# Tutorial:Migrating Your Apps to DirectX* 12 – Part 3



# Chapter 3 Migrating From DirectX 11 to DirectX 12

## 3.0 Links to the Previous Chapters

Chapter 1: Overview of DirectX* 12

Chapter 2: DirectX 12 Tools

## 3.1 Interface Mapping

If your upper-level rendering logic is written based on DirectX11, then the best way to migrate is building an interface layer that's fully compatible with DX11 because the upper-level logic is not required to do a lot of code refactoring in order to adapt to DX12. This kind of migrating is very fast. In our practices, we only spent a total of about six weeks or so to complete the migrating and testing of the vast majority of functions. Nevertheless, it also has some disadvantages. Because a lot of DX11's render objects have been integrated or removed in DX12, the wrapper classes of DX12 need to do a lot of runtime state transitions. These operations will consume some CPU time and you cannot completely remove them. So if you have plenty of time for development, it is recommended that you abstract DX12-like graphic interfaces instead to adapt backwards to DX11 features. DX12 will be the future trend, after all.

In order to adapt to DX11 APIs, we re-implemented almost all of the interfaces in D3D11.h document. The following is part of the code sample.
For example:

Table 3.1: Interface Mapping

```
class CDX12DeviceChild : public IUnknown
{
public:
        void GetDevice(ID3D11Device **ppDevice);
        HRESULT GetPrivateData(REFGUID guid, UINT *pDataSize, void *pData);
        HRESULT SetPrivateData(REFGUID guid, UINT DataSize, const void *pData);
        HRESULT SetPrivateDataInterface(REFGUID guid, const IUnknown *pData);
        HRESULT QueryInterface(REFIID riid, void **ppvObject);
};
class CDX12Resource : public CDX12LDeviceChild
{
public:
        void GetType(D3D11_RESOURCE_DIMENSION *pResourceDimension);
        void SetEvictionPriority(UINT EvictionPriority);
        UINT GetEvictionPriority(void);
};
typedef class CDX12Resource ID3D11Resource;
typedef class CDX12DeviceChild ID3D11DeviceChild;
```

It's important to note here that the project could not include D3D11 headers, otherwise definition conflicts might occur.

## 3.2 Pipeline State Object

Pipeline State Object is the core concept of D3D12. It consists of Shader, RasterizerState, BlendState, DepthStencilState, InputLayout and other data. Once the PSO object is delivered to the system, these states associated with PSO will be set at the same time. However, at the interface layer of D3D11, these rendering parameters are set using different APIs. In order to complete the adaptation, we must use a queriable runtime container to manage them. The most common object container is HashMap which can be used to avoid redundant PSO and the corresponding API calls.

Before using HashMap, we must first prepare the resource ID. The first thing you might think of is the memory address of resource. It is globally unique within the entire app life cycle, but it has a drawback which is taking up too much memory space: 8 bytes on 64-bit systems especially. Practical analysis shows that most apps do not use such a huge amount of objects, thus we can reduce the space that represents resource objects by means of sequential numbers, that is, using a monotonically increasing integer value to represent a resource object. The same integer can also represent different resources as long as those resources are of different types. For example, RasterizerState and BlendState can use different resource counters. An important benefit of this management approach is that it makes the coding space for resources more compact and easy to generate shorter Hash values. Otherwise, if you use the Hash value generated after stitching the memory addresses, the number of bytes of memory occupied by the Hash value will be big, which not only affects the storage of PSOs, but also affects the query speed. The upper limit defined for counters needs to be found out in practice. Different projects may have great differences, but we can first use a larger value in the test, and add assertion where the sequential number is assigned. Once it exceeds the upper limit, the system will trigger an alarm. Then you can determine whether to modify the underlying implementation or adjust the upper-level logic.

To further reduce the number of PSO instances, when generating RasterizerState, BlendState and DepthStencilState, we need to observe the state dependency between them. For example, when we disable the depth test in DepthStencilState, the settings for depth offset in RasterizerState can be ignored. To avoid producing redundant objects, we use default values in those cases.

RTV and DSV are also related to PSOs. Since DSV can control whether to read and write Depth or Stencil in the depth map, when the depth test is enable and depth write is disabled, you need to set a read-only DSV in the system. DSV has three read-only modes: 1) Depth Read Only 2) Stencil Read Only 3) Depth And Stencil Read Only. Besides, PSOs also need the Format information of RTV and DSV, thus you'd better defer the OMSetRenderTargets operation to the time when the PSO is set.

The ScissorEnable property has been removed from RasterizerState. The Scissor test will be in an always-on state on the hardware side. So if the app needs to disable Scissor test, you should set the width and height of ScissorRect to match the viewport or to the maximum resolution allowed by the hardware, such as 16k.

The primary topology type of Primitive needs to be set in PSO, which includes Point, Line, Triangle and Patch. We can use the pre-built conversion table when calling IASetPrimitiveTopology to directly convert it to the primary topology type mentioned above. PSO's HashMap can also be classified according to the primary topology type, with each topology type corresponding to a HashMap, then it will be directly positioned using the array subscript.

## 3.3 Resource Binding

Before getting to know the resource binding, we must first understand a core concept which is RootSignature. There are big differences between D3D12 and D3D11 in terms of resource binding model. Resource binding in D3D 11 is fixed. The runtime arranges a certain amount of resource Slots for each Shader and the app only needs to call the corresponding interface to be able to bind the resources to the Shader. In D3D12, the resource binding process is very flexible and does not limit the way in which you bind resources or the number of resources you bind. You can set the resource binding style on your own. The most commonly used approaches to bind resources are Descriptor Table and Root Descriptor. The Descriptor Table method is a little more complicated in that it places the Descriptors of a set of resources

in a Descriptor Heap in advance, so that when DrawCall needs to reference these resources, you only need to set up an initial handle on it. The Shader will find all subsequent Descriptors based on this handle. This is kind of like pointer array, which means that the Shader needs to do the 2nd addressing to locate the ultimate resources. While the advantage of Root Descriptor is that, instead of placing the Descriptors in a Descriptor Heap in advance, you can set the GPU address of resources into the Command List, which is equivalent to dynamically constructing a Descriptor in the Command List, so that the Shader can locate the resources by only one addressing operation. However, Root Descriptor consumes twice as much parameter space as Descriptor Table. Since the maximum size of RootSignature is limited, reasonable arrangements of the proportion between Root Descriptor and Descriptor Table is very important.

Under normal circumstances, we put SRV and UAV in the Descriptor Table (while Sampler can only exist in the Descriptor Table), but place CBV in the Root Descriptor. Since most resources consumed by CBV are dynamic, its address changes frequently, the use of Descriptor Table may cause combinatorial explosion. Not only the amount of memory occupied increases sharply, but it is troublesome to manage. By contrast, the combinations of Sampler, SRV and UAV will vary much less than CBV, especially for Sampler. As long as the upper rendering logic is properly designed, the number of Sampler combinations can be less than 128. Therefore, it is more appropriate to directly place them into a Descriptor Heap. Here, in order to reuse the Descriptor combinations in a Descriptor Heap, we have to use the PSO-like object management technology to first number each Sampler, SRV and UAV, and then in accordance with the needs of Shader, put them together and generate a unique Hash value which is used to create and index the Descriptor combinations in the Descriptor Heap. Since the maximum number of Samplers used in Shader is 16, each Sampler combination can be placed over a span of 16 per unit. SRV and UAV can also be managed using Sampler's approach, so the upper limit of Shader referring to them is better to be 16 too. Of course, variable combination span unit is also an option, but it is not very convenient to reuse them across frames, because when the texture pointed by SRV is released, its sequence number will be reclaimed by the app, and all Descriptor combinations referencing it will be marked as Deleted. At this point, if the combination blocks in the Descriptor Heap vary in size and are discontinuous, they would be very difficult to be reassigned like the memory pool fragments unless you make time consuming anti-fragmentation efforts. For this reason, a compromise is to use a fixed length of span for Descriptor combinations.

There can be only maximum two Descriptor Heaps set in the Command List, one for each type of Descriptor Heap. “Sampler” and “SRV / UAV / CBV” belong to two different types of Descriptor Heap, and cannot be mixed.

For the sake of efficiency, when we need to rewrite the Descriptor of _Descriptor Heap, we can first complete update in a CPU visible Descriptor Heap, and then copy the contents of the Heap to a GPU visible Descriptor Heap via CopyDescriptors* command. If each view is only in one location, then it should be created/updated directly in the shader-visible heap.

## 3.4 Resource Management

### 3.4.1 Static Resources

In D3D11, there are two approaches to initialize static resources. The first one applies to Immutable resources. It only allows the data within these resources to change once, passes the data that needs to be initialized into the system through Create* interface. The second one applies to Default resources. It can change data within these resources for several times, but with the help of Staging resources.

In D3D12, the initialization processes of these two resources are merged into one – the 2nd approach. Data was updated to the Default Heap through resources in an Upload Heap. As with D3D11, all resources

allocated from the Default Heap cannot be mapped, which means that apps don't have direct access to its CPU addresses, and therefore need an intermediate resource allocated from the Upload Heap as a bridge to push the data from the CPU side to the GPU side. One thing that needs attention here is when to delete this intermediate resource. In D3D11, the intermediate resource can be deleted directly after executing the Copy command, but this is not feasible in D3D12, because D3D12 does not provide resource life cycle management functions for runtime. All work must be done by the app, so the app needs to know whether those Copy tasks executed asynchronously completed, in other words, when the GPU no longer references these resources. We can easily access this information through the Fence feature of Command Queue. In addition, we can also complete the work of uploading resources through a shared memory pool of dynamic resources. After all, allocating an Upload Heap resource for each Default Heap resource is rather inefficient. Not only the reuse rate is very low, but it's easy to produce excessive fragments in the system. Therefore, using the dynamic resource memory pool technique described later, it is possible to avoid these problems.

You should first record the current frame number each time before applying resources to a Command List. This frame number can be used to determine whether a resource can be deleted directly when it is released on condition that the number of frames between the frame that has this frame number and the current frame exceeds the total number of Command Lists, or you can buffer it up into the delayed release list of Command List, and release it after the Command List finishes execution on GPU.

### 3.4.2 Dynamic Resources

In D3D11, the Dynamic Usage resources should be very familiar to us. It is widely used in the Vertex Buffer, Index Buffer, and Constant Buffer. The related application scenarios include particles and interfaces. The Map function normally provides a Write Discard feature which allows the app to use the same resource repeatedly, provided that the initial size of this resource meets the needs of rendering logic. As mentioned earlier, we know that the APIs of D3D are performed asynchronously, that is, the end of API calls does not mean that the execution of the task has been finished at the same time, instead, it is likely that there's still time before the final completion. At this point, if subsequent DrawCalls have modified this resource, there is a certain probability of causing competition for resources. But if you've used the Write Discard feature, it can prevent this from happening. Because the runtime or driver will automatically rename the resource so that it looks like a reference to the same object externally, but in fact it has been switched to another free resource internally. This new resource will take over the old resource for external update. To avoid prolonged occupation of a large amount of memory, the system puts these old resources in the memory pool for unified management. When they are no longer referenced by GPU, the system will reclaim and recycle them.

In D3D12, we must implement a similar function. First, we need to establish a resource pool which consists of a list of resources. As the size of resources we request each time may vary, it is best to allocate a larger resource in advance, and then classify child resources by different offsets for upper logic use. By doing so, we can reduce the number of system allocation, as well as the number of excessive fragments generated as a result of the discontinuity of memory. In general, we recommend using resource blocks in a unit of 4MB. When the resource pool is ready, we can start to allocate resources. But before that, we also need to know the memory address of resources. In D3D11, the memory address is derived from the Map function. D3D12 also returns the memory address through this function. The difference is that, unlike D3D11, D3D12 doesn't require to call the Unmap function each time you map and fill the data. As the mapping of D3D12 dynamic resources is continuous, which means that their memory addresses will always be valid, there's no need to tell the system to cancel the mapping of resources through the Unmap function. So under normal circumstances, in the lifecycle of a dynamic resource, it's only required to call

the Map function once. You can save the memory address it returns for repeated use. Before this resource is released, you need to call the Unmap function once to ensure that this memory address space can be reclaimed by the system.

Reclaiming occupied resources is also very simple. We just need to put those resource blocks allocated by the current frame into a pending queue numbered by the current Command List. Each frame will detect if the Command List corresponding to this queue is finished. If it is done, you can link all the resources in this queue to the Free List for subsequent distribution and reuse. The above method is suitable for the data that's updated and used per frame. The resources that are referenced across frames and may be updated once for a number of frames need to be maintained using another method. We should record the current Command List number before each resource is referenced. When it is renamed again, you should first check if the Command List corresponding to this number has already been executed. If not, you should put it into the To-Be-Reclaimed list of the current Command List and wait for the Command List to complete before reclaiming and using it. Since this method doesn't discard resources for every frame, it can ensure the use of resources across frames, but it needs to check the last time usage of resources every time the resources are renamed.

Due to the dynamic Buffer, GPU address changes with every request. For this reason, it is better for the external rendering logic to place the buffered requests before the resources are set to the Command List, otherwise you will need to defer the setting of resources to the time when DrawCall makes calls.

**Special reminder: CPU-side logic shall not read the memory space mapped out of resources allocated in the Upload Heap, otherwise it will cause a significant performance loss because this memory should be accessed in Write-Combine mode.**

### 3.4.3 Update of Dynamic Texture

In D3D11, the dynamic Texture is updated basically in the same way as dynamic Buffer. You directly pull out the memory address after mapping, and then fill the Texture with data. However, compared to Buffer, extra consideration needs to be given to the span value of Row Pitch and Depth Pitch. But in D3D12, you cannot fill the Texture like you did in D3D11, because the Texture is stored in the form of Swizzle in GPU, while the memory layout of buffer is linear. So the Buffer can be directly filled on the CPU side without conversion. However, for GPU read efficiency considerations, the Texture has to be uploaded in another way. As with Buffer, the first step is to allocate an Upload Heap resource of appropriate size. According to the GetCopyableFootprints API, we can learn of the allowed space layout of the Texture uploaded to the Default Heap in the Upload Heap, and then use the Copy * command to upload the data filled on the CPU side. As seen from the description, the Texture used by GPU is actually also a static resource. When we look at the implementation of D3D11, the Texture resource that can be mapped also went through a similar process as D3D12 internally, but D3D12 externalizes these tasks, and thus gives apps more possibilities for optimization.

### 3.4.4 Readback of GPU Data

In D3D11, there are two types of GPU data readback. One is the readback of the GPU data of Buffer and Texture, which is handled by the Staging resource. First you copy the static resource that needs to be read back to the Staging resource, and then use the Map function to return an address that can be read by CPU. But before you actually start reading the data, you also need to determine whether copy of the resources to the readback heap has been completed because the Copy operation is asynchronous. In D3D12, the process is similar. Unlike the Upload Heap mentioned earlier, D3D12 provides a Heap type dedicated to data readback. The ReadBack Heap can be used almost in the same way as the Upload Heap.

Likewise, you first allocate a resource from the Heap, then copy the Default Heap resource that needs to be read back to this resource through the Copy* function. What's different from D3D11 is that its Map function does not provide the capability to wait and check whether the readback has been completed. At this point, we need to apply the mechanism we mentioned in the static and dynamic resource management sections to the readback operation, determine whether the readback has been completed by way of Fence. We don't want to encourage persistent map of readback (write-back) memory. You can keep it mapped, but before reading data that was written by the GPU, you should do another Map() with a range to ensure the cache is coherent. This is free on systems which don't need it, but ensures correctness on systems which do.

Another type is the readback of hardware Query. The process for this type of readback is basically the same as the readback of Buffer and Texture, except that the Resolve function is used instead of the Copy function. Since the Resolve function can readback Query data in bulk, when assigning the Query Heap, it's not required for every single Query object to call the create function. Instead, you can allocate the Query object collection with contiguous memory space, and then conduct subsequent positioning by way of offsetting within resources.

Since we are using D3D11 interface encapsulation, externally, upload and readback operations may both use the Staging resource. So how do we distinguish between them internally? First, we need to determine whether the Map operation takes place before or after the Copy* command is executed when we first use this resource. Under normal circumstances, the Map operation takes place before the Copy* command is executed because we believe that users want to upload data to GPU. And when users want to read back data from GPU, the Map operation takes place after the Copy* command is executed. Of course there is a precondition here, that is, for the same resource, the external logic is not allowed to use it to both upload CPU data and read back GPU data. Otherwise it would not be able to identify this ambiguity internally. Fortunately, such cases rarely happen in practice.

## 3.5 Resource Barrier

This is a new concept. Prior to D3D12, resource state management was done by the driver. Now D3D12 strips it out from the driver layer, and lets the app control when and how to handle it.

There are three different types of resource barriers. The most common type is Transition which is mainly used to switch the state of a resource. When the application scenario of a resource changes, we will place a corresponding Resource Barrier before this resource is used.

A very common Transition Barrier in practice is that, a resource switches back and forth between ShaderResourceView and RenderTargetView. So we need to add a member variable in the wrapper class of a resource to record the current resource state. When the upper logic calls OMSetRenderTargets, we should first check whether the current state is RenderTargetView, if not, place a Barrier. Its StateBefore is filled with the state value stored in the member variable, while its StateAfter is filled with RenderTargetView state. If the rendering logic calls XXSetShaderResources, then we proceed with similar actions by following the above process, except that the StateAfter should be filled with ShaderResourceView state.

It is better to defer the setting of the Transition Barrier until we really start to use resources, so as to avoid unnecessary synchronization, because it will block the execution of subsequent commands.

Resources passed in through Copy*, Resolve* and Clear* commands all have a corresponding target state that needs to be set.

## 3.6 Command List/Queue

D3D12 Command List/Queues are features transformed out of D3D11 Device Context. Command List is responsible for buffering rendering commands which are then built into hardware commands known by the driver and finally executed by the Command Queue. Since each Command List can independently fill rendering commands without any intermediate lock protection, it operates faster than its counterpart in D3D11.

The Command List can be re-used repeatedly. When the app wants to re-submit the same Command List for GPU execution, it must wait until the execution of this Command List on GPU is complete; otherwise, the behavior is undefined. App can also reset the Command List after it is closed, no matter if the Command List is still being executed on GPU or not. A Command List that's been reset is equivalent to a blank Context. It no longer inherits any previous rendering state, so you need to set them again, such as PSO, Viewport, ScissorRect, RTV and DSV.

In general, in order to avoid the need for each frame to wait synchronously for the Command List being executed in the previous frame, we can prepare a number of spare Command Lists, and then check the previous pending Command Lists at the end of each frame. If a recent Command List has been executed, then it indicates that the previous Command Lists have also been executed because the Command List is executed strictly in sequence, which is equivalent to a FIFO queue. In order to determine whether a Command List has been executed, we need to use an object like Fence. When the Command Queue calls the ExecuteCommandList function, the Signal function allows the system to immediately notify the Fence object when the Command List has been executed by setting the expected value passed to the Signal function into the Fence object. So under normal circumstances, we will take the accumulated frame number of each frame as the expected value and pass it to the Signal function. The way to query whether a Command List is completed is determining whether the return value of GetCompletedValue is equal to or greater than the expected value.

The Command Queue will be bound with the SwapChain, so the first argument passed in when you create the SwapChain is Command Queue. At present, the common pattern of SwapChain is Flip*. In this mode, BufferCount should be greater than one, which means that there will be more than one BackBuffer in the SwapChain. In order to render them alternately, you need to switch the next BackBuffer to the current RenderTarget after the Present operation, automatically rewind based on the total number of BackBuffer that you've created. If you haven't performed the Present operation in a frame, then you cannot switch the BackBuffer, otherwise it will cause the system to crash.

There are three types of Command Queue: Direct, Copy and Compute. These three types of Command Queue can be performed in parallel.

- Direct Command Queue is responsible for handling Graphics rendering commands.
- Copy Command Queue is responsible for data upload or readback operation.
- Compute Command Queue is responsible for handling commands for general purpose (has nothing to do with rasterization) calculation.

For example, while we use Direct Command Queue to render a scene on a worker thread, we can use Copy Command Queue on another thread to handle the upload of Texture data. The operation in Direct

Command Queue that refers to the Texture data uploaded in the background must wait until the Copy Command Queue has been executed.

## Coming Soon:  Links to the Following Chapters

Chapter 4: DirectX 12 Features

Chapter 5: DirectX 12 Optimization