

备忘录

笔记本: DirectX 12

创建时间: 2022/8/2 0:42

更新时间: 2022/8/3 16:42

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 `D3D12CreateDevice` 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. 注意, **顶点缓冲区描述符** 不需要 创建 descriptor heap

4.

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. 缓冲区资源需要用 ID3D12Device::CreateCommittedResource 方法来创建。缓冲区资源由 D3D12_RESOURCE_DESC 结构体表示

（Direct3D 12 提供了一个 C++ 包装类 CD3DX12_RESOURCE_DESC 结构体，它派生自 D3D12_RESOURCE_DESC，它提供了如下简化的构造函数：

CD3DX12_RESOURCE_DESC::Buffer，

CD3DX12_RESOURCE_DESC::Tex1D，Tex2D，Tex3D 等）。

1. pHeapProperties：（资源欲提交至的）堆所具有的属性。有一些属性是针对高级用法而设。目前只需关心 D3D12_HEAP_PROPERTIES 中的 D3D12_HEAP_TYPE 枚举类型这一主要属性，其中的成员列举如下。
 - a) D3D12_HEAP_TYPE_DEFAULT：默认堆（default heap）。向这堆里提交的资源，唯独 GPU 可以访问。举一个有关深度/模板缓冲区的例子：GPU 会读写深度/模板缓冲区，而 CPU 从不需要访问它，所以深度/模板缓冲区应被放入默认堆中。
 - b) D3D12_HEAP_TYPE_UPLOAD：上传堆（upload heap）。向此堆里提交的都是需要经 CPU 上传至 GPU 的资源。
 - c) D3D12_HEAP_TYPE_READBACK：回读堆（read-back heap）。向这种堆里提交的都是需要由 CPU 读取的资源。
 - d) D3D12_HEAP_TYPE_CUSTOM：此成员应用于高级场景——更多信息可详见 MSDN 文档。

2. IDXGISwapChain::GetBuffer，可用于获得存于交换链中的缓冲区资源

3. 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 $j = (0, 1, 0)$ 。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

投影顶点

$$x' = \frac{x}{z \tan(a/2)}$$

$$y' = \frac{y}{z \tan(a/2)}$$

NDC，计算时，默认高为 2 $([-1,1])$ ，宽为 $2r$ ，纵横比为 r

$$x' = \frac{x}{r z \tan(a/2)}$$

$$y' = \frac{y}{z \tan(a/2)}$$

1. **矩阵表示**，为了保持变换的一致性，需要用矩阵来表示投影变换。但是上面的公式的非线性特征（存在 z ），无法矩阵化，所以需要一分为二，线性部分和非线性部分 $1/z$ 。

可以看到，我们在矩阵当中设置了常量 A 和常量 B （在下一节中会推导它们的定义），利用它们即可把输入的 z 坐标变换到归一化范围。令任意点 $(x, y, z, 1)$ 与该矩阵相乘将会得到：

$$[x, y, z, 1] \begin{bmatrix} \frac{1}{r \tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix} \quad (5.2)$$
$$= \left[\frac{x}{r \tan\left(\frac{\alpha}{2}\right)}, \frac{y}{\tan\left(\frac{\alpha}{2}\right)}, Az + B, z \right]$$

2. **归一化深度值**，将 z 坐标从区间 $[n, f]$ 映射到区间 $[0, 1]$

$$g(z) = A + \frac{B}{z}$$

$$g(z) = \frac{f}{f - n} + \frac{nf}{(f - n)z}$$

3. 投影透视矩阵

既然已经求得了 A 和 B ，我们就可以确定出完整的透视投影矩阵（perspective projection matrix）：

$$P = \begin{bmatrix} \frac{1}{r \tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-nf}{f-n} & 0 \end{bmatrix}$$

顶点 \times 投影透视矩阵 但未进行透视除法前，几何体会处于 **齐次剪裁空间** 或 **投影空间** 中，待完成透视除法后，便是用规格化设备坐标NDC来表示几何体了。

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
XMATRIX 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）。

```
XMATRIX 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;
}
```

14. 输入布局描述，对顶点结构体的描述

D3D12_INPUT_LAYOUT_DESC

— D3D12_INPUT_ELEMENT_DESC

元素构成的数组

— UINT

数组中的元素数量

```
typedef struct D3D12_INPUT_ELEMENT_DESC
{
    LPCSTR SemanticName;
    UINT SemanticIndex;
    DXGI_FORMAT Format;
    UINT InputSlot;
    UINT AlignedByteOffset;
    D3D12_INPUT_CLASSIFICATION InputSlotClass;
    UINT InstanceDataStepRate;
} D3D12_INPUT_ELEMENT_DESC;
```

- 1, SemanticName：语义
- 2, SemanticIndex：语义的索引
- 3, Format：DXGI_FORMAT 格式类型
- 4, InputSlot：输入槽
- 5, AlignedByteOffset：起始地址的偏移量

6, InputSlotClass : 暂定为

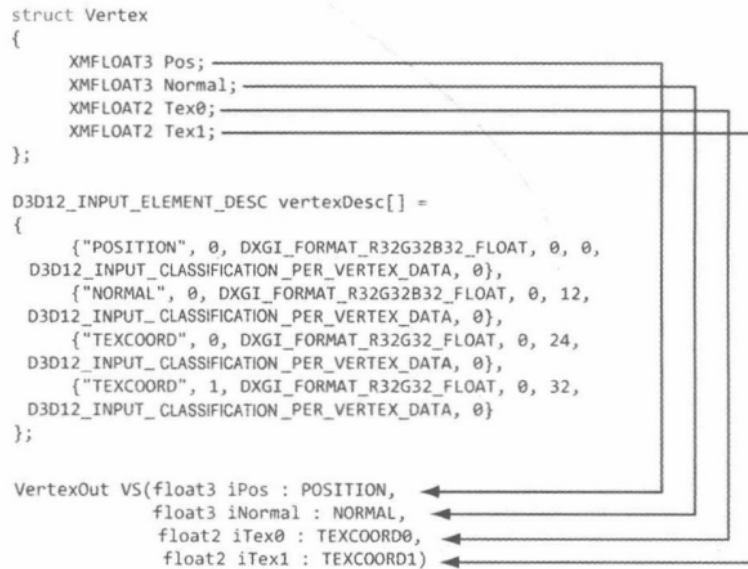
D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 另一个选项用于实例化

7, InstanceDataStepRate : 目前指定为0, 采用实例化则为1

```
struct Vertex
{
    XMFLOAT3 Pos;
    XMFLOAT3 Normal;
    XMFLOAT2 Tex0;
    XMFLOAT2 Tex1;
};

D3D12_INPUT_ELEMENT_DESC vertexDesc[] =
{
    {"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
     D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
    {"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12,
     D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
    {"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24,
     D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0},
    {"TEXCOORD", 1, DXGI_FORMAT_R32G32_FLOAT, 0, 32,
     D3D12_INPUT_CLASSIFICATION_PER_VERTEX_DATA, 0}
};

VertexOut VS(float3 iPos : POSITION,
             float3 iNormal : NORMAL,
             float2 iTex0 : TEXCOORD0,
             float2 iTex1 : TEXCOORD1)
```



15. 顶点缓冲区

一般来说, 顶点缓冲区都会被置于默认堆来优化性能 (CPU不能向默认堆的顶点缓冲区写入数据)

为了向顶点缓冲区复制顶点数据 (置于默认堆), 就需要创建一个中介位置的上传缓冲区

1. **顶点缓冲区描述符**, 通过 `D3D12_VERTEX_BUFFER_VIEW` 结构体来表示, 并且它并不需要创建描述符堆
2. **与输入槽绑定**, 在顶点缓冲区及其对应描述符创建完成后, 便可以将它与渲染流水线上的一个输入槽绑定, 通过

`ID3DGraphicsCommandList::IASetVertexBuffers` 方法实现

```
void ID3D12GraphicsCommandList::IASetVertexBuffers(
    UINT StartSlot,
    UINT NumView,
    const D3D12_VERTEX_BUFFER_VIEW *pViews);
```

1. StartSlot: 在绑定多个顶点缓冲区时, 所用的起始输入槽 (若仅有一个顶点缓冲区, 则将其绑定至此槽)。输入槽共有 16 个, 索引为 0~15。
2. NumViews: 将要与输入槽绑定的顶点缓冲区数量 (即视图数组 pViews 中视图的数量)。如果起始输入槽 StartSlot 的索引值为 k , 且我们要绑定 n 个顶点缓冲区, 那么这些缓冲区将依次与输入槽 $I_k, I_{k+1}, \dots, I_{k+n-1}$ 相绑定。
3. pViews: 指向顶点缓冲区视图数组中第一个元素的指针。

下面是该函数的一个调用示例。

```
D3D12_VERTEX_BUFFER_VIEW vbv;
vbv.BufferLocation = VertexBufferGPU->GetGPUVirtualAddress();
vbv.StrideInBytes = sizeof(Vertex);
vbv.SizeInBytes = 8 * sizeof(Vertex);

D3D12_VERTEX_BUFFER_VIEW vertexBuffers[1] = { vbv };
mCommandList->IASetVertexBuffers(0, 1, vertexBuffers);
```

3. 顶点缓冲区资源的虚拟地址, 可以通过

`ID3D12Resource::GetGPUVirtualAddress` 来获取

4. 以上仅为顶点数据送至渲染流水线做好准备, 最后还需要通过

`ID3D12GraphicsCommandList::DrawInstanced` 方法真正绘制顶点

16. 索引缓冲区

与顶点缓冲区相似

1. 索引缓冲区描述符由 `D3D12_INDEX_BUFFER_VIEW` 来表示
2. 通过 `ID3D12GraphicsCommandList::IASetIndexBuffer` 方法将索引缓冲区绑定到输入装配器阶段
3. 最后不同的是，需要通过 `ID3D12GraphicsCommandList::DrawIndexedInstanced` 代替 `DrawInstanced` 进行绘制