7 Multithreaded Rendering

7.1 Introduction

As we have discussed throughout the first half of this book, Direct3D 11 has introduced some very interesting and powerful new concepts and abilities to its arsenal. However, some of the most important new features of the API revolve around *multithreading*. The average number of CPU cores in a typical user's PC has been steadily increasing, while the frequencies at which those cores operate have plateaued, and this trend is expected to continue for the foreseeable future (Sutter). With the availability of these additional processing cores, the developer must find techniques to use them for tasks that have traditionally been performed in single threads of execution. It is in the developer's interest to convert as many tasks as possible to use parallel processing.

Direct3D 11 has been specifically designed to allow an application's rendering system to take advantage of multiple cores, and hence, of multiple threads of execution. The core interfaces that an application uses to interact with Direct3D 11 have been carefully designed to exhibit well-defined behaviors in multithreaded programs. This promotes the use of multithreaded programming, while at the same time not explicitly requiring it either. All prior versions of Direct3D had either poor or non-existent multithreading support, making D3D11 a fundamentally different type of API. This also requires a fundamental review of existing software designs to take advantage of these new abilities. As we will see later in this chapter, these multithreading, capabilities have been designed to provide the potential for increased rendering performance by spreading the work required to render

a frame across multiple CPU cores when they are available. When used properly, these designs can provide flexibility and the ability for your program's performance to scale with an increasing number of CPU cores in a user's system. This is referred to as the *scalability* of the software, and it will become increasingly more important as the average number of CPU cores continues to increase.

This chapter introduces and discusses the tools that have been added to the Direct3D 11 API to facilitate multithreaded rendering operations. This is distinct from, and should not be confused with, *general multithreaded computation.*[1] Here, we are concerned with using multiple threads of execution to render a frame as quickly as possible, without considering the other processing tasks that need to be performed within the application. However, if the time required to perform a frame's rendering requests is minimized by the use of multiple threads, the overall performance of the application improves, leaving more processing time for other tasks. These new Direct3D 11 tools allow the developer to use multiple threads for two types of operations—resource creation, and *draw submission* sequences. These are the two primary tasks that any Direct3D 11 application performs, creating the objects it needs, and then using them to feed the GPU with work to keep it busy. Efficiently performing these two key tasks lets a program use multithreaded code for the majority of the interactions with the D3D11 API, allowing its performance to scale with additional CPU cores.

7.2 Motivations for Multithreaded Rendering

Before we dive into the details of the Direct3D 11 threading model, it is important to understand what we want to be able to achieve with by introducing multithreading to rendering-related operations. After all, if there is no benefit, why should we introduce the complexity of multithreading into our rendering code? The truth is that there are very tangible reasons to want to break the rendering code path out of serial execution. The areas that we will focus on in this section are *resource creation andpipeline manipulation.*

During these discussions, it is useful to understand how other recent iterations of Direct3D were used. In both D3D9 and D3D10, it was possible to create a device with a *multithreaded* flag that would make the methods of the rendering device thread-safe. However, this thread-safe property was achieved by using coarse synchronization primitives to effectively serialize access to the code within the methods. Therefore, if two threads simultaneously called the same device method, one thread would have to wait until the other method invocation was completed. This hinders any performance gains from using

[1] Many resources are available to begin exploring this vast topic. The reader is encouraged to view the following threading library resources as a place to start (Microsoft Corporation) (OpenMP Architecture Review Board).

multiple threads, because of the frequent synchronization forced through the API. For this reason, the general advice when using these APIs was to only interact with the API from a single thread, often referred to as the *rendering thread.* This lets a program use general multithreading for other areas of the program, but restricts all rendering operations to being performed from a single serial thread of execution.

7.2.1 Resource Creation

The topic of resource creation spreads across a wide variety of tasks, all of which must interact with the Direct3D 11 API to create API resources. This includes the simple object creation tasks such as shader objects and pipeline state objects, and also extends to device memory based resources such as buffers and textures. To gain an insight into how multithreading can help with this type of operation, we can consider a few use cases from traditional serial execution applications.

A simple application structure can create all of the API based resources that it needs at startup, and then simply use the loaded resources during the execution of the application. Since we are considering serial execution of these tasks, the application must create all of its resources before proceeding into its normal operation. Depending on the number of resources to create, and how complex they are to load, this can lead to a significant time period during which the user is waiting to use the program. Resources such as textures require reading texture data from the hard disk, which can introduce delays for each read request. Other types of resources, such as shader objects, require some processing to be performed on them before they are ready to use. In the case of a shader, the source code for the shader must be compiled before the shader object is created by the API. When there are many of these tasks to perform, the net effect can produce a significant overall delay.

In more complex application structures, it is common for the application to require more resources than can fit into memory. A flight simulator provides a typical scenario, in which there is simply too much data to naively create resources for the entire run of the program. In this scenario, the terrain and world object data must be dynamically loaded into memory depending on where the viewer currently is within the scene. This means reading the data from hard disk (or perhaps over a network connection) during the execution phase of the program. Depending on how large the amount of data to be loaded is, this can cause a short-term reduction in the frame rate of an application, which is very objectionable for the user.

If a user has a multicore CPU, the impact of these two issues can be somewhat reduced in some cases. As described above, in recent modern iterations of Direct3D, the APIs can be used in a multithreaded environment even though they don't directly support multithreading themselves. This means that in situations where a delay is caused by reading from the hard disk, a program could create and use a secondary thread to load the desired

data into memory, and could then allow the main rendering thread to use the data and create the API resource. This technique provides some relief from the delays described above, but unfortunately, it is not available or applicable in all situations. For example, when creating shader objects out of the compiled byte code, multiple threads are not helpful, since the device is only accessible from one thread.

In older Direct3D editions, the general use of multiple threads for resource creation is limited to trying to find ways around the limitations imposed by the fact that the API was not designed for use in a multithreaded environment. This adds complexity to the overall application structure, and it requires additional development time to ensure that all of the interactions between secondary threads and the rendering thread are properly implemented.

7.2.2 Draw Submission Sequences

The second area of rendering that can benefit from multiple threads of execution is draw submission sequences. These consist of all interactions that the application has with the pipeline, including all pipeline state settings, and any pipeline executions done with draw calls or `dispatch` calls. Once an application moves out of the startup phase and into the normal execution mode, this is the primary work that gets performed to render a frame. This type of work is depicted in Figure 7.1, with several state changes followed by draw calls.

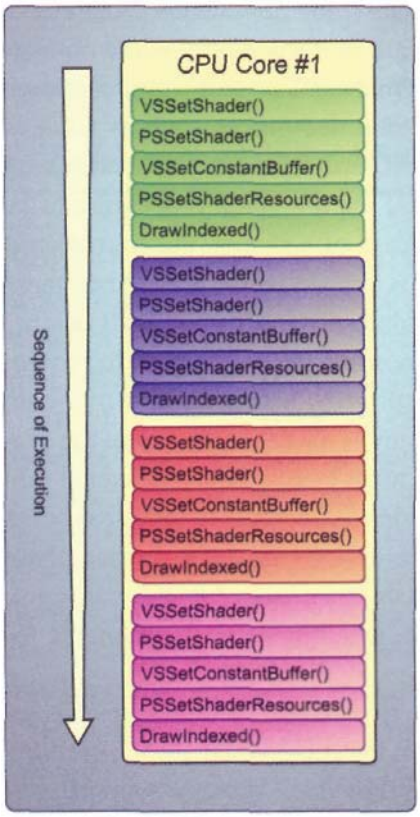

Figure 7.1. A series of API calls required to perform a rendering operation.

As you can see in Figure 7.1, the stream of API calls are grouped together to render one object after another, with each object identified by a different color. You can also see that all of these API calls are performed in a sequential manner, because rendering operations are traditionally restricted to a single thread. Due to the high number of these API calls that are commonly used to render a single frame, the CPU may become a bottleneck to the application's overall performance. A large portion of the driver model changes made between D3D9 and D3D10 were intended to reduce the amount of overhead for each of these API calls, and hence, to increase the maximum number of calls programmers could make for a given frame while still achieving a target frame rate. In fact, many optimization techniques for the

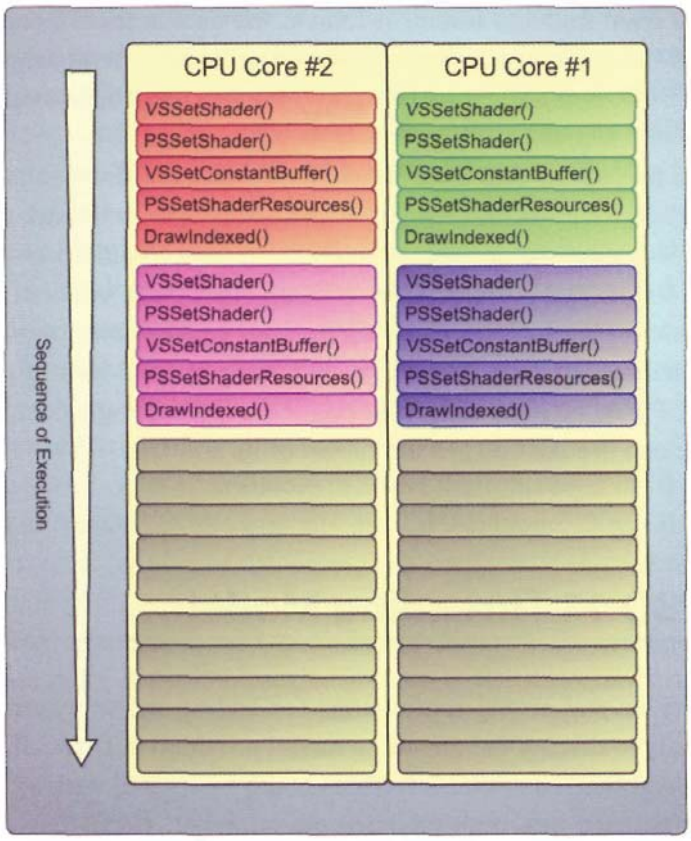

Figure 7.2. Parallelizing the API calls required for a rendering operation.

older APIs are geared toward increasing *batch sizes. A batch* is a group of objects that can be rendered either together, in the same draw call, or one after another, without other state changes—all aimed at reducing the number of API calls that need to be executed to render a frame.

This is clearly a task that could benefit from parallelization—the operations being performed are already neatly grouped and can be executed independent of one another. The colored groups could easily be split up and processed by other threads, effectively dividing the time used to submit the API calls by the number of available processing cores. That is, if the rendering API allowed the calls to be submitted on multiple threads, which up until Direct3D 11 they could not be! Such a parallel execution system is shown in Figure 7.2.

This can also be taken a step further, with the concept of using objects to represent a series of API calls. These objects, which are commonly referred to as *display lists* or *command lists,* could be generated once by recording a set of all of the desired API functions. Then the list could be executed quickly and efficiently by the driver, with minimal CPU time required to submit all of the API calls (since they aren't necessary anymore). The list

can also be reused from frame to frame, as long as the data in them doesn't change. These types of command list objects have been implemented in other rendering systems (such as OpenGL or the XBOX 360 variant of Direct3D) but have not been available in D3D9 or D3D10.

In summary, we can say that the overall desire regarding draw submission sequences is to increase the scalability of the rendering software. If the software can correctly parallelize its workload, then we are able to reduce the time required to perform the CPU work for a given frame, and automatically become even more efficient as the number of CPU cores increases. This includes both spreading the API calls over multiple threads, and reducing the number of serialized API calls with command lists. If we can reduce the potential for the CPU to be a bottleneck in the overall rendering system, we increase the likelihood that we can use the GPU to the fullest of its abilities.

7.3 Direct3D 11 Threading Model

Now that we have several strong motivations for adding multithreading to a rendering system, we are ready to discuss the threading model provided in Direct3D 11. As described earlier, the D3D9 and D3D10 device interfaces could be created with a flag to specify that the device would be used in a multithreaded environment. If this flag was not used, the devices were considered to be *thread-unsafe* and therefore should only be accessed from a single thread. When devices are created with the multithreading flag, they are created with coarse synchronization primitives to ensure that the device methods are only used by one thread at a time. This is referred to as *thread-safe,* which means that multiple threads can call the same method of a device, while synchronization ensures that there is no interference between the calls.

Direct3D 11 was designed with multithreading as one of the primary requirements. Devices, in comparison to how they were handled in D3D9/10, have been split into two different interfaces: the *device* and the *device context.* We will discuss each of these interfaces, and how they have been implemented for multithreading, in the following sections.

7.3.1 The Device

Before Direct3D 11, the responsibilities of the device interface had remained more or less unchanged for the most recent generations of Direct3D. It represented the entire GPU, including all draw submission sequences, as well as all resource creation activities. This made the device the primary interface that was used in Direct3D. Now, Direct3D 11 has taken the device and split up its responsibilities, leaving only a subset of its original functionality

in the new version of the device. This new, leaner device retains resource creation responsibilities, but it has given up the direct management of the pipeline to the device context. This change in responsibilities is described in great detail in Chapters 1-3.

However, this was not just a splitting of responsibilities. The split was made along the lines where there are different multithreaded requirements for the functionality. The resource creation methods have been implemented as *free-threaded,* which means that similar to thread-safe, they can be called simultaneously from multiple threads, but an important difference is that that the methods now don't use the heavyweight synchronization primitives. Instead, they are designed to be reentrant, so that there are no dependencies between multiple invocations of a given method. This lets the device perform object creation on multiple threads without the artificial serialization that was used to provide thread safety in previous versions of Direct3D.

With resource creation now available in a true multithreaded environment, and with minimal overhead introduced by the implementation technique, Direct3D 11 has provided facilities to allow parallel loading of resources with very straightforward implementations. Now, the same thread that has loaded a resource from disk can directly create a texture or buffer resource with the loaded data. This minimizes the amount of communication needed between threads and reduces the complexity of these types of operations.

7.3.2 The Device Context

The other half of functionality that has been split off from the device has been put into its own interface, called the *device context.* The device context comes in two flavors: the *immediate context* and the *deferred context.* We will discuss both of these objects, which implement the same interface, in the following sections.

The Immediate Context

The immediate context represents more or less the pipeline draw submission abilities from previous iterations of Direct3D's device interface—it provides the gateway for an application to directly interface with the GPU. The pipeline state setting methods are sent to the driver as soon as they are called, and are executed more or less "immediately" (in reality, this may not be immediate, because of driver implementations that use a buffer to queue up operations, but it is as close to immediate as the application can get).

This new interface has been implemented to be *thread-unsafe,* meaning that it doesn't use synchronization primitives, and that it hasn't been designed to intrinsically allow multiple threads to simultaneously use it. This is essentially the opposite of the multithreading behavior exhibited by the device, which is frep-threaded. Why would this be implemented in such a thread-unfriendly way? The answer is that this is actually a very thread-friendly

design—the immediate context should be used only from within one thread, but the other threads in a multithreaded program can use a different type of context—the *deferred context.*

The Deferred Context

The deferred context provides much of the same functionality as the immediate context, and it also serves as an interface to the pipeline. This includes the same multithreading behavior, meaning that the deferred context should be used from a single thread. However, as its name implies, all state changes and pipeline execution requests are deferred until a later point in time. Instead of immediately executing each state change and each draw and `dispatch` call, these are instead queued into what is called a *command list.* The command list can then be executed either by the immediate context, which would apply the list to the GPU immediately, or by another deferred context, which would insert the command list into that context's current command list. This functionality provides a fairly simple and intuitive introduction to generating a command list, since the deferred context is used in the same way that the immediate context is used. The only difference is that the immediate context applies the calls immediately, and the deferred context applies the calls to a command list. You can consider the deferred context to be a per-thread, per-core command list generator.

There are a few other functionalities that don't apply to the deferred context that can be used on the immediate context. The deferred context does not support directly performing queries—they must be performed on the immediate context instead. In addition, the deferred context cannot read data back from resources. This makes sense when you consider that the deferred context is creating a list of its instructions, instead of immediately doing the work. However, the deferred context is allowed to write to the contents of resources, as long as the resources are mapped with the D3D11_MAP_WRITE_DISCARD flag. This also makes sense, since this allows the command list to contain requests to write the entire contents of the resource, further enforcing the concept that the deferred context may not access the contents of a resource.

7.3.3 Command Lists

Since command lists play such an important role in the overall multithreading implementation of Direct3D 11, it is worth understanding precisely how they work. Command lists are generated by calling the deferred context's `FinishCommandList()` method. This creates a command list that includes all of the state change and `draw`/`dispatch` calls since the most recent previous call to `FinishCommandList()`. Once a command list has been created, the sequence of operations it contains cannot be changed—the object itself is immutable. The

command list is submitted for replaying by calling ExecuteCommandList() on the immediate context or another deferred context. So the general paradigm for using command lists is that they are generated by a deferred context, then are either consumed by the immediate context or applied to another deferred context, which effectively inserts that list into another list's stream of operations.

After a command list has been used, it is disposed of by calling its Release () method. However, determining when a command list should be released is up to the developer. Even though the command list itself is immutable, there is no limit to the number of times it can be executed. This introduces some interesting possibilities with respect to caching particular subsets of calls into command lists and permanently reusing them throughout the duration of the application. Later in this chapter, we will see several different levels of granularity for command lists, which can take advantage of this property.

This discussion also requires us to provide a distinction between *static* and *dynamic states* for a given object rendering sequence. A *static state* is any pipeline configuration that doesn't change from frame to frame, while *dynamic states* do change in some way over time. Examples of static states could be something like the blend state, a vertex shader, or a texture. Examples of dynamic states would be any transformation matrices that change from frame to frame, and any other state that doesn't remain the same over time. Typically, every object rendering requires some combination of both static and dynamic states. At first, this concept of reusing command lists may seem somewhat unusable, since you would normally want to render an object in a different location in each frame, because the camera, or the object itself, is moving. This would require new transformation matrices to be generated and uploaded for use with the vertex shader, which would seem to be incompatible with reusing command lists, which are immutable.

However, a command list records the states that are set on the deferred context. To use the example of transformation matrices from above, the application would bind a constant buffer (constant buffer *A)* that contains the transformation matrices to the vertex shader. Once the command list has been generated with the FinishCommandList() method, the specific constant buffer bound to the vertex shader cannot be changed in the command list. This means that there is no way to bind a different constant buffer (constant buffer *B)* in its place. However, you can modify the *contents* of constant buffer *A* to contain the desired transformation matrices, without requiring a new command list. This is the same concept as passing function arguments by value or by reference in C/C++. The command list essentially records a pointer to a particular resource, while the value that it points to can be changed independently of the command list. This lets you inject dynamic state into command lists, even though the command list itself cannot change. To summarize this idea, the command list cements the resources that are bound in its API call sequence—but it does not cement the contents of those resources. This concept is depicted graphically in Figure 7.3.

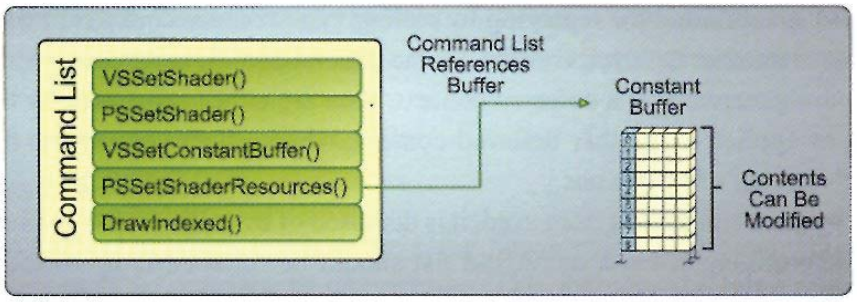

Figure 7.3. A depiction of binding resources in a deferred context, while the contents of the resource can be changed at a later time.

This gives us two general ways to use command list objects. The first is to generate and use a command list within every frame. This allows for simple dynamic updates to any state that is used in the rendering of an object, and the command list is released after the rendering is complete. This usage is intended to allow spreading the costs of API calls across multiple threads. The second way to use a command list is to generate it once, and then update its resources from frame to frame, and reuse it many times. This reduces the cost of generating the command list in every frame, at the expense of some additional complexity to ensure that only resources contain the dynamic state for the object rendering.

7.3.4 Using the Device and Context Interfaces

With this understanding of how the deferred context works, now would be a good opportunity to consider how a rendering system could be implemented to use deferred contexts. The general advice regarding deferred contexts is to use at most one context/thread per core in the user system's CPU.2 Then the application would have each thread perform some rendering work that can be queued into a command list. Once the command lists have all been generated, the immediate context would iterate through them, executing them one by one in the proper order. This basic setup is depicted below in Figure 7.4 for a quad core CPU.

As you can see from the figure, this new setup significantly reduces the amount of time needed to submit the draw state change calls to the API. In addition to reducing the overall time to submit the rendering requests, command lists are created in a format that lets the driver quickly and efficiently play back the sequence, which will further reduce the

[2] There are several ways to determine the number of CPU cores on a user's computer. The *GetLogicalProcessorInformation()* function has been in existence since Windows XP, but it includes hyper-threaded CPUs as a separate core. While this is partially correct, we are more interested in the actual number of CPU cores. Windows 7 has introduced the *GetLogicalProcessorInformationEx()* function, with extra flags to better specify precisely which count to return. Since Direct3D 11 runs on both Windows Vista and Windows 7, it may be necessary to use both functions, depending on which operating system is in use.

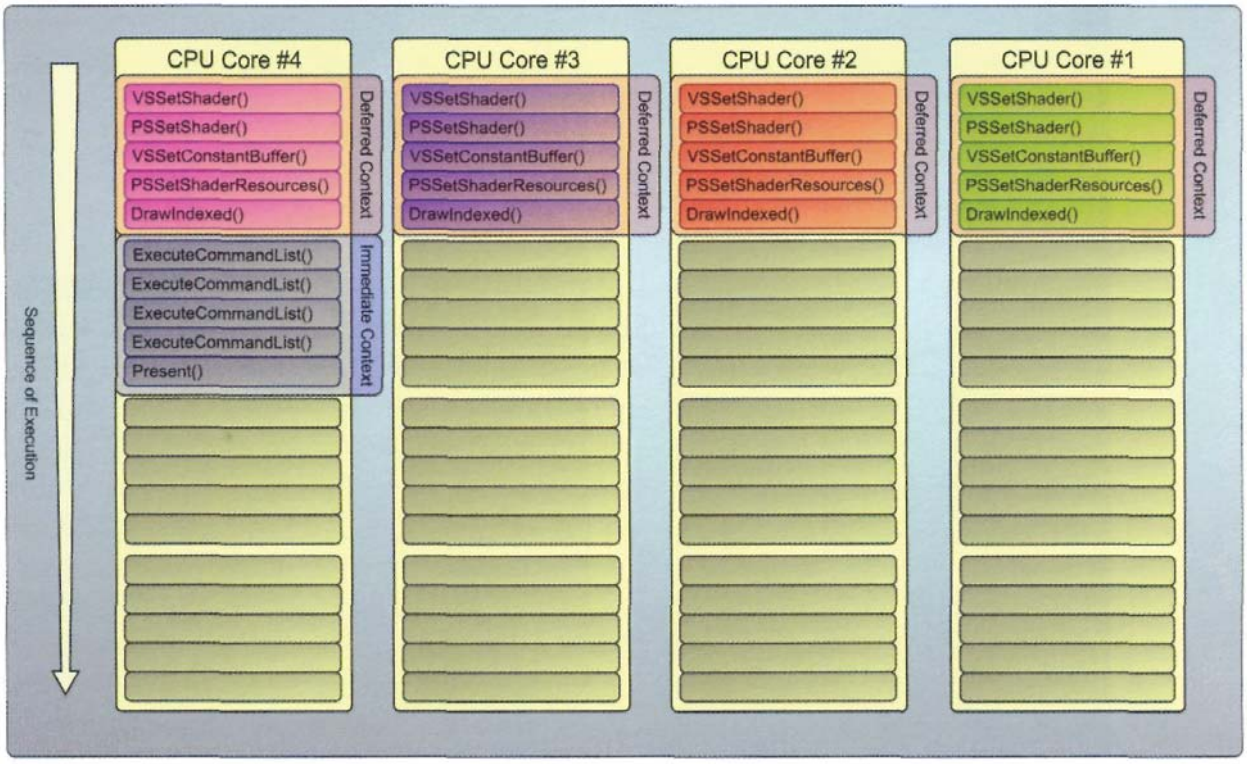

Figure 7.4. Using four deferred contexts to generate command lists to be executed on the immediate context.

total time required to finish rendering the final output on the GPU side of the application. The modified design of the Direct3D 11 interfaces provides the potential for a significant speed increase, without requiring too many changes in the application itself.

7.4 Context Pipeline State Propagation

With the structural details explained, we need to consider another topic concerning how command lists are generated and executed. Specifically, we need to understand how pipeline state is preserved, restored, and eliminated during the various operations that are performed when using command lists.

7.4.1 The Immediate Context State

We will start by investigating the behavior of the immediate context's pipeline state throughout a command list execution phase. Figure 7.5 shows the immediate context and

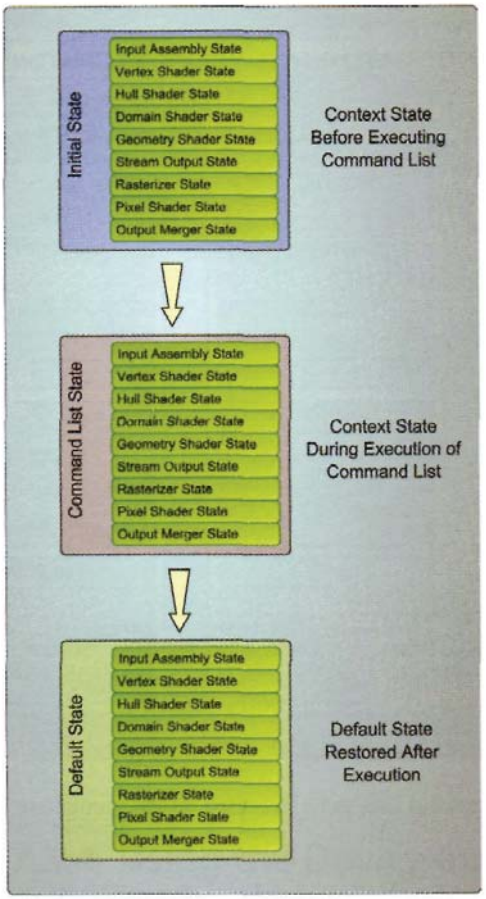

Figure 7.5. Immediate context state before, during, and after the execution of a command list.

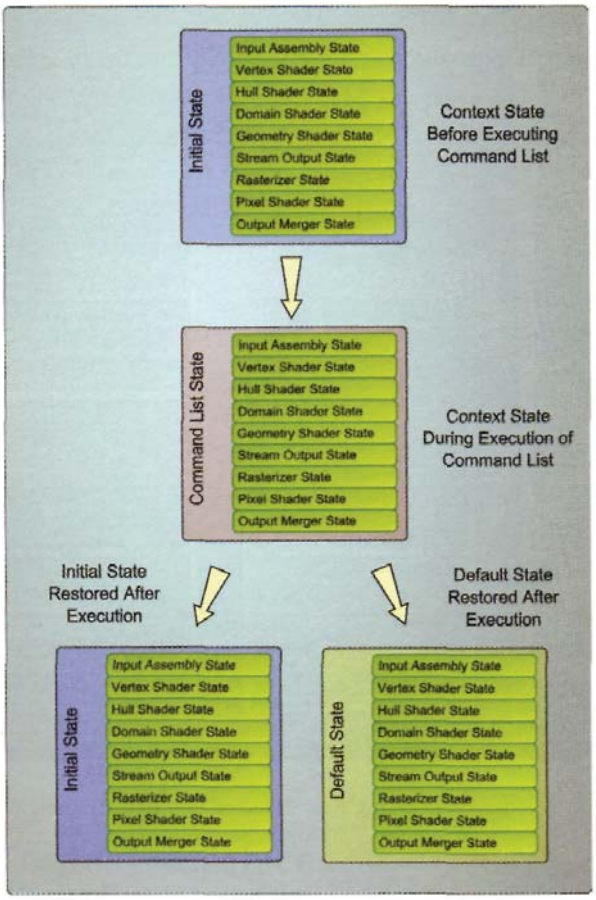

Figure 7.6. The two possible ways to handle the immediate context pipeline state after executing a command list.

its pipeline state prior to executing a command list. It's current configuration reflects any state change requests that were performed directly on the immediate context with its SetXXX() methods. When a command list is executed on the immediate context, its pipeline state is replaced with the default pipeline state. This default state is the same state in which the device was initially created.[3]

There are two possible outcomes for the existing pipeline state that was originally in the immediate context. Which one is chosen depends on a Boolean parameter passed to the ExecuteCommandList() method. If the passed parameter is `true`, the original pipeline state is stored until the command list has been executed, and is then restored to the immediate context. If the passed parameter is set to `false`, the original state is simply deleted, and the immediate context is reset to the default state after the command list has been executed.

[3] The individual values that comprise the default pipeline state are further described in the DXSDK documentation.

In both cases, the existing state of the pipeline is never shared with the command list, and the state introduced by the command list execution is never left active on the immediate context. These two possible execution paths are depicted below in Figure 7.6.

7.4.2 The Deferred Context State

The deferred context also contains the same pipeline state structure that the immediate context possesses. When the deferred context is created, it starts out with a default pipeline state. As the pipeline state is modified through use of the deferred context, the state-changing calls are accumulated until the application calls FinishCommandList () to generate a command list. Similar to the immediate context state when calling ExecuteCommandList(), the deferred context pipeline state will be handled by a Boolean parameter passed to the FinishCommandList() method. If the passed parameter is set to `false`, the deferred

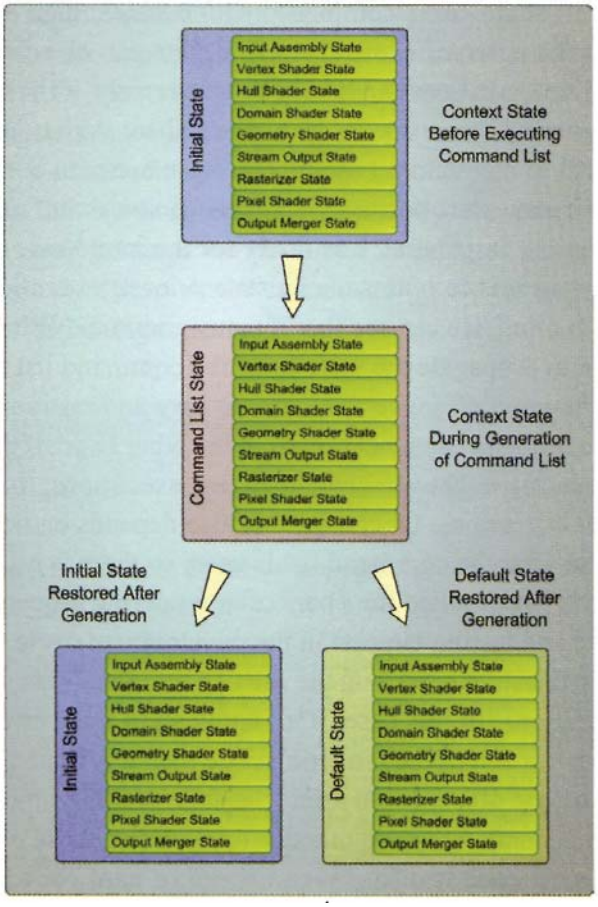

Figure 7.7. The two possible ways to handle the deferred context pipeline state after generating a command list.

context pipeline state is reset to the default pipeline state after the command list is created. If the passed parameter is set to `true`, the deferred context pipeline state is preserved beyond the `FinishCommandList()` invocation. This is depicted in Figure 7.7.

The differences between these two options determine the pipeline state with which the deferred context's *next* command list starts out with. This allows the application to control the starting state of the command list, and potentially shares the cost of setting that state initially across several command lists if the same settings are always used.

7.4.3 Performance Considerations

With some level of control over whether a context either retains or resets its pipeline state, it is helpful to consider a few situations in which it would be potentially advantageous to use these different behaviors. The primary considerations depend on the size of the command lists being used, as well as on the command-list execution strategy that the application will use. If the application is generating many small command lists within a single frame, and those command lists all share some common pipeline states, then a net reduction in the number of calls to set the deferred context pipeline state can be achieved by maintaining the state between calls to `FinishCommandList()`. Conversely, if the subsequent command lists that are being generated are very long or don't share common states, then there is likely little or no benefit to maintaining the context state between command lists.

The immediate context state propagation depends somewhat more on the frequency of command list execution. In general, it is faster for the immediate context to return to a default pipeline state than it is to both save its state prior to execution and then restore it afterwards. With this in mind, we can see that if many command lists will be executed in a row, there is no reason to propagate the state since the command list is executed in a fresh default pipeline state. It wouldn't make any sense to save and restore the original state if it isn't used in between command list executions. On the other hand, if the immediate context is used for normal rendering in between command list executions, then it may make sense to restore its state after executions. Of course, this also depends on how many of the states are common before and after the command list is used, so there is no general rule to determine which technique is better suited for a particular rendering sequence. This is a decision best left up to profiling and testing later on in the development cycle.

One final point to consider regarding the use of a previously set state relates to debugging of rendering operations. In this case, if an application inadvertently relies on a state that is set in some other portion of the rendering code then it is quite possible that a change to the rendering code of one object could affect the state of other objects that occur later in the rendered frame. In general, it is good practice to minimize these types of dependencies, especially if the rendering code is data-driven instead of hard coded. At the very least, it is a good idea to occasionally run the rendering code with no state propagation active to determine if these dependencies exist.

Determining Driver Support

The two main multithreading features provided by Direct3D 11, resource creation and draw submission, are guaranteed to be safe to use in a multithreaded development environment. However, they also rely on the GPU driver to operate optimally. If the driver cannot provide fully free-threaded resource creation, then the runtime will default to using its own lightweight synchronization. Similarly, if the driver does not provide a native implementation for multithreaded command lists, then the runtime will emulate the feature itself and provide a similar functionality. In both of these cases, the performance of the individual implementations may vary slightly, but having an emulation mode still allows the multithreading features to be used, even when the driver doesn't support them. The benefits of using multiple threads should far outweigh any discrepancy in performance between native implementations and emulated ones.

The ID3DllDevice interface provides a facility that indicates the level of support for these two multithreading features in the user's video card driver. The process for determining multithreading support requires an initialized device interface that can then be queried for its level of support for multithreading. (The process for creating a device is described in detail in Chapter 1.) When an application intends to use multithreading, it must initialize the device without the D3D11_CREATE_DEVICE_SINGLETHREADED device creation flag. Once the device has been successfully initialized, it can be queried for multithreading support using the ID3D11Device::CheckFeatureSupport() method. Listing 7.1 demonstrates how to do this. Once this method returns successfully, the passed-in D3D11_FEATURE_DATA_THREADING structure contains two Boolean variables: DriverConcurrentCreates and DriverCommandLists. These variables indicate the driver level support for resource creation and command lists, respectively.

```
D3D11_FEATURE_DATA_THREADING ThreadingOptions;
m_pDevice->CheckFeatureSupport( D3D11_FEATURE_THREADING,
                                &ThreadingOptions, sizeof( ThreadingOptions ) );
```

Listing 7.1. Checking for driver support of multithreading features.

While it is not mandatory to check these flags to determine the level of multithreading support (since emulation is available), doing so provides nice side benefit for previous-generation GPU hardware. Since the D3D11 runtime can be used with down-level hardware, it is possible for a GPU manufacturer to release updated driver implementations for D3D10 hardware. This provides a potentially significant performance improvement by allowing true multithreading on older hardware simply by having an updated driver and using D3D11. In this case, having the ability to poll the driver for support can be extremely beneficial for the end user.

7.5 Potential Usage Scenarios

The discussion of the multithreading behavior of the Direct3D 11 interfaces up to this point has focused on what they are capable of from a technical standpoint, and on how they function. Now we can take a look at a few situations where these tools could be used in a rendering application. We will look at potential cases where the use of multithreading can improve the performance of a given task, and will touch on several design considerations for designing a new rendering system.

7.5.1 Multithreaded Terrain Paging

We will start with how to apply Direct3D 11's multithreading capabilities to a *terrain paging system.* A *page* is a term for a memory management technique that uses more memory than is physically available on a computer. As data is needed, it can be *paged in* to a system memory block, replacing whatever page is already there. When paging is used to manage the data is needed at any given time, more data can be used than fits into the physical system memory. When designing a terrain paging system, the focus is typically on providing your scene with a very large space that can be explored. Because the scene data is often quite large, it is normally not possible to directly create all of the resources needed to hold all of the terrain data. So, that data must be paged into and out of a resource at runtime, piece by piece, based on the current location of the viewer. This provides a good use case for the multithreaded systems that have been discussed, since it can use both parallel resource creation and as parallel draw submission.

Such a system could use *worker threads* during startup and runtime. Each of these threads would be assigned a deferred context to use for the duration of the application, and would also have a reference to the device. Since we would likely have more terrain pages than CPU cores, the threads would have to be assigned in some repeating pattern to manage a given terrain page. At startup, each worker thread could start loading one terrain page from disk. It could then use the device's free threaded buffer resource creation method to allocate a vertex buffer and initialize it with the loaded terrain data. Since terrain loading is happening in parallel, the startup speed would likely be limited only by the available I/O bandwidth, which would vary, depending on what media is being used to store the data. For example, the hard disk, DVD drive, or network storage might all be used, each with different access characteristics. In addition, since resources can be created in parallel to rendering operations, the application can simultaneously start rendering a startup screen or debriefing message while the terrain pages are being loaded and created.

During runtime, each worker thread could generate a command list that configures the rendering pipeline with the appropriate shaders, resources (such as textures or constant

buffers), and states needed to render that terrain page. For each frame to be rendered, the command list of every visible terrain pages would be executed on the immediate context. These command lists could potentially be reused, since the view matrix required for rendering (or some combination of matrices including the view matrix) would be updated in a constant buffer, whose contents are not included in the command list. Only the reference to the constant buffer would be included in the list, not its contents.

As the viewer moves around in the scene, the worker threads could dynamically load new terrain pages from disk, as needed. Using their deferred contexts, they could add the Map/UnMap sequences for updating the terrain page's vertex buffers with the new data into their command list sequences. Then, the next time that particular terrain page is rendered, the vertex buffer resource could be updated and made available for rendering. This provides a simple approach to updating the terrain pages, with minimal synchronization required between the main rendering thread and the worker threads.

7.5.2 Multithreaded Shader Creation

The terrain paging system discussed in the previous section would certainly benefit from being run on a multithreaded system. However, not all applications need to use a data set that is so large that it must be paged into memory dynamically. However, there are some tasks that most, if not all, applications need to implement, and that could also benefit from multithreaded resource creation. Our next example fits into this broader context—creating shader objects at startup to be used during rendering. This seems like a somewhat trivial task, but depending on the number of shader programs your application will use, as well as their complexity, and the situations that they need to be used in, a "combinatorial explosion" can produce a very large number of required shader objects. The time required to create all of these objects can easily become unmanageable.

Creating a shader program requires two steps. First, the shader source code must be compiled to a byte-code format. Then, that byte code is submitted to the free-threaded device methods to create a shader program for a given programmable pipeline stage (additional details about shader creation are available in Chapter 6). Since shader compilation is relatively CPU intensive, it presents a good opportunity for parallelization when more than one shader must be compiled on a system with more than one CPU core.

In the second step, the free-threaded methods of the device allow multiple threads to simultaneously create shader objects after the compilation step. This eliminates the need to synchronize multiple threads to create the shader objects, which would have been required in older versions of Direct3D. A simple implementation to allow parallel loading of shader programs would create one worker thread for each available processing core. This arrangement is essentially a *thread pool*, a concept that may be familiar to the reader from standard multithreading designs. Then the list of shader programs to be loaded, and

their respective types and required shader models, would be distributed among the worker threads. Each thread would compile and create its list of shader objects in complete isolation from the others, eliminating possible synchronization issues. Then the main application thread would simply wait for all of the worker threads to complete, and would then read the resulting objects and store them centrally for use later on.

7.5.3 Multithreaded Submissions

The previous two examples are interesting, and they provide some insight into how resource-intensive tasks can use some of the Direct3D 11 multithreading tools to gain some performance advantage. However, the largest performance potential lies with the ability to split the rendering work for a frame among several CPU cores. As described above, this allows the overall CPU/driver costs for rendering a scene to be amortized over several threads simultaneously, reducing the time spent to send work to the GPU. Of course, there are many variations of how to implement this concept, and some may fit a particular application better than others. We will discuss a few possibilities here, and try to provide some context about why each of them would be a good choice in a particular situation.

The general scenario for the following discussion is the following. You have one main thread that can house the immediate context, and several worker threads (one for each CPU core in the system), each with a deferred context. When a frame needs to be rendered, its total workload is split up in some manner among the worker threads to generate a command list. When all of the command lists are executed in the proper order on the immediate context the final rendered image is produced. The order of execution is only important when the contents of one resource must be modified before the resource is used. For example, if one command list generates a shadow map, and then a second command list uses the shadow map to render a scene, the shadow map must be generated before being used. There are other cases where the order will not matter, such as per-object command lists (discussed below).

Deciding how to split up the work among the threads will likely depend on the types of scenes being rendered, as well as their contents. Let's look a little closer at three potential techniques for splitting up the scene's rendering workload. Please note that these are only three possible techniques that could be used, and that many other variations may or may not perform better in a given situation.

Per-View Command Lists

The first method of dividing a scene for rendering is perhaps the least intrusive of the three. The general idea is to generate a command list for each *view* of a scene. In this sense, a view can be considered one complete rendering pass in which the complete scene is rendered.

For example, the generation of a shadow map would be one view of a scene. The generation of an environment map would be another, and the regular perspective rendering of a scene would also be considered a view. If you consider the normal single-threaded series of rendering commands to be one continuous list, then a general rule of thumb for splitting the list into views would be to break them whenever the render target is changed or cleared (or in the case of computation pipeline execution, when the UAVs are changed). This splitting is shown in Figure 7.8.

With this partitioning, we would break up the rendering workload into these discrete chunks and hand them off to the worker threads to be processed into command lists. Each scene view would proceed with the rendering code it normally uses, except that it would be working with a deferred context, instead of an immediate context. Implementing this segmentation of the work may be somewhat difficult, depending on how a rendering pass is currently implemented in a rendering framework. One topic that must be considered is that of data synchronization. As an example, let's assume that an application uses a scene graph to contain its scene elements. If the graph is traversed once for each of our rendering views, and these transversals are carried out on different threads, we must ensure that the state of the scene graph itself is not modified in any way by any of the threads—otherwise, we risk (and almost certainly induce) data corruption between rendering threads. We also would like to avoid using manual synchronization code, since it will detract from any performance benefit we get from multithreading. In general, all updating for a given frame must be done prior to the rendering pass, or else any data modification must be very carefully planned out!

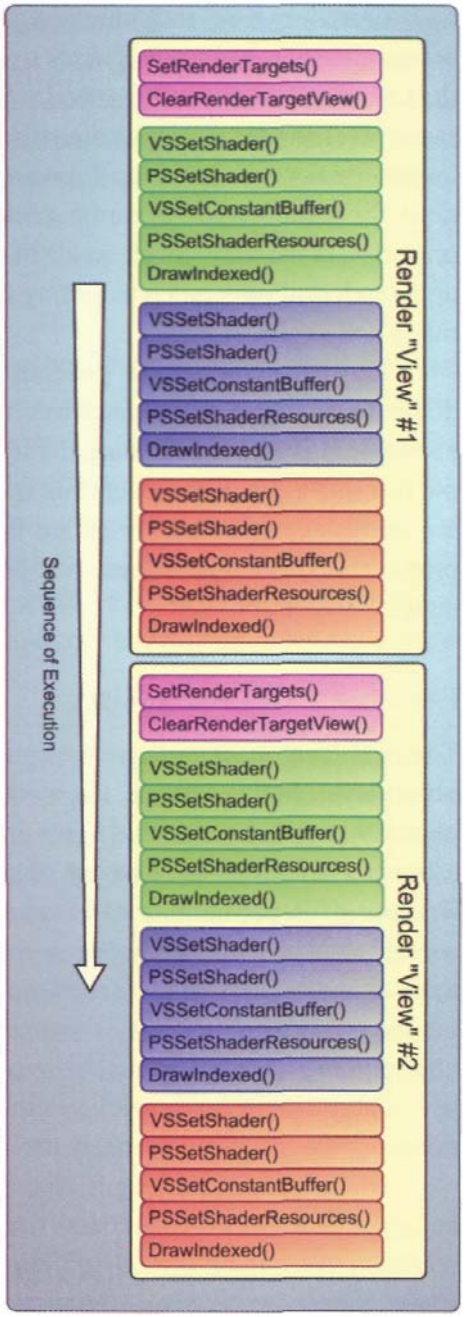

Figure 7.8. High-level rendering operations can be used to split a rendering sequence into "views."

Our view-level multithreading granularity provides good potential for reduced CPU overhead. Some rendering passes use very similar rendering effects for most, if not all of the scene objects. Consider a shadow map generation pass—most objects will use the exact same pixel shader to output the appropriate depth value, and most will also use a similar transformation shader setup (vertex and/or tessellation based shaders) with some variations for static versus dynamic geometry. These types of rendering passes are typically presorted at the view level, so all of the operations that a deferred context executes to set up a rendering pass (such as setting render targets or stencil setup) will be amortized over many draw calls.

However, due to the generally larger command lists, this view-based processing doesn't have much of a chance to reuse command lists from frame to frame. As discussed above, it is possible to update the dynamic state of objects by modifying the contents of the resources used. However, this does not allow the application to change which objects are rendered and which are culled in a given frame. This should not be seen as a critical problem, however, since there will not be very many view-sized command lists being generated for each frame.

Per-Object Command Lists

The next level of granularity we could use is to render the scene objects at the individual object level. In this scheme, the worker threads would generate one command list for each object that will be rendered. This introduces a much finer level of processing and consequently increases the number of command lists that must be generated and executed. Because of larger number of command lists, it is probably advantageous to use deferred context state propagation between command list generations. This would alleviate the need to make more frequent calls to higher-level rendering setup functions, such as setting render targets, because so many command lists are used. The additional command lists would also require a higher number of FinishCommandList() and ExecuteCommandList() calls, which may or may not impact performance. Since the command lists are generated in isolation from the rest of the scene, it also reduces the amount of batching that can be performed.

However, this paradigm also has some interesting side effects that could prove to be beneficial. Once a command list is generated for a particular object for a particular rendering pass, there is likely no reason to release and recreate the command list for every frame. Since any per-frame dynamic rendering data, such as view or skinning matrices, is provided to the shader programs in constant buffers, the command list will not change from frame to frame as long as the same constant buffers are updated and used for every frame. With no overhead for generating the command list, any additional costs discussed above could largely be overcome. In addition, the simplicity of such a scheme would be attractive as well—each object would simply receive its own command list and use it as necessary.

Another potential optimization for this technique is provided by the fact that deferred contexts can also consume command lists. This means that it is possible to build larger command lists from a group of smaller ones to minimize the number of command lists needed to be executed on the immediate context. Depending on the implementation of this feature in drivers, this could also be used to reduce the overall cost associated with rendering a frame with low-level command lists.

Per-Material Command Lists

The final solution that we will consider is to process a scene in per-material chunks. When it is time to render a frame, some preprocessing of the scene is required to group objects that share common materials, so that they can be processed together as a group by a single worker thread. Then, each newly generated command list is passed on to the main rendering thread for processing by the immediate context. This type of object sorting was a fairly standard practice in D3D9 programs, due to the relatively high CPU cost for each API call. By sorting objects, the total number of state changes could be reduced by finding a more optimal sequence to render them in. Thus, this technique could draw upon some of these existing routines for grouping suitable objects together.

This model strikes somewhat of a balance between the two prior methods. The size of the groups of objects that are rendered depends solely on how common a given material is, so this method provides the potential to handle the largest number of objects within a single rendering batch. Since the per-material command lists encompass more objects with a relatively small number of state changes between them, they provide better performance than the per-object command lists since there would be fewer lists to submit to the immediate context. However, there is also the additional cost of having to sort the objects by material before creating the command list, so the benefits of this technique will vary with the scene complexity and variations in material types. For scenes that contain relatively homogeneous rendering materials, this technique can provide a good way to efficiently process and submit them to the GPU.

7.6 Practical Considerations and Tips

We will conclude this chapter with a section devoted to providing some general design advice that should be useful when you start to build an application or framework that will use the multithreading features of Direct3D 11. Due to the typical complexity of multithreaded programming, it is best to minimize potential issues from the beginning to avoid trying to find (and fix) them later on.

7.6.1 Things to Do

We begin this section with a discussion of some design points that should be incorporated if possible into your initial design. These are suggested practices that are not required by Direct3D, but that have been found to be helpful in the authors' experiences with the API.

Variable Levels of Threading

The first topic is to ensure that your application/rendering framework is not locked into a particular number of threads. This is important for several reasons. First, there is no way to know how many CPU cores an end user's system will have, so it makes sense to dynamically decide the number of threads to create and use. If possible, this should be extended to allow the number of threads to be selectable at runtime, to allow for future possible optimizations for switching additional threads on or off as the rendering workload needs them.

The second reason for using a variable number of threads is related to debugging your programs. If you are not receiving the final rendered frame that you want, this could be due to the multithreaded rendering itself (or more precisely, to how it is being used). To help minimize debugging time, it is extremely advantageous to be able to switch among the following setups:

Multi-threaded, deferred context rendering: Multiple threads are used with deferred contexts to generate command lists, which are then rendered on the immediate context.

- Single-threaded, deferred context rendering: A single thread is used with a deferred context to generate command lists, which are then rendered on the immediate context.
- Full single-threaded rendering: A single thread is used with only the immediate context to directly render without generating command lists.

These various mixtures allow for isolating rendering issues between a regular rendering error, an incorrect state setting due to the multiple contexts being used, and data contention between multiple threads of execution. This type of flexibility is assisted by the fact that the immediate and deferred contexts outwardly appear identical. Therefore, if the rendering system batches are incorporated into objects that can be passed to a thread for processing, then each thread would require one context to do the processing. The rendering framework could then decide dynamically at runtime how many threads to use, and which types, and then pass the rendering work to the corresponding objects.

Using PIX

Some additional considerations are needed when using PIX for debugging and analyzing your application. PIX is a very useful tool that can let you inspect what your program has

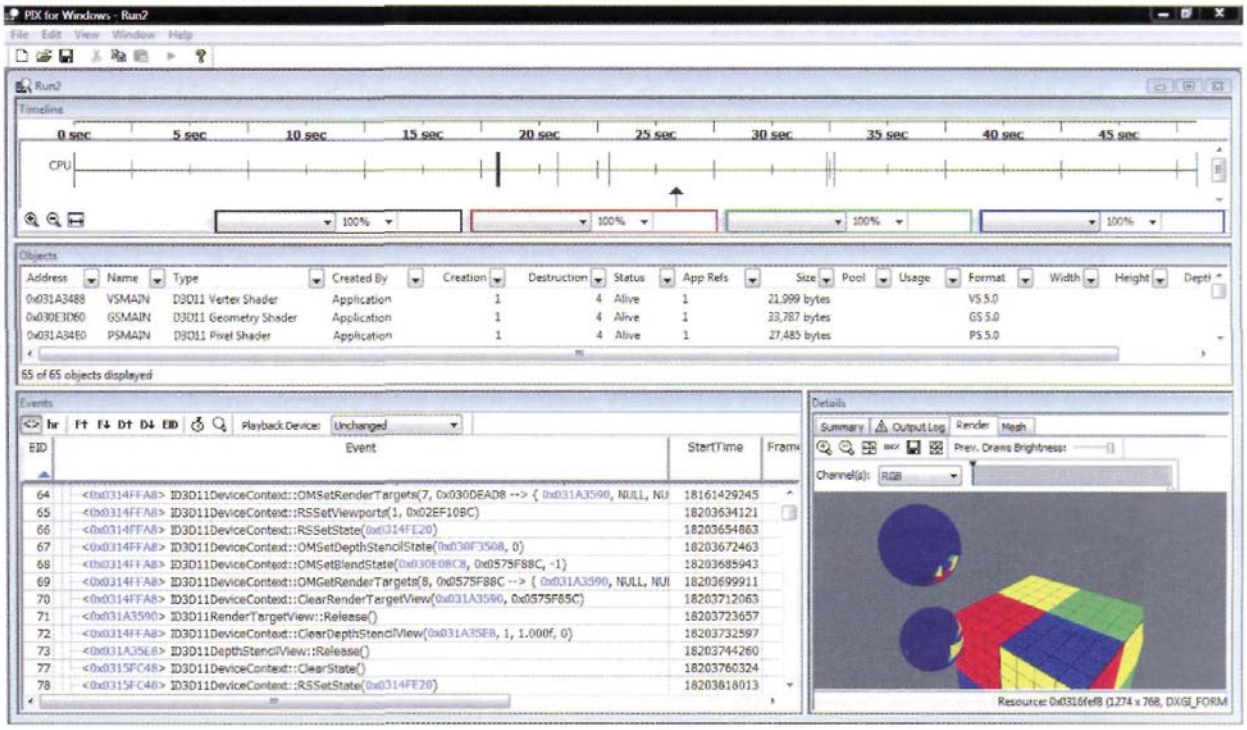

Figure 7.9. A sample screen shot taken from a multithreaded application PIX frame capture.

done with the D3D11 API for a given frame. When using multiple contexts and multiple threads to generate and consume command lists during a frame, you can still perform a frame capture and obtain significant information about how the frame was constructed. Specifically, you can see each of the API calls that are made from each context, with the calling context identified by its address. Figure 7.9 shows a snapshot of the information available from a frame capture.

Unfortunately, at the time of this writing some of the visualization tools in PIX are not operational when used in conjunction with deferred contexts/command lists. In the single context frame capture, you can step through each API call and visualize surface contents and the geometry being rendered, and you can inspect the contents of memory resources. But when deferred contexts are used, the API calls are wrapped into a command list, and the visualizations are not possible since the entire command list is executed at the same time, which is much later than when the command is actually sent to the deferred context. Even so, the time and order of the calls are still available and can help you understand if there is a mistake somewhere in the sequencing of the API calls.

This also provides another reason to be able to switch from using deferred contexts to only using the immediate context—you can use the immediate context mode to provide a complete PIX frame capture with visualizations to check for errors, and then switch back to deferred rendering once you are sure there aren't any errors in the rendering code.

7.6.2 Things to Avoid

In addition to advice about what to do, there are also several things you should try not to do. This section briefly describes a few of these. Once again, these are simply suggestions based on past experiences.

Pipeline Stalls

Although using deferred contexts and command lists lets you use multiple CPU cores to perform your rendering work, the same hazards and advice about rendering in general still apply. The command lists efficiently submit many state changes and draw/dispatch calls to the Direct3D 11 runtime and driver. However, if there is an option to submit the command lists in a variable order, then be sure to put them into the order that will allow

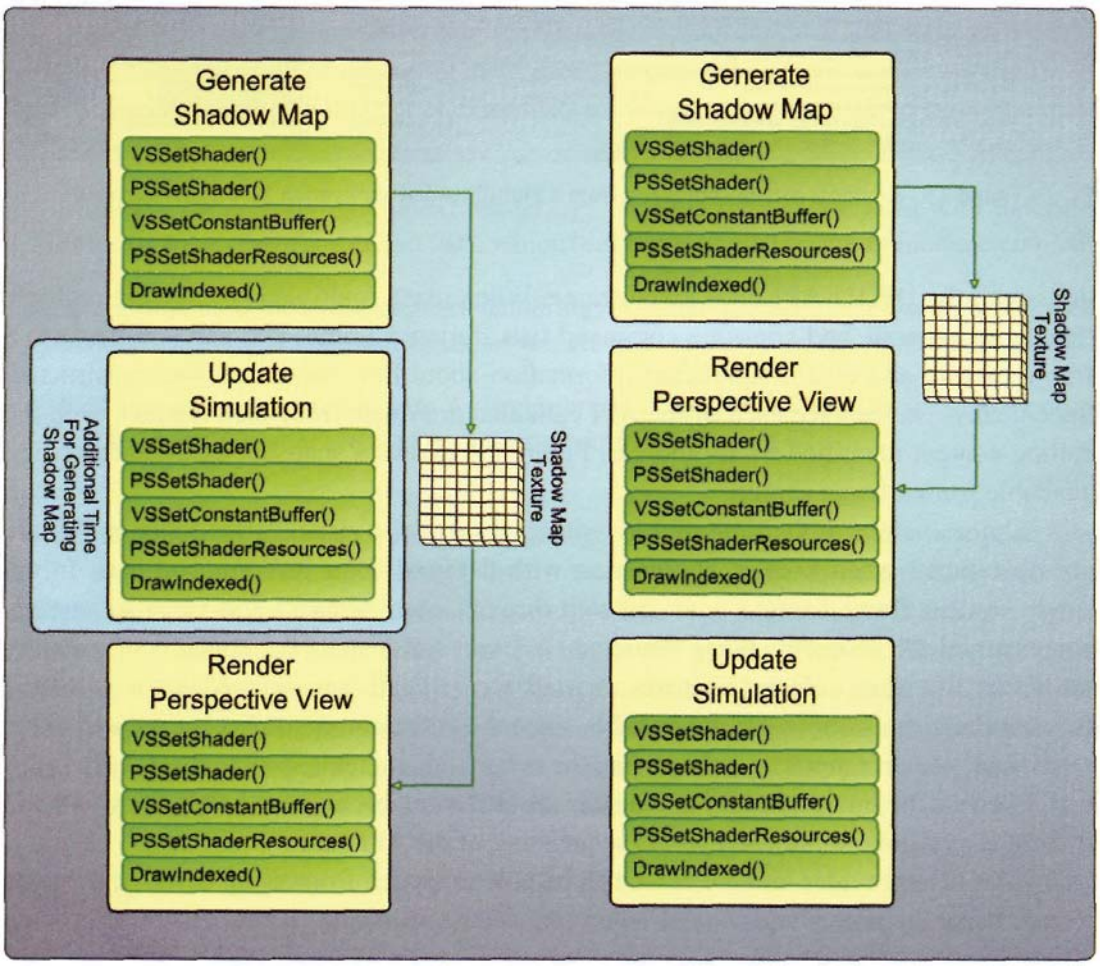

Figure 7.10. Determining the proper order to execute command lists, based on pipeline stalls.

overall execution to continue without pipeline stalls. For example, consider if we have one command list that generates a shadow map, another one that updates a simulation with the compute shader, and a third one that performs the final scene rendering. If the simulation contains two buffer resources for containing its state, then we can render the current frame with the current buffer and use the GPU resources to update the buffer for the next frame. With this set of command lists, we can execute them in the following order: the shadow map first, then the simulation, and finally, the perspective rendering. The reason behind this is depicted in Figure 7.10.

When the command lists are submitted in this order, execution of the final rendering pass doesn't start until the shadow map has already been completed. This means that the GPU will remain busy the entire time, without waiting for one pass to complete before continuing with the final pass.

Switching Modes

Another general piece of advice regarding the ordering of command lists is to minimize the number of times an application switches between using the GPU for *rendering* and using it for *computation.* If a full-frame rendering consists of several computation workloads (i.e. using Dispatch calls) as well as several rendering workloads (such as using Draw calls) then they should be ordered such that the computation command lists are executed together (as much as possible) to minimize how many times the GPU switches modes. This is general advice based on current GPU implementations, which may or may not continue to be the case in future generations of hardware.

Context State Assumptions

If you are porting an existing rendering system to Direct3D 11, it is quite possible that your previous rendering system makes some optimizations about setting pipeline states to eliminate redundant state changes and minimize the number of API calls in every frame. A typical example is to keep a reference to the current render target, and ensuring that when your application tries to set a render target, it checks to see if the current target already matches the desired one.

In previous versions of Direct3D, this would reduce the number of API calls and relieve the application code from having to manage the state itself. However, due to the way that context state is implemented in Direct3D 11, there are significant differences to how any such state-caching system will need to operate. These systems can still be used, but they must be aware that the complete pipeline state may or may not be reset based on the calls to FinishCommandList and ExecuteCommandList, as detailed above. State caching can still reduce unnecessary API calls within a command list generation pass, but the caching system needs to be reset at the appropriate points, depending on how the command lists are generated and executed.

7.7 Conclusion

With an API architecture specifically designed to allow multithreaded programming, there is a very rich set of possible performance-improving techniques available to the developer. In addition to simplifying some portions of an application, such as resource loading during startup, there is also great potential for parallelizing the rendering sequence API calls to minimize the cost of submitting work to the GPU. In addition, as the number of available CPU cores continues to increase, it is quite likely that the rest of an application framework will become multithreaded. This trend applies pressure on the rendering framework to fit into such a processing system, which provides even further incentive to build support for multithreading into a Direct3D 11-based renderer.

We can also consider these additional features in a higher-level view, as well. By directly supporting multithreading, Direct3D 11 essentially allows developers to break down the sequential rendering processes that have been used for many generations of D3D. From a design perspective, you can now logically think of a frame as being a collection of tasks and dependencies, rather than as a monolithic sequence of events.