# 立方体贴图(Cube Map)

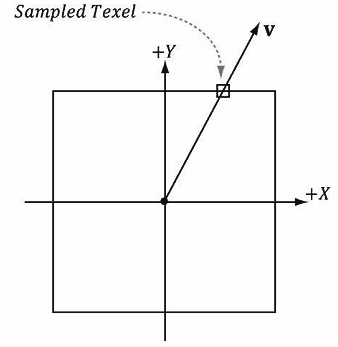
# 概述

立方体贴图本质上是一种由 6 幅纹理组成的，按特定方式解释的纹理数组。通过使用立方体贴图映射，我们可以很容易地对一个天空进行纹理 映射或模拟反射。

其实现思路是：将6幅纹理想像为关于某个坐标系原点和轴对齐的立方体的6个平面。由于立方体纹理是轴对齐的，它的每个平面都沿着3个主轴的方向放置；所以，我们可以根据与平面相交的主轴方向（±X，±Y，±Z）标识立方体贴图的每个平面。Direct3D提供了D3D11\_TEXTURECUBE\_FACE枚举类型来表示每个平面。

|  |
| --- |
| typedef **enum** D3D11\_TEXTURECUBE\_FACE  {  D3D11\_TEXTURECUBE\_FACE\_POSITIVE\_X = 0,  D3D11\_TEXTURECUBE\_FACE\_NEGATIVE\_X = 1,  D3D11\_TEXTURECUBE\_FACE\_POSITIVE\_Y = 2,  D3D11\_TEXTURECUBE\_FACE\_NEGATIVE\_Y = 3,  D3D11\_TEXTURECUBE\_FACE\_POSITIVE\_Z = 4,  D3D11\_TEXTURECUBE\_FACE\_NEGATIVE\_Z = 5,  } D3D11\_TEXTURECUBE\_FACE; |

与2D纹理映射不同，要指定一个立方体贴图中的纹理元素，我们必须定义一个从原点引出的查找向量v，通过3D纹理坐标指定纹理元素。查找向量v与立方体贴图相交的地方就是3D纹理坐标v对应的纹理元素。



在HLSL中，立方体纹理由TextureCube类型表示。下面的代码片段说明了应该如何对立方体贴图进行采样：

|  |
| --- |
| TextureCube gCubeMap;    ...    float3 v = float3(x,y,z); // 用于查找纹理坐标的向量  float4 color = gCubeMap.Sample(gSampler,v); |

注意：查找向量和立方体贴图应该使用相同的坐标系。例如，当立方体贴图使用世界坐标系时（即，立方体平面与世界空间的主轴对齐），查找向量也应该使用世界坐标系。

# 环境贴图

立方体贴图的主要用途是实现环境贴图映射（environment mapping）。它的实现思路是：在场景中的某个物体O的中心位置放置一架摄像机，将（水平和垂直）视域角设为90º。然后沿着±X轴、±Y轴和±X轴方向，从6种不同的角度各拍摄一张照片（在照片中不包含物体O）。因为视域角为90º，所以这6张照片完全可以从物体O的角度捕捉到各个方向上的环境信息。我们把这6张照片存入到一个立方体贴图中，就得到了所谓的环境贴图。换句话说，环境贴图就是在立方体平面上存入一个环境的全景照片。

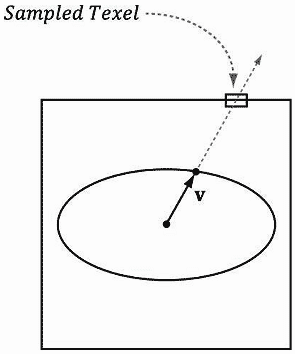


上述内容表明，在场景中有多少个使用环境贴图映射的物体，我们就需要创建多少个环境贴图。不过，环境贴图通常只用于表现远处的“背景”信息，故近景物体可以共享相同的环境贴图。

环境贴图使用的这6幅图像一般不是在Direct3D程序中生成的，虽然有时也可以这么做。它们的内容通常是由美术师提前画好的（就像我们之前使用的2D纹理一样）。我们也可以在一个3D世界编辑器中创建场景，然后在编辑器中预渲染立方体贴图平面上的这6幅图像。

# 天空纹理映射

环境贴图可以用来对天空进行纹理映射。我们可以创建一个包围整个场景的椭圆体，使用一个环境贴图对椭圆体进行纹理映射，将环境贴图投影到椭圆体表面上，模拟天地相接处的远山假象。

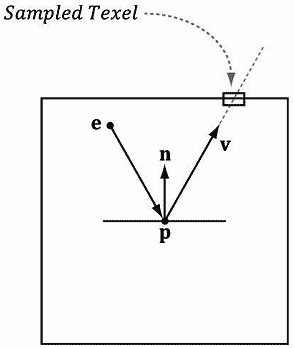


我们假设天空椭圆体无限遥远，无论摄像机如何在世界空间中移动，我们永远都无法靠近或远离天空椭圆体表面。要实现这个无限遥远的天空，我们只需要在世界空间中将天空椭圆体的中心和摄像机的位置对齐，让它跟着摄像机移动。这样无论摄像机移动到哪，我们都不会靠近天空椭圆体表面。

|  |
| --- |
| TextureCube gCubeMap : register(t1);  SamplerState gSampler : register(s0);    ...    struct VertexIn  {  float3 PosL : POSITION;  };    struct VertexOut  {  float4 PosH : SV\_POSITION;  float3 PosL : POSITION;  };    VertexOut VS(VertexIn vin)  {  VertexOut vout;    // Use local vertex position as cubemap lookup vector.  vout.PosL = vin.PosL;    // Transform to world space.  float4 posW = mul(float4(vin.PosL, 1.0f), gWorld);    // Always center sky about camera.  posW.xyz += gEyePosW;    // Set z = w so that z/w = 1 (i.e., skydome always on far plane).  vout.PosH = mul(posW, gViewProj).xyww; // Remember to set DepthFunc to COMPARISON\_LESS\_EQUAL;    **return** vout;  }    float4 PS(VertexOut pin) : SV\_Target  {  **return** gCubeMap.Sample(gSampler, pin.PosL);  } |

# 模拟反射

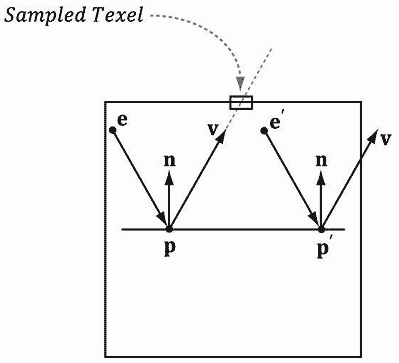
环境贴图的另一个主要用途是为任意物体模拟反射（该技术只能反射环境贴图中的图像）。下图说明了如何使用环境贴图来实现镜面反射。这个表面就像是一面镜子：观察点e可以看到由p点反射回来的环境图像。



我们为每个像素计算反射向量，然后用它来对环境贴图进行采样

|  |
| --- |
| float3 incident = -toEyeW;  float3 reflectionVector = reflect(incident, pin.NormalW);  float4 reflectionColor = gCubeMap.Sample(gSampler, reflectionVector);    litColor += gMaterial.Reflect\*reflectionColor; |

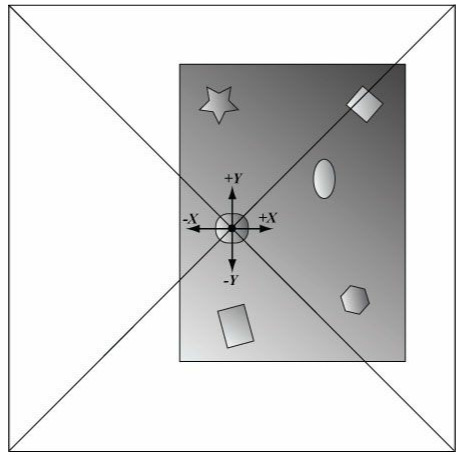
下图说明通过环境贴图映射无法达到令人满意的平面反射效果。这是因为反射向量无法表达完整的信息（它不包含坐标位置）；我们实际需要的是一条反射射线，并且要让该射线与环境贴图相交。射线具有位置和方向，而向量只具有方向。我们可以从图中看到，两条反射射线r(t) = p + tv和rʹ(t) = pʹ + tv与不同的立方体贴图元素相交，所以应该得到不同的颜色。不过，由于两条射线具有相同的方向向量v，而方向向量v又被唯一地用于查找立方体贴图，所以当观察点位于e和eʹ时，p和pʹ会映射到同一个纹理元素上。对于平面物体来说，这是环境贴图映射的一个严重缺陷。对于曲面来说，环境贴图映射的这一缺陷并不明显，因为曲面会产生反射向量的变化。



# 动态立方体贴图

上面描述的都是静态立方体贴图，存储的是预先绘制好的固定图像。但如果我们需要在场景中创建一些会移动的动态物体，则事先生成的立方体贴图当然无法捕捉到这些物体，所以无法绘制出动态物体的反射镜像。

为了克服这种限制，我们可以在运行时动态地构建立方体贴图。在每一帧里，以摄像机作为原点，沿着坐标轴共6个方向将场景渲染6次，存储到立方体贴图的对应面里。这样就能捕捉到场景中动态物体的反射镜像。



因为每一帧都需要将场景绘制6次，所以我们应将场景中用到动态立方体贴图的地方降到最少。而且我们一般使用相对较低分辨率的立方体图(如256\*256)，这样可以减少要处理的像素数量。

**构建立方体贴图**

我们通过创建一个包含6个元素的纹理数组，并且指定D3D11\_RESOURCE\_MISC\_TEXTURECUBE作为MiscFlags来创建立方体贴图。

|  |
| --- |
| CD3D11\_TEXTURE2D\_DESC texDesc(DXGI\_FORMAT\_R8G8B8A8\_UNORM,m\_cubeMapSize,m\_cubeMapSize);  texDesc.ArraySize = 6;  texDesc.BindFlags = D3D11\_BIND\_SHADER\_RESOURCE | D3D11\_BIND\_RENDER\_TARGET;  texDesc.MiscFlags = D3D11\_RESOURCE\_MISC\_TEXTURECUBE | D3D11\_RESOURCE\_MISC\_GENERATE\_MIPS;    HRESULT hr = m\_d3dDevice->CreateTexture2D(&texDesc, nullptr, m\_dynamicCubeMapTexture.GetAddressOf());  DX::ThrowIfFailed(hr); |

我们还需要指定D3D11\_RESOURCE\_MISC\_GENERATE\_MIPS，这样我们就可以调用ID3D11DeviceContext::GenerateMips来生成mipmap。因为我们只是渲染到最高层的mipmap，如果不手动生成的话，其他层会是没有定义的状态。

接下来我们需要为立方体贴图的每一个面都创建RenderTargetView

|  |
| --- |
| CD3D11\_RENDER\_TARGET\_VIEW\_DESC rtvDesc(D3D11\_RTV\_DIMENSION\_TEXTURE2DARRAY);  rtvDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;  rtvDesc.Texture2DArray.ArraySize = 1;  **for** (**int** i = 0; i < 6; ++i)  {  rtvDesc.Texture2DArray.FirstArraySlice = i;    hr = m\_d3dDevice->CreateRenderTargetView(m\_dynamicCubeMapTexture.Get(), &rtvDesc, m\_dynamicCubeMapRTV[i].GetAddressOf());  DX::ThrowIfFailed(hr);  } |

最后我们创建ShaderResourceView，将ViewDimension指定为D3D11\_SRV\_DIMENSION\_TEXTURECUBE

|  |
| --- |
| CD3D11\_SHADER\_RESOURCE\_VIEW\_DESC srvDesc(D3D11\_SRV\_DIMENSION\_TEXTURECUBE);  srvDesc.Format = DXGI\_FORMAT\_R8G8B8A8\_UNORM;    hr = m\_d3dDevice->CreateShaderResourceView(m\_dynamicCubeMapTexture.Get(), &srvDesc, m\_dynamicCubeMapSRV.GetAddressOf());  DX::ThrowIfFailed(hr); |

**构建深度缓冲区和视口**

因为立方体贴图的每个面与后台缓冲区的主分辨率不一致，故需要创建相应的深度缓冲区和视口

|  |
| --- |
| D3D11\_TEXTURE2D\_DESC depthStencilDesc;  depthStencilDesc.Width = m\_cubeMapSize;  depthStencilDesc.Height = m\_cubeMapSize;  depthStencilDesc.MipLevels = 1;  depthStencilDesc.ArraySize = 1;  depthStencilDesc.Format = DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT;  depthStencilDesc.SampleDesc.Count = 1;  depthStencilDesc.SampleDesc.Quality = 0;  depthStencilDesc.Usage = D3D11\_USAGE\_DEFAULT;  depthStencilDesc.BindFlags = D3D11\_BIND\_DEPTH\_STENCIL;  depthStencilDesc.CPUAccessFlags = 0;  depthStencilDesc.MiscFlags = 0;    ComPtr<ID3D11Texture2D> mDepthStencilBuffer;  hr = m\_d3dDevice->CreateTexture2D(&depthStencilDesc, nullptr, mDepthStencilBuffer.GetAddressOf());  DX::ThrowIfFailed(hr);    CD3D11\_DEPTH\_STENCIL\_VIEW\_DESC dsvDesc(D3D11\_DSV\_DIMENSION\_TEXTURE2D, DXGI\_FORMAT\_D24\_UNORM\_S8\_UINT);    hr = m\_d3dDevice->CreateDepthStencilView(mDepthStencilBuffer.Get(), &dsvDesc, m\_dynamicCubeMapDSV.GetAddressOf());  DX::ThrowIfFailed(hr);    ...    CD3D11\_VIEWPORT dynamicCubeMapViewport(0.f, 0.f, m\_cubeMapSize, m\_cubeMapSize);  m\_d3dContext->RSSetViewports(1, &dynamicCubeMapViewport); |

**设置摄像机**

如前所述，我们以摄像机作为原点，沿着坐标轴的6个方向将场景渲染6次，存储到相应的立方体贴图的对应面里。

通过以下代码计算出相应的观察矩阵

|  |
| --- |
| // Generate the cube map about the given position.  XMFLOAT4 center(x, y, z, 1.f);    // Look along each coordinate axis.  XMFLOAT4 targets[6] =  {  XMFLOAT4(x + 1.0f, y, z, 1.f), // +X  XMFLOAT4(x - 1.0f, y, z, 1.f), // -X  XMFLOAT4(x, y + 1.0f, z, 1.f), // +Y  XMFLOAT4(x, y - 1.0f, z, 1.f), // -Y  XMFLOAT4(x, y, z + 1.0f, 1.f), // +Z  XMFLOAT4(x, y, z - 1.0f, 1.f) // -Z  };    // Use world up vector (0,1,0) for all directions except +Y/-Y. In these cases, we  // are looking down +Y or -Y, so we need a different "up" vector.  XMFLOAT4 ups[6] =  {  XMFLOAT4(0.0f, 1.0f, 0.0f, 0.f), // +X  XMFLOAT4(0.0f, 1.0f, 0.0f, 0.f), // -X  XMFLOAT4(0.0f, 0.0f, -1.0f, 0.f), // +Y  XMFLOAT4(0.0f, 0.0f, +1.0f, 0.f), // -Y  XMFLOAT4(0.0f, 1.0f, 0.0f, 0.f), // +Z  XMFLOAT4(0.0f, 1.0f, 0.0f, 0.f) // -Z  };    XMVECTOR pos = XMLoadFloat4(&center);  XMVECTOR target = XMLoadFloat4(&targets[i]);  XMVECTOR up = XMLoadFloat4(&ups[i]);    view = XMMatrixLookAtLH(pos, target, up); |

投影矩阵也要计算相应的值

|  |
| --- |
| XMMATRIX proj = XMMatrixPerspectiveFovLH(0.5\*XM\_PI, 1.f, m\_nearZ, m\_farZ);  XMStoreFloat4x4(&m\_proj, proj); |

最后我们只需设置相应的RenderTarget，绘制出6个方向即可。