环境搭建

首次完全自己使用C++来编写D3D11的Demo，在搭建环境时遇见很多挫折，因为龙书Demo是使用以前的版本，而我这里用的是VS2017，很多地方不一样，所以记录下来。

1、D3DX库在win8.1SDK以上已经自带，所以我们不用单独下载SDK了。

2、直接包含 <d3d11.h> 并且链接时加入 d3d11.lib 就可以编写D3D11的程序了。

3、以前是用 XnaMath库，现在改成 DirectXMath 这个算法库也是肯定要用的，而且多了一个namespace DirectX，如果用到了XMColor 还需要 #include DirectXPackedVector.h ，并且名称空间是DirectX::PackedVector

4、以前用的 dxerr.lib 和VS2017 不兼容了，所以我选择的方式是直接拷贝的源码 dxerr.h 和dxerr.cpp 加入到工程中，但是还是会报错，必须还要 #pragma comment(lib, "legacy\_stdio\_definitions.lib") 具体原因见：<https://blogs.msdn.microsoft.com/chuckw/2012/04/24/wheres-dxerr-lib/>，或者根本不要用这个库，代码中的HR 宏里面用到了 DXTrace 自己改成其他提示。

5、Effects11 库已经单独开源，这个库虽然官方不推荐了，但是我们学习的例子全部是用的这个库，所以还是先用着吧，开源地址：<https://github.com/Microsoft/FX11>

6、DirectXTK 库也是要用的，这个库功能很多是一个工具集，包括载入DDS 纹理、SimpleMath等等

还有些额外的插曲，本来我为了更容易些HLSL着色器代码（本来自己英文也不好），所以找了一下可以智能提示的插件，结果弄了半天我写C++代码反而 Visual Assist X 会报错了，弄了很久才发现是和那个着色器插件有冲突，害得我还重装了VS2017。

整体流程

1、创建顶点缓冲区和索引缓冲区

2、编译或载入着色器代码（Fx）

3、创建输入布局

4、清除渲染视图和深度缓冲区视图

5、设置输入布局 和 图元拓扑

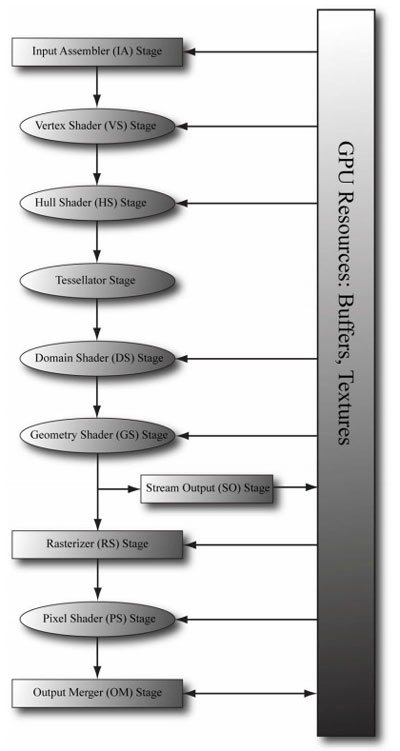
6、设置顶点和索引缓冲区

7、设置着色器的常量缓冲区（这里通常会有矩阵、光源、材质 和 纹理资源等）

8、最后根据 tech 获取描述，循环每一个pass 然后应用再画

D3D11的渲染管线如下图所示，IA（输入装配），VS（顶点着色），HS、TS和DS （三个阶段为曲面细分阶段），GS（计算着色），RS（光栅化）、PS（像素着色）、OM输出合并。

所以D3DdeviceContext 接口的方法都有很多这些字母开头的方法，分别代表不同阶段。



着色器代码为HLSL代码，这个语言的语法相对比较简单，也是类C的语言。

最基本需要 VS 和 PS 两个阶段的代码，可以看成两个C++方法， 比如顶点着色器VS,我们输入的数据顶点缓冲区， 然后通常通过一些矩阵变换，将顶点变换为需要的坐标系，比如我们输入的是本地坐标，转换成世界坐标，最后返回新的顶点数据，这些顶点数据会传递到下一个阶段的着色器代码，当然其他的都可以忽略，但是最少还需要一个像素着色器PS。

光照模型

LightHelper C++里面定义了三种光源的结构体，着色器代码LightHelper.fx 里面也定义了相应的结构体，C++里面注意需要和着色器代码的结构体字节对齐。

LightHelper.fx 提供了三种光源的计算，通过参数返回出计算结果，将计算结果相加并且透明通道设置为材质的漫反射光的透明通道，最后返回这个颜色，HLSL关键代码

pin.NormalW = normalize(pin.NormalW); //归一化法线

float3 toEye = gEyePosW - pin.PosW; // toEye 在计算光源时参数需要

float distToEye = length(toEye); //表面到观察点距离，这个距离其实后面才会用到，本来可以直接归一化 toEye

toEye /= distToEye; //归一化

//初始化值

float4 ambient = float4(0.0f, 0.0f, 0.0f, 0.0f);

float4 diffuse = float4(0.0f, 0.0f, 0.0f, 0.0f);

float4 spec = float4(0.0f, 0.0f, 0.0f, 0.0f);

//开始计算

float4 A, D, S;

ComputeDirectionalLight(gMaterial, gDirLights[i], pin.NormalW, toEye, A, D, S);

ambient += A;

diffuse += D;

spec += S;

//最后合并返回

float4 litColor = ambient + diffuse + spec;

litColor.a = gMaterial.Diffuse.a;

return litColor;

纹理采样

顶点的颜色直接从一幅图片采样，整个图片左上被视为0,0, 右下为1，1 ，然后每个顶点都需要一个纹理坐标，表示你是采样图片哪个部分的颜色。HLSL关键代码：

Texture2D gDiffuseMap; //声明2D纹理

// 设置好采样状态，如何采样，这里是用的各项异性

SamplerState samAnisotropic

{

Filter = ANISOTROPIC; //各项异性

MaxAnisotropy = 4; //设置为各项异性，还必须设置这个最大值，这个值介于1至16之间。大数值需要耗费更多的资源，但效果更好

AddressU = WRAP; //寻找模式为重复

AddressV = WRAP;

};

texColor = gDiffuseMap.Sample( samAnisotropic, pin.Tex ); // 这里就采样到颜色了，当然后面我们还要把这个颜色融入到光照模型中。

//还是之前的光源计算

float4 A, D, S;

ComputeDirectionalLight(gMaterial, gDirLights[i], pin.NormalW, toEye, A, D, S);

ambient += A;

diffuse += D;

spec += S;

//下面是关键，纹理颜色和光照模型合并，纹理颜色\* (环境光 + 漫反射光) + 高光

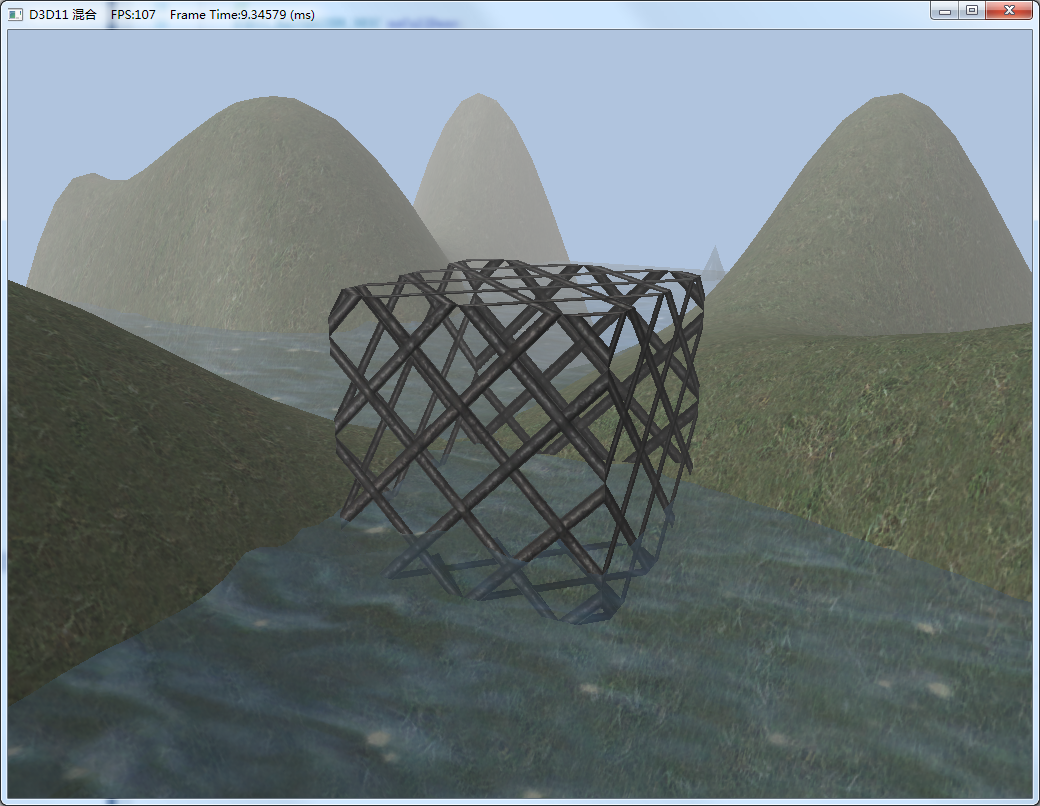
litColor = texColor\*(ambient + diffuse) + spec;

//透明通道也是要多乘以一个纹理的透明通道

litColor.a = gMaterial.Diffuse.a \* texColor.a;

混合

首先看看截图：



这里的水是透明的，就是因为水是最后画的，并且画水之前设置了混合状态

m\_pImmediateContext->OMSetBlendState(RenderStates::TransparentBS, blendFactor, 0xffffffff);

创建混合的代码

D3D11\_BLEND\_DESC transparentDesc = {0};

transparentDesc.AlphaToCoverageEnable = false; //一种多重采样技术

transparentDesc.IndependentBlendEnable = false; // 是否启用独立混合，如果是false 那么RenderTarget[0]有效

transparentDesc.RenderTarget[0].BlendEnable = true;

transparentDesc.RenderTarget[0].SrcBlend = D3D11\_BLEND\_SRC\_ALPHA;

transparentDesc.RenderTarget[0].DestBlend = D3D11\_BLEND\_INV\_SRC\_ALPHA;

transparentDesc.RenderTarget[0].BlendOp = D3D11\_BLEND\_OP\_ADD;

transparentDesc.RenderTarget[0].SrcBlendAlpha = D3D11\_BLEND\_ONE;

transparentDesc.RenderTarget[0].DestBlendAlpha = D3D11\_BLEND\_ZERO;

transparentDesc.RenderTarget[0].BlendOpAlpha = D3D11\_BLEND\_OP\_ADD;

transparentDesc.RenderTarget[0].RenderTargetWriteMask = D3D11\_COLOR\_WRITE\_ENABLE\_ALL;

HR(device->CreateBlendState(&transparentDesc, &TransparentBS));

另外一个知识点，就是我们的笼子是如何透明的呢？这个是在HLSL代码中当我们采样完了笼子纹理之后

texColor = gDiffuseMap.Sample( samAnisotropic, pin.Tex );

if(gAlphaClip)

{

clip(texColor.a - 0.1f);

}

clip() 函数会将把参数小于0的颜色丢弃，这里用了 texColor.a - 0.1f 那么如果texColor.a < 0.1 则得到的结果就小于0

最后就是雾的效果，HLSL关键代码：

if( gFogEnabled )

{

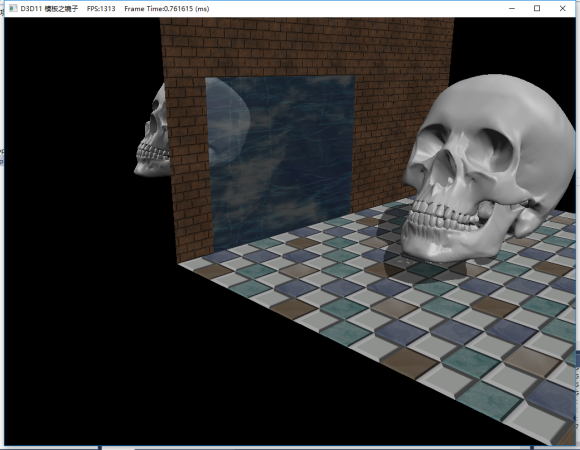
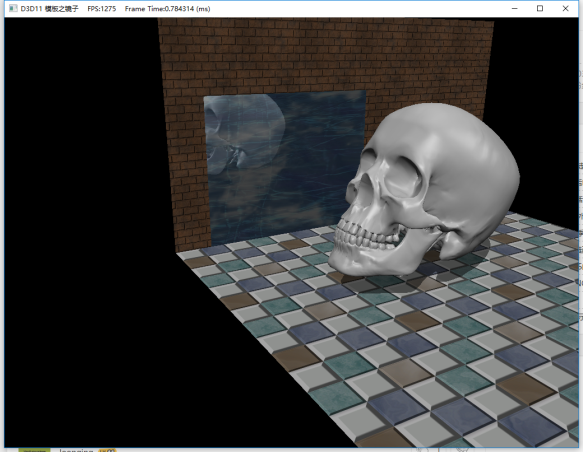
float fogLerp = saturate( (distToEye - gFogStart) / gFogRange ); //获取雾的线性插值

litColor = lerp(litColor, gFogColor, fogLerp); //线性插值算法混合光源颜色

}

模板状态

好比在现实生活中，用喷漆在墙上喷文字，我们会一张纸来挡住，让喷漆只喷到纸镂空的部分。所以我们在D3D11中也是类似这个过程，先看截图



镜子里面的骷髅头，实际是我们计算出一个和正常骷髅反射的一个矩阵坐标，再画一次骷髅才显示出来的。但是如果不做模板处理会很奇怪，如图2所示。

知道作用后这里记录下关键代码

1、生成一个模板

D3D11\_BLEND\_DESC noRenderTargetWritesDesc = { 0 };

noRenderTargetWritesDesc.AlphaToCoverageEnable = false;

noRenderTargetWritesDesc.IndependentBlendEnable = false;

noRenderTargetWritesDesc.RenderTarget[0].BlendEnable = false;

noRenderTargetWritesDesc.RenderTarget[0].SrcBlend = D3D11\_BLEND\_ONE;

noRenderTargetWritesDesc.RenderTarget[0].DestBlend = D3D11\_BLEND\_ZERO;

noRenderTargetWritesDesc.RenderTarget[0].BlendOp = D3D11\_BLEND\_OP\_ADD;

noRenderTargetWritesDesc.RenderTarget[0].SrcBlendAlpha = D3D11\_BLEND\_ONE;

noRenderTargetWritesDesc.RenderTarget[0].DestBlendAlpha = D3D11\_BLEND\_ZERO;

noRenderTargetWritesDesc.RenderTarget[0].BlendOpAlpha = D3D11\_BLEND\_OP\_ADD;

noRenderTargetWritesDesc.RenderTarget[0].RenderTargetWriteMask = 0;

HR(device->CreateBlendState(&noRenderTargetWritesDesc, &NoRenderTargetWritesBS));

m\_pImmediateContext->OMSetBlendState(RenderStates::NoRenderTargetWritesBS, blendFactor, 0xffffffff);

先设置为无渲染目标写入的混合状态

D3D11\_DEPTH\_STENCIL\_DESC mirrorDesc;

mirrorDesc.DepthEnable = true;

mirrorDesc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ZERO;

mirrorDesc.DepthFunc = D3D11\_COMPARISON\_LESS;

mirrorDesc.StencilEnable = true;

mirrorDesc.StencilReadMask = 0xff;

mirrorDesc.StencilWriteMask = 0xff;

mirrorDesc.FrontFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

mirrorDesc.FrontFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_KEEP;

mirrorDesc.FrontFace.StencilPassOp = D3D11\_STENCIL\_OP\_REPLACE;

mirrorDesc.FrontFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

//我们不渲染背面多边形，所以这些设置无关紧要。

mirrorDesc.BackFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

mirrorDesc.BackFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_KEEP;

mirrorDesc.BackFace.StencilPassOp = D3D11\_STENCIL\_OP\_REPLACE;

mirrorDesc.BackFace.StencilFunc = D3D11\_COMPARISON\_ALWAYS;

HR(device->CreateDepthStencilState(&mirrorDesc, &MarkMirrorDSS));

m\_pImmediateContext->OMSetDepthStencilState(RenderStates::MarkMirrorDSS, 1);

再设置深度模板状态为不写入深度缓冲区，但是要写入模板缓冲区。

2、开始画镜子，这个时候因为设置为了没有渲染目标写入，所以只会写入模板缓冲区。

3、那么这个模板就设计好了。然后我们就可以画反射的骷髅头了

XMVECTOR mirrorPlane = XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f); // xy plane

XMMATRIX R = XMMatrixReflect(mirrorPlane);

XMMATRIX world = XMLoadFloat4x4(&mSkullWorld) \* R;

上面是计算反射的矩阵。

XMFLOAT3 oldLightDirections[3];

for (int i = 0; i < 3; ++i)

{

oldLightDirections[i] = mDirLights[i].Direction;

XMVECTOR lightDir = XMLoadFloat3(&mDirLights[i].Direction);

XMVECTOR reflectedLightDir = XMVector3TransformNormal(lightDir, R);

XMStoreFloat3(&mDirLights[i].Direction, reflectedLightDir);

}

上面是保存之前的光源方向，并设置为镜像的光源方向

4、最后再画上正常的镜子，但是这个时候我们要加上混合来画，才能把骷髅透明出来。

D3D11\_BLEND\_DESC transparentDesc = { 0 };

transparentDesc.AlphaToCoverageEnable = false;

transparentDesc.IndependentBlendEnable = false;

transparentDesc.RenderTarget[0].BlendEnable = true;

transparentDesc.RenderTarget[0].SrcBlend = D3D11\_BLEND\_SRC\_ALPHA;

transparentDesc.RenderTarget[0].DestBlend = D3D11\_BLEND\_INV\_SRC\_ALPHA;

transparentDesc.RenderTarget[0].BlendOp = D3D11\_BLEND\_OP\_ADD;

transparentDesc.RenderTarget[0].SrcBlendAlpha = D3D11\_BLEND\_ONE;

transparentDesc.RenderTarget[0].DestBlendAlpha = D3D11\_BLEND\_ZERO;

transparentDesc.RenderTarget[0].BlendOpAlpha = D3D11\_BLEND\_OP\_ADD;

transparentDesc.RenderTarget[0].RenderTargetWriteMask = D3D11\_COLOR\_WRITE\_ENABLE\_ALL;

HR(device->CreateBlendState(&transparentDesc, &TransparentBS));

m\_pImmediateContext->OMSetBlendState(RenderStates::TransparentBS, blendFactor, 0xffffffff);

在这个例子中随便还颜色了平面阴影的效果，可以看见骷髅有阴影的，但是这个阴影不会投射到墙上，所以叫平面阴影效果，具体实现关键代码：

XMVECTOR shadowPlane = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f); // xz plane

XMVECTOR toMainLight = -XMLoadFloat3(&mDirLights[0].Direction);

XMMATRIX S = XMMatrixShadow(shadowPlane, toMainLight);

XMMATRIX shadowOffsetY = XMMatrixTranslation(0.0f, 0.001f, 0.0f); //比地面高一点点

XMMATRIX world = XMLoadFloat4x4(&mSkullWorld)\*S\*shadowOffsetY;

上面是计算阴影的矩阵

D3D11\_DEPTH\_STENCIL\_DESC noDoubleBlendDesc;

noDoubleBlendDesc.DepthEnable = true;

noDoubleBlendDesc.DepthWriteMask = D3D11\_DEPTH\_WRITE\_MASK\_ALL;

noDoubleBlendDesc.DepthFunc = D3D11\_COMPARISON\_LESS;

noDoubleBlendDesc.StencilEnable = true;

noDoubleBlendDesc.StencilReadMask = 0xff;

noDoubleBlendDesc.StencilWriteMask = 0xff;

noDoubleBlendDesc.FrontFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

noDoubleBlendDesc.FrontFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_KEEP;

noDoubleBlendDesc.FrontFace.StencilPassOp = D3D11\_STENCIL\_OP\_INCR;

noDoubleBlendDesc.FrontFace.StencilFunc = D3D11\_COMPARISON\_EQUAL;

//我们不渲染背面多边形，所以这些设置无关紧要。

noDoubleBlendDesc.BackFace.StencilFailOp = D3D11\_STENCIL\_OP\_KEEP;

noDoubleBlendDesc.BackFace.StencilDepthFailOp = D3D11\_STENCIL\_OP\_KEEP;

noDoubleBlendDesc.BackFace.StencilPassOp = D3D11\_STENCIL\_OP\_INCR;

noDoubleBlendDesc.BackFace.StencilFunc = D3D11\_COMPARISON\_EQUAL;

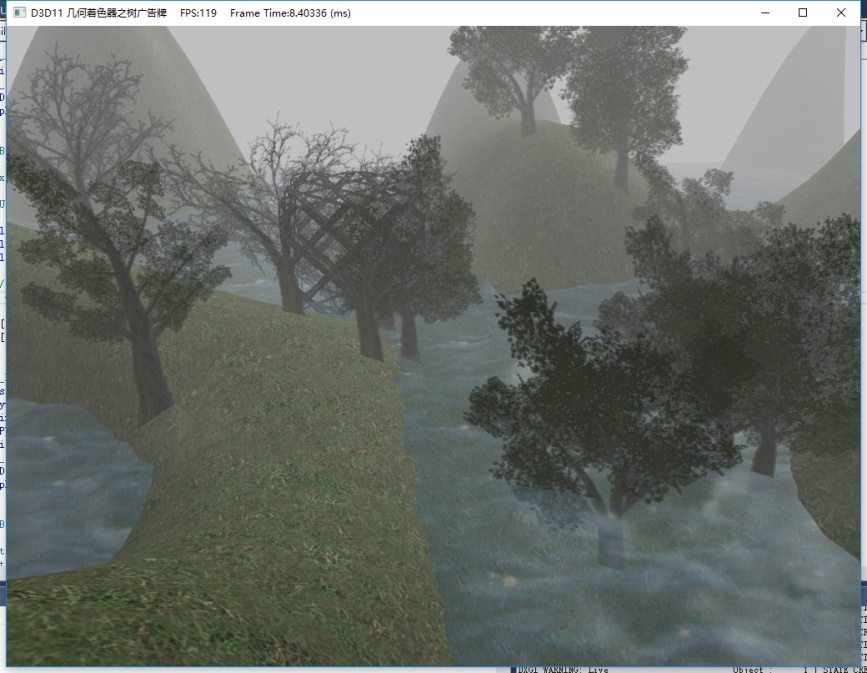
HR(device->CreateDepthStencilState(&noDoubleBlendDesc, &NoDoubleBlendDSS));

m\_pImmediateContext->OMSetDepthStencilState(RenderStates::NoDoubleBlendDSS, 0);

上面是设置深度模板状态为不双重混合

几何着色器

例子是树广告牌，效果就是一个平面始终面向摄像机



关键的HLSL代码：

[maxvertexcount(4)]

void GS(point VertexOut gin[1], uint primID : SV\_PrimitiveID, inout TriangleStream<GeoOut> triStream)

maxvertexcount 是最大输出的顶点最大数，这里是4

第一个参数是point 代表输入的图元是点，它总是一个数组，具体输入类型有：

**1．point**：输入图元为点。

**2．line**：输入图元为直线（列表或线带）。

**3．triangle**：输入图元为三角形（列表或线带）。

**4．lineadj**：输入图元为带有邻接信息的直线（列表或线带）。

**5．triangleadj**：输入图元为带有邻接信息的三角形（列表或线带）。

第三个参数是一个流的输出参数，用triStream.Append() 添加进去，总是用inout 标记，流的类型有：

**1．PointStream<OutputVertexType>**：描述单个点的顶点列表。

**2．LineStream<OutputVertexType>**：描述直线线带的顶点列表。

**3．TriangleStream<OutputVertexType>**：描述三角形线带的顶点列表

第二个参数是可选的图元ID，就是一个索引从0开始。

[maxvertexcount(4)]

void GS(point VertexOut gin[1],

uint primID : SV\_PrimitiveID,

inout TriangleStream<GeoOut> triStream)

{

// 计算精灵相对于世界的本地坐标系

// 使广告牌与y轴对齐并面向摄像机。

float3 up = float3(0.0f, 1.0f, 0.0f);

float3 look = gEyePosW - gin[0].CenterW;

look.y = 0.0f; // y轴对齐，所以投影到xz平面

look = normalize(look);

float3 right = cross(up, look);

// 计算世界空间中的三角形条带顶点（四边形）。

float halfWidth = 0.5f\*gin[0].SizeW.x;

float halfHeight = 0.5f\*gin[0].SizeW.y;

float4 v[4];

v[0] = float4(gin[0].CenterW + halfWidth\*right - halfHeight\*up, 1.0f);

v[1] = float4(gin[0].CenterW + halfWidth\*right + halfHeight\*up, 1.0f);

v[2] = float4(gin[0].CenterW - halfWidth\*right - halfHeight\*up, 1.0f);

v[3] = float4(gin[0].CenterW - halfWidth\*right + halfHeight\*up, 1.0f);

//

// 将四个顶点转换为世界空间并输出

// 他们作为三角形带。

//

GeoOut gout;

[unroll]

for(int i = 0; i < 4; ++i)

{

gout.PosH = mul(v[i], gViewProj);

gout.PosW = v[i].xyz;

gout.NormalW = look;

gout.Tex = gTexC[i];

gout.PrimID = primID;

triStream.Append(gout);

}

}

当然后面的像素着色器 flaot4 PS（GeoOut pin）就输入参数就是几何着色器传递过来的。

计算着色器

示例是模糊，先看截图：



计算着色器的计算能力和处理器数量有关，比如如果你的GPU有16个多处理器，那么就可以开启16个线程组。每个线程组，可以有n个线程。

：

Texture2D gInput;

RWTexture2D<float4> gOutput;

#define N 256

#define CacheSize (N + 2\*gBlurRadius)

groupshared float4 gCache[CacheSize];

//numthreads(N, 1, 1)，代表着线程的数量，可以是多维的

//groupThreadID，线程在线程组中的ID

//dispatchThreadID，在一次dispatch线程中的ID

//水平模糊

[numthreads(N, 1, 1)]

void HorzBlurCS(int3 groupThreadID : SV\_GroupThreadID,

int3 dispatchThreadID : SV\_DispatchThreadID)

{

...

}

再看C++代码

D3D11\_TEXTURE2D\_DESC blurredTexDesc;

blurredTexDesc.Width = width;

blurredTexDesc.Height = height;

blurredTexDesc.MipLevels = 1;

blurredTexDesc.ArraySize = 1;

blurredTexDesc.Format = format;

blurredTexDesc.SampleDesc.Count = 1;

blurredTexDesc.SampleDesc.Quality = 0;

blurredTexDesc.Usage = D3D11\_USAGE\_DEFAULT;

blurredTexDesc.BindFlags = D3D11\_BIND\_SHADER\_RESOURCE | D3D11\_BIND\_UNORDERED\_ACCESS;

blurredTexDesc.CPUAccessFlags = 0;

blurredTexDesc.MiscFlags = 0;

ID3D11Texture2D\* blurredTex = 0;

HR(device->CreateTexture2D(&blurredTexDesc, 0, &blurredTex));

D3D11\_SHADER\_RESOURCE\_VIEW\_DESC srvDesc;

srvDesc.Format = format;

srvDesc.ViewDimension = D3D11\_SRV\_DIMENSION\_TEXTURE2D;

srvDesc.Texture2D.MostDetailedMip = 0;

srvDesc.Texture2D.MipLevels = 1;

HR(device->CreateShaderResourceView(blurredTex, &srvDesc, &mBlurredOutputTexSRV));

D3D11\_UNORDERED\_ACCESS\_VIEW\_DESC uavDesc;

uavDesc.Format = format;

uavDesc.ViewDimension = D3D11\_UAV\_DIMENSION\_TEXTURE2D;

uavDesc.Texture2D.MipSlice = 0;

HR(device->CreateUnorderedAccessView(blurredTex, &uavDesc, &mBlurredOutputTexUAV));

// 视图保存对纹理的引用，以便我们可以释放我们的引用。

ReleaseCOM(blurredTex);

用一个2D纹理资源，分别创建了着色器资源视图和无序访问视图，一个用于输入，一个用于输出。当然这两个是属于中转的，因为他们都是同一个资源，不能即时输入又是输出。

Effects::BlurFX->SetInputMap(inputSRV);

Effects::BlurFX->SetOutputMap(mBlurredOutputTexUAV);

Effects::BlurFX->HorzBlurTech->GetPassByIndex(p)->Apply(0, dc);

// 我们需要分派多少组以覆盖一行像素，其中每行都是

// 组涵盖了256个像素（256是在ComputeShader中定义的）。

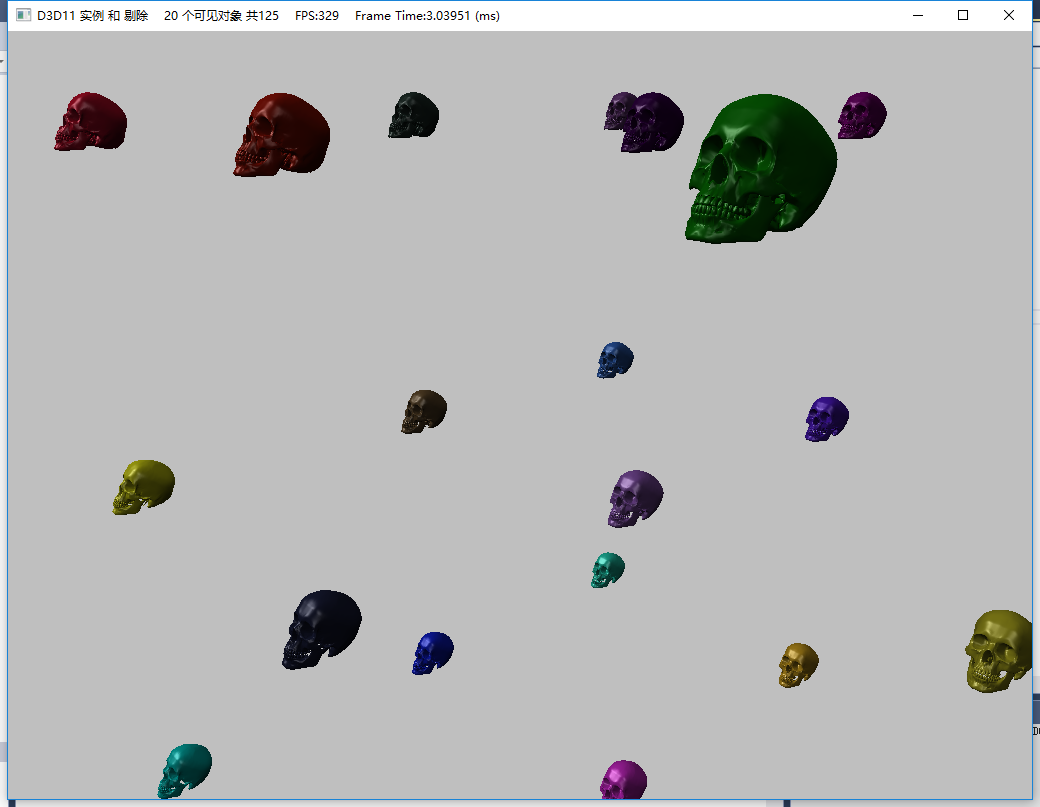
UINT numGroupsX = (UINT)ceilf(mWidth / 256.0f);

dc->Dispatch(numGroupsX, mHeight, 1);

最终调用Dispatch 方法，就开始计算了。

实例和剔除

我们在画多个相同物体时，应该使用实例的方式，首看截图



例子中我们首先定义了实例的结构体

struct InstancedData

{

XMFLOAT4X4 World;

XMFLOAT4 Color;

};

第一步：

创建实例的顶点缓冲区，也就是如果我们有100个实例，就有100个上面的结构体元素的一个缓冲区，和之前的缓冲区一样，m\_pD3dDevice->CreateBuffer() ，就搞定了，当然我们这次保存了一份变量mInstancedData，这个主要用于后面的剔除工作。

第二步：

当然骷髅本身的顶点缓冲区我们还是要一份的，和之前一样就不说了

第三步：

画的时候用 m\_pImmediateContext->DrawIndexedInstanced(mSkullIndexCount, mVisibleObjectCount, 0, 0, 0);

当然我们还要看看着色器代码的不同

struct VertexIn

{

float3 PosL : POSITION;

float3 NormalL : NORMAL;

float2 Tex : TEXCOORD;

row\_major float4x4 World : WORLD;

float4 Color : COLOR;

uint InstanceId : SV\_InstanceID; //这里我们的InputLayout并没有设置这个元素，说明这个是系统设置的应该是每个实例的ID标识。

};

struct VertexOut

{

float4 PosH : SV\_POSITION;

float3 PosW : POSITION;

float3 NormalW : NORMAL;

float2 Tex : TEXCOORD;

float4 Color : COLOR;

};

VertexOut VS(VertexIn vin)

{

……

vout.PosH = mul(float4(vout.PosW, 1.0f), gViewProj); //这里用的 vout.PosW，也就是每个实例不同的位置

vout.Color = vin.Color;

}

当然像素着色器为了使用不同的颜色也在光照计算中有所不同

float4 A, D, S;

ComputeDirectionalLight(gMaterial, gDirLights[i], pin.NormalW, toEye, A, D, S);

ambient += A\*pin.Color; //环境光 和 实例的颜色 相乘

diffuse += D\*pin.Color; //漫反射光同上

spec += S;

另外一个知识点就是剔除，也就是我们仅仅需要画出能显示在投影范围之内的实例，这个就要用到一些测试方式了，

首先要包含 #include <DirectXCollision.h>

其次我们要定义骷髅和摄像机的边缘

BoundingBox mSkullBox;

BoundingFrustum mCamFrustum;

在窗口尺寸被改变时创建视图空间中构建投影矩阵的平截头体。

BoundingFrustum::CreateFromMatrix(mCamFrustum, mCam.Proj());

然后在每次画之前做一些测试，判断实例的坐标是否在摄像机范围内

//得到摄像机视图逆矩阵

XMVECTOR detView = XMMatrixDeterminant(mCam.View());

XMMATRIX invView = XMMatrixInverse(&detView, mCam.View());

//得到实例的逆矩阵

XMMATRIX W = XMLoadFloat4x4(&mInstancedData[i].World);

XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(W), W);

// 视图空间到对象本地空间

XMMATRIX toLocal = XMMatrixMultiply(invView, invWorld);

// 将矩阵分解为各个部分。

XMVECTOR scale; //缩放

XMVECTOR rotQuat; //旋转

XMVECTOR translation; //平移

XMMatrixDecompose(&scale, &rotQuat, &translation, toLocal);

// 将相机视锥体从视图空间转换为对象的本地空间。

BoundingFrustum localspaceFrustum;

mCamFrustum.Transform(localspaceFrustum, XMVectorGetX(scale), rotQuat, translation);

// 在本地空间中执行立方体/锥体相交测试。

if(localspaceFrustum.Intersects(mSkullBox))

{

/ 将实例数据写入可见对象的动态顶点缓冲区中。

dataView[mVisibleObjectCount++] = mInstancedData[i];

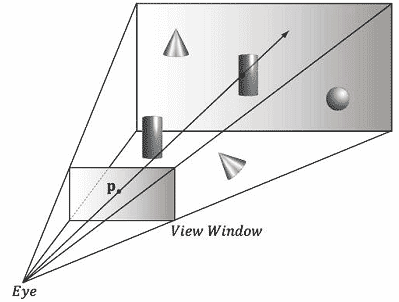
}

当然上面这一切是在我们的缓冲区运行CPU读写时，才能在CPU端动态改变缓冲区数据，通过

m\_pImmediateContext->Map 就可以更新数据了。

拾取

拾取技术，就是将2d屏幕位置（以像素为单位，鼠标所在的位置）转换为世界空间中的3d射线，随后检测该射线是否会和屏幕上的其他物体相交。



void PickingDemo::Pick(int sx, int sy)

{

XMMATRIX P = mCam.Proj();

XMFLOAT4X4 fp;

XMStoreFloat4x4(&fp, P);

// 在视图空间中计算拾取。

float vx = (+2.0f\*sx / m\_clientWidth - 1.0f) / fp.\_11;

float vy = (-2.0f\*sy / m\_clientHeight + 1.0f) / fp.\_22;

// 射线在视图空间中的定义

XMVECTOR rayOrigin = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f); //射线起点

XMVECTOR rayDir = XMVectorSet(vx, vy, 1.0f, 0.0f); //射线方向

// 转换射线到本地空间

XMMATRIX V = mCam.View();

XMMATRIX invView = XMMatrixInverse(&XMMatrixDeterminant(V), V);

XMMATRIX W = XMLoadFloat4x4(&mMeshWorld);

XMMATRIX invWorld = XMMatrixInverse(&XMMatrixDeterminant(W), W);

XMMATRIX toLocal = XMMatrixMultiply(invView, invWorld);

rayOrigin = XMVector3TransformCoord(rayOrigin, toLocal);

rayDir = XMVector3TransformNormal(rayDir, toLocal);

// 为相交测试制作射线方向单位长度。

rayDir = XMVector3Normalize(rayDir);

// 如果我们点击网格的边界框，那么我们可能会选择一个网格三角形，

// 射线/三角测试也是如此。

//

// 如果我们没有碰到边界框，那么我们不可能碰到

// 网格，所以不要浪费精力进行射线/三角测试。

// 假设我们还没有选择任何东西，所以初始化为-1。

mPickedTriangle = -1;

float tmin = 0.0f;

if (mMeshBox.Intersects(rayOrigin, rayDir, tmin)) //在网格内

{

// 找到最近的射线/三角形交叉点。

tmin = MathHelper::Infinity;

for (UINT i = 0; i < mMeshIndices.size() / 3; ++i)

{

// 三角形索引

UINT i0 = mMeshIndices[i \* 3 + 0];

UINT i1 = mMeshIndices[i \* 3 + 1];

UINT i2 = mMeshIndices[i \* 3 + 2];

// 三角形顶点

XMVECTOR v0 = XMLoadFloat3(&mMeshVertices[i0].Pos);

XMVECTOR v1 = XMLoadFloat3(&mMeshVertices[i1].Pos);

XMVECTOR v2 = XMLoadFloat3(&mMeshVertices[i2].Pos);

// 我们必须迭代所有三角形才能找到最近的交点。

float t = 0.0f;

if (TriangleTests::Intersects(rayOrigin, rayDir, v0, v1, v2, t)) //射线是否和三角形相交

{

if (t < tmin)

{

// 这是最近的选择三角形。

tmin = t;

mPickedTriangle = i;

}

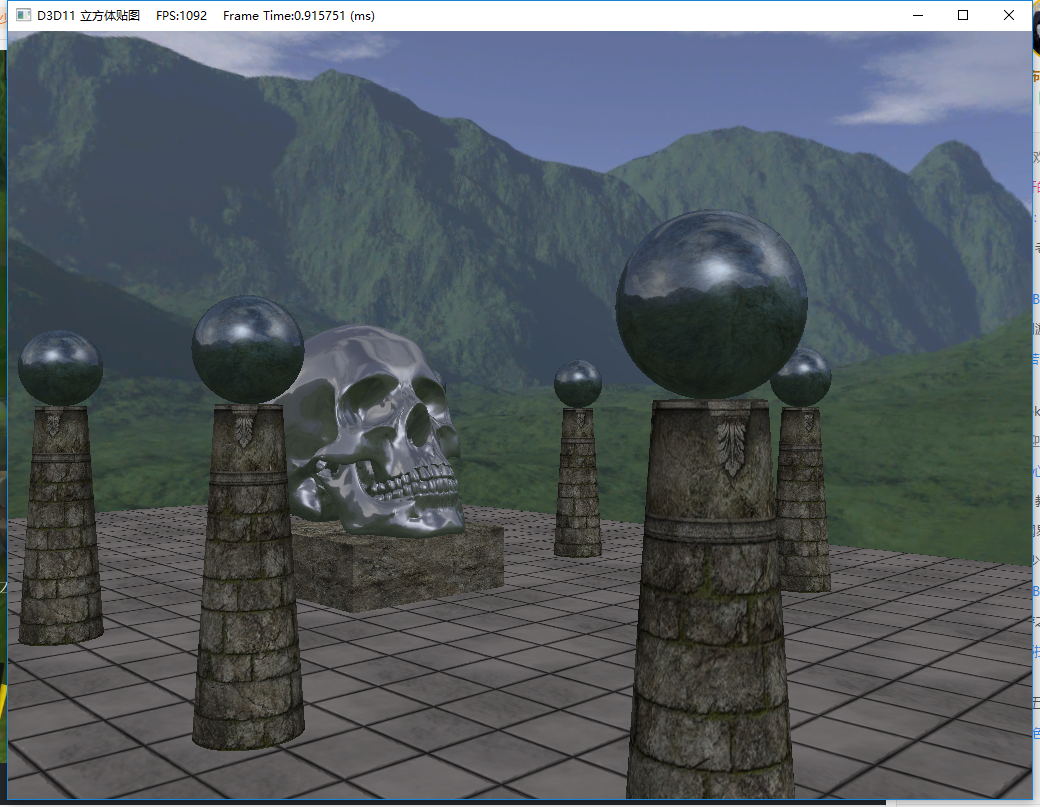
}

}

}

}

立方体贴图



周围的环境其实就是一个很大的球体，而我们画的这些圆柱、骷髅等东西就在这个球体里面。我们还是需要一幅图片，不过这个图片有6个面，所以叫立方体贴图。重点就是在着色器代码上：

TextureCube gCubeMap; //声明立方体着色器资源

VertexOut VS(VertexIn vin)

{

VertexOut vout;

// 设置 z = w 所以 z/w = 1 (即天空苍穹总是在远平面).

vout.PosH = mul(float4(vin.PosL, 1.0f), gWorldViewProj).xyww;

// 使用本地顶点位置作为立方体贴图查找向量。

vout.PosL = vin.PosL;

return vout;

}

float4 PS(VertexOut pin) : SV\_Target

{

return gCubeMap.Sample(samTriLinearSam, pin.PosL); //采样用本地坐标空间

}

当然我们在一个超大球体里面，所以还要设置两个状态，这状态可以在FX里面设置，也可以C++里面设置

RasterizerState NoCull

{

CullMode = None;

};

DepthStencilState LessEqualDSS

{

// 确保深度函数是<=而不仅仅是<

// 否则，如果深度缓冲区被清除，在z = 1（NDC）处的归一化深度值将会为1

// 则深度测试失败。

DepthFunc = LESS\_EQUAL;

};

另外一个知识点就是反射立方体贴图，如截图里面的球体，本身已经有了自己的纹理，然后还要反射立方体贴图，关键HLSL代码，在计算三种光源的循环中：

if( gReflectionEnabled )

{

float3 incident = -toEye;

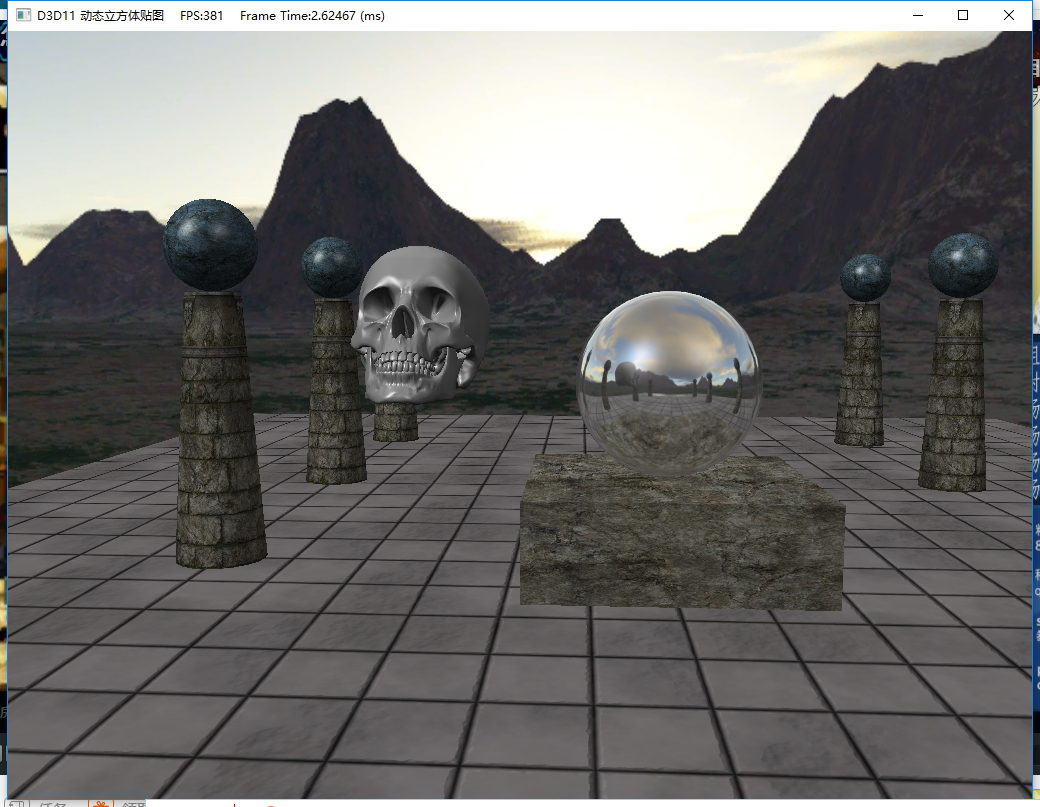
float3 reflectionVector = reflect(incident, pin.NormalW);

float4 reflectionColor = gCubeMap.Sample(samAnisotropic, reflectionVector);

litColor += gMaterial.Reflect \* reflectionColor;

}

动态立方体贴图



首先我们要创建一个6个元素的数组纹理资源

D3D11\_TEXTURE2D\_DESC texDesc;

texDesc.Width = CubeMapSize;

texDesc.Height = CubeMapSize;

texDesc.MipLevels = 0;

texDesc.ArraySize = 6;

texDesc.SampleDesc.Count = 1;

texDesc.SampleDesc.Quality = 0;

texDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;

texDesc.Usage = D3D11\_USAGE\_DEFAULT;

texDesc.BindFlags = D3D11\_BIND\_RENDER\_TARGET | D3D11\_BIND\_SHADER\_RESOURCE;

texDesc.CPUAccessFlags = 0;

texDesc.MiscFlags = D3D11\_RESOURCE\_MISC\_GENERATE\_MIPS | D3D11\_RESOURCE\_MISC\_TEXTURECUBE;

ID3D11Texture2D\* cubeTex = 0;

HR(m\_pD3dDevice->CreateTexture2D(&texDesc, 0, &cubeTex));

// 为每个立方体贴图面创建一个渲染目标视图（即纹理数组中的每个元素）。

D3D11\_RENDER\_TARGET\_VIEW\_DESC rtvDesc;

rtvDesc.Format = texDesc.Format;

rtvDesc.ViewDimension = D3D11\_RTV\_DIMENSION\_TEXTURE2DARRAY;

rtvDesc.Texture2DArray.ArraySize = 1;

rtvDesc.Texture2DArray.MipSlice = 0;

for (int i = 0; i < 6; ++i)

{

rtvDesc.Texture2DArray.FirstArraySlice = i;

HR(m\_pD3dDevice->CreateRenderTargetView(cubeTex, &rtvDesc, &mDynamicCubeMapRTV[i]));

}

// 为立方体贴图创建着色器资源视图。

D3D11\_SHADER\_RESOURCE\_VIEW\_DESC srvDesc;

srvDesc.Format = texDesc.Format;

srvDesc.ViewDimension = D3D11\_SRV\_DIMENSION\_TEXTURECUBE;

srvDesc.TextureCube.MostDetailedMip = 0;

srvDesc.TextureCube.MipLevels = -1;

HR(m\_pD3dDevice->CreateShaderResourceView(cubeTex, &srvDesc, &mDynamicCubeMapSRV));

ReleaseCOM(cubeTex);

// 我们需要深度纹理将场景渲染到立方体贴图中

// 具有与立方图贴图相同的分辨率。

D3D11\_TEXTURE2D\_DESC depthTexDesc;

depthTexDesc.Width = CubeMapSize;

depthTexDesc.Height = CubeMapSize;

depthTexDesc.MipLevels = 1;

depthTexDesc.ArraySize = 1;

depthTexDesc.SampleDesc.Count = 1;

depthTexDesc.SampleDesc.Quality = 0;

depthTexDesc.Format = DXGI\_FORMAT\_D32\_FLOAT;

depthTexDesc.Usage = D3D11\_USAGE\_DEFAULT;

depthTexDesc.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;

depthTexDesc.CPUAccessFlags = 0;

depthTexDesc.MiscFlags = 0;

ID3D11Texture2D\* depthTex = 0;

HR(m\_pD3dDevice->CreateTexture2D(&depthTexDesc, 0, &depthTex));

// 为整个立方体创建深度模板视图

D3D11\_DEPTH\_STENCIL\_VIEW\_DESC dsvDesc;

dsvDesc.Format = depthTexDesc.Format;

dsvDesc.Flags = 0;

dsvDesc.ViewDimension = D3D11\_DSV\_DIMENSION\_TEXTURE2D;

dsvDesc.Texture2D.MipSlice = 0;

HR(m\_pD3dDevice->CreateDepthStencilView(depthTex, &dsvDesc, &mDynamicCubeMapDSV));

ReleaseCOM(depthTex);

// 用于绘制到立方体贴图的视口。

mCubeMapViewport.TopLeftX = 0.0f;

mCubeMapViewport.TopLeftY = 0.0f;

mCubeMapViewport.Width = (float)CubeMapSize;

mCubeMapViewport.Height = (float)CubeMapSize;

mCubeMapViewport.MinDepth = 0.0f;

mCubeMapViewport.MaxDepth = 1.0f;

然后是分别生成6个元素的摄像机

// 生成给定位置的立方体贴图。

XMFLOAT3 center(x, y, z);

XMFLOAT3 worldUp(0.0f, 1.0f, 0.0f);

// 沿每个坐标轴看。

XMFLOAT3 targets[6] =

{

XMFLOAT3(x + 1.0f, y, z), // +X

XMFLOAT3(x - 1.0f, y, z), // -X

XMFLOAT3(x, y + 1.0f, z), // +Y

XMFLOAT3(x, y - 1.0f, z), // -Y

XMFLOAT3(x, y, z + 1.0f), // +Z

XMFLOAT3(x, y, z - 1.0f) // -Z

};

// 使用世界坐标向上（0,1,0）除了+ Y / -Y之外的所有方向。 在这些情况下，我们

// 正在向下看+ Y或-Y，所以我们需要一个不同的“向上”向量。

XMFLOAT3 ups[6] =

{

XMFLOAT3(0.0f, 1.0f, 0.0f), // +X

XMFLOAT3(0.0f, 1.0f, 0.0f), // -X

XMFLOAT3(0.0f, 0.0f, -1.0f), // +Y

XMFLOAT3(0.0f, 0.0f, +1.0f), // -Y

XMFLOAT3(0.0f, 1.0f, 0.0f), // +Z

XMFLOAT3(0.0f, 1.0f, 0.0f) // -Z

};

for (int i = 0; i < 6; ++i)

{

mCubeMapCamera[i].LookAt(center, targets[i], ups[i]);

mCubeMapCamera[i].SetLens(0.5f\*XM\_PI, 1.0f, 0.1f, 1000.0f);

mCubeMapCamera[i].UpdateViewMatrix();

}

做好这些准备工作后，我们只需要循环6次，将这些准备好的渲染目标视图和深度视图绑定好，然后开始画，画好之后，之前准备的着色器资源视图就是动态生成的立方体贴图，然后和之前的静态立方体贴图一样处理就好了。

// 生成立方体贴图

m\_pImmediateContext->RSSetViewports(1, &mCubeMapViewport);

for (int i = 0; i < 6; ++i)

{

// 清除立方体贴图面和深度缓冲区。

m\_pImmediateContext->ClearRenderTargetView(mDynamicCubeMapRTV[i], reinterpret\_cast<const float\*>(&Colors::Silver));

m\_pImmediateContext->ClearDepthStencilView(mDynamicCubeMapDSV, D3D11\_CLEAR\_DEPTH | D3D11\_CLEAR\_STENCIL, 1.0f, 0);

// 将立方体贴图面绑定为渲染目标。

renderTargets[0] = mDynamicCubeMapRTV[i];

m\_pImmediateContext->OMSetRenderTargets(1, renderTargets, mDynamicCubeMapDSV);

// 将除中心球外的场景绘制到此立方体贴图面上。

DrawScene(mCubeMapCamera[i], false);

}

// 恢复原来的视口和渲染目标。

m\_pImmediateContext->RSSetViewports(1, &m\_screenViewPort);

renderTargets[0] = m\_pRenderTargetView;

m\_pImmediateContext->OMSetRenderTargets(1, renderTargets, m\_pDepthStencilView);

// 硬件生成立方体贴图较低级别的mipmap

m\_pImmediateContext->GenerateMips(mDynamicCubeMapSRV);

// 现在画出正常的场景，但要画中心球。

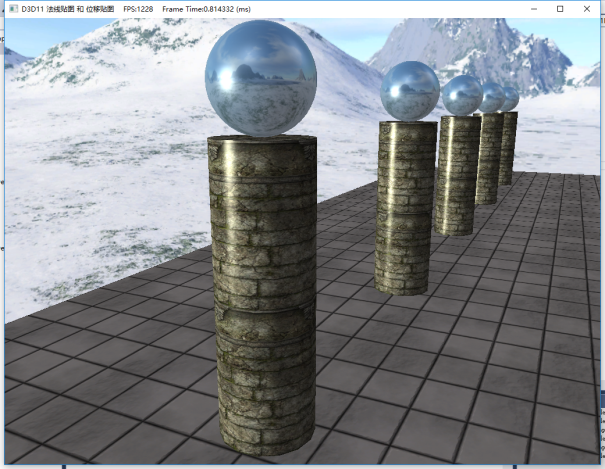
m\_pImmediateContext->ClearRenderTargetView(m\_pRenderTargetView, reinterpret\_cast<const float\*>(&Colors::Silver));

m\_pImmediateContext->ClearDepthStencilView(m\_pDepthStencilView, D3D11\_CLEAR\_DEPTH | D3D11\_CLEAR\_STENCIL, 1.0f, 0);

DrawScene(mCam, true);

法线贴图(Normal Mapping)

这个还是比较重要的技术，否则一些纹理本来很粗糙，但是光源的反光看起来又很光滑，这个就不正常了。原理就是改变顶点的法线方向，使光源的反射更加真实。我们需要一幅法线图片，这个图片可以用纹理的图片直接通过工具生成（PS的DDS插件也可以）。这个图片本来保存像素的地方改成了保存的是法线，这就是图片看起来都偏蓝的原因。



右边的就是用了法线贴图技术。

HLSL关键代码：

//同样的需要一个着色器资源

Texture2D gNormalMap;

//同样用纹理坐标采样，得到RGB

float3 normalMapSample = gNormalMap.Sample(samLinear, pin.Tex).rgb;

//这里是封装的一个方法将采样的RGB转换为法线，当然参数还需要用对象本身原来的法线和切线

float3 bumpedNormalW = NormalSampleToWorldSpace(normalMapSample, pin.NormalW, pin.TangentW);

//下面是方法的实现

float3 NormalSampleToWorldSpace(float3 normalMapSample, float3 unitNormalW, float3 tangentW)

{

// Uncompress each component from [0,1] to [-1,1].

float3 normalT = 2.0f\*normalMapSample - 1.0f;

// Build orthonormal basis.

float3 N = unitNormalW;

float3 T = normalize(tangentW - dot(tangentW, N)\*N);

float3 B = cross(N, T);

float3x3 TBN = float3x3(T, B, N);

// Transform from tangent space to world space.

float3 bumpedNormalW = mul(normalT, TBN);

return bumpedNormalW;

}

得到转换后的法线就可以把这个法线传递给光源的计算，就不再用对象本身的法线了。

float4 A, D, S;

ComputeDirectionalLight(gMaterial, gDirLights[i], bumpedNormalW, toEye,

A, D, S);

位移贴图（Displacement Mapping）

例子采样的是同样使用法线贴图 + 曲面细分，这种技术是真实的改变了顶点数据，不仅仅是光源的反射了。关键HLSL代码：

cbuffer cbPerFrame

{

DirectionalLight gDirLights[3];

float3 gEyePosW;

float gFogStart;

float gFogRange;

float4 gFogColor;

float gHeightScale; //高度缩放

float gMaxTessDistance; //最大细分距离

float gMinTessDistance; //最小细分距离

float gMinTessFactor; //最小细分因子

float gMaxTessFactor; //最大细分因子

};

在顶点着色阶段：

float d = distance(vout.PosW, gEyePosW);

// 归一化曲面细分因子。

// 曲面细分是

// 0，如果 d> = gMinTessDistance 且

// 1如果d <= gMaxTessDistance。

float tess = saturate( (gMinTessDistance - d) / (gMinTessDistance - gMaxTessDistance) );

// 重新调整[0,1] - > [gMinTessFactor，gMaxTessFactor]。

vout.TessFactor = gMinTessFactor + tess\*(gMaxTessFactor-gMinTessFactor);

曲面细分阶段：

struct PatchTess

{

float EdgeTess[3] : SV\_TessFactor;

float InsideTess : SV\_InsideTessFactor;

};

PatchTess PatchHS(InputPatch<VertexOut,3> patch,

uint patchID : SV\_PrimitiveID)

{

PatchTess pt;

// 沿着边缘的平均细分因子，并选择边缘细分因子

// 内部曲面细分。 做细分因子很重要

// 基于边缘属性的计算，以便共享边

// 多于一个三角形将具有相同的细分因子。

// 否则，可能会出现间隙。

pt.EdgeTess[0] = 0.5f\*(patch[1].TessFactor + patch[2].TessFactor);

pt.EdgeTess[1] = 0.5f\*(patch[2].TessFactor + patch[0].TessFactor);

pt.EdgeTess[2] = 0.5f\*(patch[0].TessFactor + patch[1].TessFactor);

pt.InsideTess = pt.EdgeTess[0];

return pt;

}

struct HullOut

{

float3 PosW : POSITION;

float3 NormalW : NORMAL;

float3 TangentW : TANGENT;

float2 Tex : TEXCOORD;

};

[domain("tri")]

[partitioning("fractional\_odd")]

[outputtopology("triangle\_cw")]

[outputcontrolpoints(3)]

[patchconstantfunc("PatchHS")]

HullOut HS(InputPatch<VertexOut,3> p,

uint i : SV\_OutputControlPointID,

uint patchId : SV\_PrimitiveID)

{

HullOut hout;

// 直接通过着色器

hout.PosW = p[i].PosW;

hout.NormalW = p[i].NormalW;

hout.TangentW = p[i].TangentW;

hout.Tex = p[i].Tex;

return hout;

}

struct DomainOut

{

float4 PosH : SV\_POSITION;

float3 PosW : POSITION;

float3 NormalW : NORMAL;

float3 TangentW : TANGENT;

float2 Tex : TEXCOORD;

};

// 为细分器创建的每个顶点调用域着色器。

// 这是细分之后的顶点着色器。

[domain("tri")]

DomainOut DS(PatchTess patchTess,

float3 bary : SV\_DomainLocation,

const OutputPatch<HullOut,3> tri)

{

DomainOut dout;

// Interpolate patch attributes to generated vertices.

dout.PosW = bary.x\*tri[0].PosW + bary.y\*tri[1].PosW + bary.z\*tri[2].PosW;

dout.NormalW = bary.x\*tri[0].NormalW + bary.y\*tri[1].NormalW + bary.z\*tri[2].NormalW;

dout.TangentW = bary.x\*tri[0].TangentW + bary.y\*tri[1].TangentW + bary.z\*tri[2].TangentW;

dout.Tex = bary.x\*tri[0].Tex + bary.y\*tri[1].Tex + bary.z\*tri[2].Tex;

// 插值法线可以对其进行非归一化处理，因此将其归一化。

dout.NormalW = normalize(dout.NormalW);

//

// 位移贴图。

//

// 根据到眼睛的距离选择mipmap级别; 具体来说，选择

// 下一个miplevel，每个MipInterval单位，并在[0,6]中限制miplevel。

const float MipInterval = 20.0f;

float mipLevel = clamp( (distance(dout.PosW, gEyePosW) - MipInterval) / MipInterval, 0.0f, 6.0f);

// 采样高度图（存储在alpha通道中）。

float h = gNormalMap.SampleLevel(samLinear, dout.Tex, mipLevel).a;

// 沿法线偏移顶点。

dout.PosW += (gHeightScale\*(h-1.0))\*dout.NormalW;

// 转换到投影坐标空间

dout.PosH = mul(float4(dout.PosW, 1.0f), gViewProj);

return dout;

}

其他的就和法线贴图一样的代码了，最后看一下效果，确实顶点也改变了，当然更耗资源了。

