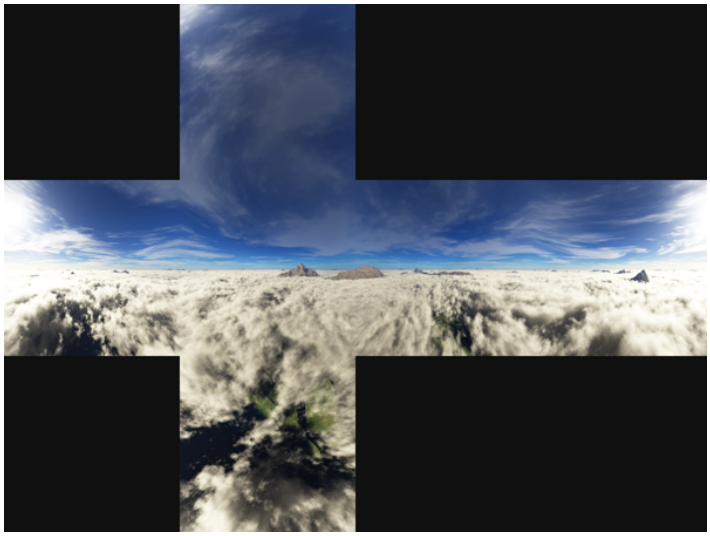
20. Cube Mapping (Skybox)

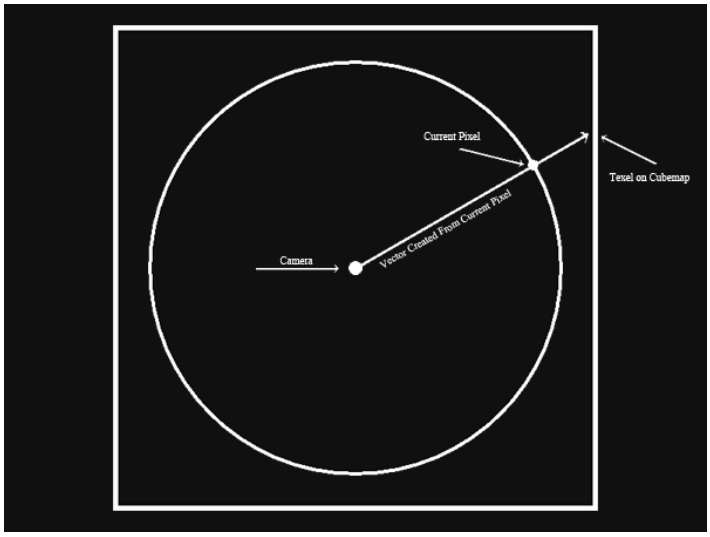
In this lesson, we will learn how to use a 3D texture to texture a sphere. This technique is called cube mapping, and we will learn how to make a skybox using this technique.

Introduction

Cube Mapping is a technique often used to create a surrounding environment infinitely far away, such as the sky. We can create a sky box using this technique by loading a 3D texture, then using this 3D image to texture a sphere that surrounds the camera. To keep this illusion up, the skybox must always be centered around the camera, so no matter how far in any direction we move, we will never get closer to the skybox. Another thing is the skybox is always in the back of the depth buffer, so everything drawn onto the screen will always be in front of the skybox. We can do this by setting the z value in our view space to 1. We will learn how to create a cube texture using the directx texture tool. A cube texture is six images, two for each axis (one for +x, -x, +y, -y, +z, and -z). In the file, the image would be stored like the one below:



To map the "cube" texture onto a sphere, its actually not too difficult. All we have to do is take the position of the current pixel we are rendering, and turn it into a vector to find which texel from the cube map we will be coloring the pixel. The below diagram shows you the idea in 2D, which we would be using (u,v) texture coordinates, but in 3D, the idea is the same, but instead of the 2D (u,v) texture coordinates, we will be using 3D (u,v,w) texture coordinates.



You can search the skymap from [Braynzar Vision]

We used a program called Terragen to create these pictures. Very cool program. two things we should mention though, is first, all six of the images must be the exact same dimensions, such as 1024x1024. also if you do decide to use Terragen to create cube maps, you might have a problem with "seams" between each picture, this is usually caused by "global illumination" or GI. you will need to turn both the GI settings to zero in the render window if you are having problems with seams.

Here is how to create a 3D texture using the directx texture tool.

Step 1 - Create 6 pictures representing the different sides of a cube; East (+X), West (-X), Up (+Y), Down (-Y), North (+Z), South (-Z) using a program such as terragen.

Step 2 - Open the DirectX texture tool (usually by going to start->all programs->microsoft directx SDK->DirectX Utilities->DirectX Texture Tool)

Step 3 - Go to File-New Texture...

Step 4 - A window should pop up. under the first section, where it says texture type, mark Cubemap

Texture. Change the second part to the x and y dimensions of your textures (remember all six of the images must be the same dimensions). Then you can change to format to a more compressed type if you'd like since these files can get pretty large. Hit OK.

Step 5 - Go to View->Cube Map Face-> and choose one of the faces (you will eventually choose all faces). Then go to File->Open Onto This Cubemap Face and choose the image corresponding with the current direction you chose (if you chose to view Positive X, then you will need to load the image in the East direction). Do this for all six directions.

Step 6 - After you have done that, save the file, and thats it.

Now that we have all the preparation done, lets move on to the code

This lesson was based of the last lesson, First person camera. However, we modified it quite a bit and we are not going to go through all the modifications, so you'll need to look at the source code to see what was changed

Global Declarations, New Include & New Function

We have a new function called CreateSphere(). This function will do what it says and create an index and vertex buffer we will use to map our sky to. This function uses Vectors, which are dynamic arrays, so we must include the vector header. Our skybox will also use a separate vertex and pixel shader from our other geometry, so we need to declare those here. Since we are loading in a texture we will pass to the shader to texture the sphere, we need to create a new shader resource view, which we call smrv for skymap resource view. You can see we have two new render states. one is a rasterizer state, which we will use to disable backface culling, since when we create our sphere some triangles are facing the opposite way as the others. And a Depth/Stencil state, we will use to make sure our sky is always behind all other geometry in the world. We have two new integers, one for the vertices of our sphere, and the other for the faces. And last we have a new matrix which defines our spheres world space.

```
#include <vector>

ID3D11Buffer* sphereIndexBuffer;
ID3D11Buffer* sphereVertBuffer;

ID3D11VertexShader* SKYMAP_VS;
ID3D11PixelShader* SKYMAP_PS;
ID3D10Blob* SKYMAP_VS_Buffer;
ID3D10Blob* SKYMAP_PS_Buffer;

ID3D11ShaderResourceView* smrv;

ID3D11DepthStencilState* DSLessEqual;
ID3D11RasterizerState* RSCullNone;

int NumSphereVertices;
int NumSphereFaces;

XMMATRIX sphereWorld;

void CreateSphere(int LatLines, int LongLines);
```

Clean Up

Remember!

```
void CleanUp()
{
    SwapChain->SetFullscreenState(false, NULL);
    PostMessage(hwnd, WM_DESTROY, 0, 0);

    //Release the COM Objects we created
    SwapChain->Release();
    d3d11Device->Release();
    d3d11DevCon->Release();
    renderTargetView->Release();
    squareVertBuffer->Release();
    squareIndexBuffer->Release();
    VS->Release();
    PS->Release();
    VS_Buffer->Release();
    PS_Buffer->Release();
    vertLayout->Release();
    depthStencilView->Release();
    depthStencilBuffer->Release();
    cbPerObjectBuffer->Release();
    Transparency->Release();
    CCWcullMode->Release();
    CWcullMode->Release();

    d3d101Device->Release();
    keyedMutex11->Release();
    keyedMutex10->Release();
    D2DRenderTarget->Release();
    Brush->Release();
    BackBuffer11->Release();
    sharedTex11->Release();
    DWriteFactory->Release();
    TextFormat->Release();
    d2dTexture->Release();
```

The Sky Geometry!

This is the function we will call to create a sphere, which we will map our sky to. Since you could just map the sky onto a cube, I'm not going to explain this function, you can look through it if you would like though, but this certain function is not exactly needed to map a sky.

```
void CreateSphere(int LatLines, int LongLines)
{
    NumSphereVertices = ((LatLines-2) * LongLines) + 2;
    NumSphereFaces  = ((LatLines-3)*(LongLines)*2) + (LongLines*2);

    float sphereYaw = 0.0f;
    float spherePitch = 0.0f;

    std::vector<Vertex> vertices(NumSphereVertices);

    XMVECTOR currVertPos = XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f);

    vertices[0].pos.x = 0.0f;
    vertices[0].pos.y = 0.0f;
    vertices[0].pos.z = 1.0f;

    for(DWORD i = 0; i < LatLines-2; ++i)
    {
        spherePitch = (i+1) * (3.14/(LatLines-1));
        Rotationx = XMMatrixRotationX(spherePitch);
        for(DWORD j = 0; j < LongLines; ++j)
        {
            sphereYaw = j * (6.28/(LongLines));
            Rotationy = XMMatrixRotationZ(sphereYaw);
            currVertPos = XMVector3TransformNormal( XMVectorSet(0.0f, 0.0f, 1.0f, 0.0f)
            currVertPos = XMVector3Normalize( currVertPos );
            vertices[i*LongLines+j+1].pos.x = XMVectorGetX(currVertPos);
            vertices[i*LongLines+j+1].pos.y = XMVectorGetY(currVertPos);
            vertices[i*LongLines+j+1].pos.z = XMVectorGetZ(currVertPos);
        }
    }

    vertices[NumSphereVertices-1].pos.x =  0.0f;
```

Call the Create Sphere Function

Here we will call the function to create our sphere. We will give it 10 latitude lines, and 10 longitude lines. By the way, when the sphere is created, its actually created on its side, so the "south" and "north" poles are parallel to the ground.

```
CreateSphere(10, 10);
```

The Skymap's VS and PS

Next we need to create our new shaders.

```
hr = D3DX11CompileFromFile(L"Effects.fx", 0, 0, "SKYMAP_VS", "vs_4_0", 0, 0, 0, &SKYMAP_VS_Buff
hr = D3DX11CompileFromFile(L"Effects.fx", 0, 0, "SKYMAP_PS", "ps_4_0", 0, 0, 0, &SKYMAP_PS_Buff

hr = d3d11Device->CreateVertexShader(SKYMAP_VS_Buffer->GetBufferPointer(), SKYMAP_VS_Buffe
hr = d3d11Device->CreatePixelShader(SKYMAP_PS_Buffer->GetBufferPointer(), SKYMAP_PS_Buffer
```

Loading the Cube Map

A Cube Map is actually an array of six 2D textures, two for each axis (x, -x, y, -y, z, -z). The first thing we do when loading a cube map is tell D3D we will be loading a texture cube, by creating a D3DX11_IMAGE_LOAD_INFO structure and setting its MiscFlags member with D3D11_RESOURCE_MISC_TEXTURECUBE.

Then we will create a 2D texture from the file. This 2D texture will actually be an array now since we said we are loading a texture cube.

Next, we get the description of our texture so we can create a resource view description that matches the texture we loaded in.

Now, we will create the shader resource view description. We will say that this resource view is a texture cube, or an array of 2D textures, so when the pixel shader is texturing a pixel, it will know how to use the 3D coordinates we give it, which are used to find the texel on the texture cube. Remember a 2D texture uses (u, v) coordinates, well a 3D texture uses (u, v, w) coordinates.

And finally we create the resource view using the texture we loaded in from a file, the shader resource views description, and storing the shader resource view in smrv.

```
D3DX11_IMAGE_LOAD_INFO loadSMInfo;
loadSMInfo.MiscFlags = D3D11_RESOURCE_MISC_TEXTURECUBE;

ID3D11Texture2D* SMTexture = 0;
hr = D3DX11CreateTextureFromFile(d3d11Device, L"skymap.dds",
    &loadSMInfo, 0, (ID3D11Resource**)&SMTexture, 0);

D3D11_TEXTURE2D_DESC SMTextureDesc;
SMTexture->GetDesc(&SMTextureDesc);

D3D11_SHADER_RESOURCE_VIEW_DESC SMViewDesc;
SMViewDesc.Format = SMTextureDesc.Format;
SMViewDesc.ViewDimension = D3D11_SRV_DIMENSION_TEXTURECUBE;
SMViewDesc.TextureCube.MipLevels = SMTextureDesc.MipLevels;
SMViewDesc.TextureCube.MostDetailedMip = 0;

hr = d3d11Device->CreateShaderResourceView(SMTexture, &SMViewDesc, &smrv);
```

The Skymap's Render States

At the bottom of our InitScene() function, we create two new render states. One Rasterizer state, and one depth/stencil state. The rasterizer state will be used to disable culling, so that both the back and front sides of our skybox's geometry will be rendered. We do this because when we created our sphere, half of the sides are backwards, and the other half are forwards.

Now let's look at the depth/stencil state. We create a depth/stencil state by filling out a D3D11_DEPTH_STENCIL_DESC structure, calling ID3D11Device::CreateDepthStencilState(), and storing the state in an ID3D11DepthStencilState object.

In our skybox's VS, we will be setting its z value to 1.0f (which we will do by setting z to w, since w is 1.0f). We do this so that it will always be the furthest possible from the camera, to make it look like it is infinitely far away, since all objects in your scene will be under the sky (usually). Just setting the z value to 1.0f in the vertex shader is not enough though, as some other object in your scene that are very far away might also end up with a z value of 1.0f. Because of this, we need to make sure that any other objects with the same z value or less (closer to the screen) are drawn, and not the pixels from the skybox. We can do this by setting the DepthFunc member to D3D11_COMPARISON_LESS_EQUAL.

```
cmdesc.CullMode = D3D11_CULL_NONE;
hr = d3d11Device->CreateRasterizerState(&cmdesc, &RSCullNone);

D3D11_DEPTH_STENCIL_DESC dssDesc;
ZeroMemory(&dssDesc, sizeof(D3D11_DEPTH_STENCIL_DESC));
dssDesc.DepthEnable = true;
dssDesc.DepthWriteMask = D3D11_DEPTH_WRITE_MASK_ALL;
dssDesc.DepthFunc = D3D11_COMPARISON_LESS_EQUAL;

d3d11Device->CreateDepthStencilState(&dssDesc, &DSLessEqual);
```

Keeping the Skybox Infinitely Far Away

To keep up the skybox illusion, we need to make sure that no matter how far in any direction the camera moves, we will never get any closer to the skybox, or else the illusion will not work. We will do this by making sure the skybox sphere is centered around the camera every frame. Another thing is it does not actualy matter what the size of the skybox is. You can try changing the size of the skybox, but it will always appear the same no matter how big or small it is.

To give the sky a slightly "flatter" look, you might want to keep the y value slightly less than the x and z values when scaling the sphere, so its kind of like a "squashed" sphere.

```
void UpdateScene(double time)
{
    //Reset cube1World
    groundWorld = XMMatrixIdentity();

    //Define cube1's world space matrix
    Scale = XMMatrixScaling( 500.0f, 10.0f, 500.0f );
    Translation = XMMatrixTranslation( 0.0f, 10.0f, 0.0f );

    //Set cube1's world space using the transformations
    groundWorld = Scale * Translation;

    /////////////**************new**************////////////////////
    //Reset sphereWorld
    sphereWorld = XMMatrixIdentity();

    //Define sphereWorld's world space matrix
    Scale = XMMatrixScaling( 5.0f, 5.0f, 5.0f );
    //Make sure the sphere is always centered around camera
    Translation = XMMatrixTranslation( XMVectorGetX(camPosition), XMVectorGetY(camPosit

    //Set sphereWorld's world space using the transformations
    sphereWorld = Scale * Translation;
    /////////////**************new**************////////////////////
}
```

Drawing the Sky

The last thing we need to do (besides the effect file), is draw the sphere our sky will be mapped to. First, we set the spheres index and vertex buffer. Then we set its shader variables, WVP and World matrices, texture, and sampler. After that we set its VS, PS, Depth/Stencil State and Rasterizer state. Then we draw the sphere. After, we need to make sure we set the states back to default in case other geometry drawn after this one do not explicitly set the default states and shaders.

```
d3d11DevCon->IASetIndexBuffer( sphereIndexBuffer, DXGI_FORMAT_R32_UINT, 0);
d3d11DevCon->IASetVertexBuffers( 0, 1, &sphereVertBuffer, &stride, &offset );

WVP = sphereWorld * camView * camProjection;
cbPerObj.WVP = XMMatrixTranspose(WVP);
cbPerObj.World = XMMatrixTranspose(sphereWorld);
d3d11DevCon->UpdateSubresource( cbPerObjectBuffer, 0, NULL, &cbPerObj, 0, 0 );
d3d11DevCon->VSSetConstantBuffers( 0, 1, &cbPerObjectBuffer );
d3d11DevCon->PSSetShaderResources( 0, 1, &smrv );
d3d11DevCon->PSSetSamplers( 0, 1, &CubesTexSamplerState );

d3d11DevCon->VSSetShader(SKYMAP_VS, 0, 0);
d3d11DevCon->PSSetShader(SKYMAP_PS, 0, 0);
d3d11DevCon->OMSetDepthStencilState(DSLessEqual, 0);
d3d11DevCon->RSSetState(RSCullNone);
d3d11DevCon->DrawIndexed( NumSphereFaces * 3, 0, 0 );

d3d11DevCon->VSSetShader(VS, 0, 0);
d3d11DevCon->OMSetDepthStencilState(NULL, 0);
```

The Effect File

The first thing thats new in our effect file is a variable called SkyMap, which is a TextureCube resource. TextureCube is like the Texture2D, but instead of one texture, it holds an array of 6 texture2D's, two for each axis.

```
TextureCube SkyMap;
```

We also have a new structure. This structure is used to send the information we need from the skymaps VS to the skymaps PS. We will only be needing the position and texture coordinates in our PS.

```
struct SKYMAP_VS_OUTPUT    //output structure for skymap vertex shader
{
    float4 Pos : SV_POSITION;
    float3 texCoord : TEXCOORD;
};
```

Here is our skymaps VS. As you can see, we are taking in normal values for each vertex still, this is because we are going to be using the same input layout for all our geometry (so we don't have to do more code). However, we will not be using the normals. And in fact, we did not define texture coordinates for our vertices either. This is because the texture coordinates will be defined by our vertices position. We can use our vertices position as a vector, describing the texel in our texturecube to color the pixel with. You can see how we do that with the line output.texCoord = inPos;. Also, notice when we are defining the output position, we take the values .xyww instead of .xyzw. This is because w is equal to 1.0f. We want to make sure our skymap is always the furthest object in our scene, so we want to set our z value to 1.0f too, which is what w is. Remember, 1.0f is the furthest value from the screen.

```
SKYMAP_VS_OUTPUT SKYMAP_VS(float3 inPos : POSITION, float2 inTexCoord : TEXCOORD, float3 norma
{
    SKYMAP_VS_OUTPUT output = (SKYMAP_VS_OUTPUT)0;

    //Set Pos to xyww instead of xyzw, so that z will always be 1 (furthest from camera)
    output.Pos = mul(float4(inPos, 1.0f), WVP).xyww;

    output.texCoord = inPos;

    return output;
}
```

The only thing thats left to do now is create our skymaps PS. This is very easy. All we have to do is sample the texturecube using the 3D texture coordinate passed in from the VS, and return the color value of that texel!

```
float4 SKYMAP_PS(SKYMAP_VS_OUTPUT input) : SV_Target
{
    return SkyMap.Sample(ObjSamplerState, input.texCoord);
}
```

Pretty simple actually, be sure to let me know if you have any problems or comments!

Exercise:

1. Make a sphere in the center of your scene which reflects the skymap surrounding the scene.

Hint: To do this, you will need to send the cameras position to the skymaps VS stage. Take the position of the camera, and the position of the vertex, and create a unit length vector describing the direction from the pixel to the camera. After that you will set the texture coordinate of that vertex to the reflection of that vector. To create a reflection vector, you will also need the normal of the vertex. If you are doing a simple sphere or square, the normal could just be the vertices position, if the cube or sphere was centered around (0.0f, 0.0f, 0.0f).