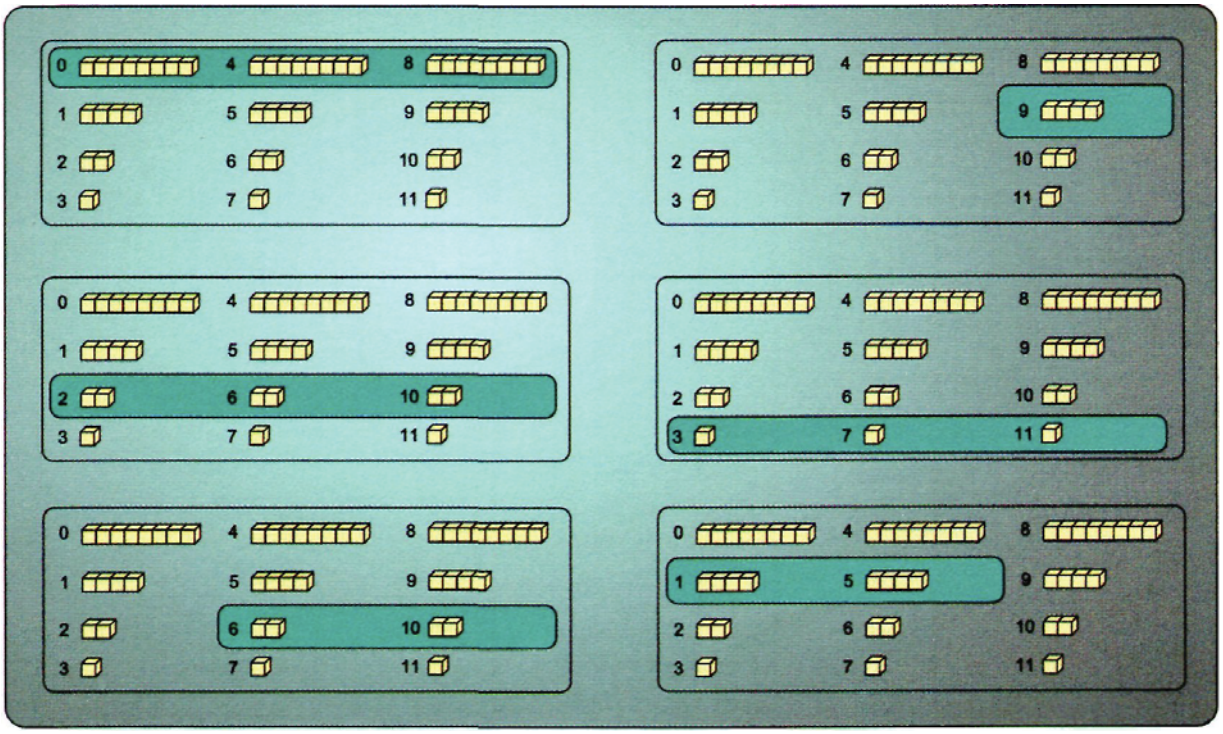

Figure 2.30. Various subresource selections for use in unordered access views.

For array-based ID texture resources, the unordered access view still specifies a single mip-map level, but it also specifies an array subset in the same way as the render target view did. These parameters are specified in the D3D11_TEX1D_ARRAY_UAV structure within the D3D11_UNORDERED_ACCESS_VIEW_DESC structure. This range is depicted graphically in Figure 2.30 for a variety of different unordered access views.

```
struct D3D11_TEX1D_ARRAY_UAV {
    UINT MipSlice;
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.23. The members of the D3D11_TEX1D_ARRAY_UAV structure.

Texture1D render target view. When a 1D texture resource is used as a render target, the render target view only specifies a single mip-map level. This is due to the way that a render target is written to, which does not allow for multiple mip-map subresources to be written to simultaneously. This mip-map level selection is performed with the D3D11_TEX1D_RTV structure within the D3D11_RENDER_TARGET_VIEW_DESC structure, as shown in Listing 2.24.

```
struct D3D11_TEX1D_RTV {
    UINT MipSlice;
}
```

Listing 2.24. The members of the D3D11TEX1DRTV structure.

When a render target view is used with a ID texture array resource, it exposes a single mip-map level of a subrange of the array elements of the resource. This selection pattern is the same as that of the unordered access view with array-based resources. The subresource selection is performed with the D3D11_TEX1D_ARRAY_RTV structure within the D3D11_RENDER_TARGET_VIEW_DESC structure, as shown in Listing 2.25.

```
struct D3D11_TEX1D_ARRAY_RTV {
    UINT MipSlice;
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.25. The members of the D3D11 TEX1DARRAYRTV structure.

Texture 1D depth stencil view. In both the array and non-array Texture1D resources, the depth stencil view uses the same subresource regions as indicated for the render target views. This makes sense, since the render target size and dimension are always required to match that of the depth stencil target. The two structures used to select these subresource regions are shown in Listing 2.26.

```
struct D3D11_TEX1D_DSV {
    UINT MipSlice;
}
Struct D3D11_TEX1D_ARRAY_DSV {
    UINT MipSlice;
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.26. The members of the D3D11_TEX1D_DSV and D3D11_TEX1D_ARRAY_DSV structures.

HLSL objects. One-dimensional texture resources may be accessed from within the HLSL programs that are executed in the programmable shader stages. However, they interact with the resources through resource objects that must be declared in the HLSL source file. These objects essentially form the connection between the shader program and the data that is exposed through the resource view used to bind a resource to the pipeline. There are two possibilities for binding resources to a pipeline stage so that they can be accessed from HLSL, either using a shader resource view or an unordered access view. The shader resource view provides read-only access to resources. If a resource is bound with a shader resource view, its resource object in HLSL will be declared as either a Texture1D or a Texture1DArray, depending on whether the resource is array based or not. The unordered access view allows read and write access to the resource data that it exposes. To indicate this difference in usage, its resource objects in HLSL are declared with the same names, but prepended with an *RW* to indicate their read/write nature. Listing 2.27 demonstrates several example declarations in HLSL using these various resource objects.

```
Texture1D<float>   tex01;
Texture1DArray<uint3>   tex02;
RWTexture1D<float4>   tex03;
RWTexture1DArray<int2>   tex04;
```

Listing 2.27. Several resource object declarations for use with 1D texture resources.

When the texture is declared in HLSL, it specifies a template-style parameter to indicate what format the resource must be compatible with. This is used to ensure that when a resource

view is bound to the pipeline for a particular resource object declaration, it contains the appropriate format. The usage of HLSL resource objects is described in more detail in Chapter 6.

Texture 2D

The second texture type that we will explore is the 2D texture. Since this texture type is closely related to a standard 2D image layout, it is typically one of the most widely used types of texture. This resource is organized as a two dimensional grid of elements, where each element is a member of the DXGI_FORMAT enumeration. The various subresource forms of a 2D texture are shown in Figure 2.31.

As seen in Figure 2.31, these textures support the use of mip-map levels and texture arrays in a similar manner as the ID texture did. However, they also support multisampled resources to implement MSAA. This is the only texture resource type that allows multisampling, and it is hence a very important resource type.

Using **2D textures.** Two-dimensional textures are the primary texture resource used in Direct3D 11. As such, they have many different uses within a real-time rendering application. The first and most visible use is that of a render target. The Direct3D 11 rendering pipeline is the backbone of any real-time rendering software, and a render target is needed to receive the results of all of the calculations performed in it. In addition, during this rendering process, 2D textures are typically used for the depth/stencil target as well, since the render and depth/stencil targets are required to match in size and dimension. Once a complete scene rendering has been performed, the contents of the render target are displayed in the desired output window for viewing onscreen. We will see this usage pattern many times throughout the remainder of the book.

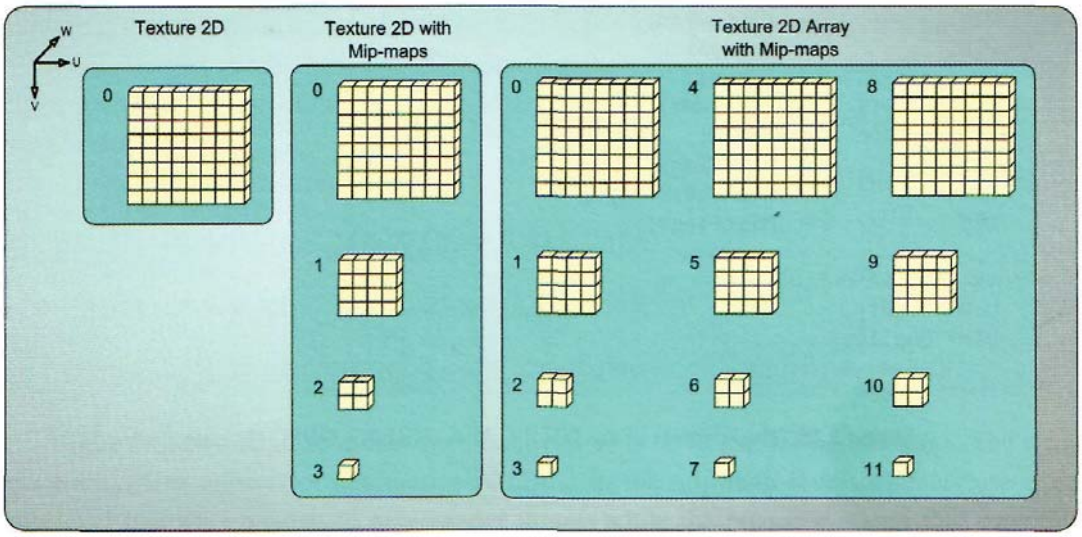

Figure 2.31. Various subresource forms of a 2D texture resource.

Outside of the standard render and depth targets, 2D textures are commonly used for applying surface properties to geometry being rendered. For example, if a brick wall is being rendered, the color of the bricks is typically supplied in a 2D texture that is read during the rendering process. The surface properties supplied don't necessarily need to be limited to the color of the surface, either. It is very common in modern rendering systems for surface normal vectors to be modified with what is called a *normal map.* This is a texture that supplies a 3D vector to represent a surface normal, where the *X, Y,* and *Z* coordinates are stored in the R, G, and B channels of the texture. Still other properties are encoded into textures, such as surface glossiness, displacement, and emissivity. It is fairly common to hear these types of textures referred to as *maps.* For example, the textures mentioned above would be referred to as *gloss maps, displacement maps,* or *emissivity maps.*

In addition to the per-object uses described above, it is also becoming increasingly important to perform post-processing of a scene rendering to increase the quality of the output presented to the user. In these situations, the output rendering is used as an input to various algorithms that process the image further or combine the image with other information to enhance it in some way. The algorithms in Chapter 10 demonstrate how this can be performed.

Creating 2D textures. The creation of a 2D texture is quite similar to the ID texture case. Along with the addition of the extra dimension specification, an extra structure parameter is added to determine the multisample level to be used in the texture. The texture is created with the ID3DDevice: :CreateTexture2D() method, which takes a pointer to the D3D11_TEXTURE2D_DESC structure, as shown in Listing 2.28.

```
struct D3D11_TEXTURE2D_DESC {
    UINT              Width;
    UINT              Height;
    UINT              MipLevels;
    UINT              ArraySize;
    DXGI_FORMAT       Format;
    DXGI_SAMPLE_DESC  SampleDesc;
    D3D11_USAGE       Usage;
    UINT              BindFlags;
    UINT              CPUAccessFlags;
    UINT              MiscFlags;
}
struct DXGI_SAMPLE_DESC {
    UINT Count;
    UINT Quality;
}
```

Listing 2.28. The contents of the D3D11_TEXTURE2D_DESC structure.

As mentioned above, this structure is quite similar to that which is used for 1D texture creation. The size of the texture is specified with the Width and Height parameters, and the

number of mip-map levels and array slices are specified in the MipLevels and ArraySize elements, respectively. The texture format also operates in the same manner as for the 1D texture and must be set to a member of the DXGI_FORMAT enumeration. The SampleDesc parameter is a structure that describes the multi-sampling features that will be included with the 2D texture. It contains two members: Count and Quality. The Count parameter specifies the number of subsamples to include in the texture, while the Quality parameter indicates the pattern and algorithm used to resolve the subsamples into a single pixel value.

The quality level is a vendor-specific metric, and it may or may not support more than one quality level. Since support for various quality levels is left up to the GPU manufacturer, the application must query what the available quality level is for a particular texture format and sample count. This is performed with the ID3DllDevice: :CheckMultisample QualityLevels() method. An example usage of this method is shown in Listing 2.29.

```
UINT NumQuality;
HRESULT hr = m_pDevice->CheckMultisampleQualityLevels(
                                DXGI_FORMAT_R8G8B8A8_UNORM, 4, &NumQuality );
```

Listing 2.29. A demonstration of how to check the available quality level of a texture format and multi-sample count.

In this example, we can see that the desired format is DXGI_FORMAT_R8G8B8A8_ UNORM and the desired number of samples is 4. The available quality level is returned in the NumQuality variable. If the result is 0, that particular format and sample count are not supported. If the value is 1 or greater, any quality level can be used, up to the returned value in the 2D texture description structure described above. It is important to note that if *n* is returned, that the value of n-1 should be used in the sample description structure!

The Usage, BindFlags, and CPUAccess parameters are selected according to the options specified in the beginning of this chapter and define where and how the texture resource can be used.

Finally, we have MiscFlags. The flags that correspond to 2D textures are listed below:

- D3D11_RESOURCE_MISC_GENERATE_MIPS
- D3D11_RES0URCE_MISC_RES0URCE_CLAMP
- D3D11_RESOURCE_MISC_TEXTURECUBE

The first two flags have the same behavior as seen in the 1D texture case. The mips generation flag allows for automatically filling in the mip-map levels of a texture if its top level has been written to as a render target, while the resource clamp flag provides a mechanism to manually control which mip-map levels must reside in video memory.

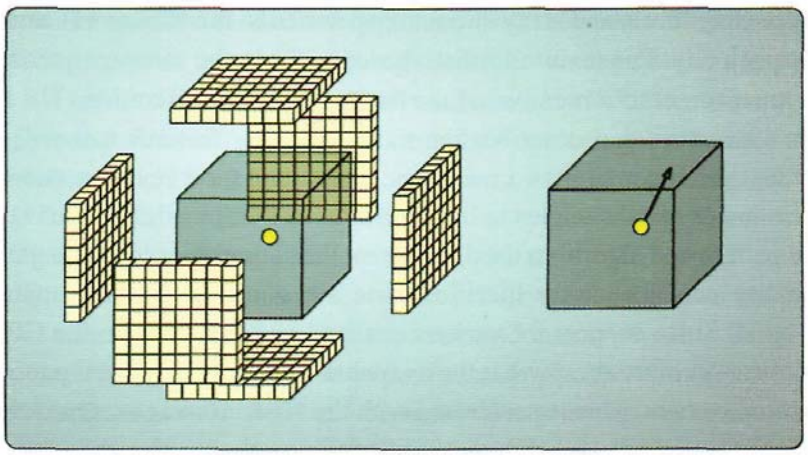

Figure 2.32. A texture 2D resource used as a cube map, and a demonstration of how it is sampled.

The texture cube flag is unique to the 2D texture type. This flag indicates that a 2D texture that is created with six texture slices can be used to generate all six faces of a cube map simultaneously. A *cube map* is a collection of six textures that can be sampled with a three-component vector positioned at the cube center, returning the texture element that is intersected by that vector. This is depicted in Figure 2.32. Intrinsic sampling functions are available to use this particular type of texture resource object, which we will see in more detail later in this section.

The other parameter for creating the texture is the initial data to load into the resource. This is provided with the D3D11_SUBRESOURCE_DATA point, which represents an array of these structures, with one structure for each subresource. The data layout for initializing the resource is provided Figure 2.33. If the texture is multisampled, the initial data parameter must be NULL, since these types of resources are not allowed to be initialized.

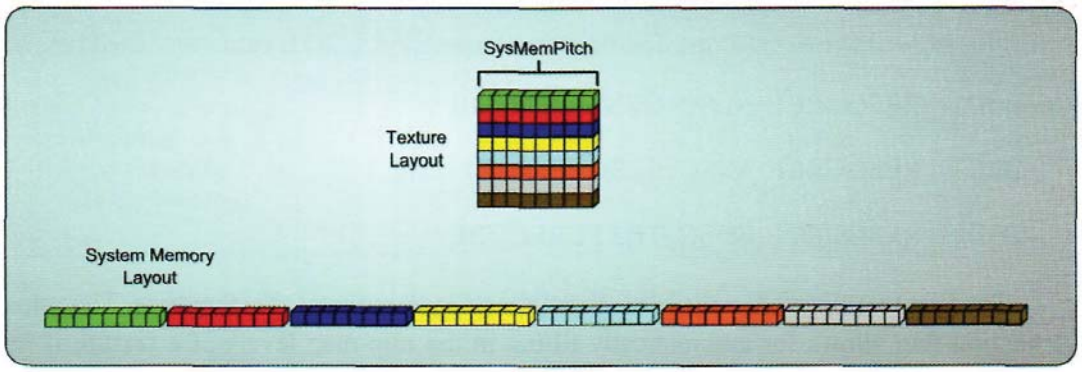

Figure 2.33. The layout of the subresource data to be applied to the initial contents of a 2D texture.

Resource view requirements. Like the 1D texture resource, the 2D texture resource can also be used with all four resource view types and can specify a subset of a resource to be exposed by the resource view. The 2D texture provides a large variety of resource creation options and also has a correspondingly large number of resource view configurations. We examine each of these possibilities below.

Texture2D shader resource view. For non-array, non-multisampled 2D texture resources, the shader resource view can select a range of mip-map levels from the resource. This is essentially the same selection mechanism that we have seen for 1D texture resources, except that each mip-map level is a 2D subresource, instead of 1D. This is specified in the D3D11_TEX2D_SRV structure within D3D11_SHADER_RESOURCE_VIEW_DESC, as shown in Listing 2.30.

```
struct D3D11_TEX2D_SRV {
    UINT MostDetailedMip;
    UINT MipLevels;
}
```

Listing 230. The members of the D3D11_TEX2D_SRV structure.

The starting mip-map level is specified in the MostDetailedMip parameter, and the maximum number of mip-map levels to use is specified in the MipLevels parameter.

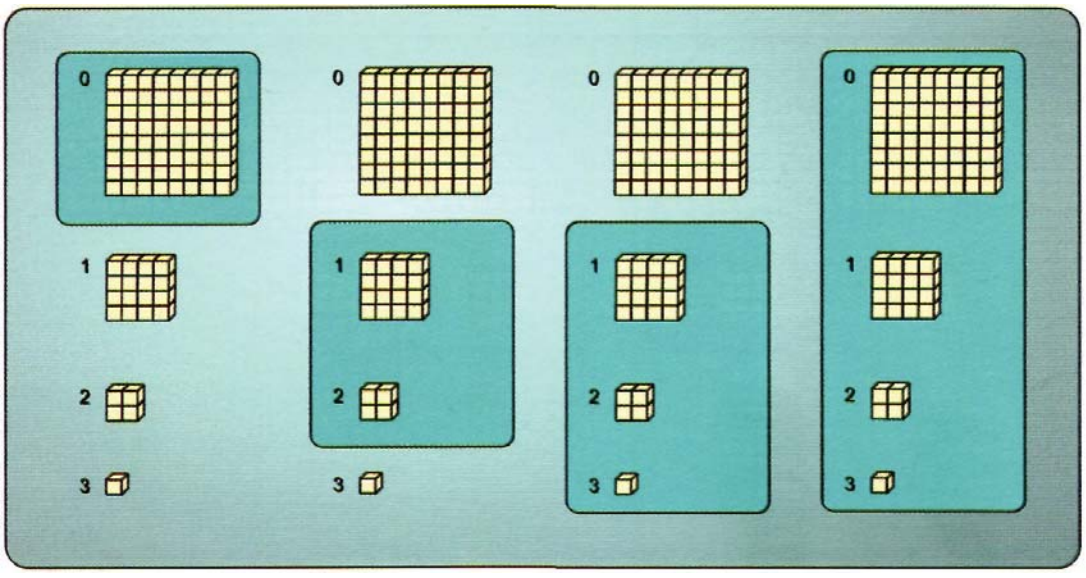

Figure 2.34. The various shader resource view subresource selections available for 2D texture resources.

Figure 2.34 demonstrates some example resource regions selected within a standard 2D texture resource.

The 2D texture array resources also select their ranges in the same way that their 1D resource counterparts do. That is, they select a range of mip-map levels, as well as an array element range. This is specified in the D3D11_TEX2D_ARRAY_SRV structure within the D3D11_SHADER_RESOURCE_VIEW_DESC structure shown in Listing 2.31.

```
struct D3D11_TEX2D_ARRAY_SRV {
    UINT MostDetailedMip;
    UINT MipLevels;
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.31. The members of the D3D11_TEX2D_ARRAY_SRV structure.

The mip-map level range is selected in the MostDetailedMip and MipLevel parameters, as seen above. In addition, the array element range is selected with the FirstArraySlice and ArraySize parameters. A few example ranges are shown in Figure 2.35.

Multisample 2D texture resources can also be used with shader resource views. Since multisampled textures do not contain mip-maps, there is no sub-range of the standard resource to select. This is reflected in the D3D11_TEX2DMS_SRV structure by its lack of parameters. However, the multisampled 2D texture array resources do allow the selection

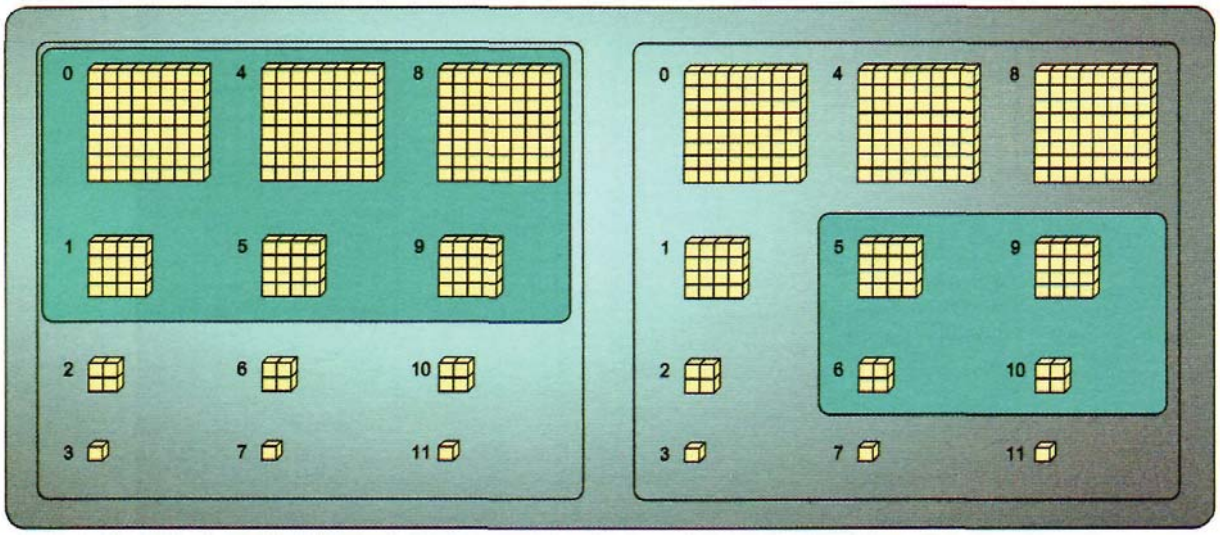

Figure 2.35. The various shader resource view subresource selections available for 2D texture array resources.

of a subrange of the elements of the texture array. This is done with the D3D11_TEX2DMS_ARRAY_SRV structure within the D3D11_SHADER_RESOURCE_VIEW_DESC structure, as shown in Listing 2.32.

```
struct D3D11_TEX2DMS_SRV {
    UINT UnusedField_NothingToDefine;
}
struct D3D11_TEX2DMS_ARRAY_SRV {
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.32. The members of the D3D11_TEX2DMS_SRV and D3D11_TEX2DMS_ARRAY_SRV structures.

As we have seen before, the array range is selected with the FirstArraySlice and the ArraySize parameters. A few examples of this range selection are shown in Figure 2.36.

There are two additional special uses for 2D texture array resources when they are created with the D3D11_RESOURCE_MISC_TEXTURECUBE miscellaneous flag. These shader resource views can be used to provide cube map access to six array elements, where each element represents one face of the cube map. For single cube map shader resource views, the D3D11_TEXCUBE_SRV structure is used. In this case, the texture array resource must have six elements, and a subrange of each face's mip-map levels can be specified. This is demonstrated in Listing 2.33.

```
struct D3D11_TEXCUBE_SRV {
    UINT MostDetailedMip;
    UINT MipLevels;
}
```

Listing 2.33. The members of the D3D11_TEXCUBE_SRV structure.

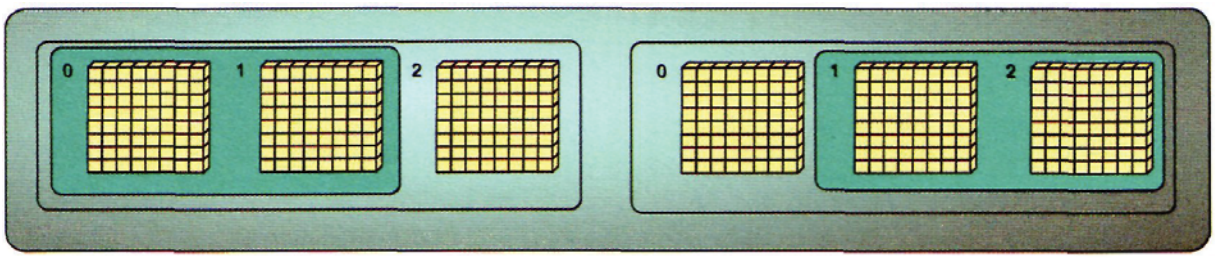

Figure 2.36. The various shader resource view subresource selections available for 2D multisampled texture array resources.

For arrays of cube maps to be specified in the shader resource view, the D3D11_TEXCUBE_ARRAY_SRV structure is used. The members of this structure allow for selecting the mip-map level range to be used, similar to the single cube map version. However, they also allow for specification of the first texture array element to be used, and then the number of cube maps that will be visible through the shader resource view. The number of cube maps is multiplied by six (for the number of faces on the cube) to determine how many array elements are needed. This is shown in Listing 2.34. Several example resource subregions are shown in Figure 2.37.

```
struct D3D11_TEXCUBE_ARRAY_SRV {
    UINT MostDetailedMip;
    UINT MipLevels;
    UINT First2DArrayFace;
    UINT NumCubes;
}
```

Listing 2.34. The members of the D3D11 TEXCUBE ARRAYSRV structure.

Texture2D unordered access view. The unordered access view can also be used to bind 2D texture resources to the pipeline. Two different subresource configurations can be used, which follow the same range selection paradigm as seen for the 1D texture resources. Non-array-based 2D texture resources can only specify a single mip-map level to be exposed to the unordered access view. This is performed with the D3D11_TEX2D_UAV structure within the D3D11_UNORDERED_ACCESS_VIEW_DESC structure, which is shown in Listing 2.35.

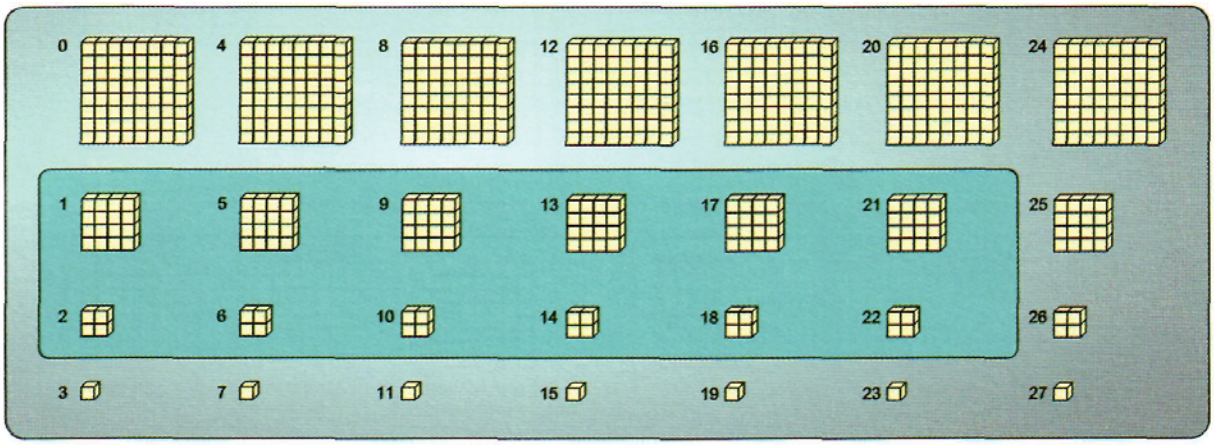

Figure 2.37, Various subresource regions of 2D texture arrays for use as texture cubes with shader resource views.

```
struct D3D11_TEX2D_UAV {
    UINT MipSlice;
}
```

Listing 2.35. The members of the D3D11TEX2D_UAV structure.

The mip-map level is selected with the MipSlice member. The difference from the shader resource view subresource is that only a single mip-map level is allowed. This is shown in Figure 2.38.

The array-based 2D texture resources are also able to be exposed to the programmable shader programs with an unordered access view. This allows a single mip-map level to be specified over a subrange of the array elements. These selections are made with the D3D11_TEX2D_ARRAY_UAV structure in the D3D11_UNORDERED_ACCESS_VIEW_DESC structure, as shown in Listing 2.36.

```
struct D3D11_TEX2D_ARRAY_UAV {
    UINT MipSlice;
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.36. The members of the D3D11_TEX2D_ARRAY_UAV structure.

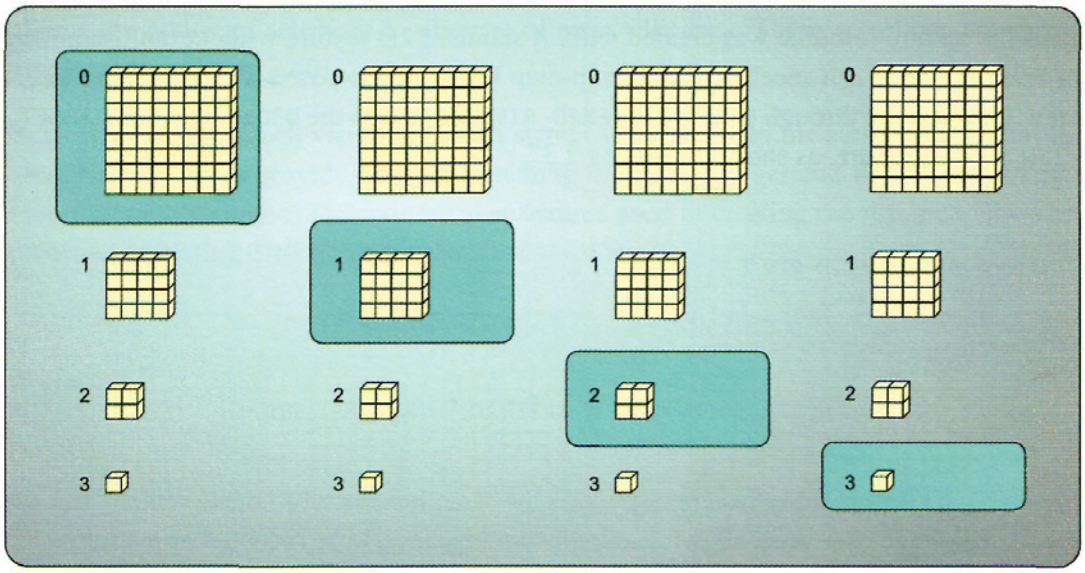

Figure 2.38. The various subresource selections available for 2D texture resources within unordered access views.

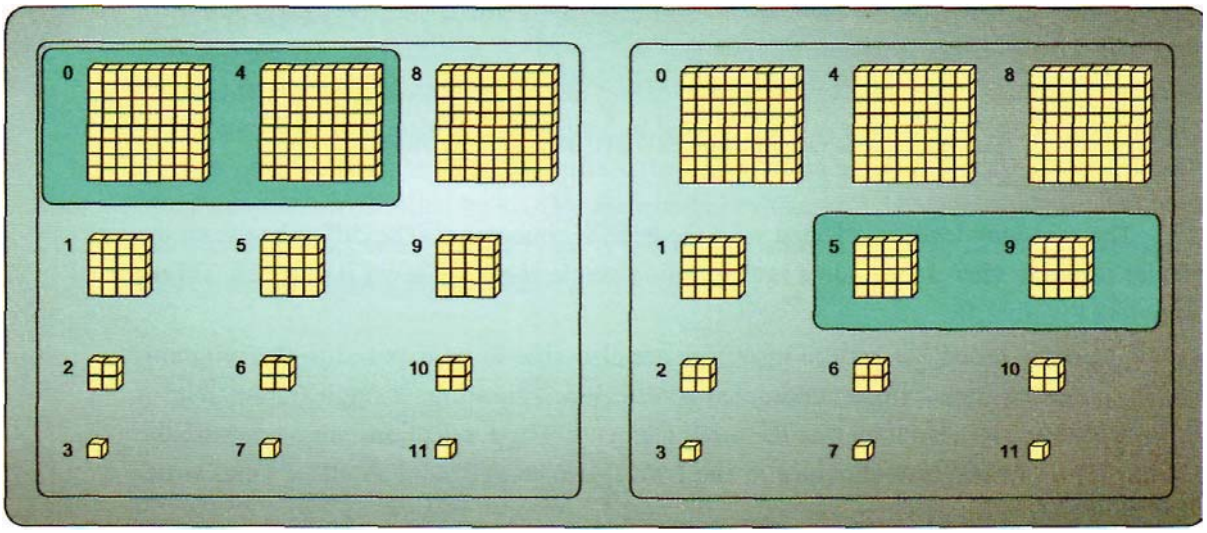

Figure 2.39. The unordered access view subresource selection for 2D texture array resources.

The array range selection is performed in the familiar manner, with the FirstArraySlice marking the beginning of the range, and the ArraySize specifying how many slices to use. For clarification, this subresource selection is demonstrated in Figure 2.39.

Texture2D render target view. As we have seen in the "Using 2D Textures" section, one of the primary uses for a 2D texture is as a render target. Four different configurations are allowed for these resources to be used with a render target view, depending on which options the texture resource was created with. A standard 2D texture without multisampling or array elements can specify a single mip-map level to be exposed with the render target view. This is done through the D3D11_TEX2D_RTV structure in the D3D11_RENDER_TARGET_VIEW_DESC structure, as shown in Listing 2.37.

```
struct D3D11_TEX2D_RTV {
    UINT MipSlice;
}
```

Listing 2.37. The members of the D3D11_TEX2D_RTV structure.

The 2D texture array resources provide the same type of subresource options that the unordered access view does, allowing a single mip-map level to be specified over a selectable range within the texture array elements. This is done through the D3D11_TEX2D_ARRAY_RTV structure within the D3D11_RENDER_TARGET_VIEW_DESC structure, as shown in Listing 2.38.

```
Struct D3D11_TEX2D_ARRAY_RTV {
    UINT MipSlice;
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.38. The members of the D3D11_TEX2D_ARRAY_RTV structure.

The multisampled 2D texture resource's primary responsibility is to receive the results of rendering operations as render targets. The render target views are the primary means of attaching these resources to the pipeline. The subresource selection structures are shown in Listing 2.39 for both the non-array and array versions of the resource.

```
struct D3D11_TEX2DMS_RTV {
    UINT UnusedField_NothingToDefine;
}
Struct D3D11_TEX2DMS_ARRAY_RTV {
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.39. The members of the D3D11_TEX2DMS_RTV and D3D11_TEX2DMS_ARRAY_RTV structures.

As you can see from Listing 2.39, there are no subresource selections possible for standard 2D multisampled texture resources. However, a 2D multisampled texture array allows the option of selecting a subrange of array elements. These selections are shown graphically in Figure 2.40.

Texture2D depth stencil view. The depth stencil view provides the same options that the render target views provide, making matching the render target and depth stencil target views relatively simple. The individual structures used in creating the resource views are provided in Listing 2.40 for reference.

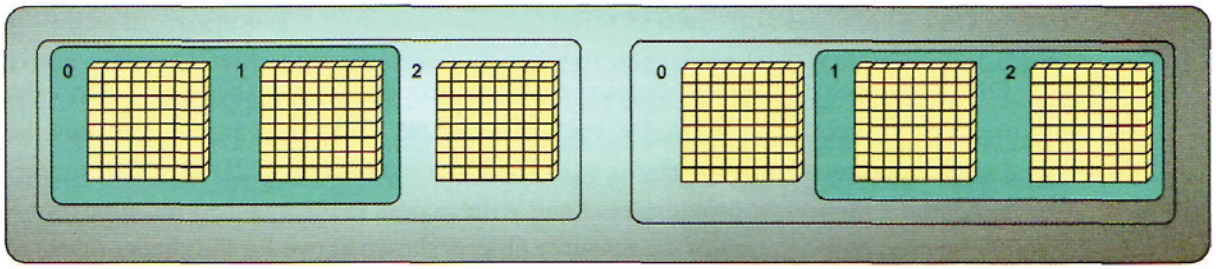

Figure 2.40. Various render target view subresource selections for a 2D multisampled texture array resource.

```
Struct D3D11_TEX2D_DSV {
    UINT MipSlice;
}
Struct D3D11_TEX2D_ARRAY_DSV {
    UINT MipSlice;
    UINT FirstArraySlice;
    UINT ArraySize;
}
Struct D3D11_TEX2DMS_DSV {
    UINT UnusedField_NothingToDefine;
}
struct D3D11_TEX2DMS_ARRAY_DSV {
    UINT FirstArraySlice;
    UINT ArraySize;
}
```

Listing 2.40. The various depth stencil view subresource selection structures.

HLSL objects. As we have seen in the 1D texture resource case, it is possible to interact with texture resources in HLSL by declaring an appropriate resource object. This resource object must then be matched with either a shader resource view or an unordered access view with an appropriate resource attached to it. When a shader resource view is used, the resource object allows only read access to the resource contents. This is reflected in the types of resource objects that can be declared for use with shader resource views. The read-only resource types are listed below.

Texture2D

Texture2DArray

Texture2DMS

Texture2DMSArray

TextureCube

TextureCubeArray

Of course, each of these resources specifies a particular type of shader resource view that it can be bound to. For example, the Texture2DMS HLSL resource object must be bound to a shader resource view that is connected to a multisampled 2D texture resource. The unordered access view provides read and write access to the attached resource, but it can only be used with a subset of the resource objects shown above for the shader resource views. The available resource objects are shown below.

- RWTexture2D
- RWTexture2DArray

As you can see, only the standard 2D texture resource and its array based counterpart can be used with unordered access views. This leaves out both multisampled resources and cube map textures, although the resources of the latter can still be manipulated, as with an RWTexture2DArray. An example HLSL declaration for each of these resource objects is shown in Listing 2.41.

```
Texture2D<float>             tex01;
Texture2DArray<int3>         tex02;
Texture2DMS<float4, 4>       tex03;
Texture2DMS<float2, 16>      tex04;
TextureCube<float3>          tex05;
TextureCubeArray<float>      tex06;
RWTexture2D<uint3>           tex07;
RWTexture2DArray<float3>     tex08;
```

Listing 2.41. Example resource object declarations in HLSL for the various types of 2D texture resources.

As can be seen in these declarations, the multisampled texture resource objects specify their format and can also optionally specify the number of subsamples that are used in the resource. This sample count used to be required in Direct3D 10, but since Direct3D 10.1, the count specification has become optional. Now it is possible to query the number of samples through the HLSL GetDimensions() method, making the direct specification unnecessary. This ensures that any methods that access the subsamples only try to use the appropriate number of samples. The semantics surrounding the usage of each of these resource objects will be covered in more detail in Chapter 6.

Texture 3D

The third texture type continues the trend and implements the expected increase in dimension to provide a 3D texture type. This texture type is organized similarly to the previous two textures, except that a third axis is added to the texture. It is essentially a 3D grid of texture elements, with each element consisting of one of the DXGI_FORMAT enumeration types. The various types of 3D texture resources are depicted in Figure 2.41.

As you can see from Figure 2.41, the 3D textures allow for single textures and mipmapped textures. Texture arrays are not supported in 3D textures. In addition, multi-sampling cannot be used with a 3D texture.

Using 3D textures. The use cases for 3D textures are typically somewhat more specialized than those for the previous two texture types. Due to the nature of a 3D representation, this

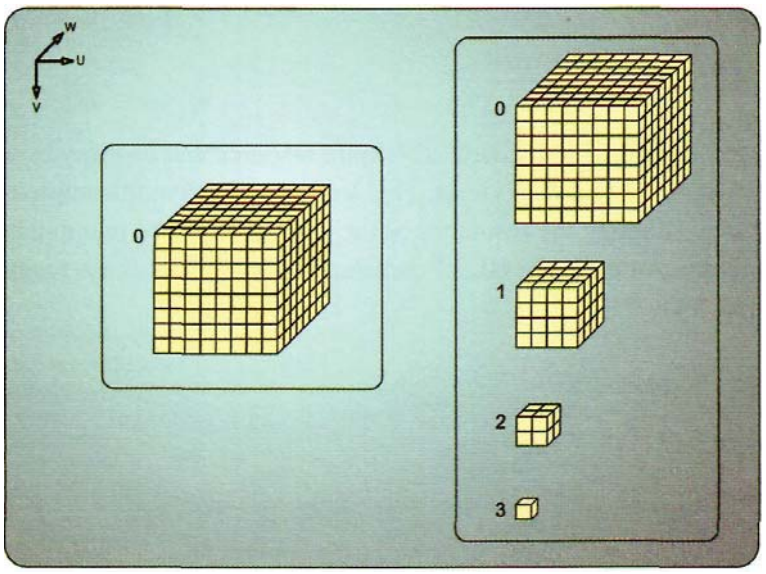

Figure 2.41. The various 3D texture resource configurations.

texture type requires a significant amount of memory to hold all of its data, and hence it is typically either limited in resolution or is used in cases where it is absolutely mandatory to have a full 3D representation. Of course, as the amount of memory in GPUs continues to increase, this situation will continually improve. At first 3D textures seem to be almost exactly like a 2D texture array, and in some ways they are very similar. The single biggest difference is that a texture sample performed an a texture array will only sample + filter from the specified texture slice, while a sample performed on a 3D texture will filter between two adjacent slices. This means that on a 3D texture, a single trilinear filtered sample will perform 16 memory fetches!

Even with the current performance and size limitations, many interesting and creative uses have been found for 3D texture resources. The storage of voxel data is perhaps the prototypical example. A *voxel* is an extension of a 2D pixel (as a picture element) to 3D (as a volume element). This data structure stores a scalar value at each point within a 3D grid. The data in the grid can then be used to extract isosurface information to produce a model. For example, we could extract a surface from the model where the scalar value is equal to 0.5. This is commonly done with the well-known marching cubes algorithm (Lorensen, 1987) or some variant of it. An easy-to-visualize example of such a voxel-based data set is the results of a magnetic resonance imaging (MRI) scan, where the biological material density of a patient is represented as a scalar value at each point within the 3D grid.

Other increasingly popular uses are global illumination lighting solutions. In these scenarios, light propagation is calculated over a complete scene volume, which is represented by a 3D texture. Since the 3D grid is limited in resolution, this provides a lower calculation cost than trying to perform ray-tracing, which more or less entails tracing

individual paths of light through a scene. Many recent research papers have been dedicated to these types of algorithms, such as seen in (Kaplanyan, 2010).

Creating 3D textures. As we have also seen in the 1D and 2D texture cases, the creation of the 3D texture is performed by passing a texture description structure to one of the device methods. The ID3DllDevice: :CreateTexture3D() method is used for 3D textures. It takes as input a pointer to a D3D11_TEXTURE3D_DESC structure, which is shown Listing 2.42.

```
struct D3D11_TEXTURE3D_DESC {
    UINT          Width;
    UINT          Height;
    UINT          Depth;
    UINT          MipLevels;
    DXGI_FORMAT   Format;
    D3D11_USAGE   Usage;
    UINT          BindFlags;
    UINT          CPUAccessFlags;
    UINT          MiscFlags;
}
```

Listing 2.42. The members of the D3D11_TEXTURE3D_DESC structure.

This structure is quite similar to the 1D and 2D versions. In a 3D texture, its dimensions are specified in the Width, Height, and Depth parameters. The number of mip-map levels and the element format are declared in the same manner as before, with the MipLevels and Format parameters. The Usage, BindFlags, and CPUAccessFlags parameters are used to determine the resource usage patterns, as described at the beginning of this chapter. Finally, the MiscFlags parameter provides the following optional flags for 3D textures:

- D3D11_RESOURCE_MISC_GENERATE MIPS
- D3D11_RESOURCE_MISC_RESOURCE_CLAMP

We have seen both of these flags before. The first flag indicates the ability to auto-generate mip-maps from a render target texture, and the second flag allows for capping the highest resolution mip-map level, which allows the driver to manage the GPU memory more flexibly.

Resource view requirements. The 3D texture is the only texture resource that is not available for use with all resource view types. Specifically, it can be used for shader resource views, unordered access views, and render target views, but cannot be used in conjunction with a depth stencil view. At first consideration, this would seem to violate the rule that a

depth stencil target always matches the size, dimension, and subsample count of the render target being used. However, for render target views, the 3D texture is actually treated like a 2D texture array, where each depth level of the texture acts like a texture slice. In this case, a depth stencil view would use a 2D texture array as its required resource type, allowing the render and depth target types to match. We will take a closer look at each of the available resource view configurations below.

Texture3D shader resource view. A single type of sub-resource specification is available for using texture 3D resources within a shader resource view. This is provided in the D3D11_TEX3D_SRV structure within the D3D11_SHADER_RES0URCE_VIEW_DESC structure. This structure is shown in Listing 2.43.

```
struct D3D11_TEX3D_SRV {
    UINT MostDetailedMip;
    UINT MipLevels;
}
```

Listing 2.43. The members of the D3D11_TEX3D_SRV structure.

As seen in Listing 2.43, a subset of the total mip-map levels can be chosen for use with a shader resource view, effectively hiding the other mip-map levels from the resource view. We have seen these mip-map level range selections before, with the most detailed mip-map level specified in the MostDetailedMip parameter, and the number of levels to

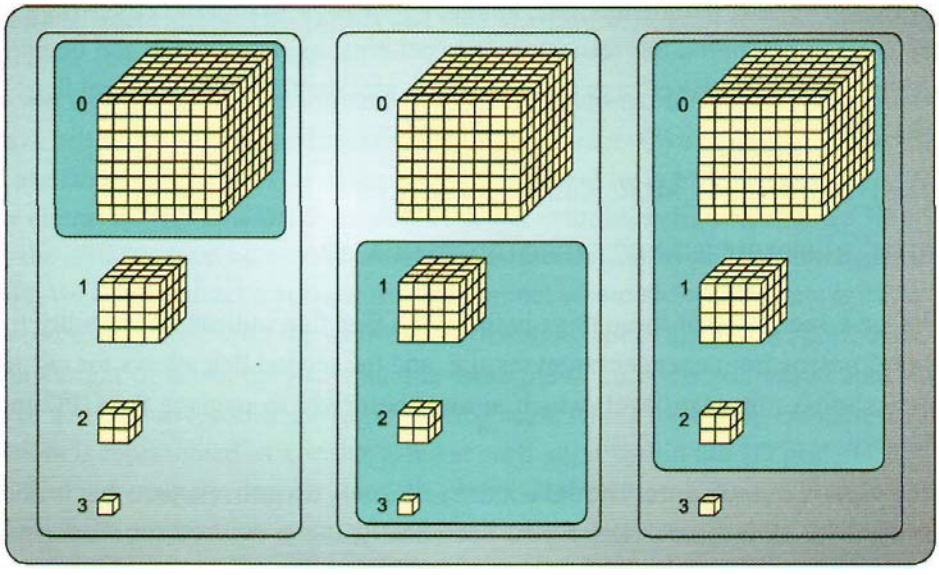

Figure 2.42. Various 3D texture subresource selections for use with shader resource views.

be used specified in the MipLevels parameter. A visualization of the mip level subset range is shown in Figure 2.42.

Texture3D unordered access view. The unordered access view also only supports one form of 3D texture access, which supplies a single mip-map level and allows for the selection of a depth range. This is performed with the D3D11_TEX3D_UAV structure within the D3D11_UNORDERED_ACCESS_VIEW_DESC structure, as shown in Listing 2.44.

```
struct D3D11_TEX3D_UAV {
    UINT MipSlice;
    UINT FirstWSlice;
    UINT WSize;
}
```

Listing 2.44. The members of the D3D11_TEX3D_UAV structure.

This structure allows for the mip-map level to be specified in the MipSlice parameter, while the range of depth slices is selected with FirstWSlice and WSize. This makes it fairly simple to use a single 3D texture resource for multiple different uses. This special range selection is visualized in Figure 2.43.

Texture3D render target view. The 3D texture resources can be used as a render target, although they are actually interpreted as 2D render target arrays. In this case, each depth

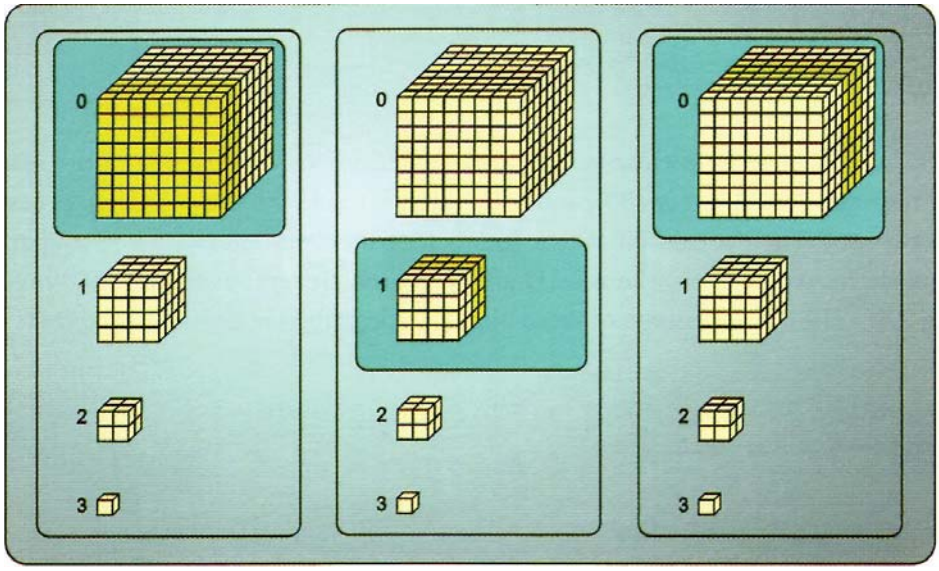

Figure 2.43. A 3D texture subresource selection for use with an unordered access view.

layer is interpreted as a separate array element. The subset of the resource to be bound with the render target view is specified in the D3D11_TEX3D_RTV structure within the D3D11_RENDER_TARGET_VIEW_DESC structure, as shown in Listing 2.45. This selection range is exactly the same as the one shown for the unordered access views above, so we will not repeat the description here.

```
struct D3D11_TEX3D_RTV {
    UINT MipSlice;
    UINT FirstWSHce;
    UINT WSize;
}
```

Listing 2.45. The members of the D3D11_TEX3D_RTV structure.

Texture3D depth stencil view. As mentioned above, the 3D texture cannot be used with the depth stencil view. Instead, a 2D texture array resource should be used in conjunction with a 3D texture render target.

HLSL objects. The 3D texture objects are used in shader programs in the same way that we have seen for 1D and 2D texture types. There are significantly fewer resource objects available for declaration and use, which corresponds to the reduced selection of resource view configurations. The shader resource view and the unordered access views can both be bound to a single matching resource object. The available objects are listed below.

- Texture3D
- RWTexture3D

By now it should be clear which of these objects can be used with the read-only shader resource view (Texture3D), and which can be used with the unordered access view (RWTexture3D). The declaration syntax for each of these resource types is similar to the other examples we have seen in the 1D and 2D cases. Several examples are provided in Listing 2.46. The detailed usage of these objects is described in depth in Chapter 6.

```
Texture3D<f!oat4>  tex01;
RWTexture3D<uint3>  tex02;
```

Listing 2.46. Sample HLSL resource object declarations for 3D texture resources.

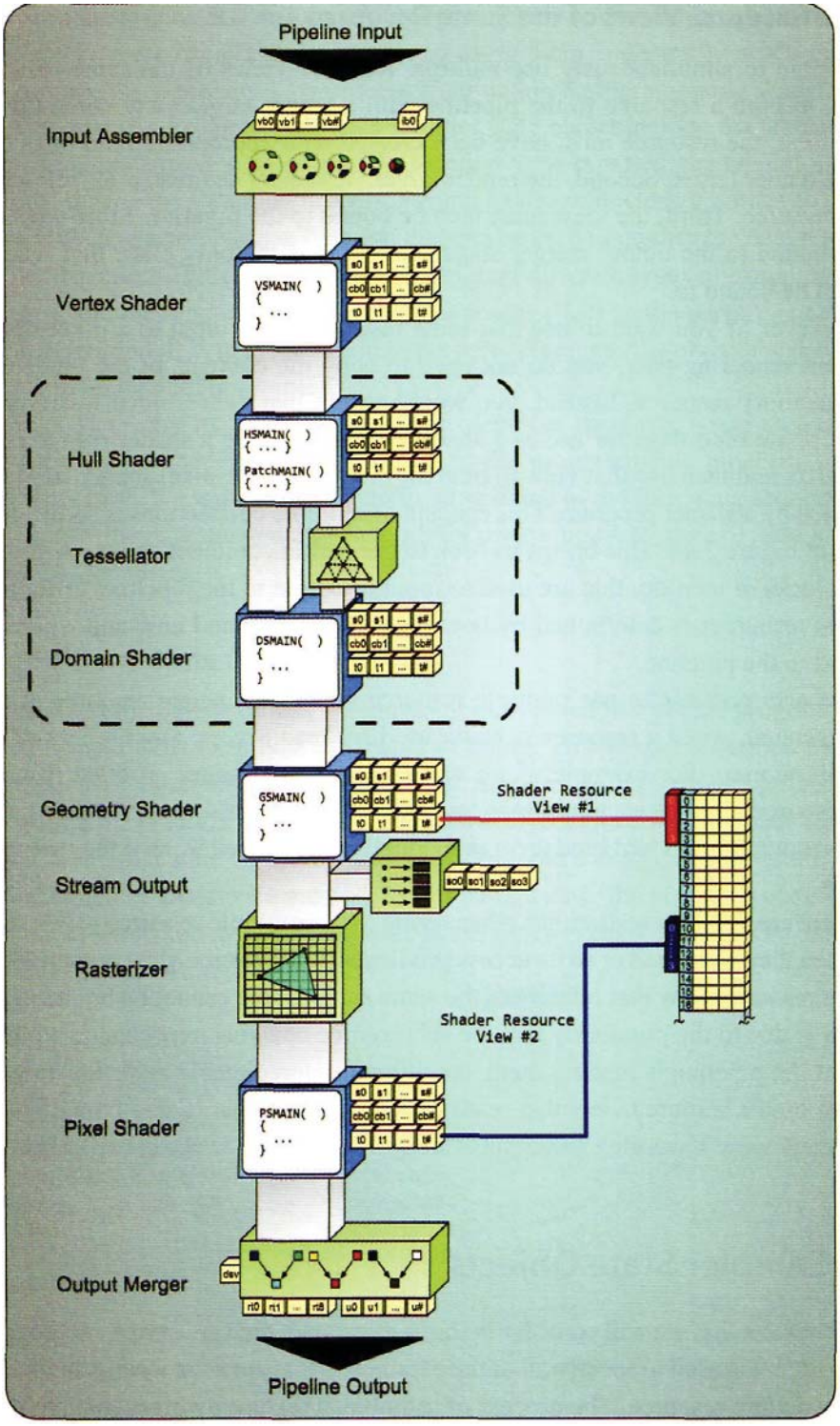

Figure 2.44. Using the same resource with different resource views in separate passes.

Multiple Resource Views of the Same Resource

It is possible to simultaneously use multiple resource views of the same resource. For example, to bind a resource to the pipeline with a render target view, three things must happen. First, the resource must have been created with appropriate bind flags for being used as a render target. Second, the render target view must be created for referencing the desired resource. Third, the view must then be bound to the pipeline. More specifically, it must be bound to the output merger stage, since that is the only place that render target views can be bound to.

However, if you want to use that same resource as an input to a pixel shader on a subsequent rendering pass, you do not need to copy the contents of the render target to another memory resource. Instead, you would ensure that the resource was created with the appropriate bind flags for use as a shader resource, create a shader resource view for the resource, and then use that view to bind the resource to the pixel shader stage, where it is then used by a shader program. This concept of multiple connections is shown in a block diagram in Figure 2.44. This brings us back to our earlier comment about resources—they are just blocks of memory that are used as input or output to the pipeline, or for both. The use of the resources is determined by how they are created, and how and where they are connected to the pipeline.

It is also possible to use multiple resource views of a single resource at the same time. In general, when a resource is being used for read access, any number of resource views can be used. For example, if we wanted to write a shader program that used one shader resource view for each mip-map level of a 2D texture resource, we could just create separate resource views and bind them individually to the pipeline, as if they were separate resources.

There are different restrictions when trying to use multiple resource views simultaneously when they have read or write access privileges. If a resource view is used for writing, any other resource view that references the same subresource cannot be bound to the pipeline. This is due to the possibility that the subresource contents may change while another portion of the pipeline is reading them, resulting in unpredictable behavior. In general, if you try to bind a resource view that reads from a resource that is also bound with a writable resource view, a warning and/or an error will be reported in the debug output window.

2.2.3 Sampler State Objects

The final objects that we will consider in this section are sampler `state objects`. A sampler state object is used to specify all of the parameters that are used when a shader program samples a texture resource. The process of sampling a texture involves looking up several pixels of texture resource and combining them with a mechanism selected by the developer, to reduce several types of common artifacts, as well as to increase the performance of

texture lookup operations. All major GPU designs include additional hardware functionality to implement these sampling operations, so using them is generally much faster than trying to emulate the same sampling process from within a programmable shader.

Options that can be specified with a sampler state object include the texture coordinate addressing mode used to determine the location to be sampled, the type of filtering to be performed during the sampling process, several LOD parameters, a border color to line the texture with, and a comparison function that can be used to modify the data returned by the sampling functions. This provides quite a range of different types of sampling that can be performed, and these options can have quite a potent effect on the performance of a texture sample operation. It is thus important to properly configure the object for the intended use case of the texture being sampled.

Sampler state objects are actually not resources in the same sense as buffers and textures, but they are managed like resources in the programmable pipeline, so we are including them here. We will explore them in more detail in the following sections by first considering how they are created and then briefly reviewing how they are used in the programmable shader stages.

Creating Sampler State Objects

Before looking at the details of how a sampler object is used in a shader program, we first need to understand what exactly it is capable of doing. This is best described by looking at the process used to create the sampler object, since all of its various options must be specified during the creation process. Like the other object types we have seen throughout this chapter, a sampler state object is created by filling out a description structure and then passing that structure to a device method. In this case, we use the ID3DllDevice:: CreateSamplerState() method to generate a ID3DllSamplerState object. The description structure is shown in Listing 2.47.

```
struct D3D11_SAMPLER_DESC {
    D3D11_FILTER               Filter;
    D3D11_TEXTURE_ADDRESS_MODE AddressU;
    D3D11_TEXTURE_ADDRESS_MODE AddressV;
    D3D11_TEXTURE_ADDRESS_MODE AddressW;
    FLOAT                      MipLODBias;
    UINT                       MaxAnisotropy;
    D3D11_COMPARISON_FUNC      ComparisonFunc;
    FLOAT                      BorderColor[4];
    FLOAT                      MinLOD;
    FLOAT                      MaxLOD;
}
```

Listing 2.47. The D3D11_SAMPLER_DESC structure and its members.

The first parameter is perhaps the most visible of all. The `Filter` structure member specifies the type of filtering that will be performed during the sampling process, in several different scenarios. Texture minification, magnification, and mip-mapping are all different types of sampling operations to which different filter qualities can be applied.

Texture minification occurs when a texture appears far enough from the viewer that its individual pixels (often referred to as *texels,* which is short for *texture pixels)* are smaller than the output render target pixels. Because the texels from the texture are smaller than the screen pixels, the texture can appear to sparkle or pop as multiple texels pass through a pixel from frame to frame. If an appropriate filtering mechanism is chosen, this effect can largely be reduced, or even eliminated. *Texture magnification* occurs in the opposite situation. When a texture is viewed up close, a single texel will cover multiple screen pixels, resulting in a "blocky" appearance. In some cases, this is the desired behavior, but in other cases a better quality image can be produced if the texture is filtered to reduce the visibility of the large texels. The final texture sampling situation to consider is *mip-mapping.* We have already discussed mip-mapping in the discussion of texture resources earlier in this chapter. Essentially, a mip-map provides a multi-resolution representation of the texture resource, and it is possible to specify the type of filtering that is performed during the mip-map sampling process.

The description structure has one filter parameter, which indicates that all three of these filtering scenarios are specified with a single enumeration. The available filtering modes are shown in Listing 2.48. As you can see, each enumerated filter type specifies how it will sample the texture under the three conditions mentioned above—MIN corresponds to texture minification, MAG corresponds to texture magnification, and MIP corresponds to texture mip-mapping.

```
enum D3D11_FILTER {
    D3D11_FTLTER_MIN_MAG_MIP_P0INT,
    D3D11_FILTER_MIN_MAG_P0INT_MIP_LINEAR,
    D3D11_FILTER_MIN_P0INT_MAG_LINEAR_MIP_P0INT,
    D3D11_FILTER_MIN_P0INT_MAG_MIP_LINEARJ
    D3D11_FILTER_MIN_LINEAR_MAG_MIP_P0INTJ
    D3D11_FILTER_MIN_LINEAR_MAG_P0INT_MIP_LINEARJ
    D3D11_FILTER_MIN_MAG_LINEAR_MIP_P0INTJ
    D3D11_FILTER_MIN_MAG_MIP_LINEARJ
    D3Dll_FTLTER_ANISOTROPIC,
    D3Dll_FILTER_COMPARISON_MIN_MAG_MIP_POINT,
    D3Dll_FILTER_COMPARISON_MIN_MAG_POINT_MIP_LINEAR,
    D3Dll_FILTER_COMPARISON_MIN_POINT_MAG_LINEAR_MIP_POINTJ
    D3D11_FILTER_C0MPARIS0N_MIN_P0INT_MAG_MIP_LINEAR,
    D3D11_FILTER_C0MPARIS0N_MIN_LINEAR_MAG_MIP_P0INT,
    D3Dll_FILTER_COMPARISON_MIN_LINEAR_MAG_POINT_MIP_LINEAR,
    D3Dll_FILTER_COMPARISON_MIN_MAG_LINEAR_MIP_POINTJ
    D3Dll_FILTER_COMPARISON_MTN_MAG_MIP_LINEARJ
    D3Dll_FILTER_COMPARISON_ANISOTROPICJ
    D3D11_FILTER_TEXT_1BIT
}
```

Listing 2.48. The available filter types in the `D3D11_FILTER` enumeration.

The available types of filtering for each of these scenarios are point sampling, linear sampling, and anisotropic sampling. *Point sampling* simply returns the texel that the input texture coordinates happen to fall on. This is the least expensive form of sampling, but it also produces the lowest image quality in most situations. *Linear sampling* provides a higher quality of sampling, in which the texels that surround an input texture coordinate are interpolated to find an approximated value somewhere between the selected texels. This produces smoother transitions between texels, but it requires additional texels to be read, in addition to performing some arithmetic on the sampled data before returning the result. The final option is to utilize *anisotropic sampling,* a much higher-quality sampling technique. When a texture is viewed at any angle that is not perpendicular to the camera's line of sight, anisotropic filtering performs a number of samples to determine the average color that is visible to the viewer and returns this combined color. The maximum number of samples is specified in the MaxAnisotropy parameter of the sampler state description structure. While this filtering mode produces the best results, it can be extremely expensive to use, due to the much higher bandwidth and computation needed to produce the final sample value.

In the first half of the D3Dll_FILTER enumeration, we see all of the various combinations of filtering types that can be used. In the second half of the enumeration, we see a repeat of the list from the first half, except for the addition of COMPARISON in the value names. This indicates that each of the individual samples used in the sampling process will be used in the comparison function (which will be discussed shortly) prior to combining the result. This allows the result of the comparisons to be filtered, instead of filtering the texture first and then performing a comparison.

The next three options in the sampler description structure specify the texture addressing modes in the *U, V,* and *W* directions that correspond to the *X, Y,* and *Z* coordinates of the texture.[3] These modes let you specify what action to take when a texture coordinate that lies outside of the [0,1] range is used during sampling. A different addressing mode can be specified for each direction. You can have the sampler wrap the texture coordinates back around to 0 when they are greater than 1, effectively "wrapping" the texture over again. If the texture coordinate is greater than 2, the texture would be wrapped again, and so on. This allows many copies of a texture to appear on a textured surface. Another possible mode is the *mirror* addressing mode, which essentially flips the texture at every integer coordinate. This also allows many copies of the texture to be visible, but it flips the orientation for each consecutive copy of the texture. You can also specify a *clamp* mode, in which the texture coordinates are "clamped" to the [0,1] range. This effectively makes the boundary values of the texture appear at any location outside of the texture. An alternative to this mode is a *border addressing* mode, in which the sampler state can specify a replacement color to return when the coordinates are outside of the [0,1] range. The border color is specified in the BorderColor parameter of the sampler description object. Finally, you can use a

[3] These coordinates are subject to the same dimensionality that was described for the texture coordinates. For example, a 1D texture does not have a notion of a *Y* or Z coordinate, while a 3D texture has all three.

mirror once mode, in which the absolute value of the texture coordinates are used and then clamped in the same way as in the clamp texture addressing mode.

The MipLODBias, MinLOD, and MaxLOD parameters are all used to manipulate the mip-map level that is used during the sampling process. The min and max parameters specify the minimum and maximum levels that can be accessed during sampling, and the bias parameter provides a constant offset to add to the mip-map level selected by the sampling hardware. In all of these parameters, 0 is the most detailed mip-map level, so adding a positive bias will select a less detailed mip-map level.

The final member in the sampler state description structure is the ComparisonFunc parameter. This indicates what type of comparison should be performed on the texture samples prior to performing the filtering operation. The other value that is used in the comparison is supplied in the texture object sampling method call. The result is either a 1 if the comparison passes, or a 0 if it fails. The results of each individual comparison are then combined according to the chosen filtering method, and returned.

Using Samplers

To use a sampler object, it must first be created as detailed above and then bound to one of the programmable pipeline stages. Each of the programmable stages can simultaneously bind up to 16 different sampler objects, and the same sampler state object can be used in more than one programmable stage at once. After the state objects have been bound to a pipeline stage, the HLSL program that will run in that stage must declare an appropriate HLSL Sampler-State object, which corresponds to a particular sampler slot. Then the shader program can pass the SamplerState object to one of the many sampling methods provided by the various texture resource types. By having to pass the SamplerState object as an argument in the sample methods, D3D11 allows multiple texture resources within a shader program to simultaneously use the same sampling object. Due to the large number of available resources that can be used in a shader program (up to 128), sharing these states becomes a significant benefit if the same set of options is needed many times.[4]

2.3 Resource Manipulations

We have seen throughout this chapter what resource types are available, their typical usage, how they are constructed, and how to declare them for use in HLSL. We also saw in the

[4] In previous versions of Direct3D, the sampler could only be used with a single texture at a time. This limited the overall number of textures that could be used in a shader program. Without this limitation, Direct3D 11 provides a significantly larger number of textures (or more specifically resources) that can be accessed in a shader.

"Resource Creation" section that resources are created with very specific usage and access patterns. Depending on what a resource will be used for, it can be advantageous to only allow it to be accessible by the GPU. In other cases, it may be necessary to be able to directly manipulate the resources in the host C/C++ application.

Direct3D 11 provides a number of different techniques to modify or manipulate the contents of a resource. We will explore each of these methods in detail in this section, and will also consider how each of these methods provides useful operations in different situations.

2.3.1 Manipulating Resources

The first group of methods that we will look at is used to modify the contents of a resource. As we will see shortly, it is also possible to read the contents of a resource with some of the methods.

Mapping Resources

The primary way to manipulate the contents of a buffer for both reading and writing from the CPU is the *Map / Unmap* method of the device context. If a resource has CPU read or write access flags set, an application can map the contents of the resource to system memory. The *map* and *unmap* method prototypes are shown in Listing 2.49.

```
HRESULT Map(
    ID3DllResource *pResource,
    UINT Subresource,
    D3D11_MAP MapType,
    UINT MapFlags,
    D3D11_MAPPED_SUBRES0URCE  *pMappedResource
);
void Unmap(
    ID3DllResource *pResource,
    UINT Subresource
);
```

Listing 2.49. The method prototypes of the map and unmap methods.

The resource and subresource to be mapped are specified in the pResource and Subresource parameters, respectively. Next, the mapping type is specified in the MapType parameter, which indicates if the resource will be read or written, and how any writing will be performed. The members of the D3D11_MAP enumeration are shown in Listing 2.50.

```
enum D3D11_MAP {
    D3D11_MAP_READ,
    D3D11_MAP_WRITE,
    D3D11_MAP_READ_WRITE,
    D3D11_MAP_WRITE_DISCARD,
    D3D11_MAP_WRITE_N0_0VERWRITE
}
```

Listing 2.50. The members of the D3D11_MAP enumeration.

The first three values in the enumeration are fairly self-explanatory. If the resource is to be read, it must have been created with the D3D11_CPU_ACCESS_READ flag, and if the resource is to be written to, it must have been created with the D3D11_CPU_ACCESS_WRITE flag. When reading a resource, its contents will be available in the D3D11_MAPPED_SUBRESOURCE structure pointed to by the pMappedResource parameter. When writing to a resource, the same D3D11_MAPPED_SUBRES0URCE structure is used by the application to know where to write the desired resource data to. When both reading and writing, the contents of the resource are made available and can then be overwritten by the application, thus "mapping" the contents of the resource to system memory. The call to unmap the resource will make any written changes take effect in the resource.

The D3D11_MAP_WRITE_DISCARD and D3D11_MAP_WRITE_NO_OVERWRITE flags are used together to allow for dynamic resource usage. The D3D11_MAP_WRITE_DISCARD mapping type invalidates the contents of the resource when it is mapped, even if only a subresource is currently being mapped. The D3D11_MAP_WRITE_NO_OVERWRITE mapping type allows writing to a portion of a buffer that has not previously been updated since the last call to D3D11_MAP_WRITE_DISCARD. In this way, the contents of the resource that have been loaded into a dynamic resource may start to be used while other portions of the resource are still being filled up.

After the resource has been read or written from the mapped subresource structure, the resource must eventually be unmapped to allow the GPU to continue using it. This is done simply with the ID3D11DeviceContext: :Unmap() method, which identifies the resource and subresource index to be unmapped.

An important consideration for the mapping of resources is that a complete subresource must be mapped at a time. This means that it is not possible to read/write a portion of a subresource, although the entire subresource can be mapped while only updating a smaller portion of it. Depending on the size of the subresource, this can involve a large amount of memory being transferred to and from system memory. In some cases this is unavoidable, but in other cases it would be beneficial to have a method for updating only a portion of the subresource. We will see such a method in the "Update Subresource" section.

A common example of using the map and unmap methods is to update the contents of a constant buffer before using it in a pipeline execution. In this case, the resource would be mapped with the D3D11_MAP_WRITE flag, since we are not interested in reading the contents

of the constant buffer. Instead, we will just write the new data into the buffer. Once the data has been written to the system memory provided with the pMappedResource structure, the resource would be unmapped and could then be bound to the pipeline for use.

Update Subresource

The other mechanism that can be used to modify the contents of a resource is the device context's UpdateSubresource() method. This method differs from mapping of resources in that it only writes to the resource and does not allow reading its contents. The prototype of this method is shown in Listing 2.51.

```
void UpdateSubresource(
    ID3D11Resource *pDstResource,
    UINT DstSubresource,
    const D3D11_B0X *pDstBox,
    const void *pSrcData,
    UINT SrcRowPitch,
    UINT SrcDepthPitch
);
```

Listing 2.51. The update subresource method prototype.

In Listing 2.50 we can see that data to be written to the resource is given a pointer to its memory location. This is accompanied by the row pitch and depth pitch of the data block, measured in bytes. The destination is identified with a pointer to the resource and a subresource index. It is possible to write to only a portion of a subresource, by identifying it within a D3D11_B0X. While the source data is provided in terms of bytes and is thus format dependent, the destination is provided in indices and is not dependent on the format of the resource.

In contrast to methods for mapping a resource, the update subresource method can write to only a portion of a subresource. If only a small part of a large resource needs to be updated, the update subresource method can perform better under some conditions, since it would use less bandwidth.

An example of such a resource update could be changing the contents of a texture based on an action taken by the user. If the user selects an option that requires modifying a small portion of a texture, this can easily be performed using the update subresource method.

2.3.2 Copying Resources

There are many situations when the contents of one resource must be copied to another. This can be done for the needs of a particular algorithm, or to copy data to a second resource that has different access capabilities. In both cases, multiple methods are available

to perform the copying. We will examine each of these methods and then present a few situations they can be employed in. These methods are distinct from those presented in the "Manipulating Resources" section, in that all of the data that is being moved is already located within a resource, while the previous methods were used to read or write data with the CPU.

Copy Resource

To copy the complete contents of one resource to another resource, the device context's copy resource method would be used. The prototype of this method is shown in Listing 2.52.

```
void CopyResource(
    ID3DllResource *pDstResource,
    ID3DllResource *pSrcResource
);
```

Listing 2.52. The copy resource method prototype.

This is a fairly simple method, which only takes pointers to the source and destination resource. The resources must be the same type and size and have compatible formats. Neither immutable resources nor depth stencil resources can serve as the destination. In addition, multisampled resources cannot be used as either the source or destination (we will see later in this section how to retrieve the contents of a multisampled resource).

Since this method copies the complete resource, it is most likely to be used to copy data to the second resource with the desired usage configuration. For example, if a compute shader is used to generate a data buffer and the application needs to save that buffer to the hard disk, it cannot directly map and read the buffer. This is because a buffer used for the unordered access view must have the default usage, which means it cannot be read by the CPU. This is true of all output from the pipeline—only resources with the default usage can receive output from the pipeline, and the CPU cannot directly read resources that have default usage. Instead, a second buffer resource would be created with the `staging` usage, and the contents of the default buffer would be copied to the staging buffer. Resources with the staging usage can be read by the CPU, which solves the problem of getting the data to the application.

An example of using the copy resource method would be to retrieve the contents of an append structured buffer that has been filled with data from the compute shader. Since the append structured buffer has the default usage flags, its contents would need to be copied to a secondary staging buffer using the CopyResource() method. The staging buffer could then be used by the CPU to read the contents using the map/unmap methods.

Copy Subresource Region

The next technique for copying resources is the device context's CopySubresourceRegion() method. This method allows a portion of a resource to be copied, as opposed to always copying the complete resource, as is required in the CopyResource() method described above. Listing 2.53 shows the prototype of the method.

```
void CopySubresourceRegion(
    ID3D11Resource *pDstResource,
    UINT DstSubresource,
    UINT DstX,
    UINT DstY,
    UINT DstZ,
    ID3DIIResource *pSrcResource,
    UINT SrcSubresource,
    const D3D11_B0X *pSrcBox
);
```

Listing 253. The copy subresource region method prototype.

Since this method allows copying a subset of one resource to another, the resources do not need to have the same size. However, the two resources must have the same type, and compatible formats. As with the source, the destination can be a subresource. This is specified by using a zero-based subresource index in the DstSubresource parameter. In addition, a destination offset can be specified in each of the three coordinates, to positioning the copied data. The source resource data is also determined with a subresource index, and the region to be copied is determined with a D3D11_BOX selection.

The ability to selectively copy portions of a resource to another resource can be used to create dynamic texture atlases, which combine several textures into a single larger texture. This can reduce the number of individual draw calls needed to render a scene (topics such as these are discussed in more detail in Chapter 3).

Copy Structure Count

One final method allows copying a portion of one resource into another resource, although this is a less direct copying technique. The device context's CopyStructureCount() method allows copying a buffer resource's hidden counter into another resource. The method prototype is shown in Listing 2.54.

```
void CopyStructureCount(
    ID3DIIBuffer *pDstBuffer,
    UINT DstAlignedByteOffset,
    ID3D11UnorderedAccessView *pSreView
);
```

Listing 2.54. The copy structure count method prototype.

As we have seen earlier in this chapter, there are two unordered access view description flags that create a hidden counter for managing the contents of a buffer. The first is the D3D11_BUFFER_UAV_FLAG_APPEND flag, which lets a buffer be used as an append/consume buffer. The second is the D3D11_BUFFER_UAV_FLAG_COUNTER flag, which directly adds the hidden counter for manipulation with the IncrementCounter() and DecrementCounter() methods in HLSL. The CopyStructureCount() method works by taking a pointer to the UAV that contains the counter as the source and then copies that value into a destination buffer at the offset specified in the DstAlignedByteOffset parameter.

The ability to copy the counter value into another buffer has uses in indirect rendering operations. When a buffer is filled with vertex data by a compute or pixel shader program, it can be used to render a model *indirectly* with a buffer resource that indicates how many primitives to render. The counter value can be placed at the proper location in the buffer by using the appropriate index when copying it. Then, the application simply passes the indirect buffer to the draw call. Indirect rendering has been discussed in the "Buffer Resources" section of this chapter, and is also covered in Chapter 3.

This functionality can also be used to stage the structure count data for eventual reading by the CPU. This is performed in the same way we described in the "Copy Resource" section.

2.3.3 Generating Resource Contents

There is also a pair of device context methods that can be used to generate the contents of a resource. We discuss both of these methods here.

Generate Mips

Earlier in this chapter, we saw the miscellaneous resource creation flag for indicating that a texture resource should be able to generate the lower-resolution mip-map levels when its top level is rendered. This is indicated with the D3D11_RESOURCE_MISC_GENERATE_MIPS flag. The generation of the mip-map level data is initiated with the ID3D11DeviceContext::GenerateMips() method. The method prototype is shown in Listing 2.55. The method only takes a pointer to the shader resource view to be updated, whose attached resource must have been created with the flag mentioned above. In addition, the number of mip-map levels that will be generated is defined by the subresource range selected by the shader resource view.

```
void GenerateMips(
    ID3D11ShaderResourceView *pShaderResourceView
);
```

Listing 2.55. The generate mips method prototype.

Resolve Subresource

The other method that can be used to generate resource content is the ID3DllDeviceContext ::ResolveSubresource() method. This is used to take a source multisampled texture resource and use the subsamples to calculate the final color value of the corresponding pixel in a non-multisampled destination resource. This is a required step before displaying the contents of a multisampled render target on the screen, and hence is a vital operation for high-quality rendered image output. In the case of a DXGI swap chain, it supports automatically performing the resolve function during the buffer swap triggered with the Present() method. This technically allows the application to skip the manual resolve process in some cases, but if a multisampled render target is used as an input to any further algorithms (such as any post-processing techniques) the render target will need to be resolved well in advance of any per-pixel algorithms being applied to it. The prototype of the method is shown in Listing 2.56.

```
void ResolveSubresource(
    ID3D11Resource *pDstResource,
    UINT DstSubresource,
    ID3DllResource *pSrcResource,
    UINT SrcSubresource,
    DXGI_FORMAT Format
);
```

Listing 2.56. The resolve subresource method prototype.

This method allows for subresource indices to be provided for both the source and destination resource. This also means that a complete subresource must be resolved at the same time, and it does not allow portions of a subresource to be manipulated in isolation. In addition, a format identifier is passed to indicate the format to be used when the two resources have compatible TYPELESS formats.