Tutorial 12: Pixel Shaders

Collapse All

Summary

This tutorial focuses on the pixel shader and its capabilities. There are many things that can be accomplished with the pixel shader, and some of the more common functions are listed. The pixel shader will apply an environment map to the object.

The outcome of the tutorial is that the model will become shiny, and it will reflect its surrounding environment.

Source

SDK root\Samples\C++\Direct3D10\Tutorials\Tutorial12

Pixel Shaders

Pixel shaders are used to manipulate the final color of a pixel before it reaches the output merger. Each pixel that is rendered will be passed through this shader before output. Once the pixel has passed through the pixel shader, the only operations that can be performed are those performed by the output merger, such as alpha blending, depth testing, stencil testing, and so on.

The pixel shader evolved from the texture mapping found in early hardware. Instead of just a simple texture lookup, it is possible to compute the final color from multiple sources, as well as alter it according to the vertex data. For general applications, however, a pixel shader does multiple lookups on different textures.

Environment Map

On surfaces which are reflective, it is not possible to just do a texture lookup on fixed texture coordinates to determine the final color. This is because whenever the object or viewer moves, the reflection alters. Thus, we must update reflection every time something moves.

An effective method to trick the observer to believe it's a dynamic reflection of the environment is to generate a special texture that can wrap around the object. This texture is referred as a cube map. A cube map is effectively placing an object in the middle of a cube, with each face of the cube being a texture. An environment map is a cube map which has the textures which correspond to the view of the environment on that face. If the environment is static, these environment maps can be pre-generated. If the environment is dynamic, the cubemap needs to be updated on-the-fly. See the CubeMapGS sample for an illustration of how to do this.

From this environment map, we can calculate what a camera will see as the reflection. First we can find the direction the camera is viewing the object, and from that, reflect it off the normal of each pixel and perform a lookup based on that reflected vector.

Setting up the Environment Map

The setup of the environment map is not the emphasis of this tutorial. The procedure to follow is very similar to that of a normal texture map. Please refer to Tutorial 7, Texture Mapping and Constant Buffers, for an explanation of how to properly initialize a texture map and its associated resource view.

```
// Load the Environment Map
ID3D10Resource *pResource = NULL;
V_RETURN( DXUTFindDXSDKMediaFileCch( str, MAX_PATH, L"Lobby\\LobbyCube.dds" ) );
V_RETURN( D3DX10CreateTextureFromFile( pd3dDevice, str, NULL, NULL, &pResource ) );
if(pResource)
{
    g_pEnvMap = (ID3D10Texture2D*)pResource;
    pResource->Release();
    D3D10_TEXTURECUBE_DESC desc;
    g_pEnvMap->GetDesc( &desc );
    D3D10_SHADER_RESOURCE_VIEW_DESC SRVDesc;
```

```
    ZeroMemory( &SRVDesc, sizeof(SRVDesc) );
    SRVDesc.Format = desc.Format;
    SRVDesc.ViewDimension = D3D10_SRV_DIMENSION_TEXTURECUBE;
    SRVDesc.TextureCube.MipLevels = desc.MipLevels;
    SRVDesc.TextureCube.MostDetailedMip = 0;
    V_RETURN(pd3dDevice->CreateShaderResourceView( g_pEnvMap, &SRVDesc, &g_pEnvMapSRV ));
}

// Set the Environment Map
g_pEnvMapVariable->SetResource( g_pEnvMapSRV );
```

Implementing the Lookup

We will step through the simple algorithm described above and perform a proper lookup of an environment map. The calculations are done in the vertex shader and then interpolated to the pixel shader for the lookup. It is better to compute it in the vertex shader and interpolate to the pixel shader since there are fewer computations necessary.

To compute the reflected vector for the lookup, we require two pieces of information. First is the normal of the pixel in question, second is the direction of the eye.

Since the operation is done in view space, we must transform the pixel's normal to proper viewing space.

```
float3 viewNorm = mul( input.Norm, (float3x3)View );
```

Next we need to find the direction of the camera. Note, however, that we are already in view space, and thus, the camera's direction is that of the Z-axis (0,0,-1.0), since we are staring directly at the object.

Now that we have our two pieces of information, we can calculate the reflection of the vector with the **reflect** command. The variable **ViewR** is used to store the resulting reflection for processing in the pixel shader.

```
output.ViewR = reflect( viewNorm, float3(0,0,-1.0) );
```

Once the correct vector has been calculated in the vertex shader, we can process the environment map in the pixel shader. This is done by calling a built-in function to sample the environment map and return the color value from that texture.

```
// Load the environment map texture
float4 cReflect = g_txEnvMap.Sample( samLinearClamp, viewR );
```

Since it is all normalized (and texture coordinates range from 0 to 1), the *x* and *y* coordinates will correspond directly to the texture coordinates that it needs to look up.

As a bonus, we included the code to do the original flat texture lookup, and computed that as the diffuse term. To play with the amount of reflection, you can modulate *cReflect* by a scaling factor against *cDiffuse* and experiment with the results. You can have a very reflective character or a somewhat dull character.