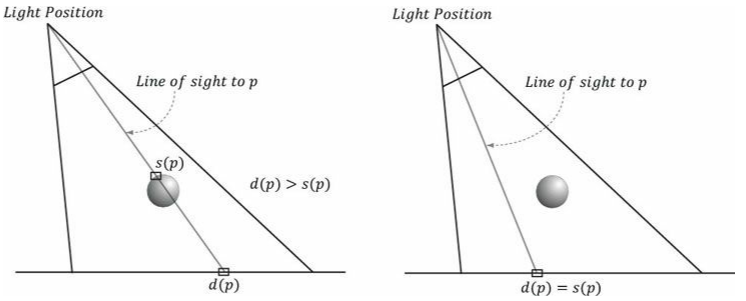
# 阴影贴图(Shadow Mapping)

# 概述

阴影可以表现场景中的光源位置，烘托物体之间的对应关系。这里我们介绍一种基本的阴影贴图算法，它是一种在游戏和3D应用程序中模拟动态阴影的常见方法。

阴影贴图映射算法的实现思路是：把灯光作为观察点，把场景的深度值渲染到一个称为阴影贴图（shadow map）的深度缓冲区中。在完成该操作后，我们可以得到一张从灯光角度看去，所有可见像素的深度值构成的贴图。（被其他像素挡住的像素不会出现在阴影贴图中，因为它们不会通过深度测试，要么被其他像素覆盖，要么从未被写入进来。

在生成阴影贴图之后，我们要像往常那样从摄像机的角度渲染场景。针对每一个要被渲染的像素p，我们以光源的视角来计算其深度值，记为d(p)。之后我们沿光源到像素的路径对阴影贴图进行采样，获取存储的深度值s(p)。它就是在观察线上与光源距离最近的像素的深度。我们将这两个值相比较，就可以知道所要渲染的像素是不是位于阴影中。我们可以从下图发现，当且仅当d(p)>s(p)时，像素p位于阴影范围内。而d(p)<=s(p)时，像素p位于阴影范围之外。



# 渲染场景深度

阴影贴图算法依赖于以光源的视角渲染场景深度，本质上来说，这也是一种渲染到纹理技术。

我们在创建深度贴图时指定bindflag为

depthStencilDesc.BindFlags = **D3D11\_BIND\_DEPTH\_STENCIL** | **D3D11\_BIND\_SHADER\_RESOURCE**;

贴图格式也需要注意，我们指定弱类型格式，因为该纹理的格式取决于特定视图。着色器资源视图使用DXGI\_FORMAT\_R24\_UNORM\_X8\_TYPELESS格式，而深度/模板缓冲区视图使用DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT格式。

depthStencilDesc.Format = **DXGI\_FORMAT\_R24G8\_TYPELESS**;

我们只需要深度值，所以会将一个空的渲染目标绑定到管线上。因为没有渲染目标就没有可以写入像素的地方，所有的像素写入操 作都会被屏蔽掉。而深度值仍然可以写入深度缓冲区。

|  |
| --- |
| ID3D11RenderTargetView\* nullTarget[] = { nullptr };  m\_d3dContext->OMSetRenderTargets(1, nullTarget, m\_shadowMapDSV.Get()); |

注意：当绑定一个空的渲染目标时，你不能直接给渲染目标数组参数传递空值。正确的做法是传递一个有效的ID3D11RenderTargetView数组指针，在该数组中包含一个空值元素。

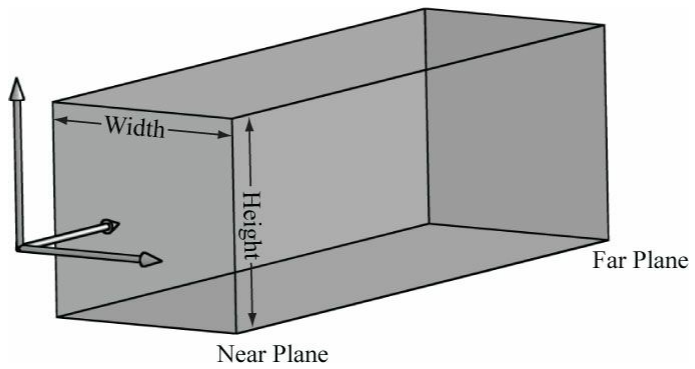
    // 错误示范

    md3dDevice->OMSetRenderTargets(1, nullptr, mDepthMapDSV);

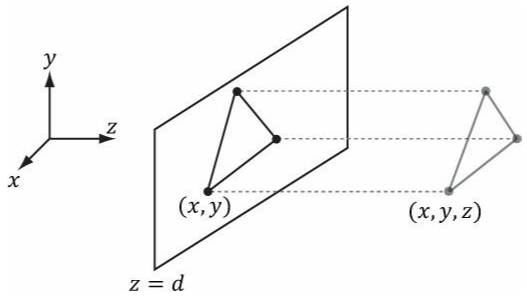
# 正交投影

到目前为止，我们一直使用的是透视投影（perspective projection）。它的主要特点是与摄像机距离越远的物体看上去越小(近大远小)。这与我们在生活中看到的情况完全相同。正交投影（orthographic projection）是另一种投影，它主要用于3D科学或工程应用中。这种投影的特点是平行线在投影之后依然保持平行(而不是像透视投影一样相交于远点)。

正交投影非常适合模拟由平行光产生的阴影。沿着z轴的正方向看，正交投影的视域体是一个宽为w、高为h，近平面为n、远平面为f的对齐于观察空间坐标轴的长方体。



使用正交投影时，其投影线均平行于观察空间的z轴，故顶点(x,y,z)的2D投影坐标为(x,y)。



我们可以使用XMMatrixOrthographicLH来生成其投影矩阵。与透视投影不同的是，正交投影是完全线性的变换，其不需要除以w分量，将原坐标乘以变换矩阵后直接就能将其转为NDC坐标。

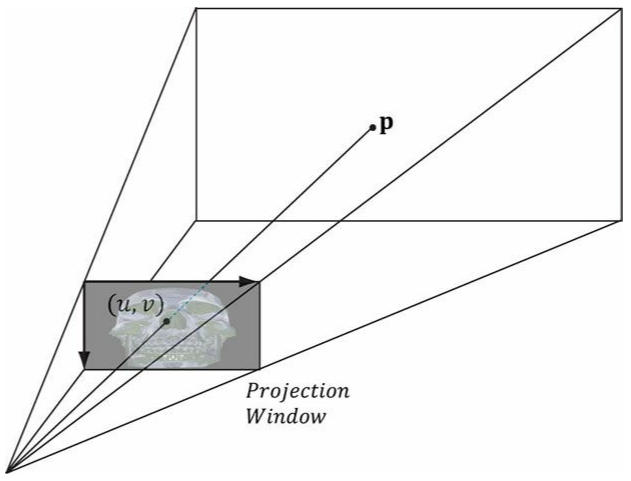
注意，因为其w分量总是1，除以w并不会影响结果，若代码中有除以w的操作，正交投影和透视投影均可正常工作。当然此时就需要在代码一致性和不必要的除法中间权衡利弊了。

# 投影纹理坐标

投影纹理映射（projective texturing）允许我们把一个纹理投影到任意几何体上，其原理与投影仪的工作方式相似。

实现投影贴图映射的关键在于要为每个像素生成一个纹理坐标，使纹理看起来就像是被投影到了几何体上一样。我们将这种纹理坐标称为投影纹理坐标（projective texture coordinate）。

从下图中可以看到，纹理坐标(u,v)指定了应当被投影在3D点p上的纹理元素。相反，给定一个点，也可以得到它在纹理空间的坐标(u,v)。



我们通过以下步骤生成投影纹理坐标。

1.把点p投影到光源的投影窗口，并把坐标变换到 NDC 空间。

2.把投影后的坐标从NDC空间变换到纹理空间，再转换为纹理坐标。

第一步我们可以通过乘以观察矩阵和投影矩阵，然后执行透视除法来完成。

第二步则是执行相应的坐标变换u=0.5x+0.5,v=−0.5y+0.5。

我们可以用矩阵来表示此变化:

|  |
| --- |
| // Transform NDC space [-1,+1]^2 to texture space [0,1]^2  // It's the same as following  //shadowPosH.x = +0.5f\*shadowPosH.x + 0.5f;  //shadowPosH.y = -0.5f\*shadowPosH.y + 0.5f;  XMMATRIX T(  0.5f, 0.0f, 0.0f, 0.0f,  0.0f, -0.5f, 0.0f, 0.0f,  0.0f, 0.0f, 1.0f, 0.0f,  0.5f, 0.5f, 0.0f, 1.0f); |

这样在shader里只需执行透视除法即可。

|  |
| --- |
| // Complete projection by doing division by w.  shadowPosH.xyz /= shadowPosH.w; |

# 视锥体之外的点

在渲染管线中，位于视锥体之外的点都会被裁剪掉。不过，当我们从灯光视角对几何体进行投影生成投影纹理坐标时，并不会有剪裁发生——因为我们只是简单地投影顶点。所以当点位于投影体之外时，它的纹理坐标会超出[0,1]区间。在进行纹理采样时，这些[0,1]区间之外的投影纹理坐标会采用与普通纹理坐标相同的处理方式，根据当前指定的寻址模式进行采样。这些点不会被投影机发出的光照到，所以我们可以使用边框寻址模式，然后边框颜色设为0。

|  |
| --- |
| CD3D11\_SAMPLER\_DESC samDesc(D3D11\_DEFAULT);  samDesc.Filter = D3D11\_FILTER\_COMPARISON\_MIN\_MAG\_LINEAR\_MIP\_POINT;  samDesc.AddressU = D3D11\_TEXTURE\_ADDRESS\_BORDER;  samDesc.AddressV = D3D11\_TEXTURE\_ADDRESS\_BORDER;  samDesc.AddressW = D3D11\_TEXTURE\_ADDRESS\_BORDER;  samDesc.ComparisonFunc = D3D11\_COMPARISON\_LESS\_EQUAL;  **float** outsideShadowFactor = 0.f;  samDesc.BorderColor[0] = outsideShadowFactor;  samDesc.BorderColor[1] = outsideShadowFactor;  samDesc.BorderColor[2] = outsideShadowFactor;  samDesc.BorderColor[3] = outsideShadowFactor;      ComPtr<ID3D11SamplerState> sam;  HRESULT hr = m\_d3dDevice->CreateSamplerState(&samDesc,sam.GetAddressOf());  DX::ThrowIfFailed(hr);  m\_d3dContext->PSSetSamplers(1, 1, sam.GetAddressOf()); |

# 阴影贴图算法

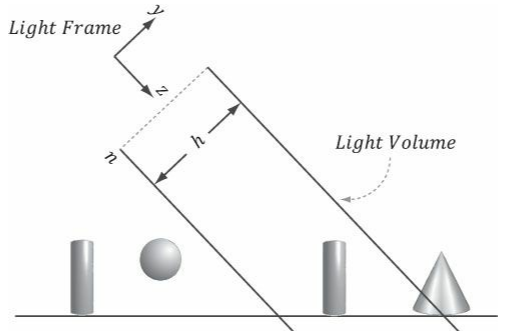
**生成阴影贴图**

创建SRV和DSV

|  |
| --- |
| // Use typeless format because the DSV is going to interpret // the bits as DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT, whereas the SRV is going to interpret  // the bits as DXGI\_FORMAT\_R24\_UNORM\_X8\_TYPELESS.      // Create Texture      D3D11\_TEXTURE2D\_DESC depthStencilDesc;  depthStencilDesc.Width = m\_outputWidth;  depthStencilDesc.Height = m\_outputHeight;  depthStencilDesc.MipLevels = 1;  depthStencilDesc.ArraySize = 1;  depthStencilDesc.Format = DXGI\_FORMAT\_R24G8\_TYPELESS;  depthStencilDesc.SampleDesc.Count = 1;  depthStencilDesc.SampleDesc.Quality = 0;  depthStencilDesc.Usage = D3D11\_USAGE\_DEFAULT;  depthStencilDesc.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL | D3D11\_BIND\_SHADER\_RESOURCE;  depthStencilDesc.CPUAccessFlags = 0;  depthStencilDesc.MiscFlags = 0;      HRESULT hr = m\_d3dDevice->CreateTexture2D(&depthStencilDesc, nullptr, m\_shadowMapTexture.GetAddressOf());  DX::ThrowIfFailed(hr);      // Create Depth Stencil View      CD3D11\_DEPTH\_STENCIL\_VIEW\_DESC dsvDesc(D3D11\_DSV\_DIMENSION\_TEXTURE2D, DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT);  hr = m\_d3dDevice->CreateDepthStencilView(m\_shadowMapTexture.Get(), &dsvDesc, m\_shadowMapDSV.GetAddressOf());  DX::ThrowIfFailed(hr);      // Create Shader Resource View      CD3D11\_SHADER\_RESOURCE\_VIEW\_DESC srvDesc(D3D11\_SRV\_DIMENSION\_TEXTURE2D,DXGI\_FORMAT\_R24\_UNORM\_X8\_TYPELESS);  hr = m\_d3dDevice->CreateShaderResourceView(m\_shadowMapTexture.Get(), &srvDesc, m\_shadowMapSRV.GetAddressOf());  DX::ThrowIfFailed(hr); |

**渲染场景到阴影贴图**

我们将从光源的角度渲染场景。我们需要定义一个将坐标从世界空间转换到光源空间的观察矩阵以及用于描述光照范围的光源投影矩阵。对于聚光灯，我们可以使用透视投影(平截头体)，对于平行光，我们可以使用正交投影(长方体)。由于平行光被限制在长方体内，它只能照到场景的一部分。对于能够照到整个场景的光源(例如太阳)，我们就可以扩展光源的体积，使其包含全部场景。



相关代码

|  |
| --- |
| **if** (bUsingSpotOrDirectionalLight)  {  pos = m\_spotLight.Position;  dir = m\_spotLight.Direction;  }  **else**  {  dir = m\_dirLight.Direction;  **float** distance = 200.f;  pos = XMFLOAT3(-dir.x\*distance, -dir.y\*distance, -dir.z\*distance);  }  XMVECTOR up = XMVectorSet(0.f, 1.f, 0.f, 0.f);  **if** (dir.y == -1.f || dir.y == 1.f)  {  up = XMVectorSet(1.f, 0.f, 0.f, 0.f);  }  XMMATRIX view = XMMatrixLookToLH(XMLoadFloat3(&pos), XMLoadFloat3(&dir), up);  XMMATRIX proj = XMLoadFloat4x4(&m\_proj);  **if** (!bUsingSpotOrDirectionalLight)  {  **float** width = 20.f;  **float** height = 20.f;  // For large scene. you need ensure it contain all objects in scene  //float width = 100.f;  //float height = 100.f;  proj = XMMatrixOrthographicLH(width, height, 1.f, 1000.f);  }  XMStoreFloat4x4(&m\_view, view);  XMStoreFloat4x4(&m\_proj, proj);  XMStoreFloat4x4(&m\_cbPerFrame.viewProj, XMMatrixTranspose(XMMatrixMultiply(view, proj)));  // Transform NDC space [-1,+1]^2 to texture space [0,1]^2  // It's the same as following  //shadowPosH.x = +0.5f\*shadowPosH.x + 0.5f;  //shadowPosH.y = -0.5f\*shadowPosH.y + 0.5f;  XMMATRIX T(  0.5f, 0.0f, 0.0f, 0.0f,  0.0f, -0.5f, 0.0f, 0.0f,  0.0f, 0.0f, 1.0f, 0.0f,  0.5f, 0.5f, 0.0f, 1.0f);  XMStoreFloat4x4(&m\_cbPerFrame.shadowTransform, XMMatrixTranspose(XMMatrixMultiply(XMMatrixMultiply(view, proj),T)));  d3dUtil::UpdateDynamicBufferFromData(m\_d3dContext, m\_constantBufferPerFrame, m\_cbPerFrame); |

**阴影因子**

我们为光照计算添加一个[0,1]的系数。当值为0是，表示其该点位于阴影中，值为1时，表示位于阴影外。一个点可能部分处于阴影中，这样它的值就是0到1之间。阴影因子直接与漫反射和高光相乘来实现阴影。由于环境光是间接光，所以阴影因子对其不会产生影响。

|  |
| --- |
| **float** shadow = 1.0f;  #ifdef ENABLE\_SHADOWMAP  shadow = CalcShadowFactor(gsamShadow, gShadowMap, pin.ShadowPosH);  #endif      ComputeDirectionalLight(gMaterial, gDirLight, bumpedNormalW, toEyeW, A, D, S);  ambient += A;  diffuse += shadow \* D;  spec += shadow \* S; |

**阴影图检测**

我们通过对阴影图进行采样来确定一个点是否位于阴影中。即我们为每个像素计算d(p)和s(p)。d(p)通过把点变换到光源的NDC空间中即可求得。此时的z坐标即为深度值。在光源的视锥体中使用投影纹理映射技术，然后对阴影贴图采样就可以求出s(p)。这样获得的d(p)和s(p)都位于光源的NDC空间中，它们可以直接进行比较。

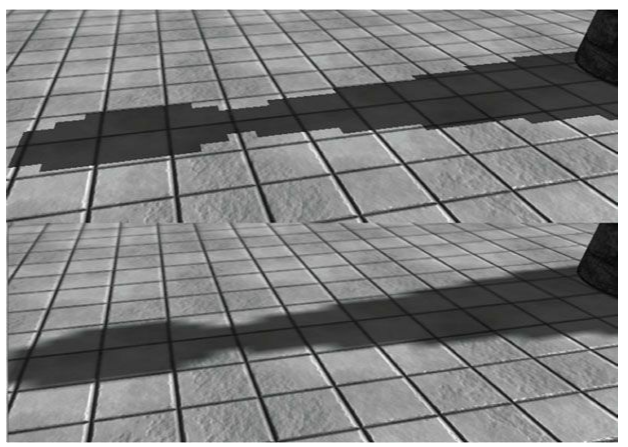
|  |
| --- |
| **static** **const** **float** SMAP\_SIZE = 2048.0f;  **static** **const** **float** SMAP\_DX = 1.0f / SMAP\_SIZE;    ...    // Complete projection by doing division by w.  shadowPosH.xyz /= shadowPosH.w;      // Depth in NDC space.  **float** depth = shadowPosH.z;      // Texel size.  **const** **float** dx = SMAP\_DX;      // Sample shadow map to get nearest depth to light.  **float** s0 = gShadowMap.Sample(gShadowSam, shadowPosH.xy).r;    // Is the pixel depth <= shadow map value?  **float** result0 = depth <= s0; |

**PCF过滤**

在使用投影纹理坐标对阴影图进行采样时，往往不会命中阴影图的准确位置，而通常是在4个纹素之间。对于彩色图，我们使用双线性插值即可解决该问题。而对于深度值采用这样的算法，可能会把像素误标入阴影中。所以，我们应该对采样的结果进行插值，而不是对深度值进行插值，这种方法称为百分比渐进过滤(Percentage Closer Filtering, PCF)。我们以点过滤(MIN\_MAG\_MIP\_POINT)的方式在坐标uv及周围的4个点进行采样。因为使用的点采样，所以这4个点会分别命中围绕坐标(u,v)最近的4个阴影图纹素。我们对这4个点进行阴影图检测，然后对结果进行双线性插值。

|  |
| --- |
| // Sample shadow map to get nearest depth to light.  **float** s0 = gShadowMap.Sample(gShadowSam, projTexC.xy).r;  **float** s1 = gShadowMap.Sample(gShadowSam, projTexC.xy + float2(SMAP\_DX, 0)).r;  **float** s2 = gShadowMap.Sample(gShadowSam, projTexC.xy + float2(0, SMAP\_DX)).r;  **float** s3 = gShadowMap.Sample(gShadowSam, projTexC.xy + float2(SMAP\_DX, SMAP\_DX)).r;  // Is the pixel depth <= shadow map value?  **float** result0 = depth <= s0;  **float** result1 = depth <= s1;  **float** result2 = depth <= s2;  **float** result3 = depth <= s3;  // Transform to texel space.  float2 texelPos = SMAP\_SIZE\*projTexC.xy;  // Determine the interpolation amounts.  float2 t = frac(texelPos);  // Interpolate results.  **return** lerp(lerp(result0, result1, t.x), lerp(result2, result3, t.x), t.y); |

这样的话，一个像素就可能局部处于阴影中，这使得阴影内外的过度更加平滑，而非棱角分明。



从上面可以看出，PCF需要4个纹理样本，而采样时GPU代价较高的操作。幸运的是，Direct3D 11硬件对PCF技术提供了内部支持，我们可以通过调用SampleCmpLevelZero方法来实现它。

    // Automatically does a 4-tap PCF.

    percentLit = shadowMap.SampleCmpLevelZero(gsamShadow, shadowPosH.xy, depth).r;

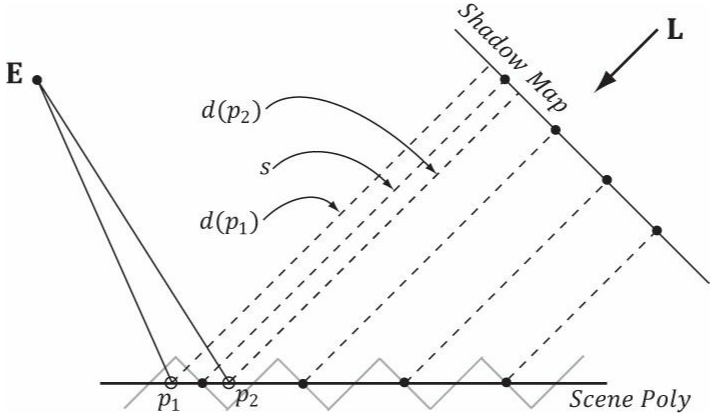
该方法名字中的LevelZero部分意味着只能在最高层的mipmap层级中执行此函数的相关任务。此方法使用的采样器是比较采样器，这使硬件能够执行阴影贴图的比较测试。我们使用**D3D11\_FILTER\_COMPARISON\_MIN\_MAG\_LINEAR\_MIP\_POINT**作为过滤器，并将比较函数设置为**D3D11\_COMPARISON\_LESS\_EQUAL**。

**偏移与走样**

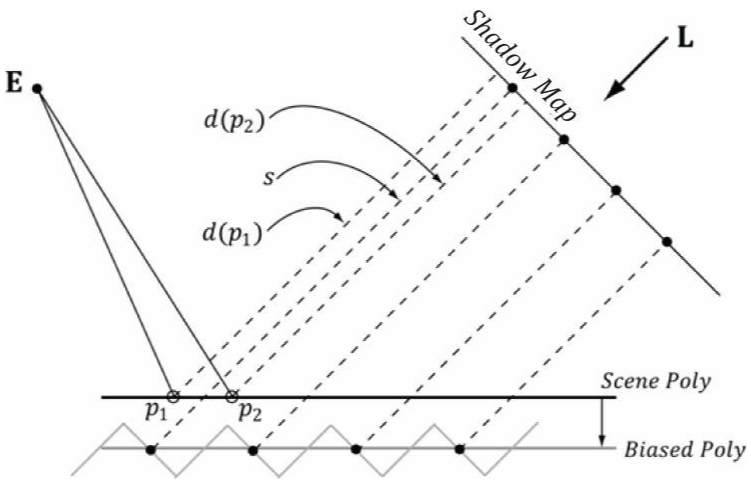
阴影图存储的是距离光源最近的像素的深度值，但是其分辨率有限，导致没一个纹素要表示场景中的一片区域。阴影图只是以光源视角针对场景深度进行的离散采样，这样会导致所谓的阴影粉刺(shadow acne)等图像走样(aliasing)的问题。

tu

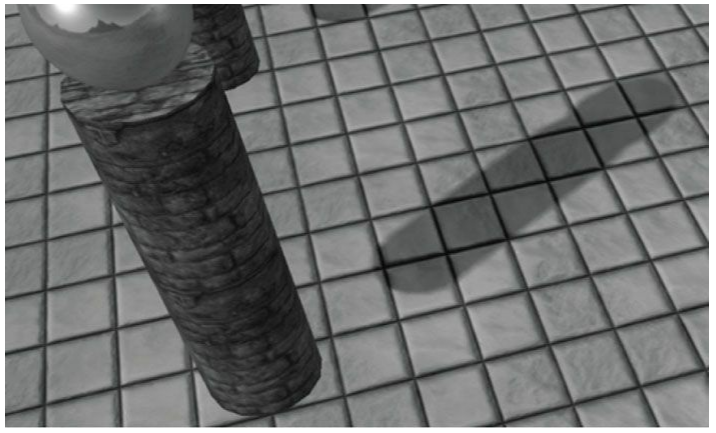
下图说明了为什么会出现这种现象。



一种简单的解决方法就是通过一个偏移量来进行调整，如图所示。



但要注意，偏移量过大会导致影子与物体分离。



然而，并没有一个固定的偏移量可以运用于所有几何体。特别是有着极大斜率的三角形，需要更大的偏移量，但这样又会造成阴影分离的问题。

因此，我们绘制时可以先以光源视角度量多边形斜率，并未斜率较大的多边形应用更大的偏移量。图形硬件内部对此提供了支持，我们通过名为斜率缩放偏移(slope-scaled-bias)的光栅化属性来实现。

|  |
| --- |
| // [From MSDN]  // If the depth buffer currently bound to the output-merger stage has a UNORM format or  // no depth buffer is bound the bias value is calculated like this:  //  // Bias = (float)DepthBias \* r + SlopeScaledDepthBias \* MaxDepthSlope;  //  // where r is the minimum representable value > 0 in the depth-buffer format converted to float32.  // [/End MSDN]  //  // For a 24-bit depth buffer, r = 1 / 2^24.  //  // Example: DepthBias = 100000 ==> Actual DepthBias = 100000/2^24 = .006  CD3D11\_RASTERIZER\_DESC rsDesc(D3D11\_DEFAULT);  rsDesc.DepthBias = 1000;  rsDesc.DepthBiasClamp = 0.f;  rsDesc.SlopeScaledDepthBias = 1.f;  ComPtr<ID3D11RasterizerState> mRSState;  hr = m\_d3dDevice->CreateRasterizerState(&rsDesc, mRSState.GetAddressOf());  DX::ThrowIfFailed(hr);  m\_d3dContext->RSSetState(mRSState.Get()); |