备忘录

笔记本： DirectX 12
创建时间： 2022/8/2 0:42 **更新时间：** 2022/8/2 18:58
作者： handsome小赞

- **COM**（组件对象模型 Component Object Model），通常将COM对象视为一种接口，当前暂时将它当作一个C++来使用

  > COM类接口大都以“ I ”开头

- **数据格式**一般以“ DXGI_FORMAT ”开头，也有以“D3D12”开头的

  > 例如，纹理格式（深度缓冲区也是一种纹理）

- **描述符**（view / descriptor）/ **描述符堆**（descriptor heap）

  > 1. CBV/SRV/UAV 描述符分别表示的是常量缓冲区视图（constant buffer view）、着色器资源视图（shader resource view）和无序访问视图（unordered access view）这 3 种资源。
  > 2. 采样器（sampler，亦有译为取样器）描述符表示的是采样器资源（用于纹理贴图）。
  > 3. RTV 描述符表示的是渲染目标视图资源（render target view）。
  > 4. DSV 描述符表示的是深度/模板视图资源（depth/stencil view）。

- **多重采样**

  > 在创建交换链缓冲区和深度缓冲区时都需要填写 DXGI_SAMPLE_DESC 结构体。当创建后台缓冲区和深度缓冲区时，多重采样的有关设置一定要相同[1]。
  >
  > 注意，上方描述不准确，Direct3D 12 并不支持创建MSAA交换链！

- **DirectX 图形基础结构/设施**（**DXGI**），使多种图形API种所共有得底层任务能借助一组通用API来处理。
- **显示适配器**（**display adapter**），适配器用接口 IDXGIAdapter 来表示
- **显示输出**（**display output**），用接口 IDXGIOutput 来表示
- **资源驻留**（residency）
- **命令队列 / 命令列表 / 命令分配器（command allocator)**

  > 记录在命令列表内的命令，实际上是存储在与之关联的命令分配器上。当通过 `ID3D12CommandQueue::ExecuteCommandLists` 方法执行命令列表的时候，命令列表就会引用分配器里的命令
  >
  > 注意，在没有缺点GPU执行完 command allocator 中所有命令前，千万不要重置命令分配器！

- **IID_PPV_ARGS** 辅助宏

```
#define IID_PPV_ARGS(ppType) __uuidof(**(ppType)), IID_PPV_ARGS_Helper(ppType)
```

其中，__uuidof(**(ppType))将获取(**(ppType))的COM 接口 ID（globally unique identifier，全局唯一标识符，GUID），在上述代码段中得到的即为 ID3D12CommandQueue 接口的 COM ID。IID_PPV_ARGS 辅助函数的本质是将 ppType 强制转换为 void**类型。我们在全书中都会见到此宏的身影，这是因为在调用 Direct3D 12 中创建接口实例的 API 时，大多都有一个参数是类型为 void**的待创接口 COM ID。

- **围栏**（fence），**刷新命令队列**，强制CPU等待，直到GPU完成所有命令的处理，达到某个指定的围栏点为止。围栏用 **ID3D12Fence** 接口来表示
- **资源屏障**
- **初始化Direct3D**

1. 用 D3D12CreateDevice 函数创建 ID3D12Device 接口实例。
2. 创建一个 ID3D12Fence 对象，并查询描述符的大小。
3. 检测用户设备对 4X MSAA 质量级别的支持情况。
4. 依次创建命令队列、命令列表分配器和主命令列表。
5. 描述并创建交换链。
6. 创建应用程序所需的描述符堆。
7. 调整后台缓冲区的大小，并为它创建渲染目标视图。
8. 创建深度/模板缓冲区及与之关联的深度/模板视图。
9. 设置视口（viewport）和裁剪矩形（scissor rectangle）。

1. 创建设备，利用函数D3D12CreateDeviece 创建接口实例**md3dDevice**
2. 创建围栏，md3dDevice -> CreateFence 创建围栏**mFence**
3. 获取描述符大小，示例如下
1. **渲染目标描述符大小** mRtvDescriptorSize = md3dDevice -> GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_RTV)
2. **深度/模板描述符大小** mDsvDescriptorSize = md3dDevice -> GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_DSV)
3. **常量缓冲区描述符大小** mCbvDescriptorSize = md3dDevice -> GetDescriptorHandleIncrementSize(D3D12_DESCRIPTOR_HEAP_TYPE_CBV_SRV_UAV)
4. 检测对 4X MSAA 质量级别的支持，md3dDevice -> CheckFeatureSupport
5. 创建命令队列和命令列表，直接参考D3DAPP::CreateCommandObjects
6. 描述并创建交换链，IDXGIFactory::CreateSwapChain
注意描述 SampleDesc 的 Quality，其值需要通过 **(4** 来获取
7. 创建描述符堆，以 ID3D12DescriptorHeap 接口来表示，用 ID3D12Device::CreateDescriptorHeap 方法来创建，示例如下（参考 D3DAPP::CreateRtvAndDsvDescriptorHeaps）
1. **mRtvHeap**，存储 SwapChainBufferCount 个 RTV
2. **mDsvHeap**，存储 DSV（暂时用一个）
3.

在本书的应用框架中有以下定义：

```
static const int SwapChainBufferCount = 2;
int mCurrBackBuffer = 0;
```

其中，mCurrBackBuffer 是用来记录当前后台缓冲区的索引（由于利用页面翻转技术来交换前台缓冲区和后台缓冲区，所以我们需要对其进行记录，以便搞清楚哪个缓冲区才是当前正在用于渲染数据的后台缓冲区）。

创建描述符堆之后，还要能访问其中所存的描述符。在程序中，我们是通过句柄来引用描述符的，并以 ID3D12DescriptorHeap::GetCPUDescriptorHandleForHeapStart 方法来获得描述符堆中第一个描述符的句柄。借助下列函数即可获取当前后台缓冲区的 RTV 与 DSV：

```
D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::CurrentBackBufferView()const
{
  // CD3DX12 构造函数根据给定的偏移量找到当前后台缓冲区的 RTV
  return CD3DX12_CPU_DESCRIPTOR_HANDLE(
    mRtvHeap->GetCPUDescriptorHandleForHeapStart(),// 堆中的首个句柄
    mCurrBackBuffer,   // 偏移至后台缓冲区描述符句柄的索引
    mRtvDescriptorSize); // 描述符所占字节的大小
}

D3D12_CPU_DESCRIPTOR_HANDLE D3DApp::DepthStencilView()const
{
  return mDsvHeap->GetCPUDescriptorHandleForHeapStart();
}
```

通过这段示例代码，我们就能够看出描述符大小的用途了。为了用偏移量找到当前后台缓冲区的 RTV 描述符[1]，我们就必须知道 RTV 描述符的大小。

8. 创建渲染目标视图 / 描述符

1. **IDXGISwapChain::GetBuffer**，可用于获得存于交换链中的缓冲区资源
2. **ID3D12Device::CreateRenderTargetView**，为获取的后台缓冲区创建渲染目标描述符。（仅仅有交换链和描述符堆还不够，还需要有描述符）

```
ComPtr<ID3D12Resource> mSwapChainBuffer[SwapChainBufferCount];
CD3DX12_CPU_DESCRIPTOR_HANDLE rtvHeapHandle(
  mRtvHeap->GetCPUDescriptorHandleForHeapStart());
for (UINT i = 0; i < SwapChainBufferCount; i++)
{
  // 获得交换链内的第 i 个缓冲区
  ThrowIfFailed(mSwapChain->GetBuffer(
    i, IID_PPV_ARGS(&mSwapChainBuffer[i])));

  // 为此缓冲区创建一个 RTV
  md3dDevice->CreateRenderTargetView(

    mSwapChainBuffer[i].Get(), nullptr, rtvHeapHandle);

  // 偏移到描述符堆中的下一个缓冲区
  rtvHeapHandle.Offset(1, mRtvDescriptorSize);
}
```

9. 创建深度 / 模板缓冲区及其视图 / 描述符，

1. 深度缓冲区其实是一种2D纹理，纹理是一种GPU资源，因此需要填写 D3D12_RESOURCE_DESC 结构体来描述纹理资源，通过 ID3D12Device::CreateCommittedResource 方法来创建它。
2. 在使用深度/模板缓冲区前，一定要创建相关的深度/模板描述符，并绑定到渲染流水线上，类似于 **(8**
3. 在创建深度/模板描述符后，需要将深度 / 模板缓冲区资源从初始状态转换为深度缓冲区

```
// 将资源从初始状态转换为深度缓冲区
mCommandList->ResourceBarrier(
  1,
  &CD3DX12_RESOURCE_BARRIER::Transition(
    mDepthStencilBuffer.Get(),
    D3D12_RESOURCE_STATE_COMMON,
    D3D12_RESOURCE_STATE_DEPTH_WRITE));
```

4. 注意，所有创建描述符的函数，第二位参数如果指定为 nullptr，表示将采用该资源创建时的格式，为它的第一个 mipmap层级创建一个描述符

10. 设置视口，通过 `ID3D12GraphicsCommandList::RSSetViewports` 方法来设置Direct3D中的视口

- 命令列表一旦被重置，视口也就需要随之重置

11. 设置剪裁矩形，通过 `ID3D12GraphicsCommandList::RSSetScissorRects` 方法来设置剪裁矩形，例如，剔除被矩形UI（UI位于某块区域的最上层）遮挡的3D空间中的像素，这部分像素将不会再被光栅化处理

- 命令列表一旦被重置，剪裁矩形也就需要随之重置

12. 观察空间

综上所述，只要给定摄像机的位置、观察目标点以及世界空间中“向上”方向的向量，我们就能构建出对应的摄像机局部坐标系，并推导出相应的观察矩阵。

DirectXMath 库针对上述计算观察矩阵的处理流程提供了以下函数：

```
XMMATRIX XM_CALLCONV XMMatrixLookAtLH(      // 输出观察矩阵 V
  FXMVECTOR EyePosition,                    // 输入虚拟摄像机位置 Q
  FXMVECTOR FocusPosition,                  // 输入观察目标点 T
  FXMVECTOR UpDirection);                   // 输入世界空间中向上方向的向量 j
```

一般来说，世界空间中的 y 轴方向与虚拟摄像机“向上”向量的方向相同，所以，我们通常将“向上”向量定为 $\boldsymbol{j} = (0, 1, 0)$。举个例子，假设我们希望把虚拟摄像机架设于世界空间内点$(5, 3, -10)$的位置，并令它观察世界空间的原点（0, 0, 0），则构建相应观察矩阵的过程为：

```
XMVECTOR pos  = XMVectorSet(5, 3, -10, 1.0f);
XMVECTOR target = XMVectorZero();
XMVECTOR up   = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);

XMMATRIX V = XMMatrixLookAtLH(pos, target, up);
```

13. 投影和齐次裁剪空间

我们可以利用 DirectXMath 库内的 XMMatrixPerspectiveFovLH 函数来构建透视投影矩阵：

```
// 返回投影矩阵
XMMATRIX XM_CALLCONV XMMatrixPerspectiveFovLH(
  float FovAngleY,     // 用弧度制表示的垂直视场角
  float Aspect,        // 纵横比 = 宽度 / 高度
  float NearZ,         // 到近平面的距离
  float FarZ);         // 到远平面的距离
```

下面的代码片段详细解释了 XMMatrixPerspectiveFovLH 函数的用法。在此例中，我们将垂直视场角指定为 45°，近平面位于 $z = 1$ 处，远平面位于 $z = 1000$ 处（这些长度皆以观察空间中的单位表示）。

```
XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*XM_PI,
AspectRatio(), 1.0f, 1000.0f);
```

纵横比采用的是我们窗口的宽高比：

```
float D3DApp::AspectRatio()const
{
  return static_cast<float>(mClientWidth) / mClientHeight;
}
```