2015

DirectX 12 技术白皮书

Ver 1.0

前言

随着微软最新一代操作系统 Windows 10 的发布，其核心图形技术也升级到了最新的 DirectX 12。为了帮助之前有 DirectX 经验的程序员尽快熟悉并上手使用 DirectX 12 来开发游戏，微软、英特尔和苏州蜗牛公司三方通力合作，结合了苏州蜗牛《九阳神功》PC DirectX 12 版本的开发经验编写了此白皮书。希望给每一位热爱微软 DirectX 技术的开发者朋友带来最前沿的第一手资料。本白皮书由来自微软的资深图形专家、DirectX12 项目经理 —— 陆建业担任技术顾问，每个章节由三家公司的技术专家合作完成：

第一章：梅颖广（微软）

第二章：梅颖广（微软）

第三章：吕文伟（蜗牛）

第四章：郭胜（英特尔）、吕文伟（蜗牛）

第五章：王凯（英特尔）、吕文伟（蜗牛）

此白皮书目前只是第一版本，我们计划在 2015 年年底会最终完成。期间特别希望能得到大家的建议和意见，这是我们的联系方式：

梅颖广：yimei@microsoft.com

郭　胜：sheng.guo@intel.com

吕文伟：lvww@snailgame.net

目录

第 1 章 DirectX 12 概述

DirectX，是微软在 Windows 操作系统平台上控制硬件底层、的 API 处理多媒体任务（特别是游戏和视频）的程序接口集。DirectX API 被划分为多个组件，组件之间的 API 相互独立，并且独立更新，这样就可以保证游戏只需要使用必须的函数。各个组件提供了访问不同硬件的能力，这些硬件可以包括显卡、声卡、 GPU 以及游戏操纵杆、鼠标等拥有标准接口的输入设备。

1.1 DirectX12 概述

DirectX12 主要包括如下一些组件：

表 1.1 DirectX12 组件

Direct2D	用于 2D 图形的绘制，是一个高性能的矢量函数渲染库
DirectWrite	使用 Direct2D 的应用程序中进行字体和文字的渲染
Direct3D	用于在 DirectX 中构建所有的 3D 图形。它就是最受注意的并且更新最频繁的 API
XAudio2	低级的音频处理 API
XACT3	是一个构建于 XAudio2 之上的高级音频处理 API
XInput	用于处理 Xbox 游戏机的等所有的输入操作
DirectCompute	这是一个新加进 DirectX 11 的 API 集，允许使用 GPU 执行通用多线程计算

1.1.1 DirectX 12 的开发环境和硬件配置

表 1.2 DirectX12 开发环境和硬件配置

开发环境

▼ 软件安装

- Windows 10
- Visual Studio 2013 + Update4 / Visual Studio 2015 (有更完善的 DX12 调试工具)
- DirectX 12 SDK

▼ 驱动安装

- AMD Driver
- Intel Driver
- NVIDIA Desktop / Mobile Driver

硬件配置

- CPU：Intel Core i5-4670K
- 内存：16GB+ RAM
- 显卡（支持 Feature Level 11.1 以上）：
  - NVIDIA：GTX 970/980(4GB+ RAM) / GTX TITAN (4GB+ RAM)
  - AMD： R9 290 (4GB+ RAM)
  - INTEL：HasWell 及以上处理器的核显
- 硬盘：1TB

1.1.2 Direct3D 12

Direct3D 是 DirectX 组件中最为重要核心的部分。随着 DirectX 12 的发布，Direct3D 也更新到了最新的一代 ——Direct3D 12。相比之前的版本，Direct3D 12 更快更有效，可以支持更丰富的场景，更多的物体，更酷炫的特性，能够充分利用主流 GPU。

性能更高效

Direct3D 12 提供了比以往更低的硬件抽象层，使游戏和应用程序能显著提高 CPU 利用率和多线程负载均衡。Direct3D 12 允许游戏和应用极大程度地自己管理内存。此外，通过使用 Direct3D 12 的新特性，可以大幅减少 GPU 开销，提高游戏和应用的表现效果。这些新特性包括：指令队列和列表(command queues and lists)，资源描述符表(descriptor tables)，管道状态对象(pipeline state objects)

等。此外 Direct3D 12 还支持很多渲染管线新技术诸如：保守光栅化算法(conservative rasterization)，volume-tiled resources，和 raster-order views。

主机更强大

在渲染功能上，Direct3D 12 是 Direct3D 11 的一个超集。在保留 Direct3D 11 渲染功能的同时，能更好地发挥现代显卡的多核 CPU 和 GPU 的效能，极大提高了主机平台 API 的效率。基于 Direct3D 12，保证充分使用主机所有的 CPU 和 GPU 内核资源的基础上，现代主机能够挖掘其最强大的渲染潜能。

工具更完善

Direct3D 12 提供了更完善的 PIX 工具用于在 D3D 应用程序运行时的调试和分析。 除了能够追踪很多有用的信息（诸如 API 的调用，时间统计），能够在 GPU 上调试着色器代码，打断点和进入代码调试之外，原本针对 Xbox 应用性能检测不够完善的情况现在也已经大大改善。目前最新的 PIX 工具提供了对于 DirectX 应用程序的全部图形化调试环境。

1.2 Direct3D 12 的核心概念

1.2.1 Work Submission

当程序给 GPU 提交渲染任务(work submission)时，Direct3D 12 给与程序很高的控制权。在提交渲染任务时，程序首先会记录渲染指令到指令列表(command lists)里，然后提交这些指令列表到一个 GPU 的指令队列(command queque)中。Direct3D 12 支持 CPU 多线程来同时处理多指令列表的记录。当然，指令列表提交至指令队列的过程也是线程自由的。

指令列表提交后的执行对于 CPU 来说是异步的。也就是，当程序提交指令列表到指令队列后，CPU 不会等命令的执行，而是直接回到程序中。这样就可以保证 CPU 在任何时刻都能提交大量的渲染指令列表。Direct3D 12 提供了在 CPU 和 GPU 之间的同步通信原语(synchronization primitives)用来获知 GPU 执行渲染任务的进展。

这种指令列表的模式是很高效的。API 指令被直接转换成原生态的 GPU 指令可以最大程度地降低显示驱动的负担，这提供了一种高性能的渲染显示方案。为了达到这样的性能，在一个应用的最终发布版本中构建渲染指令列表时，显示驱动的核心程序应只执行最少量的错误检查工作。在开发过程中，你可以使用调试工具来确保渲染的正确。为了方便输入验证、分析和调试，您也可以使用一个调试层，利用 Direct3D 12 的 API 进行全面的状态检查，来获知错误和警告（比如“无效渲染操作”的错误、“使用未定义的渲染操作”的警告等）。

1.2.2 Memory Management

Direct3D 12 API 所处理的数据存储在被称为资源堆(resource heap)的内存对象上。

资源堆可以存在于本地显存或系统内存中，并有一定的缓存(cache)、CPU 可访问或者 GPU 可访问等的特性。一个应用可以通过使用各种堆创建 API 的参数来控制资源堆内存的属性和配置。

在一个资源堆内，程序可以自由地分配资源，比如纹理和缓冲(buffer)。这是一个相对轻量级的操作，在一个现有的资源堆里创建资源是不产生实际的内存分配操作的。为了让 GPU 能正确地访问这些资源，应用程序需要创建资源描述符视图(descriptor views)，比如：着色器资源视图(shader resource views)或无序访问视图(unordered access views)。

为了读取或写入 CPU 的资源堆内存，应用程序必须映射一个基于堆的资源用于 CPU 访问。

在 Diect3D 12 中，为了确保 GPU 内存分配数据的连续性以及在流水线操作中的正确，一般采用环形缓冲(ring buffer)或相类似的技术。API 提供了同步原语确保这种类型的流水线操作。

1.3 Direct3D 11 到 Direct3D12 的重要变化

Direct3D 12 和 Direct3D 11 在编程模型上有很大的不同。Direct3D 12 让我们比以往任何时候都更接近硬件。正因为如此，Direct3D 12 才能做到更快、 更高效。当然反过来，在获得更高效率的同时应用程序也需要比使用 Direct3D 11 时承担更多的任务。

Direct3D 12 是回归到更底层的编程；它通过引入如下所述的新特性来使得程序能更好地控制游戏和应用的图形元素：使用对象来表示的管道状态(pipeline state objects)，用指令列表和集合(Command lists and bundles)来提交渲染工作以及使用描述符表(descriptor tables)来访问资源的总体状况。

1.3.1 Direct3D 12 和 Direct3D 11 抉择

使用 Direct3D 12 在提升应用效率的同时，也需要应用承担更多的任务

- 在 Direct3D 12，CPU 和 GPU 的同步工作是应用程序必须处理的，而 Direct3D 11 中是在 runtime 中隐式执行。这也意味着 Direct3D 12 不会自动检查管道危障(pipeline hazard)，而需要由应用程序去做。
- 使用 Direct3D 12 由应用程序负责流水线数据更新。也就是说，在 Direct3D 12 中，必须手动执行 Direct3D 11 中的"Map/Lock-DISCARD"模式。在 Direct3D 11 中，当你使用 D3D11_MAP_WRITE_DISCARD 标识调用 ID3D11DeviceContext::Map 时，如果 GPU 仍然使用的缓冲区，runtime 返回一个新内存区块的指针代替旧的缓冲数据。这让 GPU 能够在应用程序往新缓冲填充数据的同时仍然可以使用旧的数据。应用程序不需要额外的内存管理。旧的缓冲在 GPU 使用完后会自动销毁或重用。
- 在 Direct3D 12 中，所有的动态更新(包括 constant buffer，dynamic vertex buffer，dynamic textures 等等)都由应用程序来控制。这些动态更新包括必要的 GPU fence 或 buffering，由应用程序来保证内存的可用性。
- Direct3D 12 只将 COM 风格的引用计数用于 interface 的生命周期(通过 Direct3D 的弱引用模型关联到 device 的生命周期)。所有的 resource 和 description 内存生命周期都由应用程序负责保证，不使用引用计数。而 Direct3D 11 使用引用计数来管理 interface 相关的对象。

1.3.2 管道状态对象(pipeline state objects)

Direct3D 11 允许使用大量独立的对象的集合来操纵管线状态。例如，input assembler state，pixel shader state，rasterizer state 和 output merge state 都能够独立进行修改。这种设计提供了便利性和相对高层的图形管线表示。但是主要由于各种各样的 state 通常是互相关联的, 这种设计不能充分发挥现代硬件的性能。例如，很多 GPU 将 pixel shader 和 output merger state 合并到一个硬件表示。因为 Direct3D 11 API 允许管线状态分别设置，所以驱动必需等到 Draw 的时候才能定下管线状态。这种方案延迟了硬件的状态设置，也就意味着额外的开销和更少的每帧 DrawCall。

Direct3D 12 通过将大部分管线状态统一到不可变的管线状态对象(PSOs)来解决这个问题，PSOs 在创建的时间就决定了。硬件和驱动能够立即将 PSO 转换为硬件原生指令和状态，而使 GPU 工作。你仍然可以动态切换正在使用的 PSO，硬件只需要直接拷贝最少的预计算状态到硬件寄存器，而不是实时计算硬件状态。通过使用 PSOs，DrawCall 的开销显著的减少，每帧可以有更多的 DrawCall。

1.3.3 指令列表和集合(Command lists and bundles)

在 Direct3D 11 中，所有的任务提交都通过 immediate context 完成，immediate context 代表了一条送往 GPU 的指令流。要实现多线程负载均衡，游戏可以使用 deferred context。Direct3D 11 中的 Deferred Context 不能完美的映射到硬件，所以它们能做的事情有限。

Direct3D 12 引入了新的模型来执行任务提交。指令列表包含了在 GPU 上执行一个具体工作所需的所有信息。每个命令列表包含的信息包括使用哪个 PSO，需要什么纹理和缓冲资源和所有 DrawCall 的参数。因为每个指令列表是自包含的并且没有状态继承，驱动能够以一种自由线程化(free-

threaded)的方式预先计算所有需要的 GPU 命令。接下来唯一需要进行的处理是通过指令队列将指令列表最终提交到 GPU。
除了指令列表，Direct3D 12 还引入了一个二级的任务预计算方式：bundle。不同于指令列表：完全的自包含，典型的构造、一次提交、丢弃，bundle 提供了某种形式的状态继承用来复用。例如，如果游戏想用不同的纹理绘制两个角色模型。一种方法是用一个指令列表记录两组完整一样的 DrawCall。另一种方法是记录一个绘制单一角色模型的 bundle，然后用不同的资源在指令列表上“回放”bundle 两次。在后一种情况下，显示驱动只需要计算相应的指令一次，而创建指令列表本质上相当于两个低开销的函数调用。

1.3.4 描述符堆和表(Descriptor Heap and Table)

Direct3D 11 中的资源绑定高度抽象和便利，但很多现代硬件能力没有被利用到。在 Direct3D 11 中，游戏创建资源的视图对象，然后绑定这些视图到管线中不同 shader 阶段的 slot 中。然后 Shader 从显式绑定的 slot 读取数据，这些绑定 slot 在绘制时是固定的。这个模型意味着每当游戏使用不同的资源绘制，它必须重新绑定不同的视图到不同的 slot，然后再次调用绘制函数。这种情况表示额外的开销能够通过完全利用硬件能力来消除。

Direct3D 12 改变了绑定模型来匹配现代硬件并显著的提升了性能。和需要独立的资源视图和显式绑定到 slot 不同，Direct3D 12 提供了一个 descriptor heap 用来创建游戏中不同的资源。这个方案提供了一种机制让 GPU 预先直接写入硬件本地资源描述(descriptor)到内存。游戏可以在整个描述符堆中指定一个或者几个描述符表来声明某个 DrawCall 中哪些资源在管道中使用。因为 descriptor heap 填充的都是最恰当的硬件特定的 descriptor，改变 descriptor table 是消耗相当低的操作

除了由 descriptor heap 和 table 带来的性能提升。Direct3D 12 还允许资源在 shader 里被动态的索引，这提供了空前的灵活性并诞生了新渲染技术。举例来说，现代延迟渲染引擎一般将一个某种形式的材质或物体标识符编码到中间的 G-Buffer。在 Direct3D 11 中，这些引擎必须小心的避免使用太多的材质，因为在一个 G-Buffer 中包含太多会极大的影响最终渲染的速度。有了能动态索引的资源，一个有上千材质的场景能够最终和只有十个材质的场景一样快。

1.4 创建第一个 DirectX12 的程序

DirectX12 的程序和之前一样，采用最标准的图形循环程序流程。

表 8.3 DirectX12 主程序结构

```
Initialize();
Do
{
    Update();
    Render();
}
while (1);
Destroy();
```

1.4.1 初始化(Initialize)

这一步包括初始化全局变量和类，程序必须初始化 Pipeline 和 Assets。

1. 初始化 Pipeline 的步骤：
   创建 device 和 swap chain
   创建 command allocator

参考代码如下所示。

表 1.4 Pipeline 示例代码

```
InitPipeline(HWND hWnd)
{
        //
```

```
        // create swap chain descriptor
        //

        DXGI_SWAP_CHAIN_DESC descSwapChain;
        ZeroMemory(&descSwapChain, sizeof(descSwapChain));
        descSwapChain.BufferCount = cNumSwapBufs;
        descSwapChain.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
        descSwapChain.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
        descSwapChain.SwapEffect = DXGI_SWAP_EFFECT_FLIP_SEQUENTIAL;
        descSwapChain.OutputWindow = hWnd;
        descSwapChain.SampleDesc.Count = 1;
        descSwapChain.Windowed = TRUE;

        //
        // create the device
        //

        HRESULT hardware_driver = createDeviceAndSwapChain(
                nullptr,
                D3D_DRIVER_TYPE_HARDWARE,
                deviceFlags,
                D3D_FEATURE_LEVEL_9_1,
                D3D12_SDK_VERSION,
                &descSwapChain,
                IID_PPV_ARGS(mSwapChain.GetAddressOf()),
                IID_PPV_ARGS(mDevice.GetAddressOf()),
                IID_PPV_ARGS(mCommandQueue.GetAddressOf())
                );

        if (!SUCCEEDED(hardware_driver)) {
                createDeviceAndSwapChain(
                        nullptr,
                        D3D_DRIVER_TYPE_WARP,
                        deviceFlags,
                        D3D_FEATURE_LEVEL_9_1,
                        D3D12_SDK_VERSION,
                        &descSwapChain,
                        IID_PPV_ARGS(mSwapChain.GetAddressOf()),
                        IID_PPV_ARGS(mDevice.GetAddressOf()),
                        IID_PPV_ARGS(mCommandQueue.GetAddressOf())
                        );
        }

        //
        // create the command allocator object
        //

        mDevice->CreateCommandAllocator(D3D12_COMMAND_LIST_TYPE_DIRECT,
IID_PPV_ARGS(mCommandAllocator.GetAddressOf()));
        }
```

其中 CreateDeviceAndSwapChain() 代码如下所示。

表 1.5 CreateDeviceAndSwapChain 示例代码

```
HRESULT CreateDeviceAndSwapChain(
        _In_opt_ IDXGIAdapter* pAdapter,
        D3D_DRIVER_TYPE DriverType,
        D3D_FEATURE_LEVEL MinimumFeatureLevel,
        UINT SDKVersion,
        _In_opt_ CONST DXGI_SWAP_CHAIN_DESC* pSwapChainDesc,
        _In_ REFIID riidSwapchain,
        _COM_Outptr_opt_ void** ppSwapChain,
        _In_ REFIID riidDevice,
        _COM_Outptr_opt_ void** ppDevice,
        _In_ REFIID riidQueue,
        _COM_Outptr_opt_ void **ppQueue
        )
```

```
{
	ComPtr<ID3D12Device> pDevice;
	ComPtr<IDXGIFactory> pDxgiFactory;
	ComPtr<IDXGISwapChain> pDxgiSwapChain;
	ComPtr<ID3D12CommandQueue> pQueue;

	//
	// create the D3D 12 device
	//

	HRESULT hr = D3D12CreateDevice(
		pAdapter,
		MinimumFeatureLevel,
		__uuidof(ID3D12Device),
		IID_PPV_ARGS(&pDevice)
		);
	if (FAILED(hr)) { return hr; }

	D3D12_COMMAND_QUEUE_DESC queueDesc;
	ZeroMemory(&queueDesc, sizeof(queueDesc));
	queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
	queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;
	hr = pDevice->CreateCommandQueue(&queueDesc, IID_PPV_ARGS(&pQueue));

	hr = CreateDXGIFactory1(IID_PPV_ARGS(&pDxgiFactory));
	if (FAILED(hr)) { return hr; }

	//
	//  create the swap chain
	//

	DXGI_SWAP_CHAIN_DESC LocalSCD = *pSwapChainDesc;
	hr = pDxgiFactory->CreateSwapChain(
		pQueue.Get(), // Swap chain needs the queue so it can force a flush on it
		&LocalSCD,
		&pDxgiSwapChain
		);
	if (FAILED(hr)) { return hr; }

	//
	// get the required pointers to the device, queue and swap chain
	//

	hr = pDevice.Get()->QueryInterface(riidDevice, ppDevice);
	if (FAILED(hr)) { return hr; }

	hr = pQueue.Get()->QueryInterface(riidQueue, ppQueue);
	if (FAILED(hr)) { return hr; }

	hr = pDxgiSwapChain.Get()->QueryInterface(riidSwapchain, ppSwapChain);
	if (FAILED(hr))
	{
		reinterpret_cast<IUnknown*>(*ppDevice)->Release();
		return hr;
	}

	return S_OK;
}
```

2. 初始化 Assets 的步骤：
   Compile the shaders.
   Create an input layout.
   Create an empty root signature.
   Create a pipeline state object description, then create the object.
   Create a descriptor heap.

Create a command list.
Create a backbuffer and render target.
Setup the viewport.
Create a scissor rectangle.
Create and load the vertex buffers.
Create the vertex buffer views.
Create a fence.
Close the command list, and then execute it to initialize the GPU setup.
Create an event handle.
Wait for the GPU to finish.

可以看出加载和准备 Asserts 是比较长的一个过程。其中不少阶段是和 D3D11 类似，当然也有的过程是 D3D12 新加的。在 Direct3D 12 中需要通过管道状态对象(PSO)把管道状态(pipeline state)连接到指令列表(Command List)，可以把 PSO 存储为一个成员变量而多次使用。一个描述符堆定义了视图和如何访问资源（比如：render target view）。
参考代码如下所示。

表 1.6 Assets 示例代码

```
void InitAssets()
{
	//
	// handles to vert and pixel shaders
	//

	ComPtr<ID3DBlob> blobShaderVert, blobShaderPixel;

	//
	// compile shaders
	//

	D3DCompileFromFile(L"shaders.hlsl", nullptr, nullptr, "VShader", "vs_5_0", 0, 0,
blobShaderVert.GetAddressOf(), nullptr);
	D3DCompileFromFile(L"shaders.hlsl", nullptr, nullptr, "PShader", "ps_5_0", 0, 0,
blobShaderPixel.GetAddressOf(), nullptr);

	//
	// create input layout
	//

	D3D12_INPUT_ELEMENT_DESC layout[] =
	{
			{ "POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
D3D12_INPUT_PER_VERTEX_DATA, 0 },
			{ "COLOR", 0, DXGI_FORMAT_R32G32B32A32_FLOAT, 0, 12,
D3D12_INPUT_PER_VERTEX_DATA, 0 }
		};
	UINT numElements = sizeof(layout) / sizeof(layout[0]);

	//
	// create an empty root signature
	//

	ComPtr<ID3DBlob> pOutBlob, pErrorBlob;
	D3D12_ROOT_SIGNATURE_DESC descRootSignature;
	descRootSignature.Init(0, nullptr, 0, nullptr,
D3D12_ROOT_SIGNATURE_FLAG_ALLOW_INPUT_ASSEMBLER_INPUT_LAYOUT);
	D3D12SerializeRootSignature(&descRootSignature, D3D_ROOT_SIGNATURE_VERSION_1,
pOutBlob.GetAddressOf(), pErrorBlob.GetAddressOf());
	mDevice->CreateRootSignature(0, pOutBlob->GetBufferPointer(), pOutBlob-
>GetBufferSize(), IID_PPV_ARGS(mRootSignature.GetAddressOf()));

	//
	// create a PSO description
	//
```

```
        D3D12_GRAPHICS_PIPELINE_STATE_DESC descPso;
        ZeroMemory(&descPso, sizeof(descPso));
        descPso.InputLayout = { layout, numElements };
        descPso.pRootSignature = mRootSignature.Get();
        descPso.VS = { reinterpret_cast<BYTE*>(blobShaderVert->GetBufferPointer()),
blobShaderVert->GetBufferSize() };
        descPso.PS = { reinterpret_cast<BYTE*>(blobShaderPixel->GetBufferPointer()),
blobShaderPixel->GetBufferSize() };
        descPso.RasterizerState = CD3DX12_RASTERIZER_DESC(D3D12_DEFAULT);
        descPso.BlendState = CD3DX12_BLEND_DESC(D3D12_DEFAULT);
        descPso.DepthStencilState.DepthEnable = FALSE;
        descPso.DepthStencilState.StencilEnable = FALSE;
        descPso.SampleMask = UINT_MAX;
        descPso.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
        descPso.NumRenderTargets = 1;
        descPso.RTVFormats[0] = DXGI_FORMAT_R8G8B8A8_UNORM;
        descPso.SampleDesc.Count = 1;

        //
        // create the actual PSO
        //

        mDevice->CreateGraphicsPipelineState(&descPso,IID_PPV_ARGS(mPSO.GetAddressOf()));

        //
        // create descriptor heap
        //

        D3D12_DESCRIPTOR_HEAP_DESC descHeap = {};
        descHeap.NumDescriptors = 1;
        descHeap.Type = D3D12_DESCRIPTOR_HEAP_TYPE_RTV;
        descHeap.Flags = D3D12_DESCRIPTOR_HEAP_FLAG_NONE;
        mDevice->CreateDescriptorHeap(&descHeap, IID_PPV_ARGS(mDescriptorHeap.GetAddressOf()));

        //
        // create command list
        //

        mDevice->CreateCommandList(0, D3D12_COMMAND_LIST_TYPE_DIRECT, mCommandAllocator.Get(),
mPSO.Get(), IID_PPV_ARGS(mCommandList.GetAddressOf()));

        //
        // create backbuffer/rendertarget
        //

        mSwapChain->GetBuffer(0, IID_PPV_ARGS(mRenderTarget.GetAddressOf()));
        mDevice->CreateRenderTargetView(mRenderTarget.Get(), nullptr, mDescriptorHeap-
>GetCPUDescriptorHandleForHeapStart());

        //
        // set the viewport
        //

        mViewPort =
        {
                0.0f,
                0.0f,
                static_cast<float>(mWidth),
                static_cast<float>(mHeight),
                0.0f,
                1.0f
        };

        //
        // create scissor rectangle
        //

        mRectScissor = { 0, 0, mWidth, mHeight };
```

```
        //
        // create geometry for a triangle
        //

        VERTEX triangleVerts[] =
        {
                { 0.0f, 0.5f, 0.0f, { 1.0f, 0.0f, 0.0f, 1.0f } },
                { 0.45f, -0.5, 0.0f, { 0.0f, 1.0f, 0.0f, 1.0f } },
                { -0.45f, -0.5f, 0.0f, { 0.0f, 0.0f, 1.0f, 1.0f } }
        };

        //
        // actually create the vert buffer
        // Note: using upload heaps to transfer static data like vert buffers is not
recommended.
        // Every time the GPU needs it, the upload heap will be marshalled over.  Please read
up on Default Heap usage.
        // An upload heap is used here for code simplicity and because there are very few verts
to actually transfer
        //

        mDevice->CreateCommittedResource(
                &CD3DX12_HEAP_PROPERTIES(D3D12_HEAP_TYPE_UPLOAD),
                D3D12_HEAP_FLAG_NONE,
                &CD3DX12_RESOURCE_DESC::Buffer(3 * sizeof(VERTEX)),
                D3D12_RESOURCE_STATE_GENERIC_READ,
                nullptr,    // Clear value
                IID_PPV_ARGS(mBufVerts.GetAddressOf()));

        //
        // copy the triangle data to the vertex buffer
        //

        UINT8* dataBegin;
        mBufVerts->Map(0, nullptr, reinterpret_cast<void**>(&dataBegin));
        memcpy(dataBegin, triangleVerts, sizeof(triangleVerts));
        mBufVerts->Unmap(0, nullptr);

        //
        // create vertex buffer view
        //

        mDescViewBufVert.BufferLocation = mBufVerts->GetGPUVirtualAddress();
        mDescViewBufVert.StrideInBytes = sizeof(VERTEX);
        mDescViewBufVert.SizeInBytes = sizeof(triangleVerts);

        //
        // create fencing object
        //

        mDevice->CreateFence(0, D3D12_FENCE_FLAG_NONE, IID_PPV_ARGS(mFence.GetAddressOf()));
        mCurrentFence = 1;

        //
        // close the command list and use it to execute the initial GPU setup
        //

        mCommandList->Close();
        ID3D12CommandList* ppCommandLists[] = { mCommandList.Get() };
        mCommandQueue->ExecuteCommandLists(_countof(ppCommandLists), ppCommandLists);

        //
        // create event handle
        //

        mHandleEvent = CreateEventEx(nullptr, FALSE, FALSE, EVENT_ALL_ACCESS);

        //
```

```
        // wait for the command list to execute; we are reusing the same command list in our
main loop but for now,
        // we just want to wait for setup to complete before continuing
        //

        waitForGPU();
```

其中 CPUwaitForGPU() 代码如下所示。

表 1.7 CPUwaitForGPU 示例代码

```
Void CPUwaitForGPU()
{
        //
        // signal and increment the fence value
        //

        const UINT64 fence = mCurrentFence;
        mCommandQueue->Signal(mFence.Get(), fence);
        mCurrentFence++;

        //
        // Let the previous frame finish before continuing
        //

        if (mFence->GetCompletedValue() < fence)
        {
                mFence->SetEventOnCompletion(fence, mHandleEvent);
                WaitForSingleObject(mHandleEvent, INFINITE);
        }
}
```

1.4.2 更新(Update)

这部分就是放置游戏逻辑代码的地方

1.4.3 渲染(Render)

这部分是处理描画工作，主要过程为：

1. 填充指令列表(populateCommandLists)
3. 重置 command list allocator
4. 重置 command list
5. 设置 graphics root signature
6. 设置 viewport and scissor rectangles
7. 设置 resource barrier, 标识资源是 render target
8. 记录指令到 command list
9. 指令列表执行后标识 render target 为目前使用状态(present)
10. 关闭指令列表等待下一次的记录
2. 执行指令列表
3. 翻转前后缓冲
4. 等待 GPU 处理

参考代码如下所示。

表 1.8 Render 示例代码

```
void Render()
{
        //
        // record all the commands we need to render the scene into the command list
        //
```

```
        populateCommandLists();

        //
        // execute the command list
        //

        ID3D12CommandList* ppCommandLists[] = { mCommandList.Get() };
        mCommandQueue->ExecuteCommandLists(_countof(ppCommandLists), ppCommandLists);

        //
        // swap the back and front buffers
        //

        mSwapChain->Present(1, 0);
        mIndexLastSwapBuf = (1 + mIndexLastSwapBuf) % cNumSwapBufs;
        mSwapChain->GetBuffer(mIndexLastSwapBuf,
IID_PPV_ARGS(mRenderTarget.ReleaseAndGetAddressOf()));
        mDevice->CreateRenderTargetView(mRenderTarget.Get(), nullptr, mDescriptorHeap-
>GetCPUDescriptorHandleForHeapStart());

        //
        // wait and reset everything
        //

        waitForGPU();
}
```

其中 populateCommandLists () 代码如下所示。

表 1.9 populateCommandLists 示例代码

```
void populateCommandLists()
{
        //
        // command list allocators can be only be reset when the associated command lists have
finished execution on the GPU;
        // apps should use fences to determine GPU execution progress
        //

        mCommandAllocator->Reset();

        //
        // HOWEVER, when ExecuteCommandList() is called on a particular command list, that
command list can then be reset
        // anytime and must be before rerecording
        //

        mCommandList->Reset(mCommandAllocator.Get(), mPSO.Get());

        //
        // set the graphics root signature
        //

        mCommandList->SetGraphicsRootSignature(mRootSignature.Get());

        //
        // set the viewport and scissor rectangle
        //

        mCommandList->RSSetViewports(1, &mViewPort);
        mCommandList->RSSetScissorRects(1, &mRectScissor);

        //
        // indicate this resource will be in use as a render target
        //

        setResourceBarrier(mCommandList.Get(), mRenderTarget.Get(),
```

```
D3D12_RESOURCE_STATE_PRESENT, D3D12_RESOURCE_STATE_RENDER_TARGET);

        //
        // record commands
        //

        float clearColor[] = { 0.0f, 0.2f, 0.4f, 1.0f };
        mCommandList->ClearRenderTargetView(mDescriptorHeap-
>GetCPUDescriptorHandleForHeapStart(), clearColor, nullptr, 0);
        mCommandList->OMSetRenderTargets(&mDescriptorHeap-
>GetCPUDescriptorHandleForHeapStart(), true, 1, nullptr);
        mCommandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
        mCommandList->IASetVertexBuffers(0, &mDescViewBufVert, 1);
        mCommandList->DrawInstanced(3, 1, 0, 0);

        //
        // indicate that the render target will now be used to present when the command list is
done executing
        //

        setResourceBarrier(mCommandList.Get(), mRenderTarget.Get(),
D3D12_RESOURCE_STATE_RENDER_TARGET, D3D12_RESOURCE_STATE_PRESENT);

        //
        // all we need to do now is close the command list before executing it
        //

        mCommandList->Close();
}
```

其中 setResourceBarrier () 代码如下所示。

表 1.10 setResourceBarrier 示例代码

```
Void setResourceBarrier(ID3D12GraphicsCommandList* commandList, ID3D12Resource* resource, UINT
stateBefore, UINT stateAfter)
{
        D3D12_RESOURCE_BARRIER descBarrier = {};

        descBarrier.Type = D3D12_RESOURCE_BARRIER_TYPE_TRANSITION;
        descBarrier.Transition.pResource = resource;
        descBarrier.Transition.Subresource = D3D12_RESOURCE_BARRIER_ALL_SUBRESOURCES;
        descBarrier.Transition.StateBefore = stateBefore;
        descBarrier.Transition.StateAfter = stateAfter;

        commandList->ResourceBarrier(1, &descBarrier);
}
```

1.4.4 销毁(Destroy)

这部分主要是资源释放和关闭程序

1. 等待 GPU 结束
2. 切换出全屏状态
3. 关闭 Event

参考代码如下所示。

表 1.11 Destory 示例代码

```
void Destroy()
{
        //
        // wait for the GPU to be done with all resources
        //

        waitForGPU();
```

```
        mSwapChain->SetFullscreenState(FALSE, nullptr);

        CloseHandle(mHandleEvent);
}
```

1.4.5 代码补充

shaders.hlsl 参考如下所示。

表 1.12 shader 示例代码

```
struct VOut
{
        float4 position : SV_POSITION;
        float4 color : COLOR;
};

VOut VShader(float4 position : POSITION, float4 color : COLOR)
{
        VOut output;

        output.position = position;
        output.color = color;

        return output;
}


float4 PShader(float4 position : SV_POSITION, float4 color : COLOR) : SV_TARGET
{
        return color;
}
```

上述代码用的一些变量罗列在此处。

表 1.13 变量定义

```
int mIndexLastSwapBuf = 0;
int cNumSwapBufs = 2;
ComPtr<IDXGISwapChain> mSwapChain;

//
// vertex definition
//

struct VERTEX{ FLOAT X, Y, Z; FLOAT Color[4]; };

//
// pipeline objects
//

D3D12_VIEWPORT mViewPort;
D3D12_RECT mRectScissor;

ComPtr<ID3D12Device> mDevice;
ComPtr<ID3D12Resource> mRenderTarget;
ComPtr<ID3D12CommandAllocator> mCommandAllocator;
ComPtr<ID3D12CommandQueue> mCommandQueue;
ComPtr<ID3D12RootSignature> mRootSignature;
ComPtr<ID3D12DescriptorHeap> mDescriptorHeap;

//
// fence objects
//

ComPtr<ID3D12Fence> mFence;
UINT64 mCurrentFence;
HANDLE mHandleEvent;
```

```
//
// asset objects
//

ComPtr<ID3D12PipelineState> mPSO;
ComPtr<ID3D12GraphicsCommandList> mCommandList;
ComPtr<ID3D12Resource> mBufVerts;

D3D12_VERTEX_BUFFER_VIEW mDescViewBufVert;
```

第 2 章 DirectX12 工具

2.1 Visual Studio 的图形诊断工具

我们推荐使用 Visual Studio 2015 来开发 DirectX12 程序。下面的内容主要针对 Visual Studio 2015 版本的 Graphics Diagnostics Tool。

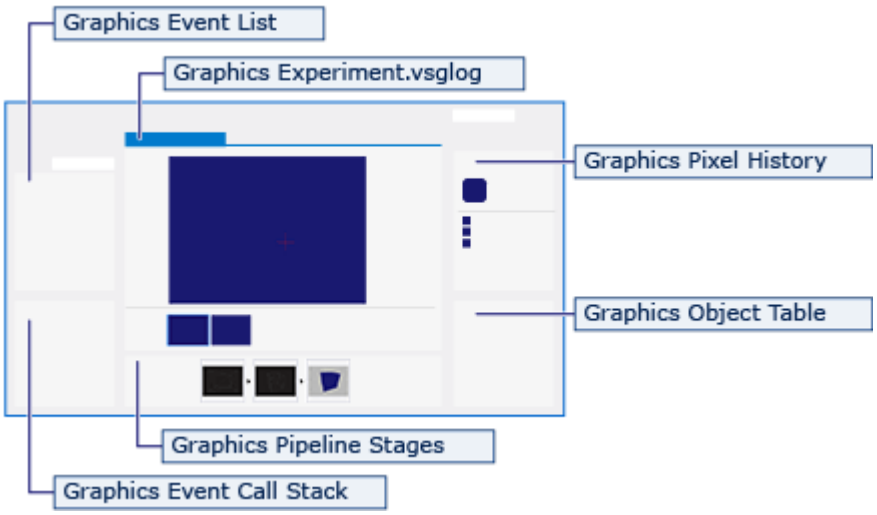
2.1.1 图形诊断工具概述

Visual Studio 2015 的图形诊断工具(Graphics Diagnostics) 是一套记录、分析 Direct3D 应用程序的表现和性能问题的工具集。图形诊断程序不仅可以用来诊断运行在 Windows PC 和 Windows 设备的模拟器的程序，也可以调试运行在远端的 PC 和设备上的程序。

为了最精确地分析程序使用 Direct3D 的情况，图形诊断工具能够在应用运行时直接捕捉应用的一个状态并立即进行分析，共享，或保存供以后分析。开发者不仅可以使用命令行工具 **dxcap.exe** 来启动和手动控制捕捉，而且 VS 也提供了从 VS 界面捕获帧、从应用界面捕获帧及使用捕获 API 自动捕获帧三种不同方式，来帮助用户编程控制启动和捕捉。

要诊断应用的性能问题，推荐使用 **Frame Analysis** 工具，它是图形诊断工具的一个新功能，来分析抓获的帧数据。相比开发者需要自己手工去修改图形参数并不断的比较前后的性能才能决定所做的修改是否合适，此工具会自动更改应用程序使用 Direct3D 的方式并为开发者 benchmark 所有的参数，从而揭示有性能优化潜力的地方。

Visual Studio 图形分析器窗口用来检查开发者已经抓获的帧的渲染和性能问题。它内置了好几个工具来帮助开发者了解应用程序的渲染行为。每中工具都揭示了捕获帧的不同的信息，每个工具都能从帧缓存开始很直观地展示渲染的问题。
下图显示了图形分析器的工具布局。



2.1.2 图形诊断工具兼容性

图形诊断程序支持使用 Direct3D 12、 Direct3D 11 和 Direct3D 10 的应用程序，部分支持使用 Direct2D 的程序。它不支持使用早期版本的 Direct3D、 DirectDraw 或其他图形 API 的应用程序。

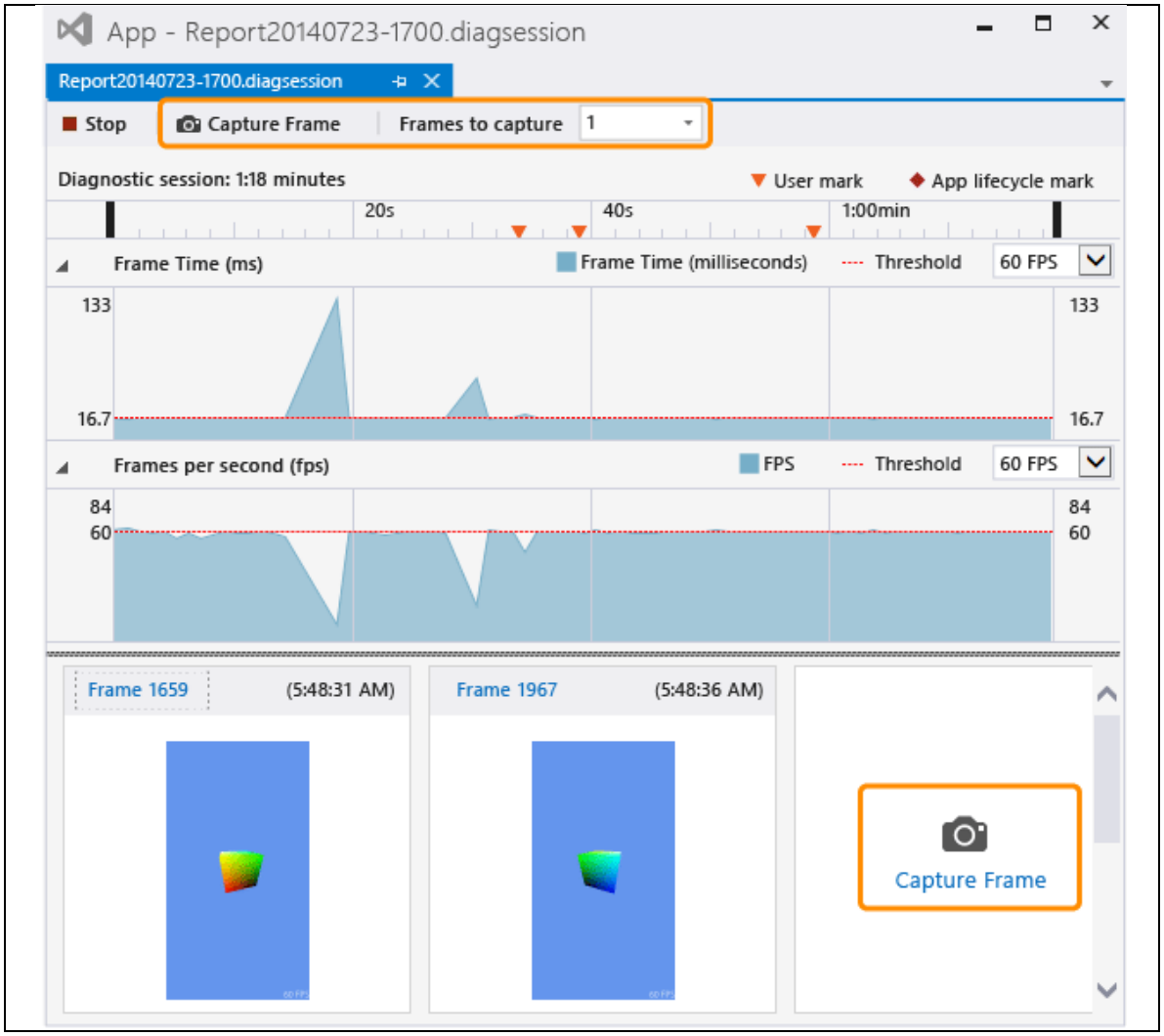
2.1.3 Visual Studio 的图形诊断功能

1. Graphic Toolbar

图形工具栏提供了快速访问图形诊断程序的命令

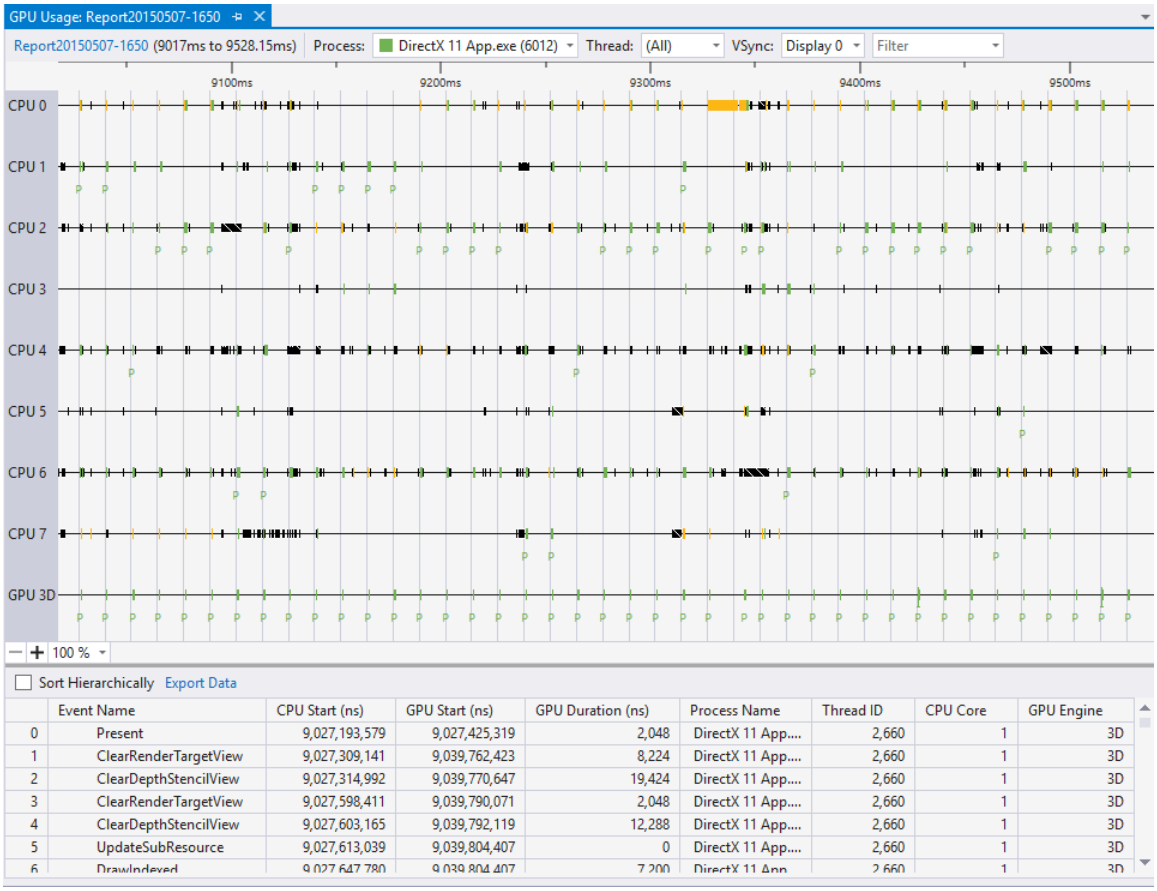
2. Capturing Graphics Information

当应用程序运行在图形诊断工具中时，Visual Studio 将显示一个诊断会话界面。开发者可以使用它来捕获的当前帧并显示帧率和每帧花费时间（GPU 和 CPU 使用率只有在启动 GPU Usage 工具才能看到）。负载显示可以帮助开发者根据画面的性能来识别想要捕获的帧，不建议使用在画面查错上。



3. GPU usage

使用 GPU 使用率 工具，能更好地理解 Direct3D 应用程序的在 GPU 和 CPU 上的性能表现。开发者可以使用它来确定应用程序的性能是否已经到达 CPU 或 GPU 的限制，帮助开发者深入了解如何更有效地使用该平台的硬件。GPU 使用率工具支持使用 Direct3D 12、 Direct3D 11 和 Direct3D 10 的程序 (VS2015 RTM 目前还不支持 DirectX12，会在之后的更新中增加);它不支持其他图形 API，比如：Direct2D 或 OpenGL 。



	Event Name	CPU Start (ns)	GPU Start (ns)	GPU Duration (ns)	Process Name	Thread ID	CPU Core	GPU Engine
0	Present	9,027,193,579	9,027,425,319	2,048	DirectX 11 App....	2,660	1	3D
1	ClearRenderTargetView	9,027,309,141	9,039,762,423	8,224	DirectX 11 App....	2,660	1	3D
2	ClearDepthStencilView	9,027,314,992	9,039,770,647	19,424	DirectX 11 App....	2,660	1	3D
3	ClearRenderTargetView	9,027,598,411	9,039,790,071	2,048	DirectX 11 App....	2,660	1	3D
4	ClearDepthStencilView	9,027,603,165	9,039,792,119	12,288	DirectX 11 App....	2,660	1	3D
5	UpdateSubResource	9,027,613,039	9,039,804,407	0	DirectX 11 App....	2,660	1	3D
6	DrawIndexed	9,027,647,780	9,039,804,407	7,200	DirectX 11 App....	2,660	1	3D

4. DirectX control panel

DirectX 控制面板是 DirectX 组件，用来帮助开发者更改 DirectX 的行为方式。 例如，开发者可以启用调试版本的 DirectX Runtime 组件，选择调试消息的种类，并禁止某些图形硬件功能被用来模拟不被支持的硬件类型。这一级别的控制可以帮助您调试并测试您的 DirectX 应用程序。你可以从 Visual Studio 中打开 DirectX 控制面板。

2.1.4 参考资源

本章节相关内容的详细使用说明，可以参考如下 MSDN 网址：
https://msdn.microsoft.com/zh-cn/library/hh315751(v=vs.140).aspx

关于 Visual Studio 2015 在 DirectX 开发中的新功能，可以参考如下视频：
https://channel9.msdn.com/Series/ConnectOn-Demand/212

第 3 章从 Direct X 11 移植到 DirectX 12

3.1 接口映射

如果你上层的渲染逻辑是基于 DirectX11 写的，那么构建一个与 DX11 完全兼容的接口层将是一个最好的移植途径，因为上层逻辑不会因为适配 DX12 而需要做大量的代码重构。虽然这种移植是很快速的，在我们的实践中，总共花费了大概 6 个星期左右就完成了绝大多数功能的移植与测试工作。但是它也有一定的弊端，因为很多 DX11 的渲染对象在 DX12 中已经被整合或者移除了，所以 DX12 的封装类里就要做很多运行时的状态转换，这些操作会消耗一定的 CPU 时间，而且你无法彻底移除它们，因此如果你有足够的开发时间，建议还是专门抽象出更亲近 DX12 的图形接口，反过来去适配 DX11 的功能，毕竟未来的趋势还是以 DX12 为主的。另外很多现代的图形 API 都与 DX12 非常相像，例如 Mantle，Metal，Vulkan 等，因此移植这些 API 也会是相当高效的。

为了适配 DX11 的 API，我们重新实现了 D3D11.h 文件里的几乎所有的接口，以下是部分代码样例。例如：

表 3.1 接口映射

```
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;
```

这里要注意的是，工程里不能包含 D3D11 的头文件，否则会发生定义冲突。

3.2 Pipeline State Object

Pipeline State Object 是 D3D12 的核心概念，它由 Shader，RasterizerState，BlendState，DepthStencilState 和 InputLayout 等数据组成，一旦 PSO 对象被投递到系统中，那么 PSO 所关联的这些状态会被同时设置。但是在 D3D11 的接口层这些渲染参数是分别使用不同的 API 进行设置的，所以我们要做适配就必须使用一个运行时的可查询容器去管理它们。最常见的对象容器是 HashMap，通过它能够避免产生冗余的 PSO，以及对应的 API 调用。

在使用 HashMap 之前，我们要先准备资源的 ID，大家可能首先想到的是资源的内存地址，它在整个应用程序的生命周期内，是全局唯一的，但它有一个弊端，就是占用了较多的内存空间，尤其在 64 位系统上，它消耗了 8 个字节。经过实践分析，大多数的应用并没有使用到这么庞大的对象量，因此我们可以通过循序号的方式来缩小资源对象的表示空间，也就是使用一个按顺序累加的整形值来表示一个资源对象。这个整形值还可以按照资源的类型分别处理，允许同一个数字表示不同类型的资源，例如 RasterizerState 和 BlendState 就可以使用不同的资源计数器。这种管理方式的一个重要的好处是让资源的编码空间变得更加的紧凑，方便生成更加短小的 Hash 值。否则如果使用内存地址拼接后生成的 Hash 值，Hash 值所占用的内存字节数就会变得很大，不仅影响了 PSO 的存储，也影响了查询的速度。定义计数器的上限需要在实践中总结，不同的项目可能会有较大的差别，不

过我们可以先使用的一个较大值来做测试，并且在分配顺序号的地方加上断言，一旦超过上限，系统就会给出报警，然后再判断究竟是修改底层实现，还是调整上层的逻辑。

为了进一步的减少 PSO 的实例个数，我们在生成 RasterizerState，BlendState，DepthStencilState 的时候，要观察它们之间的状态依赖，例如当我们在 DepthStencilState 中关闭深度测试时，RasterizerState 里的关于深度偏置的设置项就可以忽略，如果没有应用统一的默认值，就会可能产生冗余的对象。

RTV 和 DSV 也与 PSO 有关，因为 DSV 可以控制是否对深度图中的 Depth 或 Stencil 进行读写，当深度测试关闭时，需要设置一个只读的 DSV 到系统中。DSV 有三种只读方式，1.Depth Read Only;2.Stencil Read Only;3.Depth And Stencil Read Only。另外 PSO 还需要 RTV 和 DSV 的 Format 信息，所以最好把 OMSetRenderTargets 这个操作延迟到 PSO 设置时。

ScissorEnable 这个属性已经从 RasterizerState 中移除，Scissor 测试在硬件端会处于一直开启的状态，不能手动关闭，所以应用程序如果需要控制 Scissor 测试的关闭，就把 ScissorRect 的宽高设置成一个硬件允许的最大分辨率，例如 16k。

Primitive 的主拓扑类型需要在 PSO 中设置，它包括 Point，Line，Triangle 和 Patch，我们可以在调用 IASetPrimitiveTopology 时使用预先构造的转换表，直接换算成以上的主拓扑类型。PSO 的 HashMap 也可以按照主拓扑类型进行分类，每一个拓扑类型对应一个 HashMap，然后利用数组下标直接定位。

3.3 资源的绑定

在了解资源绑定之前，我们要先理解一个核心概念，它就是 RootSignature。D3D12 与 D3D11 的资源绑定模型存在较大的差异，D3D11 的资源绑定是固化的，运行时给每一个 Shader 安排一定数量的资源 Slot，应用程序只需要调用对应的接口就能够把资源绑定到 Shader 上。在 D3D12 中，资源绑定流程很灵活，没有限定资源以何种方式或何种数量进行绑定，你可以自行组织资源的绑定风格。最常用的绑定方式有两种，一个是 Descriptor Table 的绑定方式，另一个则是 Root Descriptor 的绑定方式。Descriptor Table 的方式相对比较复杂，它是将一组资源的 Descriptor 事先放置在 Descriptor Heap 上，当 DrawCall 需要引用这些资源时，只需要设置一个首句柄就可以了，Shader 会根据这个句柄找到所有后续的 Descriptor。这种方式有点像指针数组，也就是 Shader 需要进行二次寻址才能定位到最终的资源。而 Root Descriptor 的好处是不需要事先把 Descriptor 放置在 Descriptor Heap 上，而是将资源的 GPU 地址设置到 Command List 中，这相当于直接在 Command List 动态构建一个 Descriptor，让 Shader 只通过一次寻址就能定位到资源。但是 Root Descriptor 会占用比 Descriptor Table 多一倍的参数空间。由于 RootSignature 最大尺寸是有限的，所以合理安排 Root Descriptor 和 Descriptor Table 的比例很重要。

一般情况下我们把 SRV 以及 UAV 安排在 Descriptor Table 中，Sampler 则只能存在于 Descriptor Table 中，而把 CBV 放置在 Root Descriptor 里。因为 CBV 大多数使用的资源都是动态的，因此它的地址会经常发生变化，如果用 Descriptor Table 就有可能引起组合爆炸，不仅内存的占有量陡增，而且管理起来比较麻烦。相较之下 Sampler，SRV 和 UAV 的组合变化会比 CBV 少很多，尤其是 Sampler，只要上层渲染逻辑设计得当，Sampler 的组合都会在 128 个以内，所以将它们直接放到 Descriptor Heap 中比较合适。这里为了能够重用 Descriptor Heap 中的 Descriptor 组合，我们就要用到与 PSO 类似的对象管理技术，首先要给每一个 Sampler，SRV 以及 UAV 进行编号，然后按照 Shader 的需求把它们拼接生成唯一的 Hash 值，用来创建和索引 Descriptor Heap 中的 Descriptor 组合。由于 Sampler 在 Shader 中最大的使用数量是 16 个，所以每一个 Sampler 组合可以以 16 为单位的跨度来放置。SRV 和 UAV 也可以使用 Sampler 的方法进行管理，因此最好 Shader 对它们的引用上限也是

16 个，当然可变的组合跨度单位也是一种选择，只是不太方便对它们进行跨帧复用，因为当 SRV 所指向的纹理释放时，它的顺序号会被系统回收，而所有引用它的 Descriptor 组合也会被标记成已删除。这时如果 Descriptor Heap 上的组合块是大小不一且不连续的话，就会像内存池碎片一样很难被重新分配，除非进行耗时的反碎片处理，因此使用固定长度的 Descriptor 组合跨度是一种折中的选择。

设置到 Command List 里的 Descriptor Heap 最多只能有两个，每种类型的 Descriptor Heap 各一个。Sampler 和 SRV/UAV/CBV 属于两种不同类型的 Descriptor Heap，它们之间不能混用。

基于效率的考量，当需要改写 Descriptor Heap 的 Descriptor 时，我们可以先在一个 CPU 可见的 Descriptor Heap 中完成更新，然后再通过 CopyDescriptors*命令将这个 Heap 的内容拷贝到 GPU 可见的 Descriptor Heap。

3.4 资源的管理

3.4.1 静态资源

在 D3D11 中，静态的资源有两种初始化方式，第一种应用在 Immutable 的资源上，它只允许应用程序对这个资源内的数据变更一次，把需要初始化的数据通过 Create*的接口传入系统中。第二种方式对应是 Default 的资源，它可以多次变更资源内的数据，但是要通过另外一个 Staging 的资源协助完成。

在 D3D12 中，这两种资源的初始化流程被合并成一个，也就是第二种，通过一个 Upload Heap 中的资源把数据更新到 Default Heap 中。跟 D3D11 一样，凡是从 Default Heap 中分配出来的资源，都不能 Map，也就是不能直接访问它的 CPU 地址，因此需要一个从 Upload Heap 中分配的中间资源作为桥梁，把数据从 CPU 端推送到 GPU 端。这里有一个需要注意的问题，就是何时去删除这个中间资源，在 D3D11 上，中间资源可以在执行完 Copy 命令后直接删除，但是 D3D12 上却不能这样做，因为 D3D12 不提供运行时的资源生命周期管理功能，所有的工作都必须由应用程序来完成，所以应用程序需要知道那些异步执行的 Copy 任务是否已经结束，换句话说就是何时 GPU 不再引用这些资源。通过 Command Queue 的 Fence 功能我们能够方便的获取这些信息。另外我们也能通过一个共享的动态资源内存池来完成资源上传的工作，毕竟针对每一个 Default Heap 的资源都分配一个对应的 Upload Heap 资源相对比较低效，不仅复用率很低，还容易让系统产生过多的碎片。因此使用后面介绍的动态资源内存池技术，能够避免上述问题的产生。

每次应用资源到 Command List 之前，先记录下当前的帧号，这个帧号可以用来在资源释放时判断是否可以直接删除，条件是与当前帧的间隔帧数超过总 Command List 的数量，或是要把它缓冲起来，放到 Command List 的延迟释放链表里，当这个 Command List 执行完毕时再统一释放它们。

3.4.2 动态资源

在 D3D11 中，我们对 Dynamic Usage 的资源应该十分的熟悉，它广泛的使用在 Vertex Buffer，Index Buffer，以及 Constant Buffer 上，相关的应用场景有粒子和界面等。通常 Map 函数会提供一个 Write Discard 的功能，它可以让应用程序反复使用同一个资源，前提是这个资源的初始尺寸满足渲染逻辑的需求，根据先前的介绍，我们知道 D3D 的 API 都是异步执行的，也就是说当 API 调用结束时，并不意味着这个任务也同时被执行完毕了，而很可能离最后的完成还有一段时间，这时候如果后续的 DrawCall 又修改了该资源，就有一定的概率引起资源的竞争，当然如果你使用了 Write Discard 的特性，就能够避免这种情况的发生。因为运行时或驱动会自动对资源做 Rename 的处理，让外部看起来引用了同一个对象，但实际上内部已经切换成另外一个空闲的资源，这个新的资源会

接替旧资源给外部进行更新。为了避免长时间占用大量的内存，这些旧资源系统将其放入内存池中统一进行管理，当它们不再被 GPU 引用时，系统会重新回收再利用。

在 D3D12 中，我们必须实现类似的这种功能。首先要建立一个资源池，这个资源池由一个资源列表构成。由于每一次我们可能请求的资源大小不一样，所以最好事先分配一个较大的资源，然后通过不同的偏移来划分子资源供上层逻辑使用，这样做的好处是减少了系统分配的次数，而且也减少了由于内存的不连续性导致产生过多的碎片。一般情况下，建议使用 4MB 为单位的资源块。当准备好了资源池，我们就可以开始分配资源了，但是在此之前，我们还需要知道资源所处的内存地址，在 D3D11 里，是通过 Map 函数得到内存地址的，D3D12 也是通过这个函数来返回，不同的是 D3D12 不像 D3D11 那样需要每次 Map 且填充完数据后必须调用一次 Unmap 函数，由于 D3D12 的动态资源的 Map 都是持续性的，也就是说它的内存地址会一直有效，不需要通过 Unmap 函数通知系统解除对资源的映射。所以一般情况下，一个动态资源的生命周期里只要求调用一次 Map 函数，你可以把它返回的内存地址保存起来，以后反复使用。当这个资源释放之前，则需要调用一次 Unmap 函数，保证这块内存地址空间可以被系统回收。

回收已占用资源的方法也很简单，把那些当前帧所分配的资源块放入以当前 Command List 为编号的一个未决队列中，每帧检测这个队列所对应的 Command List 是否执行完毕，如果已经完成，就可以把这个队列里面的所有资源全部链接到 Free List 中供后续的分配重用。以上的方法适用于那些逐帧更新并使用的数据，对于那些可能若干帧才更新一次的跨帧引用的资源就要换另一种方法进行维护。我们在每个资源被引用前，记录当前的 Command List 编号，当它再次被 Rename 时，先检查这个编号对应的 Command List 是否已经执行完毕，如果尚未完成就把它投入当前 Command List 的待回收链表中，等待该 Command List 完成后再回收利用它。由于这种方法不是每帧都丢弃资源，所以可以保证资源的跨帧使用，只是说它需要每次 Rename 时都要检查上一次的资源使用状况。

由于动态的 Buffer，GPU 地址会随着每一次的请求而发生变化，所以外部渲染逻辑最好把 Buffer 的请求放在资源设置到 Command List 之前，否则就需要把资源的设置延迟到 DrawCall 调用时。

特别提醒：Upload Heap 中分配的资源 Map 出来的内存空间，CPU 端逻辑不要对其进行读操作，否则会造成极大的性能损失，因为这块内存属于 Write-Combine 的访问模式。

3.4.3 动态 Texture 的更新

在 D3D11 中，动态 Texture 的更新与动态 Buffer 的更新方式基本一致，直接 Map 后取出内存地址，然后进行数据填充，只是比 Buffer 多要考虑 Row Pitch 和 Depth Pitch 的跨度值。但是在 D3D12 中，就不能像 D3D11 那样对 Texture 进行填充，因为 Texture 在 GPU 中是以 Swizzle 的方式进行存储的，而 Buffer 是 Linear 的内存布局，所以在 CPU 端对 Buffer 的填充可以直接处理，不需要转换。而 Texture 为了 GPU 读取效率的考量就要换一种方式进行上传。第一步跟 Buffer 一样，先分配一个合适大小的 Upload Heap 资源，根据 GetCopyableFootprints 这个 API，我们可以从中得知上传到 Default Heap 的 Texture 在 Upload Heap 中被允许的空间布局，待数据在 CPU 端填充完毕后，再使用 Copy*的命令把数据上传上去。从上述描述可知其实 GPU 使用的 Texture 还是一个静态的资源。我们反观 D3D11 的实现，那个可以 Map 的 Texture 资源，内部也是做了类似 D3D12 的处理流程，但 D3D12 把这些工作都显性化了，也就给了应用程序更多的优化可能。

3.4.4 回读 GPU 数据

在 D3D11 中，回读 GPU 数据有两种类型，一种是回读 Buffer 和 Texture 的 GPU 数据，它是通过 Staging 资源进行处理的，首先将需要回读的静态资源拷贝到 Staging 资源上，然后使用 Map 函数返回一个 CPU 可以读取的地址，但在真正开始读取数据之前，还要判断这个资源是否已经回读完毕

了，因为 Copy 操作是异步的。在 D3D12 中，也是类似的流程，跟之前的 Upload Heap 不一样，D3D12 提供了一个专门用作回读数据的 Heap 类型，ReadBack Heap 的用法跟 Upload Heap 差不多，也是先从 Heap 中分配出一个资源，然后通过 Copy*函数将需要回读的 Default Heap 的资源拷贝到这个资源上，与 D3D11 不同的是它的 Map 函数不提供等待以及检查是否已经回读完成的功能，这时我们就需要把之前在静态和动态资源管理章节提到的机制应用在回读操作上，通过 Fence 的方式来判断这次回读是否已经完成。而且 ReadBack Heap 分配出来的资源也是可以做持续性 Map 的，整个生命周期只需要一次 Map 和一次 Unmap。

另外一种回读类型是硬件 Query 的回读。它跟 Buffer 和 Texture 的回读流程基本一致，只是把 Copy 函数换成了 Resolve 函数而已。由于 Resolve 函数可以批量回读 Query 数据，所以在分配 Query Heap 时，不必每一个 Query 对象都调用一次创建函数，而是批量分配内存空间连续的 Query 对象集合，然后通过资源内偏移的方式进行后续的定位。

由于我们使用的是 D3D11 的接口封装，所以对于外部来说上传和回读操作都可能用到了 Staging 的资源，那么内部该如何区分它们呢？首先要判断第一次使用该资源时 Map 操作发生在 Copy*命令之前还是之后，一般情况下，Map 操作发生在 Copy*命令之前的行为，我们认为是用户希望上传数据到 GPU 中，而发生在之后则是想从 GPU 中回读数据。当然这里有一个前提条件，就是对于同一个资源，不允许外部逻辑即用它来上传 CPU 数据又用它来回读 GPU 数据，否则这种歧义性内部将无法识别，幸运的是实践中这类情况很少发生。

3.5 Resource Barrier

这是一个全新的概念，在 D3D12 之前这个工作是由驱动来承担的，现在 D3D12 把它从驱动层剥离了出来，让应用程序来控制何时进行开启它们。

Resource Barrier 有三种不同的类型，最常用的类型是 Transition，主要用在资源状态的切换上。当资源的应用场景发生了变化，我们就要在这个资源被使用前放置一个对应的 Resource Barrier。
实践中很常见的 Transition Barrier 是一个资源在 RenderTargetView 和 ShaderResourceView 之间来回切换。所以我们需要在资源的封装类里加入一个成员变量，用来记录当前的资源状态，当上层逻辑调用 OMSetRenderTargets 时，要先检查当前的状态是不是 RenderTargetView，如果不是就放置一个 Barrier，它的 StateBefore 填写的是成员变量里存放的状态值，StateAfter 里填写的是 RenderTargetView 状态。如果渲染逻辑调用了 XXSetShaderResources，那么依照以上流程可以做一个类似的处理，只是把 StateAfter 里填成 ShaderResourceView 状态而已。

Transition Barrier 最好延迟到真正开始使用资源前再进行设置，这样可以避免没有必要的同步，因为它会阻塞后续命令的执行。
Copy*，Resolve*和 Clear*等命令传入的资源都有对应的目标状态需要设置。

3.6 Command List/Queue

D3D12 的 Command List/Queues 是从 D3D11 的 DeviceContext 中脱胎出来的功能，Command List 负责将渲染命令缓冲起来，然后构建成驱动认识的硬件指令，最后由 Command Queue 去执行。因为每一个 Command List 可以独立的填充渲染命令，中间没有任何的锁保护，所以执行速度较之 D3D11 要更加的快。

Command List 可以被重复使用，当需要再次使用时，必须首先确认是否 Command List 中的命令已经被 GPU 执行完毕，如果 GPU 尚未执行完毕，而去强行提交这个 Command List，那么后续的 GPU

行为将不可预期。一个被 Reset 的 Command List 相当于一个空白的 Context，它不再继承之前的任何渲染状态，所以你需要重新设置它们，例如 PSO，Viewport，ScissorRect，RTV 和 DSV 等。

一般情况下，为了避免每帧去同步等待上一帧正在执行的 Command List，于是我们可以准备若干个 Command List 备用，然后每帧结束时都去检查前面未决的 Command List，如果最近一个 Command List 已经执行完毕，那么就说明之前的 Command List 也执行完毕了，因为 Command List 是严格顺序执行的，相当于一个 FIFO 的队列。要判断 Command List 是否执行完毕，我们需要使用 Fence 这种对象，当 Command Queue 调用完 ExectueCommandList 这个函数后，Signal 函数可以让系统在 Command List 被执行完毕时立即通知 Fence 对象，通知的形式是把传入 Signal 函数的那个期待值设置到 Fence 对象中。因此通常情况下，我们会把每帧累加的帧号当成期待值设置给 Signal 函数。查询 Command List 是否完成的方式就是判断 GetCompletedValue 的返回值是否等于期待值。

Command Queue 会和 SwapChain 绑定在一起，所以在创建 SwapChain 时传入的第一个参数就是 Command Queue。目前 SwapChain 的常用模式是 Flip*，在这种模式下，你需要让 BufferCount 大于一，也就是 SwapChain 中会有超过一个的 BackBuffer。为了交替渲染他们，你需要在 Present 之后，把下一个 BackBuffer 切换成当前的 RenderTarget，根据你创建 BackBuffer 的总数自动回绕。如果某一帧你没有执行 Present 操作，那么你 BackBuffer 也不能进行切换，否则会导致系统崩溃。由于 Flip*会产生帧同步，所以最终你渲染的 FPS 会受到限制，为了移除这种限制，你可以使用 GetFrameLatencyWaitableObject 返回的内核对象进行判断，如果该对象不是通知状态，那么你可以直接跳过 Present 函数，这样就可以不受帧同步的影响了。

Command Queue 有三种类型，Direct，Copy 以及 Compute。这三种类型的 Command Queue 之间可以并行执行。
Direct 类型的 Command Queue 负责处理 Graphics 的渲染命令。
Copy 类型的 Command Queue 负责数据上传或者回读操作。
Compute 类型的 Command Queue 负责通用目(跟光栅化无关)的计算的命令处理。

例如，当我们使用 Direct 类型的 Command Queue 在某个工作线程渲染场景时，与此同时可以在另外一个线程使用 Copy 类型的 Command Queue 处理 Texture 数据的上传。在 Direct 类型的 Command Queue 对这些后台上传的 Texture 的引用操作必须等待 Copy 类型的 Command List 执行完毕后再去使用。

第 4 章 DirectX12 特性

4.1 Multiplane Overlay 的功能和用法

4.1.1 介绍

Multiplane Overlays（MPO） 最初是随 Win8.1 推出的 WDDM1.3（DX11.2）的新特性，该特性扩展到了 Win10 的 WDDM2.0（DX12）。MPO 支持使用原始分辨率显示华丽的 2D 艺术和 UI 元素，而把 3D 场景绘制到一个更小的、可拉伸的帧缓冲中，两种不同分辨率的表面的拉伸和合成，则由系统自动实现，对应用程序是透明的。

MPO 的主要作用是使游戏在不同情况下能维持稳定的、适当的帧率，从而改善整个游戏体验。一方面，分辨率对现代游戏的性能几乎总是会产生明显影响。随着高清、4K 屏幕的普及，游戏窗口的分辨率越来越高，像素增加的同时也增加了纹理采样和渲染目标带宽，对游戏的性能带来了挑战；其次，现在 3D 场景的渲染越来越多使用后处理技术，这增加了 shader 的复杂性，使每个像素的代价决定现代游戏性能成为一个趋势。因此，设置合适的分辨率对保持游戏性能很关键。

另一方面，对角色扮演、实时策略、和多人在线等游戏来说，以窗口初始分辨率渲染 GUI 组件也很重要。例如，即使在低端平台上，玩家也会希望和队友进行文字聊天。GUI 组件主要是 2D 图形，渲染开销相对较低，以窗口分辨率渲染可以提供最佳视觉体验。

MPO 为高分辨率显示和游戏性能提供了一个折中的平衡方案， 可缓解游戏在某些高分辨率渲染中性能下降太快，使游戏在广大硬件平台上流畅运行。

由于是驱动模型的新功能，MPO 的实现是由图形驱动提供给上层 D3D Runtime 或 DXGI runtime 的。它可以是 WDDM1.3 及以上版本的驱动通过软件来实现，也可以通过图形硬件来实现。后者使用 GPU 的固定功能管线，能减少很多额外的 CPU 和 GPU 资源的消耗，有利于进一步提升游戏性能和降低功耗。Intel 将在 SkyLake 及以后的处理器核显上增加对 MPO 的硬件支持。

4.1.2 应用场合

在游戏中，Multiplane Overlays 可在以下典型场合中应用。

1. 性能（如帧率）低于某个阈值的场景。

根据游戏的类型，当帧率低于某个值时，游戏的可玩性会大打折扣。比如大量游戏角色出现的集市、战斗场景中，游戏的帧率下降会很明显。为了避免性能下降到不可玩的程度，游戏在检测到帧率低于阈值时，可自动切换到 MPO 渲染模式，维持可玩的性能。

2. 释放粒子特效的场景。

在 MMOG 中，释放大量粒子特效的场景经常使帧率突降，严重时甚至导致游戏卡顿，是游戏中常见的性能瓶颈。粒子特效的渲染涉及大量像素填充，而在高分辨率的表面上渲染时对性能的影响更加显著。游戏中，可以在释放粒子特效时切换到 MPO 模式，把粒子特效渲染到一个分辨率较小的渲染目标上，有效降低像素填充量。另外，在激烈的战斗场景中，大量粒子特效的释放使画面的变化非常剧烈，玩家很难看清每帧的细节。因此 MPO 对渲染目标的拉伸对最终的视觉质量的影响几乎无法察觉。在这类场景中使用 MPO，能在保持视觉体验的同时维持适当的帧率，解决游戏中常见的性能问题。

3. 移动平台上的渲染

随着移动平台作为游戏终端日益普及，功耗变得和游戏开发相关了。当平台从 AC 电源模式切换到电池模式时，系统性能设置能引起机器 CPU、GPU 频率的降低，使游戏帧率下降。利用 MPO 方法，

游戏能在不改变窗口分辨率的情况下，自动调整渲染目标的分辨率，对整体性能进行补偿，同时可以降低功耗，使游戏运行更长的时间。MPO 的这种特性可用于实现游戏的省电模式。该模式作为一种游戏设置选项提供给玩家。对玩家而言，在省电模式下即使画面有轻微差异，也是可以接受的。

4.1.3 API 调用示例

在游戏中应用 Multiplane Overlays（MPO），首先要检测平台的软、硬件是否支持。检测代码如下：

表 4.1 检测平台是否硬件支持 Multiplane Overlay

```
BOOL supportMultiPlaneOverlay = FALSE;
IDXGIOutput* dxgiOutput;
IDXGIFactory * pDXGIFactory;
IDXGIAdapter adapter;

HRESULT hr = CreateDXGIFactory(__uuidof(IDXGIFactory),
                               (void**)(&pDXGIFactory) );

pDXGIFactory->EnumAdapters(0, &adapter);

adapter->EnumOutputs(0, &dxgiOutput);
if (dxgiOutput)
{
    IDXGIOutput2* dxgiOutput2;
    dxgiOutput->QueryInterface(IID_PPV_ARGS(&dxgiOutput2));
    SAFE_RELEASE(dxgiOutput);
    if (dxgiOutput2)
    {
        supportMultiPlaneOverlay = dxgiOutput2->SupportsOverlays();
        SAFE_RELEASE(dxgiOutput2);
    }
}
```

IDXGIOutput2::SupportsOverlays()这个 API 用于检测显卡硬件是否支持该特性。如果返回 true，说明硬件支持；如果返回 false，说明硬件不支持，但驱动以软件方式支持。Intel Skylake 平台及以后的处理器核显将从硬件上支持 MPO。

在游戏初始化阶段，要创建几个关键的 Multiplane Overlay API 对象：如 Direct Composition 的设备，缩放转换器，前景、背景 SwapChain 等。示例代码如下：

表 4.2 初始化创建 Multiplane Overlay API 对象

```
IDXGIFactory2* dxgiFactory;
adapter->GetParent(IID_PPV_ARGS(&dxgiFactory));

if (dxgiFactory)
{
    //创建一个 Direct Composition 的设备
    DCompositionCreateDevice(NULL, IID_PPV_ARGS(&m_directCompositionDevice));

     // 设备为窗口创建一个渲染目标
    m_directCompositionDevice->CreateTargetForHwnd(
   (HWND)_CORE_API->GetAppWindow(), false, &m_directCompositionTarget);

    // 设备创建多层视图
    m_directCompositionDevice->CreateVisual(&m_rootVisual);
    m_directCompositionDevice->CreateVisual(&m_mainVisual);
    m_directCompositionDevice->CreateVisual(&m_overlayVisual);

    //设备创建一个缩放变换控制器，并设置到主视图中
    m_directCompositionDevice->CreateScaleTransform(&m_mainScaleTransform);
    m_mainVisual->SetTransform(m_mainScaleTransform);
```

```
    //当主视图被拉伸时，采用线性插值滤波                m_mainVisual-
>SetBitmapInterpolationMode(DCOMPOSITION_BITMAP_INTERPOLATION_MODE_LINEAR);

    //将主视图和界面视图添加到根视图中，把根视图设置到渲染目标上
     m_rootVisual->AddVisual(m_mainVisual, FALSE, NULL);
     m_rootVisual->AddVisual(m_overlayVisual, FALSE, NULL);
    m_directCompositionTarget->SetRoot(m_rootVisual);

    //准备创建 SwapChain
    DXGI_SWAP_CHAIN_DESC1 swapChainDesc = { 0 };
       swapChainDesc.Width = width; // Match the size of the window.
       swapChainDesc.Height = height;
       swapChainDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
       swapChainDesc.Stereo = false;
       swapChainDesc.SampleDesc.Count = 1; // Don't use multi-sampling.
       swapChainDesc.SampleDesc.Quality = 0;
       swapChainDesc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
       swapChainDesc.BufferCount = 2;
       swapChainDesc.SwapEffect = DXGI_SWAP_EFFECT_FLIP_DISCARD;
       swapChainDesc.Flags = DXGI_SWAP_CHAIN_FLAG_FRAME_LATENCY_WAITABLE_OBJECT;
       swapChainDesc.Scaling = DXGI_SCALING_STRETCH;
       swapChainDesc.AlphaMode = DXGI_ALPHA_MODE_PREMULTIPLIED;

    //DXGIFactory 创建前景 SwapChain
     hr = dxgiFactory->CreateSwapChainForComposition(
            m_graphicsCommandQueue,
            &swapChainDesc,
            nullptr,
            &m_foregroundSwapChain
            );
    //把界面视图绑定到前景 SwapChain 上
     m_overlayVisual->SetContent(m_foregroundSwapChain);

    //创建背景 SwapChain
     swapChainDesc.AlphaMode = DXGI_ALPHA_MODE_IGNORE;

     hr = dxgiFactory->CreateSwapChainForComposition(
            m_graphicsCommandQueue,
            &swapChainDesc,
            nullptr,
            &m_backgroundSwapChain
            );

       //把主视图绑定到背景 SwapChain 上
     m_mainVisual->SetContent(m_backgroundSwapChain);

     m_directCompositionDevice->Commit();
     SAFE_RELEASE(dxgiFactory);
}
```

注意：D3D12 与 D3D11 区别是 CreateSwapChainForComposition 传入的第一个参数是 CommandQueue 而不是 Device。

在游戏渲染过程中，基于 MPO 渲染的代码逻辑如下：

表 4.3 基于 Multiplane Overlay 的游戏渲染方法

```
//根据帧数的变化，动态调整后台 SwapChain 的缩放比例
m_mainScaleTransform->SetScaleX(scaleRatio);
m_mainScaleTransform->SetScaleY(scaleRatio);
m_directCompositionDevice->Commit();
```

```
//渲染 3D 场景前，先设置后台 SwapChain 对应的 RenderTarget
//同时调整 Viewport 大小
OMSetRenderTargets(1, &m_backgroundRTV, m_backgroundDSV);
viewport.Width = foregroundSwapChain.Width / scaleRatio;
viewport.Height = foregroundSwapChain.Height / scaleRatio;
RSSetViewports(1, viewport);

// Draw 3D Scene...

// 当进行全屏后处理时,需要改变纹理采样坐标，控制寻址范围，因为后台的 Viewport 可能让
// RenderTarget 只填充了局部
VSSetConstantBuffer(scaleRatio);
// Draw Fullscreen PostProcess

//在渲染 UI 前，先设置前台 SwapChain 对应的 RenderTarget
//同时调整 Viewport 大小
OMSetRenderTargets(1, &m_foregroundRTV, m_foregroundDSV);
viewport.Width = foregroundSwapChain.Width;
viewport.Height = foregroundSwapChain.Height;
RSSetViewports(1, viewport);

// Draw UI..

//最后提交 SwapChain
m_backgroundResource->SetTransitionBarrier(D3D12_RESOURCE_STATE_PRESENT);
m_foregroundResource->SetTransitionBarrier(D3D12_RESOURCE_STATE_PRESENT);

m_commandList->Close();

m_backgroundSwapChain->Present(0, 0);
m_foregroundSwapChain->Present(0, 0);
```

4.1.4 总结

Multiplane Overlays 是 Win8.1 及以后平台上的一项新的图形显示功能。基于 Win10 平台的 DirectX12 游戏，能方便地使用 DX12 API 实现 Multiplane Overlays 的渲染，解决游戏的高分辨率渲染和高负载场景中的帧率突降问题，使游戏在广大硬件平台上获得流畅、精彩的游戏体

第 5 章 DirectX12 优化

5.1 DirectX 12 多线程基础

5.1.1 介绍

图形渲染是现代 3D 游戏的主要任务之一。在 DirectX 9 中，原则上所有渲染 API 都必须在一个线程中调用。DirectX 10/11 中加强了多线程支持，但各线程的负载很不平衡，渲染相关负载主要集中在游戏的主渲染线程和图形驱动中，这使得渲染任务无法充分利用现代多核处理器的能力，经常成为游戏渲染管线的主要性能瓶颈之一。

为了提高图形渲染效率，在 DirectX 12 中，多线程得到了前所未有的支持。在重新设计的 DirectX 12 中，为了让应用程序的图形渲染可以达到最大的多核 CPU 的使用效率：一方面，DirectX 12 尽可能地预处理和复用渲染命令，降低渲染状态的切换开销，提升渲染 API 在 CPU 和 GPU 上的处理效率；另一方面，为应用程序提供了更高效的多线程渲染机制，允许应用程序最大程度地利用多任务获得性能提升。通过使用多线程手段可以使图形驱动在 CPU 端的开销降低，同时也使 GPU 的工作效率显著提升。DX12 的多线程机制除了使渲染任务能更均衡地并行运行在不同的处理器核上以提升性能，还能降低 CPU 的功耗，这对移动平台上的游戏也非常重要。

英特尔公司在 SIGGRAPH 2014 上展示了用 DirectX 11 和 DirectX 12 开发的小行星演示程序。在此程序中，用户可以运行时切换 DirectX 11 或 DirectX 12 进行渲染。在渲染的一帧中，有 50000 个小行星需要被绘制，意味着 CPU 端提交的 Draw Call 就有 50000 次；同时，由于大量不同的贴图、模型等数据的随机组合，此演示程序可以反映两代图形 API 在驱动层效率的差异。借助多线程等技术，和 DirectX 11 版本相比，DirectX 12 在帧率和功耗方面均表现出较大优势，具体可参考 DirectX 开发博客：
http://blogs.msdn.com/b/directx/archive/2014/08/13/directx-12-high-performance-and-high-power-savings.aspx

5.1.2 重要基础设施

（一）Command list 和 Command Queue

Command list 以及 Command Queue 是 DirectX 12 多线程编程中重要的基础设施。在此，我们先简单地在渲染命令方面对比 DirectX12 和 DirectX 9 以及 DirectX 11。

在 DirectX 9 中，大部分的渲染命令都是通过 Device 的接口进行调用的，例如 BeginScene、Clear、DrawIndexedPrimitive 等等；而渲染状态则由 Device 的 SetRenderState 接口负责。在 DirectX 11 中，渲染命令则大多通过对 Immediate Context 上的相关接口调用实现。然而在 DirectX 12 中，为了尽可能地对单线程进行预处理同时又提高多线程并发工作的可能，我们需要使用 Command List 这个对象。上面所述的大部分渲染命令均可通过 Command List 上的接口调用实现（各接口具体定义请参考 DirectX 12 SDK 的 d3d12.h 头文件，查看其中的 ID3D12GraphicsCommandList 接口声明）。而为了把 Command List 提交给 GPU 去执行，我们需要 Command Queue 对象。在这里，Command Queue 主要负责提交 Command List，并且同步后者的执行。以下代码演示了如何创建一个 Command List 并且通过它来记录渲染命令，最后由 Command Queue 提交这些命令。
代码如下所示：

表 5.1：Command List 和 Command Queue 的使用

```
// Command Allocator 用于负责 Command List 相关的内存分配
// 参数 D3D12_COMMAND_LIST_TYPE_DIRECT 表示此分配器用于 Command List
ComPtr<ID3D12CommandAllocator> pCommandAllocator;
pDevice->CreateCommandAllocator(D3D12_COMMAND_LIST_TYPE_DIRECT,
IID_PPV_ARGS(pCommandAllocator)));
```

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

// Command Queue 的描述
// Type = D3D12_COMMAND_LIST_TYPE_DIRECT 指定此 Command Queue 适用于提交 Command List
D3D12_COMMAND_QUEUE_DESC queueDesc = {};
queueDesc.Flags = D3D12_COMMAND_QUEUE_FLAG_NONE;
queueDesc.Type = D3D12_COMMAND_LIST_TYPE_DIRECT;

// 创建一个 Command Queue
ComPtr<ID3D12CommandQueue> pCommandQueue;
pDevice->CreateCommandQueue(&queueDesc, IID_PPV_ARGS(&pCommandQueue)));

// 通过 Command List 调用渲染相关接口
// 为了示意，这里只例举少数几个，大量接口请查看 ID3D12GraphicsCommandList 接口声明
pCommandList->ClearRenderTargetView(rtvDescriptor, clearColor, 0, nullptr);
pCommandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
pCommandList->IASetVertexBuffers(0, 1, &pVertexBufferView);
pCommandList->DrawInstanced(3, 1, 0, 0);

// 通过 Command Queue 执行 Command List，CommandQueue 可以一次提交多个 Command List
ID3D12CommandList* ppCommandLists[] = { pCommandList.Get() };
pCommandQueue->ExecuteCommandLists(_countof(ppCommandLists), ppCommandLists);
```

值得一提的是，虽然 Command List 本身并不是线程自由（Thread Free）的，多个线程不能在一个 Command List 上并行地访问其接口，但我们可以使用多个 Command List，对渲染任务进行拆分，把待提交的渲染命令根据需要分配到各个 Command List 上，最后通过 Command Queue 提交 Command List 上的渲染命令。在 DirectX 12 中，所有在 Device 上的接口都是线程自由的，Command List 上的所有操作都是非线程自由的，即单线程的，但我们可以通过准备多个 Command List，在不同的线程中独立调用各自维护的 Command List 的渲染命令接口。与此同时，Command Queue 是线程自由的，应用程序的不同线程可以在 Command Queue 上任意顺序地执行各 Command List。

（二）Bundle 和 Pipeline State Object

为了使得在单个线程中，尽可能优化驱动的效率，DirectX 12 进一步引入了第二个层次的 Command List，它就是 Bundle。此对象的作用是允许应用程序把一组 API 命令事先创建（“录制”），以便之后重复使用。而在创建 Bundle 时，显示驱动可以尽可能对这些命令进行预处理，以便的对这组 API 命令最大程度地优化。一直以来，对于渲染状态的更新和维护是图形驱动性能开销不小的一部分，DirectX 12 把这部分状态抽象成为 Pipeline State Object（PSO），以便更好地和现在图形硬件的状态映射起来，减小切换和管理的代价。

（三）Resource Barrier

在 DirectX 12 中，单个资源状态的管理已经由图形驱动移交给应用程序，这很大程度上减轻了驱动对资源状态的追踪维护的成本，此时我们需要使用 Resource Barrier 机制。这种所谓的“资源屏障”的使用场景很常见，比如一张贴图既可以被作为渲染时引用的贴图资源（Shader Resource View，SRV）又可以被当成一张渲染目标（Render Target View，RTV）。举例一个现实中的例子：我们需要一张阴影图（Shadow Map），所以需要把场景深度事先渲染到这张贴图资源中，此时为此资源为 RTV；而后在渲染带阴影效果的场景时，这张贴图则被当作 SRV 使用。现在，这些都需要应用程序自己使用 Resource Barrier 进行处理，告知 GPU 某一资源状态。

代码如下所示：

表 5.2：Resource Barrier 使用方法

```
// 阴影贴图从一般状态切换到深度可写状态，得以将场景深度渲染至其中
pCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(pShadowTexture,
D3D12_RESOURCE_STATE_COMMON, D3D12_RESOURCE_STATE_DEPTH_WRITE));

// 阴影贴图将作为像素着色器的 Shader Resource 使用，场景渲染时，将对阴影贴图进行采样
pCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(pShadowTexture,
D3D12_RESOURCE_STATE_DEPTH_WRITE, D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE));

// 阴影贴图恢复到一般状态
pCommandList->ResourceBarrier(1, &CD3DX12_RESOURCE_BARRIER::Transition(pShadowTexture,
D3D12_RESOURCE_STATE_PIXEL_SHADER_RESOURCE, D3D12_RESOURCE_STATE_COMMON));
```

（四）Fence

DirectX 12 引入了 Fence 对象，来实现 GPU 到 CPU 的同步。Fence 是一种无锁的同步机制，它符合 GPU 端到 CPU 端轻量的同步原语要求。基本上，通信只需要一个整型的变量即可实现。
代码如下所示：

表 5.3：创建 Fence 对象

```
// 创建一个 Fence，其中 fenceValue 为初始值
ComPtr<ID3D12Fence> pFence;
pDevice->CreateFence(fenceValue, D3D12_FENCE_FLAG_NONE, IID_PPV_ARGS(&pFence)));
```

通过 Fence 实现的同步分为两种，第一种是 CPU 端的线程查询当前 Fence 的值，从而得到 GPU 端执行任务的进度：

表 5.4：通过查询 Fence 上的值实现同步

```
pCommandQueue->Signal(pFence.Get(), fenceValue);

// 由 CPU 端查询 Fence 上的完成值（进度）
// 如果比 fenceValue 小，则调用 DoOtherWork
if (pFence->GetCompletedValue() < fenceValue)
{
        DoOtherWork();


}
```

另一种是 CPU 端线程可以要求 GPU 在 Fence 上的值达到指定值时，将此线程唤醒以达到同步的目的，配合其它 Win32 的 API，可以满足诸多同步要求。
代码示例如下：

表 5.5：通过指定 Fence 上的值实现同步

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

5.1.3 多线程渲染示例

下面，我们尝试通过一个简单的示例说明如何使用 DirectX 12 多线程，以及如何将渲染任务进行拆分，以大幅度提高渲染效率。为了便于描述以及尽可能保持简洁易懂，我们将结合伪代码，同时我们也不得不省略函数某些的参数，但这样应该不影响理解。
在我们的例子中，OnRender 是一个典型的 DirectX 12 单线程渲染函数，它的功能是渲染游戏场景的一帧。在这个函数中，我们使用 Command List 记录所有的渲染命令，包括设置后台缓冲区的资源

屏障状态，清除颜色，对每一个网格进行绘制等等，然后使用 Command Queue 执行 Command List，最后由 SwapChain 呈现整个画面。
渲染函数大致代码如下所示：

表 5.6：原始的单线程渲染函数

```
void OnRender()
{
	// 重置 Command List
	pCommandList->Reset(...);

	// 将后台缓冲区设置屏障，从待呈现状态变成渲染目标状态
	pCommandList->ResourceBarrier(1, (..., D3D12_RESOURCE_STATE_PRESENT,
D3D12_RESOURCE_STATE_RENDER_TARGET));

	// 设置渲染目标
	pCommandList->OMSetRenderTargets(...);

	// 清除渲染目标
	pCommandList->ClearRenderTargetView(...);

	// 设置图元/拓扑类型
	pCommandList->IASetPrimitiveTopology(...);

	// 其它 Command List 上的操作
	// ...

	// 绘制每一个网格
	foreach Mesh in Meshes
	{
		pCommandList->DrawInstanced(...);
	}

	// 将后台缓冲区设置屏障，从渲染目标状态变成待呈现状态
	pCommandList->ResourceBarrier(1, (..., D3D12_RESOURCE_STATE_RENDER_TARGET,
D3D12_RESOURCE_STATE_PRESENT));
	// 关闭 Command List
	pCommandList->Close();

	// 在 Command Queue 上执行 Command List
	pCommandQueue->ExecuteCommandLists(...);

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

接下去，我们将把此渲染函数并行化，采用 DirectX 12 多线程对程序进行修改。在程序初始化阶段，我们创建了若干个工作线程，用于分担处理场景中数量众多的对象的渲染命令。对于每个工作线程，我们平均分配了场景中的同等数量的网格（Mesh），同时，我们为每个工作线程创建一多个 Command List，每个 Command List 负责记录子线程的部分渲染任务。通常，每个子线程只需管理一个 Command List，这里为每个工作线程创建多个 Command List（子任务）的好处在于：当工作线程被分配的任务比很多时，不需要完成全部任务，就可以通知主线程把渲染命令提交给 GPU，提高了 CPU/GPU 的并行度。而主线程和工作线程之间使用 Win32 的信号量和等待 API 来实现同步。
主线程渲染函数代码大致如下：

表 5.7：多线程化后的主线程渲染函数

```
void OnRender_MainThread()
{
        // 通知每一个子渲染线程开始渲染
        for workerId in workerIdList
        {
                SetEvent(BeginRendering_Events[workerId]);
        }

        // Pre Command List 用于渲染准备工作
        // 重置 Pre Command List
        pPreCommandList->Reset(...);

        // 设置后台缓冲区从呈现状态到渲染目标的屏障
        pPreCommandList->ResourceBarrier(1, (..., D3D12_RESOURCE_STATE_PRESENT,
D3D12_RESOURCE_STATE_RENDER_TARGET));

        // 清除后台缓冲区颜色
        pPreCommandList->ClearRenderTargetView(...);

        // 清除后台缓冲区深度/模板
        pPreCommandList->ClearDepthStencilView(...);

        // 其它 Pre Command List 上的操作
        // ...

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

        // Post Command List 用于渲染后收尾工作
        // 设置后台缓冲区从呈现状态到渲染目标的屏障
        pPostCommandList->ResourceBarrier(1, (..., D3D12_RESOURCE_STATE_RENDER_TARGET,
D3D12_RESOURCE_STATE_PRESENT));

        // 其它 Post Command List 上的操作
        // ...

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

        // 等待所有工作线程完成任务 1
        WaitForMultipleObjects(Task1_Events);

        // 提交已完成渲染命令(Pre Command List 和所有工作线程上的用于任务 1 的 Command List)
        pCommandQueue->ExecuteCommandLists(..., pPreCommandList +
pCommandListsForTask1);

        // 等待所有工作线程完成任务 2
        WaitForMultipleObjects(Task2_Events);

        // 提交已完成渲染命令（所有工作线程上的用于任务 2 的 Command List）
        pCommandQueue->ExecuteCommandLists(..., pCommandListsForTask2);

        // ...
```

```
        // 等待所有工作线程完成任务 N
        WaitForMultipleObjects(TaskN_Events);

        // 提交已完成渲染命令（所有工作线程上的用于任务 N 的 Command List）
        pCommandQueue->ExecuteCommandLists(..., pCommandListsForTaskN);

        // 提交剩下的 Command List（pPostCommandList）
        pCommandQueue->ExecuteCommandLists(..., pPostCommandList);

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

工作线程函数代码大致如下：

表 5.8：多线程化后的子线程渲染函数

```
void OnRender_WorkerThread(workerId)
{
        // 每一次循环代表子线程一帧渲染工作
        while (running)
        {
                // 等待主线程开始一帧渲染事件通知
                WaitForSingleObject(BeginRendering_Events[workerId]);

                // 渲染子任务 1
                {
                        pCommandList1->SetGraphicsRootSignature(...);
                        pCommandList1->IASetVertexBuffers(...);
                        pCommandList1->IASetIndexBuffer(...);
                        // ...
                        pCommandList1->DrawIndexedInstanced(...);
                        pCommandList1->Close();

                        // 通知主线程当前工作线程上的渲染子任务 1 完成
                        SetEvent(Task1_Events[workerId]);
                }

                // 渲染子任务 2
                {
                        pCommandList2->SetGraphicsRootSignature(...);
                        pCommandList2->IASetVertexBuffers(...);
                        pCommandList2->IASetIndexBuffer(...);
                        // ...
                        pCommandList2->DrawIndexedInstanced(...);
                        pCommandList2->Close();

                        // 通知主线程当前工作线程上的渲染子任务 2 完成
                        SetEvent(Task2_Events[workerId]);
                }

                // 更多渲染子任务
                // ...

                // 渲染子任务 N
                {
```

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

                    // 通知主线程当前工作线程上的渲染子任务 N 完成
                    SetEvent(TaskN_Events[workerId]);
                }
        }
}
```

这样，我们成功地把任务分配给了子线程去处理，而主线程只关注如准备以及渲染后处理这样的工作。子线程只需要适时通知主线程自己的工作情况，使用多个 Command List 可以无须打断地将一帧的渲染命令处理完成。同时，主线程也可以专心处理自己的工作，在合适的情况下，等待子线程完成阶段性工作，并地将子线程中相关的 Command List 使用 Command Queue 提交给 GPU。当然只要能确保渲染顺序正确，子线程也可以通过 Command Queue 提交 Command List 上的命令。这里为了便于说明，我们把 Command Queue 提交 Command List 的操作，放在了主线程上。另外，现代 3D 游戏中，大量地使用后期处理，我们可以将后期处理这样的任务放在主线程中，或者放在一个或多个子线程中，由于篇幅有限，我们的示例代码省略了这一部分的实现。

5.1.4 总结

作为 DirectX 12 设计目标的重要组成部分，多线程是每一个应用程序值得尝试的优化方案。DirectX 12 API 提供了良好的多线程支持，通过合适的迁移，单线程可以实现并行化，充分利用硬件性能，使得渲染效率得到较大提高。