**第4章 Direct3D的初始化**

**1，组件对象模型**

COM指针的使用：

1，通过特定函数或另一COM接口的方法，而不是C++语言中的关键字new获取COM指针。

2， Microsoft::WRL::ComPtr类(#include<wrl.h>)，可以把它当作是COM对象的智能指针。

ComPtr3个常用的方法：

1. Get：返回一个指向此底层COM接口的指针。此方法常用于把原始的COM接口指针作为参数传递给函数。
2. GetAddressOf：返回指向此底层COM接口指针的地址。凭此方法可利用函数参数返回COM接口的指针。
3. Reset：将此ComPtr实例设置为nullptr释放与之相关的所有引用。此方法的功能与将Comptr目标实例赋值为nullptr效果相同。

**2，纹理格式**

2D纹理是一种由数据元素构成的矩阵。它的用途之一是存储2D图像数据，在这种情况下，纹理中每个元素存储的都是一个像素的颜色。然而，纹理的用处并非仅此而已。

并不是任意类型的数据元素都能用于组成纹理，它只能存储DXGI\_FORMAT枚举类型中描述的特定格式的数据元素。

1，DXGI\_FORMAT\_R32G32B32\_FLOAT

2，DXGI\_FORMAT\_R16G16B16A16\_UNORM

3，DXGI\_FORMAT\_R32G32\_UINT

4，DXGI\_FORMAT\_R8G8B8A8\_UNORM

5，DXGI\_FORMAT\_R8G8B8A8\_SNORM

6，DXGI\_FORMAT\_R8G8B8A8\_SINT

7，DXGI\_FORMAT\_R8G8B8A8\_UINT

DXGI\_FORMAT\_R16G16B16A16\_TYPELESS无类型。

**3，交换链和页面翻转**

为了避免动画中出现画面闪烁的现象，最好将动画帧完整地绘制在一种称为后台缓冲区的离屏纹理内。为此，需要利用硬件管理两种纹理缓冲区：前台缓冲区和后台缓冲区。

当后台缓冲区中的动画帧绘制完成之后，两种缓冲区的角色互换：后台缓冲区变为前台缓冲区呈现新一帧的画面，而前台缓冲区则为了展示动画的下一帧转为后台缓冲区，等待填充数据。

前台缓冲区和后台缓冲区构成了交换链，在Direct3D中用IDXGISwapChain接口来表示。这个接口不仅存储了前台缓冲区和后台缓冲区两种纹理，而且还提供了修改缓冲区大小和呈现缓冲区内容的方法。（前后台缓冲的这种互换操作称为呈现）

**4，深度缓冲**

深度缓冲区这种纹理资源存储的并非图像数据，而是特定像素的深度信息。深度值的范围为0-1。0代表观察者在视椎体中能看到离自己最近的物体，1代表能看到最远的物体。

深度缓冲区中的元素与后台缓冲区内的像素呈一一对应关系，如果后台缓冲区的分辨率为1280x1024，那么深度缓冲区中就应当有1280x1024个深度元素。

若使用了深度缓冲，则物体的绘制顺序也就变得无关紧要了。

深度缓冲技术的原理是计算每个像素的深度值，并执行深度测试，具有最小深度值的像素会获得最终的胜利，它将被写入后台缓冲区中。

深度缓冲区也是一种纹理，可用的格式如下：

1，DXGI\_FORMAT\_D32\_FLOAT\_S8X24\_UINT

2，DXGI\_FORMAT\_D32\_FLOAT

3，DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT

4，DXGI\_FORMAT\_D16\_UNORM

**5，资源与描述符**

**描述符的概念**：在发出绘制命令之前，我们需要将与本次绘制调用（draw call）相关的资源绑定到渲染流水线上。但是，GPU资源并非直接与渲染流水线相绑定，而是要通过一种名为描述符的对象来对它间接引用。

**描述符的作用**：

1，GPU资源实质上都是一些普通的内存块。由于资源的这种通用性，它们便能被设置到渲染流水线的不同阶段供其使用。

2，一个资源可以绑定到渲染流水线的不同阶段。每个阶段都需要设置独立的描述符。创建描述符的最佳时机为初始化期间。

视图和描述符是同义词。每个描述符都有一种具体类型，此类型指明了资源的具体作用。本书常用的描述符如下：

1，CBV/SRV/UAV描述符分别表示的是常量缓冲区视图，着色器资源视图和无序访问视图这3种资源；

2，采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。

4，DSV描述符表示的是深度/模板视图资源。

**描述符堆**中存有一系列描述符，本质上是存放用户程序中某种特定类型描述符的一块内存。

我们需要为每一种类型的描述符都创建出单独的描述符堆。也可以为同一种描述符类型创建出多个描述符堆。

**6，多重采样技术的原理**

在不能提升显示器分辨率或在显示器分辨率受限的情况下，我们就可以运用各种反走样（也有译为抗锯齿）技术。

超级采样（SSAA）使用4倍于屏幕分辨率大小的后台缓冲区和深度缓冲区。当数据要从后台缓冲区调往屏幕显示的时候，会将后台缓冲区按4个像素一组进行解析。每组用求平均值的方法得到一种相对平滑的像素颜色。

多重采样（MSAA）同样使用4倍于屏幕分辨率的后台缓冲区和深度缓冲区。多重采样并不需要对每一个子像素都进行计算，而是仅计算一次像素中心处的颜色，再基于可见性和覆盖性将得到的颜色信息分享给其子像素。

**7，利用Direct3D进行多重采样**

typedef struct DXGI\_SAMPLE\_DESC{

UINT Count;

UINT Quality;

}DXGI\_SAMPLE\_DESC;

Count成员指定了每个像素的采样次数，Quality成员则用于指示用户期望的图像质量级别。

根据给定的纹理格式和采样数量，我们就能用ID3D12Device::CheckFeatureSupport方法查询到对应的质量级别。

Typedef struct D3D12\_FEATURE\_DATA\_MULTISAMPLE\_QUALITY\_LEVELS{

DXGI\_FORMAT Format;

UINT SampleCount;

D3D12\_MULTISAMPLE\_QUALITY\_LEVEL\_FLAGS Flags;

UINT NumQualityLevels;

} D3D12\_FEATURE\_DATA\_MULTISAMPLE\_QUALITY\_LEVELS;

ThrowIfFailed(md3dDevice->CheckFeatureSupport(

D3D12\_FEATURE\_MULTISAMPLE\_QUALITY\_LEVELS;

&msQualityLevels,

sizeof(msQualityLevels)));

此方法的第二个参数兼具输入和输出的属性。当它作为输入参数时，我们必须指定纹理格式，采样数量以及希望查询的多重采样所支持的标志。

通常会把采样数量设定为4或8。如果不希望使用多重采样，则可将采样数量设为1，并令质量级别为0.

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。

**8，功能级别**

功能级别为不同级别所支持的功能进行了严格的界定，每个功能级别所支持的特定功能可参见SDK文档。

如果用户的硬件不支持某特定功能级别，应用程序理当回退至版本更低的功能级别。

enum D3D\_FEATURE\_LEVEL{

D3D\_FEATURE\_LEVEL\_9\_1 = 0x9100,

D3D\_FEATURE\_LEVEL\_9\_2 = 0x9200,

……

D3D\_FEATURE\_LEVEL\_11\_1 = 0xb100,

} D3D\_FEATURE\_LEVEL

**9，DirectX图形基础结构**

DirectX图形基础结构（DXGI）是一种与Direct3D配合使用的API。

下面介绍一些在Direct3D初始化时会用到的相关接口：

IDXGIFactory主要用于创建IDXGISwapChain接口以及枚举**显示适配器**。而显示适配器则真正实现了图形处理能力。通常来说，显示适配器是一种硬件设备（例如独立显卡），然而系统也可以用软件显示适配器来模拟硬件的图形处理功能。一个系统中可能会存在数个适配器（比如装有数块显卡），显示器用接口IDXGIAdapter来表示。

UINT i = 0;

IDXGIAdapter\* adapter = nullptr;

std::vecter<IDXGIAdapter\*> adapterList;

while(mdxgiFactory->EnumAdapters( i, &adapter) != DXGI\_ERROR\_NOT\_FOUND){

adapterList.push\_back(adapter);

i++;

}

一个系统也可能装有数个显示设备，我们称每一台显示设备都是一个显示输出实例（适配器输出），用IDXGIOutput接口表示。每个适配器都与一组**显示输出**相关联。

UINT i = 0;

IDXGIOutput\* output = nullptr;

while(adapter->EnumOutput( i, &output) != DXGI\_ERROR\_NOT\_FOUND){

DXGI\_OUTPUT\_DESC desc;

output->GetDesc(&desc);

i++;

}

每种显示设备都有一系列它所支持的**显示模式**，可以用DXGI\_MODE\_DESC结构体中的数据成员表示。

UINT count = 0;

UINT flags = 0;

output->GetDisplayModeList(format, flags, &count, nullptr);

std::Vector<DXGI\_MODE\_DESC> modeList(count);

output->GetDisplayModeList(format, flags, &count, &modeList[0]);

for(auto& x : modeList){}

在进入全屏模式之时，枚举显示模式就显得尤为重要。为了获得最优的全屏性能，我们所指定的显示模式（包括刷新率）一定要与显示器支持的显示模式完全匹配。

**10，功能支持的检测**

HRESULT ID3D12Device::CheckFeatureSupport(

D3D12\_FEATURE Feature, void \*pFeatureSupportData, UINT FeatureSupportDataSize

)

Feature：枚举类型D3D12\_FEATURE中的成员之一，指定我们希望检测的功能支持类型：

1. D3D12\_FEATURE\_D3D12\_OPTIONS：检测图形驱动对Direct3D12各种功能的支持情况；
2. D3D12\_FEATURE\_ARCHITECTURE：检测图形适配器中GPU的硬件体系架构特性；
3. D3D12\_FEATURE\_FEATURE\_LEVELS：检测对功能级别的支持情况；
4. D3D12\_FEATURE\_\_FORMAT\_SUPPORT：检测对给定纹理格式的支持情况；
5. D3D12\_FEATURE\_MULTISAMPLE\_QUALITY\_LEVELS：检测对多重采样功能的支持情况。

**11，资源驻留**

一般来说，资源在创建时就会驻留在显存中，而当它被销毁时则清出。但是通过下面方法，我们可以自己控制资源的驻留。

HRESULT ID3D12Device::MakeResident(UINT NumObjects, ID3D12Pageable \*const \*ppObjects)

HRESULT ID3D12Device::Evict(UINT NumObjects, ID3D12Pageable \*const \*ppObjects)