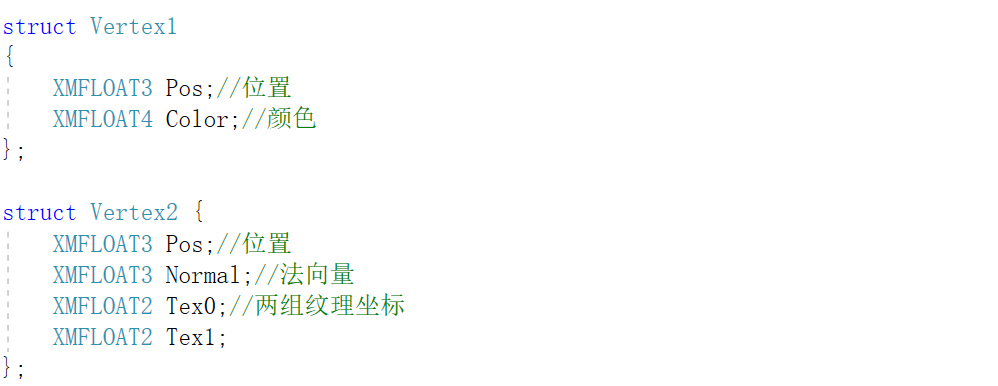
**6 利用Direct3D绘制几何体**

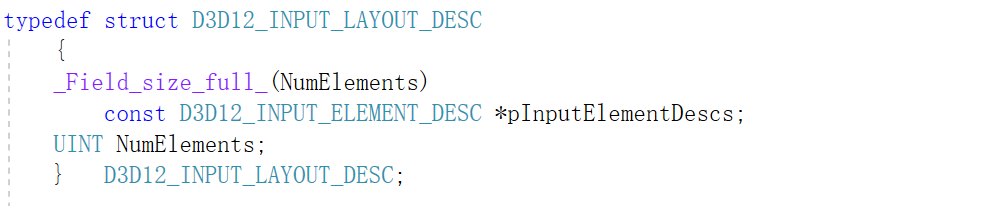
**6.1 顶点与输入布局**

除了空间位置，Direct3D中的顶点还可以存储其他属性数据。为了构建自定义的顶点格式，我们首先要创建一个结构体来容纳选定的顶点数据。

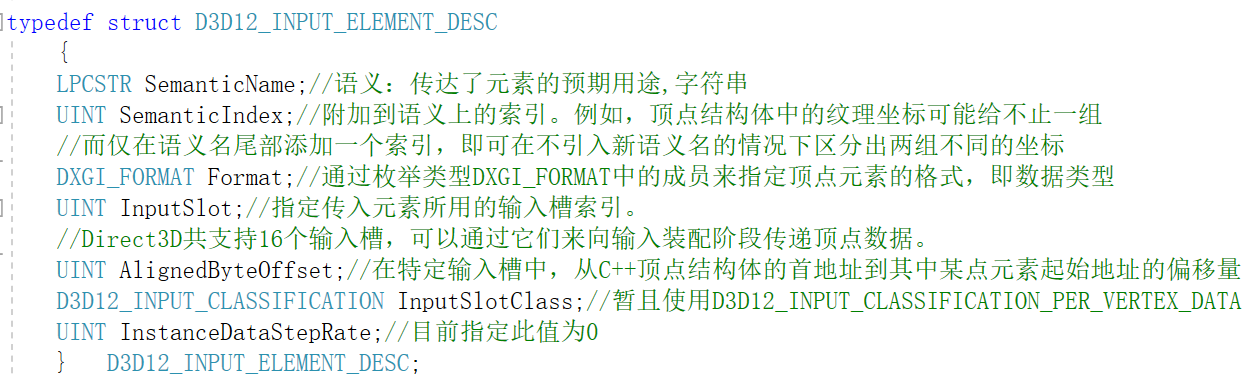
定义了顶点结构体之后，我们还需要向Direct3D提供该顶点结构体的描述，使它了解应怎样来处理结构体中的每个成员。

用户提供给Direct3D的这种描述被称为输入布局描述，用结构体

D3D12\_INPUT\_LAYOUT\_DESC来表示：

输入布局描述由两部分组成：一个以D3D12\_INPUT\_ELEMENT\_DESC元素构成的数组，以及一个表示该数组中的元素数量的整数。

D3D12\_INPUT\_ELEMENT\_DESC结构体定义如下：

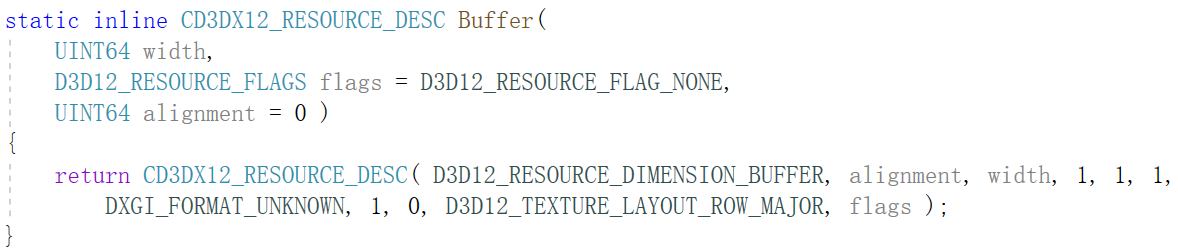


**6.2 顶点缓冲区**

为了使GPU可以访问顶点数组，就需要把它们放置在称为缓冲区的GPU资源（ID3D12Resource）里。我们把存储顶点的缓冲区叫做顶点缓冲区。

我们先通过填写D3D12\_RESOURCE\_DESC结构体来描述缓冲区资源，接着再调用ID3D12Device::CreateCommittedResource方法去创建ID3D12Resource对象。

包装类CD3DX12\_RESOURCE\_DESC，派生自D3D12\_RESOURCE\_DESC结构体，并附有多种便于使用的构造函数以及方法。



若缓冲区存储64个float类型的数据，那么width的值即为64\*sizeof(float)。

在Direct3D 12中，所有的资源均用ID3D12Resource接口表示。

对于静态几何体，即每一帧都不会发生改变的几何体而言，我们会将其顶点缓冲区置于默认堆(D3D12\_HEAP\_TYPE\_DEFAULT)中来优化性能。一般来说，游戏中的大多数几何体都是如此处理。在这种情况下，顶点缓冲区初始化完毕后，只有GPU需要从其中读取数据来绘制几何体，所以使用默认堆是很明智的做法。然而，如果CPU不能向默认堆中的顶点缓冲区写入数据，那么我们该如何初始化此顶点缓冲区？

因此，除了创建顶点缓冲区资源本身以外，我们还需用D3D12\_HEAP\_TYPE\_UPLOAD这种堆类型来创建一个处于中介位置的上传缓冲区资源。在创建了上传缓冲区之后，我们就可以将顶点数据从系统内存复制到上传缓冲区，而后再把顶点数据从上传缓冲区复制到真正的顶点缓冲区中。

由于我们需要利用作为中介的上传缓冲区来初始化默认缓冲区中的数据。因此，我们就在d3dUtil.h/.cpp文件中构建了d3dUtil::CreateDefaultBuffer工具函数，避免每次做重复的工作。

**查看d3dUtil::CreateDefaultBuffer源码**

Microsoft::WRL::ComPtr<ID3D12Resource> d3dUtil::CreateDefaultBuffer(

ID3D12Device\* device,

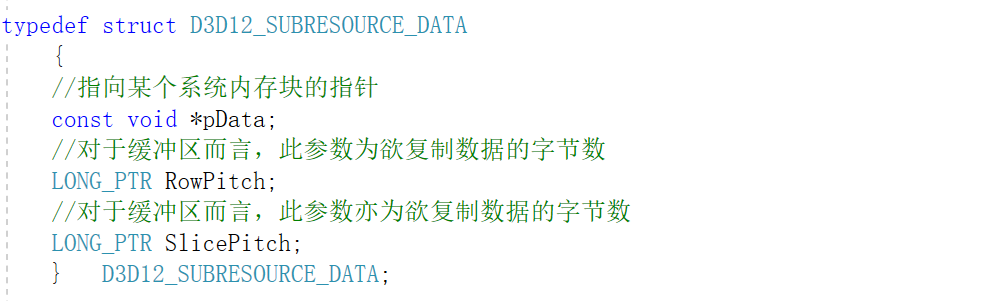
ID3D12GraphicsCommandList\* cmdList,

const void\* initData,

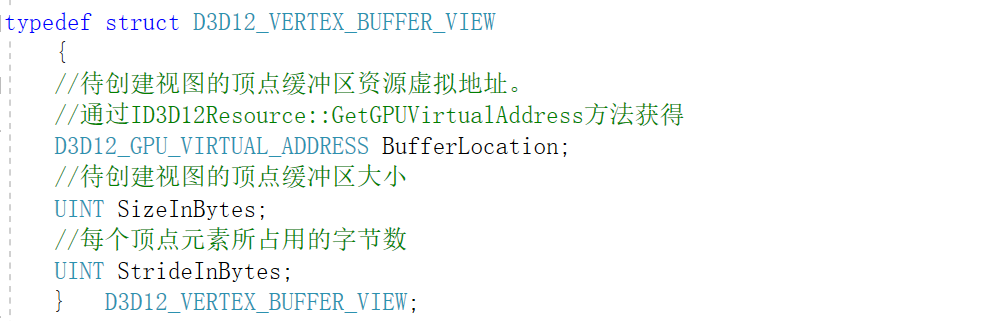
UINT64 byteSize,

Microsoft::WRL::ComPtr<ID3D12Resource>& uploadBuffer)

D3D12\_SUBRESOURCE\_DATA描述我们希望复制到默认缓冲区中的数据



为了将顶点缓冲区绑定到渲染流水线上，我们需要给这种资源创建一个顶点缓冲区视图。与RTV不同的是，我们无须为顶点缓冲区视图创建描述符堆。顶点缓冲区视图是由D3D12\_VERTEX\_BUFFER\_VIEW结构体来表示。



在顶点缓冲区及其对应视图创建完成后，便可以将它与渲染流水线上的一个输入槽相绑定。

这样一来，我们就能向流水线中的输入装配器阶段传递顶点数据了。此操作可以通过下列方法来实现：

void ID3D12GraphicsCommandList::IASetVertexBuffers(

UINT StartSlot, UINT NumViews, const D3D12\_VERTEX\_BUFFER\_VIEW \*pViews);

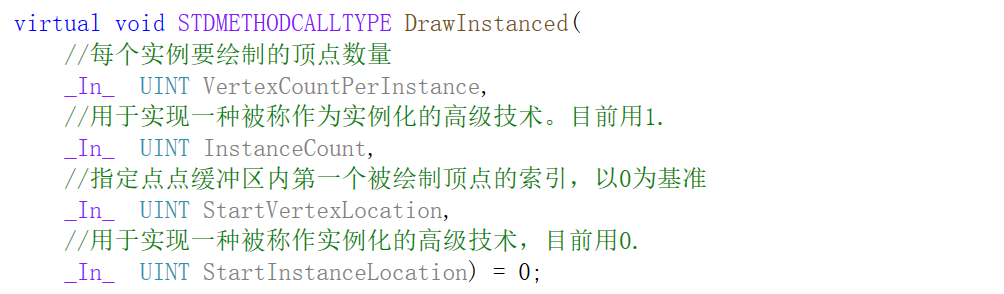
StartSlot：在绑定多个顶点缓冲区时，所用的起始输入槽

NumViews：将要输入槽绑定的顶点缓冲区数量

pViews：指向顶点缓冲区视图数组的第一个元素的指针。

将顶点缓冲区设置到输入槽上并不会对其执行实际的绘制操作，而是仅为顶点数据送至渲染流水线做好准备而已。

这最后一步才是通过ID3D12GraphicsCommandList::DrawInstanced方法真正地绘制顶点。



既然DrawInstanced方法没有指定顶点被定义为何种图元，那么，它们应该被绘制为点，线列表还是三角形列表呢？

图元拓扑状态由ID3D12GraphicsCommandList::IASetPrimitiveTopology方法来设置。

顶点的处理的流程：

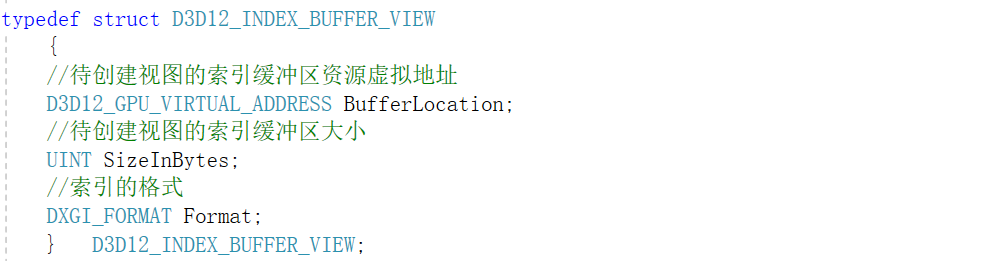
顶点结构体和输入布局描述🡪上传缓冲区🡪顶点缓冲区🡪顶点缓冲区视图🡪渲染流水线输入槽🡪绘制顶点

**6.3 索引和索引缓冲区**

与顶点相似，为了使GPU可以访问索引数组，就需要将它们放置于GPU的缓冲区资源(ID3D2Resource)内。我们称存储索引的缓冲区为索引缓冲区。

本书采用d3dUtil::CreateDefaultBuffer函数时通过void\*类型作为参数引入泛型数据，这就意味着我们也可以用此函数来创建索引缓冲区（或任意类型的默认缓冲区）。

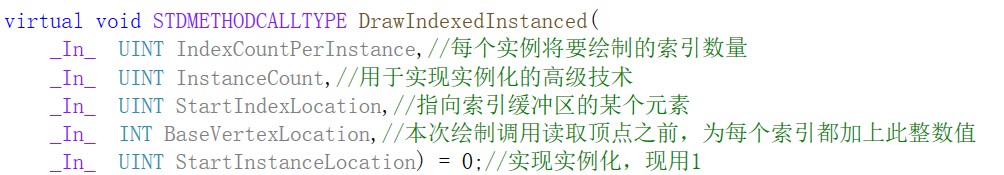
为了使索引缓冲区与渲染流水线绑定，我们需要给索引缓冲区资源创建一个索引缓冲区视图。无须为索引缓冲区创建描述符堆。索引缓冲区视图用D3D12\_INDEX\_BUFFER\_VIEW来表示。



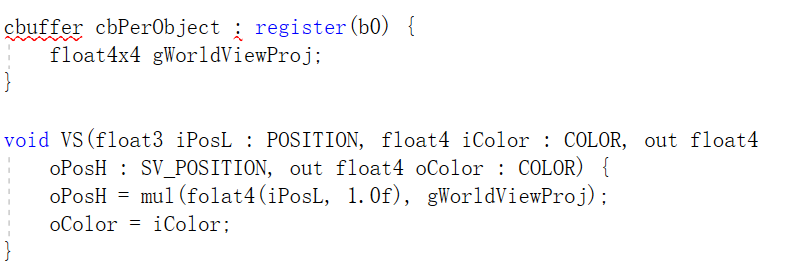
与顶点缓冲区相似，在使用之前，我们需要先将它们绑定到渲染流水线上。通过ID3D12GraphicsCommandList::IASetIndexBuffer方法即可将索引缓冲区绑定到输入装配器阶段。



在使用索引的时候，我们一定要用ID3D12GraphicsCommandList::DrawIndexedInstanced方法代替DrawInstanced方法进行绘制。



**6.4 顶点着色器示例**



在Direct3D中，编写着色器的语言为高级着色语言HLSL，其语法与C++十分相似。一般情况下，着色器通常在以.hlsl为扩展名的文本文件中编写。

SV\_POSITION语义比较特殊，它所修饰的顶点着色器输出元素存有齐次裁剪空间中的顶点位置信息。因此，我们必须为输出位置信息的参数附上SV\_POSITION语义，使GPU可以进行例如裁剪，深度测试和光栅化处理。如果没有使用几何着色器，那么顶点着色器必须用SV\_POSITION语义来输出顶点在齐次裁剪空间中的位置。

我们可以把函数的返回类型输入签名替换为结构体，从而取代过长的参数列表。

如果我们传入的顶点数据与顶点着色器所期望的输入不相符，便会导致错误。但是顶点数据与输入签名不需要完全匹配，顶点数据中也可以附带一些顶点着色器根本用不到的额外数据。

**6.5 像素着色器示例**

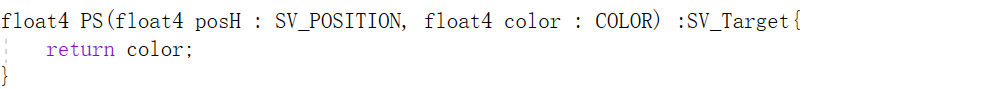
为了计算出三角形中每个像素的属性，我们会在光栅化处理期间对顶点着色器（或几何着色器）输出的顶点属性进行插值。随后再将这些插值数据传至像素着色器中作为它的输入。

像素着色器与顶点着色器有些相似：前者是针对每一个像素片段而运行的函数，后者是针对每一个顶点而运行的函数。

值得我们注意的是，这些输入像素着色器的像素片段有可能最终不会传入或留存在后台缓冲区中。例如，像素片段可能会在像素着色器中被裁剪掉，被另一个像素片段遮挡，或者被丢弃。

因此，在确定后台缓冲区某一像素的过程中，可能会存在多个候选的像素片段。这就是像素片段和像素意义的差别。

像素着色器的输入与顶点着色器的输出必须准确匹配。



SV\_Target语义表示该返回值的类型应当与渲染目标格式相匹配。

**6.6 常量缓冲区**

**6.6.1 创建常量缓冲区**

常量缓冲区是一种GPU资源，其数据内容可供着色器程序所引用。



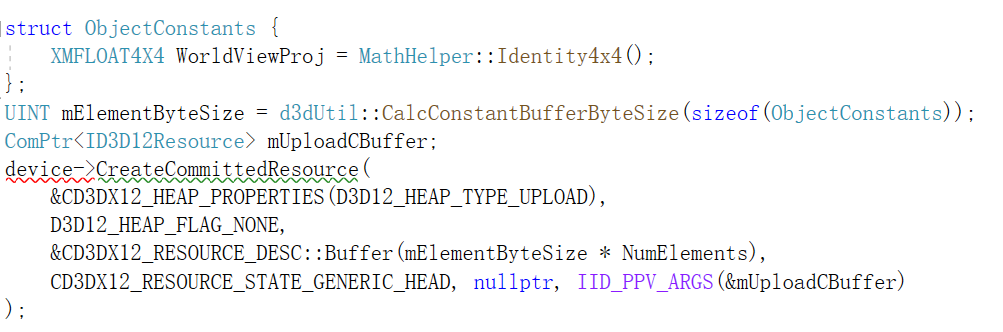
在这段代码中，cbuffer对象（常量缓冲区）的名称为cbPerObject，其中存储的是一个4x4矩阵，表示把一个点从局部空间变换到齐次裁剪空间所用的矩阵。

在HLSL中，可将一个4x4矩阵声明为内置的float4x4类型。

与顶点缓冲区和索引缓冲区不同的是，常量缓冲区通常由CPU每帧更新一次。如果摄像机每帧都不停地移动，那么常量缓冲区也需要每帧更新矩阵。所以，我们会把常量缓冲区创建到一个上传堆而非默认堆中。这样能使我们从CPU端更新常量。

常量缓冲区对硬件也有特别的要求，即常量缓冲区的大小必为硬件最小分配空间(256B)的整数倍。

我们经常需要用到多个相同类型的常量缓冲区。例如假设常量缓冲区cbPerObject内存储的是随不同物体而异的常量数据。因此，我们要绘制n个物体，则需要n个该类型的常量缓冲区。



mUploadCBuffer就是常量缓冲区，它存储了一个ObjectConstants类型的常量缓冲区数组。待到绘制物体的时候，只要将常量缓冲区视图（CBV）绑定到存有物体相应常量数据的缓冲区子区域即可。

工具函数d3dUtil::CalcConstantBufferByteSize会做适当的运算，使缓冲区的大小凑整为256B的整数倍。

6.6.2 更新常量缓冲区

首先要获得指向欲更新资源数据的指针，可用Map方法来做到这一点。

利用memcpy函数将数据从系统内存复制到常量缓冲区。

当常量缓冲区更新完成后，我们应在释放映射内存之前对其进行Unmap操作。

6.6.3 上传缓冲区辅助函数

将上传缓冲区的相关操作简单封装一下，使用起来更加方便。我们在UploadBuffer.h文件中定义了下面这个类。