Tutorial: Migrating Your Apps to DirectX® 12 – Part 5
Chapter 5 DirectX 12 Optimization

5.0 Links to Previous Chapter
Chapter 1: Overview of DirectX 12
Chapter 2: DirectX 12 Tools
Chapter 3: Migrating From DirectX 11 to DirectX
Chapter 4: DirectX 12 Features

5.1 The Basics of DirectX 12 Multi-threading

5.1.1 Introduction
Graphics rendering is one of the main tasks of modern 3D games. In DirectX 9/10, technically all rendering APIs must be called in a thread. DirectX 11 enhanced multi-threading support, but the loads for various threads are very unbalanced. Rendering related loads are mainly completed in the game's main rendering threads and graphics drivers, which makes it challenging for rendering tasks to take full advantage of the capability of modern multi-core processors, and thus often becomes one of the main performance bottlenecks of game rendering pipeline.

In order to improve the efficiency of graphics rendering, multi-threading has gained unprecedented support in DirectX 12. In the redesigned DirectX 12, in order to allow the graphics rendering of applications to achieve maximum efficiency in the use of the multi-core CPU: on one hand, DirectX 12 pre-processes and reuses rendering commands as much as possible to reduce the cost of switching rendering state and enhance the processing efficiency of rendering API on CPU and GPU; On the other hand, DirectX 12 provides more efficient multi-threaded rendering mechanism that allows applications to maximize the use of multiple tasks and improve performance. The use of multi-threading can reduce the cost of graphics driver on the CPU side and significantly improve the productivity of GPU. DX12’s multi-threaded mechanism not only allows rendering tasks to run on different processor cores in parallel in a more balanced way to improve performance, but can reduce CPU power consumption, which is also very important for games on mobile platforms.

Intel demonstrated the Asteroids Demo developed with DirectX 11 and DirectX 12 at SIGGRAPH 2014. In this program, users can switch to DirectX 11 or DirectX 12 at runtime for rendering. In one frame rendered, 50,000 asteroids need to be drawn, which means that the Draw Call is submitted on the CPU side for 50,000 times; at the same time, due to the random combination of a large number of different textures, models and other data, this demo can reflect the difference between the two generations of graphics API in driver layer efficiency. With technologies such as multi-threading, DirectX 12 has shown great advantages in frame rate and power consumption compared to DirectX 11. Please refer to the DirectX development blog for details:

5.1.2 Key Infrastructures
(1) Command List and Command Queue
Command list and Command Queue are key infrastructures in DirectX 12 multi-threaded programming. Here, we’ll first briefly compare DirectX 12, DirectX 9 and DirectX 11 for their rendering commands.

In DirectX 9, most of the rendering commands are invoked by Device interfaces, such as BeginScene, Clear, and DrawIndexedPrimitive; while the rendering state is taken care of by Device SetRenderState. In DirectX 11, the rendering commands are mostly implemented by invoking the relevant interfaces on the Immediate Context. However, in DirectX 12, in order to pre-process single thread as much as possible while increasing the likelihood that multiple threads run in parallel, we need to use the Command List object. Most rendering commands described above can be implemented by invoking interfaces on the Command List (For specific definition of each interface, please refer to the d3d12.h header file of DirectX 12 SDK for ID3D12GraphicsCommandList interface declaration). In order to submit the Command List to GPU for execution, we need the Command Queue object. Here, Command Queue is primarily responsible for submitting the Command List, and synchronizes the execution of it. The following code demonstrates how to create a Command List and use it to record rendering commands which are finally submitted by the Command Queue.

Here is the code:

```cpp
Table 5.1: The Usage of Command List and Command Queue

// Command Allocator is responsible for Command List related memory allocation
// The D3D12_COMMAND_LIST_TYPE_DIRECT parameter indicates that this allocator is used
// for the directy type of Command List
ComPtr<ID3D12CommandAllocator> pCommandAllocator;
pDevice->CreateCommandAllocator(D3D12_COMMAND_LIST_TYPE_DIRECT,
IID_PPV_ARGS(pCommandAllocator));

// Create the Command List
ComPtr<ID3D12GraphicsCommandList> pCommandList;
pDevice->CreateCommandList(0, D3D12_COMMAND_LIST_TYPE_DIRECT,
pCommandAllocator, pPipelineState, IID_PPV_ARGS(&pCommandList));

// Description of Command Queue
// Type = D3D12_COMMAND_LIST_TYPE_DIRECT specifies that this Command Queue is well
// suited to submit the Command List
D3D12_COMMAND_QUEUE_DESC queueDesc = {}; 
queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;

// Create a Command Queue
ComPtr<ID3D12CommandQueue> pCommandQueue;
pDevice->CreateCommandQueue(&queueDesc, IID_PPV_ARGS(&pCommandQueue));

// Invoke the rendering related interfaces through the Command List
// For illustration, here we just name a few. Please refer to the
ID3D12GraphicsCommandList interface declaration for other interfaces
pCommandList->ClearRenderTargetView(rtvDescriptor, clearColor, 0, nullptr);
pCommandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
pCommandList->IASetVertexBuffer(view, 1, &VertexBufferView);
pCommandList->DrawInstanced(3, 1, 0, 0);

// Execute the Command List through the Command Queue which can submit multiple
// Command Lists at a time
ID3D12CommandList* ppCommandLists[] = { pCommandList.Get() }; 
pCommandQueue->ExecuteCommandLists(_countof(ppCommandLists), ppCommandLists);
```
It is worth nothing that, all interfaces on the DirectX 12 device are thread free, except that the Command List is single-threaded. To better parallelize rendering work across CPU cores, we use multiple Command Lists to split the rendering tasks, assign the rendering commands to different Command Lists, and finally submit the rendering commands on Command Lists to the Command Queue for GPU execution. By preparing multiple Command Lists, we can independently invoke the rendering command interfaces of individually maintained Command Lists in different threads. At the same time, Command Queue is thread free, different threads of applications can execute various Command Lists in any order on the Command Queue.

(2) Bundle and Pipeline State Object

In order to optimize the efficiency of driver in a single thread, DirectX 12 further introduces a second level of Command List, which is Bundle. The purpose of this object is to allow applications to create a set of API commands beforehand ("Record") for repeated use later. And when you create the Bundle, the display driver can pre-process these commands as much as possible to optimize this set of API commands to the maximum. Updating and maintaining the rendering states have always been a part that incurs a lot of performance overhead to the graphics driver. DirectX 12 abstracts this part of state into a Pipeline State Object (PSO), so as to better map it to the current state of the graphics hardware and thus reduce switching and management costs.

(3) Resource Barrier

In DirectX 12, the management of individual resource state has been handed by the graphics driver over to the application, which substantially reduces the cost for the driver to track and maintain the resource state. At this point, we need to use the Resource Barrier mechanism. The usage scenarios for this so-called "resource barrier" are very common. For example, a map can both be used as a map resource (Shader Resource View, SRV) reference when rendering and be treated as a render target (Render Target View, RTV). Let's look at a real-world example: as we need a shadow map, it is required to render the scene depth in advance into this map resource, in which case the resource is RTV; the n when rendering a scene with shadow effect, this map will be used as SRV. Now, these all need to be processed by the application itself using Resource Barrier to inform GPU of a resource state.

Here is the code: Table 5.2: The Usage of Resource Barrier

```c
// The shadow map transitions from the Common state to the Depth Write state to render the scene depth into it
pCommandList->ResourceBarrier(1,
&CD3DX12_RESOURCE_BARRIER::Transition(pShadowTexture, D3D12_RESOURCE_STATE_COMMON,
D3D12_RESOURCE_STATE_DEPTH_WRITE));

// The shadow map will be used as the Shader Resource of pixel shader. When rendering the scene, the shadow map will be sampled.
pCommandList->ResourceBarrier(1,
&CD3DX12_RESOURCE_BARRIER::Transition(pShadowTexture,
D3D12_RESOURCE_STATE_DEPTH_WRITE, D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));

// The shadow map restores to the Common state
pCommandList->ResourceBarrier(1,
&CD3DX12_RESOURCE_BARRIER::Transition(pShadowTexture,
D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE, D3D12_RESOURCE_STATE_COMMON));
```

(4) Fence
DirectX 12 introduces the Fence object to achieve GPU-CPU and GPU-GPU synchronization. Fence is a lock-free synchronization mechanism that meets the requirements for lightweight synchronization primitives. Basically, the communication is all about an integer variable.

For GPU-CPU synchronization, here is the code:

```cpp
// Create a Fence, where the initial value is fenceValue
ComPtr<ID3D12Fence>pFence;
pDevice->CreateFence(fenceValue, D3D12_FENCE_FLAG_NONE, IID_PPV_ARGS(&pFence));
```

There are two types of GPU-CPU synchronization enabled via Fence. The first one is that the thread on the CPU side queries the current value of Fence to get the task progress on the GPU side:

```cpp
pCommandQueue->Signal(pFence.Get(), fenceValue);
// Query the completion value (progress) on Fence on the CPU side
// If the value is smaller than fenceValue, call DoOtherWork.
if (pFence->GetCompletedValue() < fenceValue) {
    DoOtherWork();
}
```

The other one is that the thread on the CPU side can ask GPU to wake it up for synchronization purposes when the value on Fence reaches a specified value, and coordinate with other Win32 APIs to meet a lot of synchronization requirements.

Here is the code:

```cpp
if (pFence->GetCompletedValue() < fenceValue) {
    pFence->SetEventOnCompletion(fenceValue, hEvent);
    WaitForSingleObject(hEvent, INFINITE);
}
```

5.1.3 Example of Multi-threaded Rendering

Now we’ll try to illustrate how to use DirectX 12 multi-threading through a simple example, and how to split rendering tasks to significantly improve rendering efficiency. For ease of description and to keep it simple and easy to understand as much as possible, we will use pseudocode in combination, at the same time, we have to omit certain parameters in the functions, but this should not affect your understanding.

In our example, OnRender is a typical single-threaded render function of DirectX 12. It is used to render one frame in a game scene. In this function, we use the Command List to log all rendering commands, including the commands for setting a resource barrier state for the back buffer, clearing color and drawing each mesh, then use the Command Queue to execute the Command List, and finally present the whole picture via SwapChain.

The code for the render function looks like:

```cpp
void OnRender() {
}
```
Next, we will parallelize this render function by using DirectX 12 multi-threading to modify the program. In the program initialization phase, we created a number of worker threads to deal with the rendering commands for a large number of objects in the scene. For each worker thread, we evenly distribute the same number of meshes in the scene. At the same time, we create multiple Command Lists for each worker thread. Each Command List is responsible for logging some rendering tasks of the child threads. Typically, each child thread only needs to manage a single Command List. Here the benefit of creating multiple Command Lists (subtasks) for each worker thread is: when the worker thread is assigned many tasks, it can notify the main thread to submit the rendering commands to GPU without completing all the tasks, thus improving the parallelism of CPU/GPU. Win32’s semaphore and wait APIs are used to achieve synchronization between the main thread and the worker thread.

The code for the main thread render function looks like:

```cpp
void OnRender_MainThread()
{
    // Reset the Command List
    pCommandList->Reset(...);

    // Set barrier for the back buffer, change the barrier state from Present to Render Target
    pCommandList->ResourceBarrier(1, (...,
                                   D3D12_RESOURCE_STATE_PRESENT,
                                   D3D12_RESOURCE_STATE_RENDER_TARGET));

    // Set the render target
    pCommandList->OMSetRenderTargets(...);

    // Clear the render target
    pCommandList->ClearRenderTargetView(...);

    // Set the primitive/topology type
    pCommandList->IASetPrimitiveTopology(...);

    // Other operations on the Command List
    // ...

    // Draw each mesh
    foreach(Mesh in Meshes)
    {
        pCommandList->DrawInstanced(...);
    }

    // Set barrier for the back buffer, change the barrier state from Render Target to Present
    pCommandList->ResourceBarrier(1, (...,
                                   D3D12_RESOURCE_STATE_RENDER_TARGET,
                                   D3D12_RESOURCE_STATE_PRESENT));

    // Close the Command List
    pCommandList->Close();

    // Execute the Command List on the Command Queue
    pCommandQueue->ExecuteCommandLists(...);

    // Present using SwapChain
    pSwapChain->Present(...);
}
```

<table>
<thead>
<tr>
<th>Table 5.7: Multi-threaded Main Thread Render Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void OnRender_MainThread();</code></td>
</tr>
</tbody>
</table>
// Notify each child rendering thread to begin rendering
for workerId in workerIdList
{
    SetEvent(BeginRendering_Events[workerId]);
}

// Pre Command List is used to prepare the rendering
// Reset the Pre Command List
pPreCommandList->Reset(...);

// Set barrier between Presentation state to Rendering Target for the back buffer
pPreCommandList->ResourceBarrier(1, (...,
D3D12RESOURCESTATE_PRESENT,
D3D12RESOURCESTATE_RENDER_TARGET));

// Clear the color of back buffer
pPreCommandList->ClearRenderTargetView(...);

// Clear the depth/template of back buffer
pPreCommandList->ClearDepthStencilView(...);

// Other operations on the Pre Command List
// ...

// Close the Pre Command List
pPreCommandList->Close();

// Post Command List is used for finishing touches after the rendering
// Set barrier between Presentation state to Rendering Target for the back buffer
pPostCommandList->ResourceBarrier(1, (...,
D3D12RESOURCESTATE_RENDER_TARGET, D3D12RESOURCESTATE_PRESENT));

// Other operations on the Post Command List
// ...

// Close the Post Command List
pPostCommandList->Close();

// Submit the Pre Command List
pCommandQueue->ExecuteCommandLists(..., pPreCommandList);

// Wait for all worker threads to complete Task1
WaitForMultipleObjects(Task1_Events);

// Submit the Command Lists for Task1 on all the worker threads
pCommandQueue->ExecuteCommandLists(..., pCommandListsForTask1);

// Wait for all worker threads to complete Task2
WaitForMultipleObjects(Task2_Events);

// Submit the completed rendering commands (Command Lists for Task2 on all the worker threads)
pCommandQueue->ExecuteCommandLists(..., pCommandListsForTask2);

// ...

// Wait for all worker threads to complete TaskN
WaitForMultipleObjects(TaskN_Events);

// Submit the completed rendering commands (Command Lists for TaskN on all the worker threads)
pCommandQueue->ExecuteCommandLists(..., pCommandListsForTaskN);

// Submit the remaining Command Lists (pPostCommandList)
pCommandQueue->ExecuteCommandLists(..., pPostCommandList);

// using SwapChain presentation
pSwapChain->Present(...);
}

The code for the worker thread function looks like:

### Table 5.8: Multi-threaded Child Thread Render Function

```cpp
void OnRender_WorkerThread(workerId)
{
    // Each loop represents rendering of one frame on child threads
    while (running)
    {
        // Wait for event notification from the main thread to begin rendering
        one frame
        WaitForSingleObject(BeginRendering_Events[workerId]);

        // Rendering subtask1
        {
            pCommandList1->SetGraphicsRootSignature(...);
            pCommandList1->IASetVertexBuffers(...);
            pCommandList1->IASetIndexBuffer(...);
            // ...
            pCommandList1->DrawIndexedInstanced(...);
            pCommandList1->Close();

            // Notify the main thread that the rendering subtask1 on the
current worker thread completes
            SetEvent(Task1_Events[workerId]);
        }

        // Rendering subtask2
        {
            pCommandList2->SetGraphicsRootSignature(...);
            pCommandList2->IASetVertexBuffers(...);
            pCommandList2->IASetIndexBuffer(...);
            // ...
            pCommandList2->DrawIndexedInstanced(...);
            pCommandList2->Close();

            // Notify the main thread that the rendering subtask2 on the
current worker thread completes
            SetEvent(Task2_Events[workerId]);
        }

        // More rendering subtasks
        // ...

        // Rendering subtaskN
        {
            pCommandListN->SetGraphicsRootSignature(...);
            pCommandListN->IASetVertexBuffers(...);
            pCommandListN->IASetIndexBuffer(...);
            // ...
```
pCommandListN->DrawIndexedInstanced(...);
pCommandListN->Close();

// Notify the main thread that the rendering subtask on the
current worker thread completes
SetEvent(TaskN_Events[workerId]);
}
}

In this way, we successfully assign the task to the child threads, while having the main thread concentrate on tasks like preparation and finishing touches after the rendering. The child threads only need to timely notify the main thread of their work, and use multiple Command Lists to complete rendering commands for one frame without interruption. Meanwhile, the main thread can also concentrate on their work, wait for the child threads to complete the phased work where appropriate, and submit the related Command Lists in the child threads to GPU via Command Queue. Of course, the child threads can also submit the commands on the Command List via Command Queue as long as they can ensure that the rendering is done in the correct order. Here, for illustration purposes, we place the operation of submitting the Command List via Command Queue on the main thread. In addition, modern 3D games have extensive post-processing. We can place tasks like post-processing on the main thread, or on one or more child threads. Given the limited pages available, we didn’t include this part of the implementation in the sample code.

5.1.4 Summary
As an important part of the DirectX 12 design objective, multi-threading is an optimization solution worthwhile to try for every applications that are CPU-bound with CPU workload that can be parallelized among multiple threads. DirectX 12 API provides good multi-threading support. Through appropriate migration, single-threaded applications can be parallelized to make full use of the hardware performance and greatly improve rendering efficiency.

Notices
All sample source code is released under the Intel Sample Source Code License Agreement.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.
Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, and Intel Core are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others

© 2015 Intel Corporation.