**6 利用Direct3D绘制几何体**

6.1 顶点与输入布局

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。

struct Vertex1{

XMFLOAT3 Pos;

XMFLOAT4 Color;

}

struct Vertex2{

XMFLOAT3 Pos;

XMFLOAT3 Normal;

XMFLOAT2 Tex0;

XMFLOAT2 Tex1;

}

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，用结构体D3D12\_INPUT\_LAYOUT\_DESC来表示。

API：

D3D12\_INPUT\_LAYOUT\_DESC

D3D12\_INPUT\_ELEMENT\_DESC

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，接着在调用ID3D12Device::CreateCommittedResource方法去创建ID3D12Resource对象。

在Dirext3D 12中提供了一个包装类CD3DX12\_RESOURCE\_DESC，它派生自D3D12\_RESOURCE\_DESC结构体，并附有多重便于使用的构造函数以及方法。

API：

D3D12\_RESOURCE\_DESC

ID3D12Device::CreateCommittedResource

CD3DX12\_RESOURCE\_DESC::Buffer

对于静态几何体而言(即每一帧都不会发生改变的几何体)，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，游戏中的大多数的几何体（如树木，建筑，地形和角色动画）都是如此处理。

然而，CPU不能向默认堆中的顶点缓冲区写入数据。因此，我们还需要用D3D12\_HEAP\_TYPE\_UPLOAD这种堆类型创建一个处于中介位置的上传缓冲区资源。

我们通过把资源提交至上传堆，才得以将数据从CPU复制到GPU显存中。

API:

d3dUtil::CreateDefaultBuffer

(例子：创建存有立方体8个顶点的默认缓冲区，并为每个顶点都分别赋予不同的颜色)

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。

顶点缓冲区视图由D3D12\_VERTEX\_BUFFER\_VIEW结构体来表示。

API:

D3D12\_VERTEX\_BUFFER\_VIEW

在顶点缓冲区及其对应的视图创建完成后，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，我们就能向流水线的输入装配器阶段传递顶点数据。

API:

ID3D12GraphicsCommandList::IASetVertexBuffers

将顶点缓冲区设置到输入槽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。通过ID3D12GraphicsCommandList::DrawInstanced方法真正地绘制顶点。

API:

ID3D12GraphicsCommandList::DrawInstanced

顶点数据的处理流程

1，顶点结构体和输入布局描述；2，上传缓冲区；3，顶点缓冲区；4，顶点缓冲区视图；5，渲染流水线输入槽；6，绘制顶点

6.3 索引和索引缓冲区

与顶点类似，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(ID3D12Resource)内。我们称存储索引的缓冲区为索引缓冲区。

由于d3dUtil::CreateDefaultBuffer函数是通过void\*类型作为参数引入泛型数据，这意味着我们也可以使用该函数来创建索引缓冲区或任意类型的默认缓冲区。

API:

d3dUtil::CreateDefaultBuffer

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。索引缓冲区视图用结构体D3D12\_INDEX\_BUFFER\_VIEW表示。无须为索引缓冲区视图创建描述符堆。

API:

D3D12\_INDEX\_BUFFER\_VIEW

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::IASetIndexBuffers方法即可将索引缓冲区绑定到输入装配器阶段。

(例子：创建一个索引缓冲区来定义构成立方体的三角形)

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

API:

ID3D12GraphicsCommandList::DrawIndexedInstanced

6.4 顶点着色器示例

通过以下代码实现一个简单的顶点着色器：

cbuffer cbPerObject : register(b0){

float4x4 gWorldViewProj;

}

void VS(float3 iPosL : POSITION, float4 iColor:COLOR, out float4 oPosH : SV\_POSITION, out float4 oColor : COLOR){

oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);

oColor = iColor;

}

在Dirext3D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，易于学习。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。

HLSL没有引用和指针概念，所以需要借助结构体或多个输入参数才能够从函数中返回多个数值。在HLSL中，所有的函数都是内联的。

SV\_POSITION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。

我们可以把函数的返回类型和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。

6.5 像素着色器示例

为了计算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。

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。

cbuffer cbPerObject : register(b0){

float4x4 gWorldViewProj;

}

void VS(float3 iPosL : POSITION, float4 iColor:COLOR, out float4 oPosH : SV\_POSITION, out float4 oColor : COLOR){

oPosH = mul(float4(iPosL, 1.0f), gWorldViewProj);

oColor = iColor;

}

float4 PS(float4 posH : SV\_POSITION, float4 color : COLOR) : SV\_Target{

return color;

}

可以使用结构体参数重写上面示例。

6.6 常量缓冲区

6.6.1 创建常量缓冲区

常量缓冲区也是一种GPU资源，其数据内容可供着色器程序所引用。就像我们在本书中将会学到的纹理等其他类型的缓冲区资源一样，它们都可以被着色器程序所引用。

cbuffer cbPerObject : register(b0){

float4x4 gWorldViewProj;

}

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gWorldViewProj。表示把一个点从局部空间转换到其次裁剪空间所用到的由世界，视图和投影3种变换组合而成的矩阵。

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。

如果摄像机每帧都在不停地移动，那么常量缓冲区也需要每一帧都随之以新的视图矩阵而更新。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。

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。

我们经常用到多个相同类型的常量缓冲区。下列代码展示了如何创建一个缓冲区资源，并利用它来存储NumElements个常量缓冲区。

struct ObjectConstants{

DirectX::XMFLOAT4X4 WorldViewProj = MathHepler::Identity4x4();

}

UINT mElementByteSize = d3dUtil::CalcConstantBufferByteSize(sizeof(ObjectConstants));

ComPtr<ID3D12Resource> mUploadCBuffer;

device->CreateCommittedResource(&CD3DX12\_HEAP\_PROPERTIES(D3D12\_HEAP\_TYPE\_UPLOAD),D3D12\_HEAP\_FLAG\_NONE,&CD3DX12\_RESOURCE\_DESC::Buffer(mElementByteSize \* NumElements),

D3D12\_RESOURCE\_STATE\_GENERIC\_READ,nullptr, IID\_PPV\_ARGS(&mUploadCBuffer));

我们可以认为mUploadCBuffer中存储了一个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。

由于mUploadCBuffer缓冲区存储的是一个常量缓冲区数组，因此，我们把它称为常量缓冲区。

将缓冲区的大小抽成256B整数倍的代码：

UINT d3dUtil::CalcConstantBufferByteSize(UINT byteSize){

return (byteSize + 255) & ~255;

}

随Direct3D 12一同推出的是着色器模型5.1。其中新引进了一条可用于定义常量缓冲区的HLSL语法：

struct ObjectConstants{

float4x4 gWorldViewProj;

uint matIndex;

}

ConstantBuffer<ObjectConstants> gObjConstants : register(b0);

此段代码中，常量缓冲区的数据元素被定义在一个单独的结构体中，随后再用此结构体来创建一个常量缓冲区。

这样一来，我们就可以利用下列获取数据成员的语法，在着色器里访问常量缓冲区中的各字段：

uint index = gObjConstants.matIndex;

6.6.2 更新常量缓冲区

由于常量缓冲区是用D3D12\_HEAP\_TYPE\_UPLOAD这种堆类型来创建的，所以我们就能通过CPU为常量缓冲区资源更新数据。使用Map方法。

ComPtr<ID3D12Resource> mUploadBuffer;

BYTE\* mMappedData = nullptr;

mUploadBuffer->Map(0, nullptr, reinterpret\_cast<void\*\*>(&mMappedData))

(Map各参数含义)

我们利用memcpy函数将数据从系统内存复制到常量缓冲区：

memcpy(mMappedData, &data, dataSizeInBytes);

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：

if(mUploadBuffer != nullptr)

mUploadBuffer->Unmap(0, nullptr)

mMappedData = nullptr;

(Unmap各参数含义)

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地封装一下，使用起来会更加方便。我们在UploadBuffer.h文件中定义了下面这个类。

(UploadBuffer类源码)

(鼠标移动改变各种矩阵的例子)

6.6.4 常量缓冲区描述符

常量缓冲区描述符都要存放在以D3D12\_DESCRIPTOR\_HEAP\_VBC\_SRV\_UAV类型所创建的描述符堆里。这种堆可以混合存储常量缓冲区描述符，着色器资源描述符和无序访问描述符。

(创建描述符堆)

(创建常量缓冲区)

6.6.5 根签名和描述符表

通常来讲，在绘制调用开始执行之前，我们应将不同的着色器程序所需的各类型的资源绑定到渲染流水线上。实际上，不同类型的资源会被绑定到特定的寄存器槽上，以供着色器程序访问。

(例子代码)

根签名定义的是：在执行绘制命令之前，那些应用程序将绑定到渲染流水线上的资源，它们会被映射到着色器的对应输入寄存器。

根签名一定要与使用它的着色器相兼容，在创建流水线状态对象时会对此进行验证。不同的绘制调用可能会用到一组不同的着色器程序，这也就意味着要用到不同的根签名。

在Direct3D中，根签名由ID3D12RootSignature接口来表示，并以一组描述绘制调用过程中着色器所需资源的根参数定义而成。

根参数可以是根常量，根描述符或者描述符表。

本章我们仅使用描述符表，描述符表指定的是描述符堆中存有描述符的一块连续区域。

(例子：创建一个根签名，根参数为一个描述符表)

(代码：创建一个根参数，将含有一个CBV的描述符表绑定到常量缓冲区寄存器0)

根签名只定义了应用程序要绑定到渲染流水线的资源，却没有真正地执行任何资源绑定操作。使用ID3D12GraphicsCommandList::SetGraphicsRootDescriptorTable方法令描述符表与渲染流水线相绑定。

(SetGraphicsRootDescriptorTable各参数含义)

(例子)

6.7 编译着色器

在Direct3D中，着色器程序必须先被编译为一种可移植的字节码。接下来，图形驱动程序将获取这些字节码，并将其重新编译为针对当前系统GPU所优化的本地指令。

我们可以在运行期间用下列函数对着色器进行编译。

(D3DCompileFromFile各参数含义)

为了能够输出错误信息，定义了辅助函数CompileShader。

(CompileShader源码)

仅对着色器进行编译并不会使它与渲染流水线相绑定以供其使用。

6.7.1 离线编译

我们不仅可以在运行期间编译着色器，还能够以单独的步骤离线地编译着色器。

我们通常用.cso作为已编译着色器的扩展名。

为了以离线的方式编译着色器，我们将使用DirectX自带的FXC命令行编译工具。

为了将color.hlsl文件中分别以VS和PS作为入口点的顶点着色器和像素着色器编译为调试版本的字节码。我们输入以下命令：

(代码见书本)

为了将color.hlsl文件中分别以VS和PS作为入口点的顶点着色器和像素着色器编译为发行版本的字节码。我们输入以下命令：

(代码见书本)

(各参数含义见书本表格)

既然已经按离线的方式把顶点着色器和像素着色器编译到.cso文件里，也就不需要在运行时对齐进行编译，无须调用D3DCompileFromFile方法。

但是，我们仍要将.cso文件中已编译好的着色器对象字节码加载到应用程序中，这可以由C++的标准文件输入机制实现。

(代码见书本)

6.7.2 生成着色器汇编代码

FXC程序根据可选参数/Fc来生成可移植的着色器汇编代码。

通过查阅着色器的汇编代码，既可核对着色器的指令数量，也能了解生成的代码细节，这是为了验证编译器所生成的代码与我们预想的是否一致。

有时，查看着色器汇编代码的目的是为了弄清它到底做了什么。

6.7.3 利用Visual Studio离线编译着色器

Visual Studio 2015会识别.hlsl文件并提供编译选项，这些UI中配置选项就是FXC程序的参数。

VS集成的HLSL工具有一个缺点，只允许每个文件中仅有一个着色器程序。因此，这条限制将令顶点着色器和像素着色器不能共存于一个文件里。

此外，有时我们希望以不同的预处理指令编译同一个着色器程序，从而获取同一个着色器的不同编译结果，也是无法做到的。

6.8 光栅器状态

如今渲染流水线中的大多数阶段都是可编程的，但是有些特定环节却只能接受配置。例如，用于配置渲染流水线中光栅化阶段的光栅器状态组由结构体D3D12\_RASTERIZER\_DESC表示。

(D3D12\_RASTERIZER\_DESC变量含义)

CD3D12\_RASTERIZER\_DESC是在扩展自D3D12\_RASTERIZER\_DESC结构体的基础上，又添加了一些辅助构造函数的工具类。

6.9 流水线状态对象

大多数控制图形流水线状态的对象被统称为流水线状态对象(PSO)，用ID3D12PipelineState接口来表示。

要创建PSO，我们首先要填写一份描述其细节的D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC结构体实例。

(D3D12\_GRAPHICS\_PIPELINE\_STATE\_DESC变量含义)

(实例)

并非所有的渲染状态都封装于PSO内，如视口和裁剪矩形等属性。

Direct3D实质上就是一种状态机，里面的事物会保持它们各自状态，知道我们将其改变。

如果把一个PSO与命令列表相绑定，那么，在我们设置另一个PSO或充值命令列表之前，会一直沿用当前的PSO绘制物体。

6.10 几何图形辅助结构体

我们通过创建一个同时存有顶点缓冲区和索引缓冲区的结构体来方便地定义多个几何体。

另外借此结构体，可以将顶点和索引数据置于系统内存中，以供CPU读取。例如，执行拾取和碰撞检测等工作。

当需要定义多个几何体时，使用MeshGeometry(定义于d3dUtil.h头文件中)结构体。

(MeshGeometry源码)

6.11 立方体演示程序