Michael Schuld – mike@thehazymind.com – http://www.thehazymind.com

Writing a Custom Model Processor and BasicEffect

The Need for our own Basic Effect

Previously, our need for our own custom effect was related to the lack of support for certain features of the built in BasicEffect class on the Xbox 360. From the looks of the updated API though, these problems have since been remedied, so our needs must be updated to show that we should still create our own custom basic effect. My major reason for still including this tutorial is the general fact that learning shaders is hard, and that a tutorial series on XNA should include as many components of XNA as it can. So, without further ado, here comes a learning experience in HLSL, XNA Content Pipeline and shaders!

[To make it a bit easier to see what is supposed to be going on, I will include a picture of my demo after each section of the shader is added so everyone can be sure they are getting correct results.]

A Quick Vector Primer

Since lighting and pretty much any other shaders use a lot of vector math I thought I'd explain in general the functions I will use here. Hopefully this will help people less familiar with vector math to get through this tutorial without too much fuss.

A dot product of two vectors is defined like this:

```
u.v = ||u|| * ||v|| * cos(theta)
```

This basically means that the dot product equals the length of the two vectors times the angle between them. Now, we are most interested in the angle between them, and so taking advantage of the fact that we have normalized all of our vectors, the lengths are always 1. This makes the equation look like this:

```
u.v = 1 * 1 * cos(theta)
```

Basically, this allows us to find a reference to the angle between two vectors just using the dot product which just happens to be a built in shader function.

Since our lighting will need to be an amount between 0.0 and 1.0 (a basic brightness scale), using the already given cosine of the angle will be perfect. This is already the percent of light that would be received at that angle and can be used directly. The second two components of our lighting model included here (a basic Phong model) rely heavily on this fact. With this simple calculation, we can create quite a few lighting effects with ease.

In addition to the shader we are going to write, we also need a better way to get the material data for the models we want to render so we can tell our shader how to go about accomplishing its job.

Tutorial content is intellectual property of Hazy Mind Interactive and is not to be reproduced in any form without express written consent

To help us with that, we will create a custom model processor that will grab the material data when the model is being loaded and store it for use later on. This processor can be extended to handle all sorts of other effects like normal and specular maps as well (which we will do a bit later.)

Our Custom Content Pipeline

To begin the process, let's create a new Class Library project called HMContentPipeline in our solution. Inside our new project we need a few classes, HMModelProcessor and HMMaterialProcessor.

For now these processors will just extend their base classes and use a few simple overrides that do nothing out of the ordinary. This will basically just make our rendering code for our models a little simpler and set up the code we will need to add more complicated lighting effects in a bit.

The HMModelProcessor will start out looking like this:

```
[ContentProcessor(DisplayName = "Hazy Mind Model Processor")]
public class HMModelProcessor : ModelProcessor {
    private String modelDirectory;

    public override ModelContent Process(NodeContent input, ContentProcessorContext context) {
        if (input == null) {
            throw new ArgumentNullException("input");
        }
        modelDirectory = Path.GetDirectoryName(input.Identity.SourceFilename);

        return base.Process(input, context);
    }

    protected override MaterialContent ConvertMaterial(
        MaterialContent material, ContentProcessorContext context
    ) {
        var basicMaterial = new EffectMaterialContent {
            Effect = new ExternalReference<EffectContent>("../HMEngineContent/Shaders/BasicShader.fx")
        };

        var processorParameters = new OpaqueDataDictionary();
        processorParameters["ColorKeyColor"] = ColorKeyColor;
        processorParameters["ColorKeyEnabled"] = ColorKeyEnabled;
        processorParameters["TextureFormat"] = TextureFormat;
        processorParameters["GenerateMipmaps"] = GenerateMipmaps;
        processorParameters["ResizeTexturesToPowerOfTwo"] = ResizeTexturesToPowerOfTwo;

        // Copy any textures already added to the built in material to our custom basicMaterial
        foreach (KeyValuePair<String, ExternalReference<TextureContent>> texture in material.Textures) {
            basicMaterial.Textures.Add(texture.Key, texture.Value);
        }

        // Convert the material using the HMMaterialProcessor.
        // This will perform any special texture processing we need
        return context.Convert<MaterialContent, MaterialContent>(
            basicMaterial, typeof (HMMaterialProcessor).Name, processorParameters
        );
    }
}
```

Our processor overrides two of the built in methods of ModelProcesser, the Process and ConvertMaterial methods. In Process we simply grab the path to the directory the model is in, which we will use later on to get to the textures that go with it. ConvertMaterial creates our custom material and references what will be our new BasicShader.fx file as its effect. Then, we set up any parameters other parts of the ModelProcessor will use, copy any textures from the model's material to our new custom

material, and finally convert it using out custom MaterialProcessor so that we can add some custom handling to any unique textures we will use later on (this is mostly used for normal mapping and other advanced texture processing.)

Since we are making calls with the HMMaterialProcessor, we'd better get code in place for it as well. Here is what it looks like to start:

```
[ContentProcessor]
[DesignTimeVisible(false)]
public class HMMaterialProcessor : MaterialProcessor {
    protected override ExternalReference<TextureContent> BuildTexture(
        string textureName, ExternalReference<TextureContent> texture, ContentProcessorContext context
    ) {
        // Named textures will each have their own custom processor
        switch (textureName) {
            default:
                // Default processing for all others
                return base.BuildTexture(textureName, texture, context);
        }
    }
}
```

Right now, this material processor overrides the built in BuildTexture method and ends up actually just calling the method it just overrode. This is only here so that we can expand on it later on and accomplishes nothing useful right now.

Starting with a Simple Transform

Now that we have our custom content processors set up, we need to modify some of our rendering code and make our BasicShader to use with our models. To begin our process, create a new file in the HMEngineContent/Shaders folder called BasicShader.fx with the following code:

```
float4x4 World;
float4x4 View;
float4x4 Projection;

texture2D Texture;
sampler DiffuseSampler = sampler_state {
    Texture = <Texture>;
};

struct VS_INPUT {
    float4 Position        : POSITION0;
    float2 Texcoord        : TEXCOORD0;
};

VS_INPUT Transform(VS_INPUT Input) {
    VS_INPUT Output;

    float4 worldPosition   = mul(Input.Position, World);
    float4 viewPosition    = mul(worldPosition, View);

    Output.Position        = mul(viewPosition, Projection);
    Output.Texcoord        = Input.Texcoord;

    return Output;
}

float4 BasicShader(VS_INPUT Input) : COLOR0 {
    return tex2D(DiffuseSampler, Input.Texcoord);
}
```

```
technique TransformBasicShader {
    pass P0 {
        VertexShader = compile vs_2_0 Transform();
        PixelShader = compile ps_2_0 BasicShader();
    }
}
```

Now that we have our processors and shader set up, we need to get the model using it. Add a reference to the HMContentPipeline in the HMDemoContent project and set the Content Processor on our model to “Hazy Mind Model Processor”. We can delete the line that sets our model.Shader as well since this is now being set during the processing stage.

All that is left now is to make some modifications to the component manager and the model render loops. We will leverage some of the built in Model rendering functionality to simplify things a bit.

In HMComponentMananger, add a reference to HMEngine.HMObjects and change Draw to this:

```
public override void Draw(GameTime gameTime) {
    foreach (IHMRenderable render in renderable.Values) {
        if (render is HMModel) {
            // HMModel has effects compiled in and handles parameters passes itself
            render.Render(GraphicsDevice, null);
        } else {
            HMEffectManager.SetActiveShader(render.Shader);
            HMEffectManager.ActiveShader.SetParameters(render);

            foreach (EffectPass pass in HMEffectManager.ActiveShader.Effect.CurrentTechnique.Passes) {
                render.Render(GraphicsDevice, pass);
            }
        }
    }

    base.Draw(gameTime);
}
```

Since we are going to use the built in Model.Draw method that handles effect passes for us, we don’t need to do that part ourselves for HMModels. For all the other types of objects we will still handle passes like we have been.

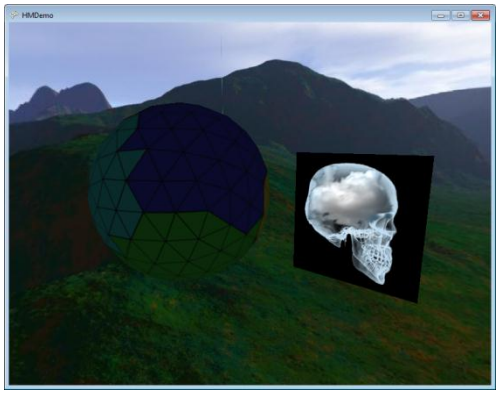
Now all that is left is to make a small addition to HMModel and update its Render method:

```
private Matrix[] transforms;

public override void Render(GraphicsDevice myDevice, EffectPass pass) {
    // Copy the current bone transformations for use below
    myModel.CopyAbsoluteBoneTransformsTo(transforms);

    Matrix world =
        HMCameraManager.ActiveCamera.World *
        Matrix.CreateScale(Scaling) *
        Matrix.CreateFromQuaternion(Rotation) *
        Matrix.CreateTranslation(Position);

    foreach (ModelMesh mesh in myModel.Meshes) {
        foreach(Effect effect in mesh.Effects) {
            effect.Parameters["World"].SetValue(transforms[mesh.ParentBone.Index] * world);
            effect.Parameters["View"].SetValue(HMCameraManager.ActiveCamera.View);
            effect.Parameters["Projection"].SetValue(HMCameraManager.ActiveCamera.Projection);
        }

        mesh.Draw();
```

```
    }
}
```

Running our demo now should look exactly as it has. Now let's get to adding our lighting!

Adding in some Ambient Light

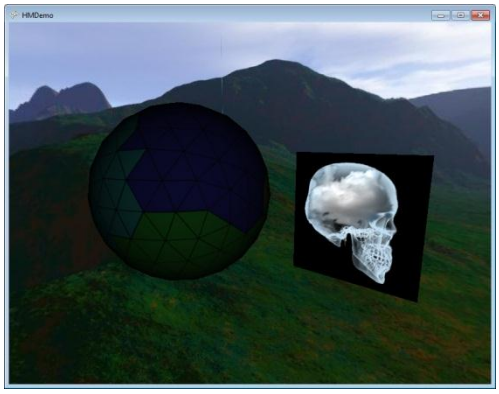


The best way I can think of describing ambient lighting (which of course involves snow, because it is minus 8 Fahrenheit outside right now) is to imagine a pitch black night in the middle of winter. Everything outside is still very visible because of the HUGE amount of scattering of any small lights around due to all the snow. This is referred to as ambient lighting. Ambient light is a very simple calculation that gives all parts of a mesh the same amount of light (although I do make it a bit more complicated in a second.) To use ambient light, we will add a constant to the top of our shader called "AmbientColor" (keeping close to BasicEffect parameters.) This will be a float3 that will store a light to base everything in the scene by. This light value will only be used by the pixel shader to calculate final pixel color. Here is the changed function for that:

```
float3 AmbientColor = (float3)0.2f;

float4 BasicShader(VS_INPUT Input) : COLOR0 {
    float4 TextureColor    = tex2D(DiffuseSampler, Input.Texcoord);
    float4 LightingColor   = float4(AmbientColor, 1.0f);
    return TextureColor * LightingColor;
};
```

We are using the LightingColor variable because it will have a lot of other components of the final light calculation added to it before we are finished.

Adding the Edge Component



An edge component is basically a number that describes the difference between the angle of a surface, and the direction that an object is being looked at from. To accomplish the addition of this component, we will need to send in the camera's current position to the shader so we can use it to find the view direction. In the shader, this will be called the "EyePosition" (keeping with the naming of parameters in the default BasicEffect). We will send this in the Render function of HMModel like the other parameters:

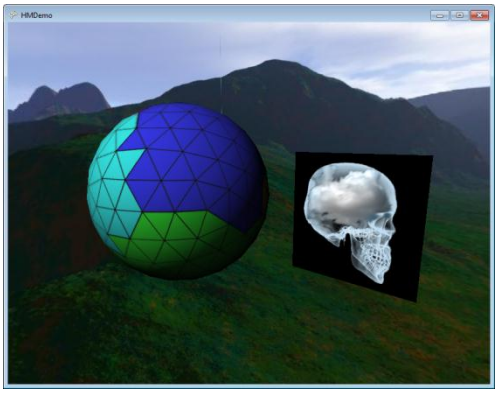
```
effect.Parameters["EyePosition"].SetValue(HMCameraManager.ActiveCamera.Position);
```

The shader itself will also use some normals calculation, but these are already being sent in by the VertexBuffer in the Render functions. To start the edge component calculation, we first need to translate the normal and create a ViewDirection for ourselves in the vertex shader and pass them into the pixel shader:

```
float3 EyePosition;

struct VS_INPUT {
    float4 Position        : POSITION0;
    float2 Texcoord        : TEXCOORD0;
    float3 Normal          : NORMAL0;
};

struct VS_OUTPUT {
    float4 Position        : POSITION0;
    float2 Texcoord        : TEXCOORD0;
    float3 Normal          : TEXCOORD1;
    float3 ViewDirection   : TEXCOORD2;
};

VS_OUTPUT Transform(VS_INPUT Input) {
    VS_OUTPUT Output;

    float4 worldPosition   = mul(Input.Position, World);
    float4 viewPosition    = mul(worldPosition, View);

    Output.Position        = mul(viewPosition, Projection);
    Output.Texcoord        = Input.Texcoord;
    Output.Normal          = mul(Input.Normal, World);
    Output.ViewDirection   = EyePosition - worldPosition;

    return Output;
}

struct PS_INPUT {
    float2 Texcoord        : TEXCOORD0;
    float3 Normal          : TEXCOORD1;
    float3 ViewDirection   : TEXCOORD2;
};

float4 BasicShader(PS_INPUT Input) : COLOR0 {
    float3 Normal = normalize(Input.Normal);
    float3 ViewDirection   = normalize(Input.ViewDirection);

    float EdgeComponent    = dot(Normal, ViewDirection);
    float3 TotalAmbient    = saturate(AmbientColor * EdgeComponent);

    float4 TextureColor    = tex2D(DiffuseSampler, Input.Texcoord);
    float4 LightingColor   = float4(TotalAmbient, 1.0f);

    return TextureColor * LightingColor;
}
```

We started by getting the object's position (after world transformation of course) and used that with the eye position to get a direction from the camera to the object. We can use this new direction vector and the passed through normal to find out whether a vertex is facing the camera or not in our pixel shader. Finally, at the end of the pixelshader, we return it and our lighting. This will be even more specific later on when only certain parts of a mesh have certain types of lighting.

Adding the Diffuse Lighting



Diffuse lighting is caused by directional lights hitting one side of an object but not the other. This is a fairly easy to visualize type of light (especially with all my lovely pictures), so I won't go into detail describing it. Diffuse lighting is most easily calculated with an essentially infinite distance light source (something like the sun), so I am going to pass in a light direction and not a position for now. We

are also going to need to pass in a color for the light. These will be passed the same way as our camera settings. To handle that, I'll add a simple light manager for now (similar to the camera manager).

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;

namespace HMEngine.HMLights {
    internal sealed class HMLightException : Exception {
        internal HMLightException(string s) : base(s) {}
    }

    public sealed class HMLightManager : GameComponent {
        private static readonly Dictionary<string, HMLight> lights = new Dictionary<string, HMLight>();
        private static HMLight DefaultLight { get; set; }

        public static HMLight ActiveLight { get; private set; }

        public HMLightManager(Game game) : base(game) {
            DefaultLight = new HMLight(new Vector3(0, 1000, 1000), Vector3.Zero, Color.White);

            AddLight("HMEngine.HMLights.Sun", DefaultLight);
            SetActiveLight("HMEngine.HMLights.Sun");
        }


        public static void AddLight(string name, HMLight light) {
            lights.Add(name, light);
        }

        public static void SetActiveLight(string name) {
            if (lights.ContainsKey(name)) {
                ActiveLight = lights[name];
            } else {
                throw new HMLightException("No light with the name " + name + " exists");
            }
        }
    }
}

// Add these parameters in the HMModel.Render loop
effect.Parameters["LightColor"].SetValue(HMLightManager.ActiveLight.Color.ToVector3());
effect.Parameters["LightDirection"].SetValue(HMLightManager.ActiveLight.Direction);
```

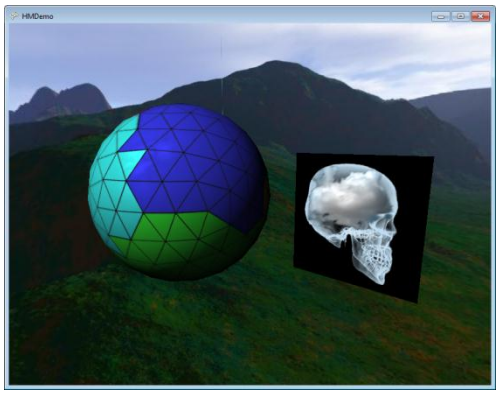
The light direction for our default sun is at 45 degrees in all directions down and away. These values will be changed by light objects later on, but for now we will just use these defaults. Diffuse light relies on the angle between the direction to the light and the surface normal. We already have the surface parts figured out for the edge component, so we just need to pass the needed values into the shader:

```
float3 LightColor;
float3 LightDirection;

float4 BasicShader(PS_INPUT Input) : COLOR0 {
    float3 Normal          = normalize(Input.Normal);
    float3 ViewDirection   = normalize(Input.ViewDirection);
    float3 NormalDirection = normalize(LightDirection);

    float EdgeComponent    = dot(Normal, ViewDirection);
    float3 TotalAmbient    = saturate(AmbientColor * EdgeComponent);

// Diffuse Calculations
    float NDotL            = dot(Normal, NormalDirection);
    float3 TotalDiffuse    = saturate(LightColor * NDotL);
// End Diffuse

    float4 TextureColor    = tex2D(DiffuseSampler, Input.Texcoord);
```

```
    float4 LightingColor   = float4(TotalAmbient + TotalDiffuse, 1.0f);

    return TextureColor * LightingColor;
};
```

The diffuse part of the basic shader just takes the dot product of the normal and the vector from the surface to the light position (-LightDirection) and multiplies it by an average between the light's and the object's diffuse colors. Adding this to the ambient color in the final part and returning it out finishes off this part of the lighting. Hopefully you read the vector primer at the beginning of this tutorial, or you already know enough about vector and lighting calculations that this isn't too complicated to look at.

Adding Specular Lighting and a Specular Map



For specular lighting, we need to once again add a couple of properties to the actual engine, but we will also be adding a specular map texture that will be handled with our custom model processor. The texture that I used for my specular map was just a simple photoshop cloud render. This makes a sort of oily or velvet looking pattern on the sphere but mostly just shows off the specular map really well. It's hard to see in the picture but moving the camera around in the running demo will show this effect well. For starters lets add a simple specular texture processor that will generate mip maps for our specular map:

```
using System.ComponentModel;
using Microsoft.Xna.Framework.Content.Pipeline;
using Microsoft.Xna.Framework.Content.Pipeline.Graphics;

namespace HMContentPipeline {
    [ContentProcessor]
    [DesignTimeVisible(false)]
    internal class HMSpecularProcessor : ContentProcessor<TextureContent, TextureContent> {
        public override TextureContent Process(TextureContent input, ContentProcessorContext context) {
            input.GenerateMipmaps(false);

            return input;
        }
    }
}
```

To use this texture processor we need to add a few things to our ModelProcessor and MaterialProcessor code. The new BuildTexture method in the MaterialProcessor looks like this:

```
protected override ExternalReference<TextureContent> BuildTexture(string textureName, ExternalReference<TextureC
ontent> texture, ContentProcessorContext context) {
    // Named textures will each have their own custom processor
    switch (textureName) {
        case HMModelProcessor.SPECULAR_MAP_KEY:
            return context.BuildAsset<TextureContent, TextureContent>(texture, typeof(HMSpecularProcessor).Name);

        default:
            // Default processing for all others
            return base.BuildTexture(textureName, texture, context);
    }
}
```

The switch-case now checks if the texture being processed is a specular map, and runs it through our custom specular processor (which just adds mipmaps for us.) In all other cases it will just use default processing.

In the ModelProcessor we need to add a few things. A constant string for naming our specular map:

```
public const string SPECULAR_MAP_KEY = "SpecularMap";
```

A setting we can use in the Visual Studio UI to tell the processor what texture to use for the map:

```
[DisplayName("Specular Map")]
[Description("This file will be used as the specular map on the model when it is set.")]
[DefaultValue("")]
public string SpecularMap { get; set; }
```

Finally we need a method to grab the specular map setting, and add the map as TextureContent to the materials loaded into our model. This will also handle adding the specular map to any child meshes:

```
private void AddSpecularMap(NodeContent node) {
    var mesh = node as MeshContent;
    if (mesh != null) {
        // If SpecularMap hasn't been set. Look in the OpaqueData for it, otherwise use the UI value
        string specularMap = String.IsNullOrEmpty(SpecularMap) ?
            mesh.OpaqueData.GetValue<string>(SPECULAR_MAP_KEY, null) : SpecularMap;

        if (specularMap == null) {
            throw new InvalidContentException("Specular Map missing!");
        }

        specularMap = Path.Combine(modelDirectory, specularMap);
        foreach (GeometryContent geometry in mesh.Geometry) {
            geometry.Material.Textures.Add(SPECULAR_MAP_KEY, new ExternalReference<TextureContent>(specularMap));
        }
    }

    foreach (NodeContent child in node.Children) {
        AddSpecularMap(child);
    }
}

// Update HMModelProcessor.Process()
public override ModelContent Process(NodeContent input, ContentProcessorContext context) {
    if (input == null) {
        throw new ArgumentNullException("input");
    }
    modelDirectory = Path.GetDirectoryName(input.Identity.SourceFilename);

    AddSpecularMap(input); // Add this line

    return base.Process(input, context);
}
```

The shader calculations for the specular component of our lighting actually takes advantage of the NDotL that we have already calculated to find the reflection vector of the light in the direction it would be bounced back off of the object, and takes another dot product between this new reflection vector and the view to see how much of that light is being bounced to the viewer. Here are the shader additions for this:

```
float SpecularPower    = 8; // Default to 8
```

```
// In BasicShader()
float3 Reflection      = normalize(2.0f * Normal * NDotL - NormalDirection);
float RDotV            = max(0.0f, dot(Reflection, ViewDirection));
float3 TotalSpecular
    = saturate(LightColor * pow(RDotV, SpecularPower) * tex2D(SpecularSampler, Input.Texcoord));

float4 LightingColor   = float4(TotalAmbient + TotalDiffuse + TotalSpecular, 1.0f);
```

There you have it. We now have all of the basic lighting allowed by BasicEffect put together into our own shader that we are free to do with what we would like. There are still a few properties in the BasicEffect that we haven't gotten to yet, but I will add those in another tutorial when we implement things like fog equations and other light types as well. For now, you should play around with different color and power properties of the light to get a better understanding of what everything in the shader is doing.

Last but not least, here is my version of the final full shader after putting this all together:

```
float4x4 World;
float4x4 View;
float4x4 Projection;

float3 AmbientColor           = (float3)0.1f;
float3 EyePosition;

float3 LightColor;
float3 LightDirection;

float SpecularPower           = 8; // Default to 8

texture2D Texture;
sampler DiffuseSampler = sampler state {
    Texture = <Texture>;
};

texture2D SpecularMap;
sampler SpecularSampler = sampler_state {
    Texture = SpecularMap;
};

struct VS_INPUT {
    float4 Position           : POSITION0;
    float2 Texcoord           : TEXCOORD0;
    float3 Normal             : NORMAL0;
};

struct VS_OUTPUT {
    float4 Position           : POSITION0;
    float2 Texcoord           : TEXCOORD0;
    float3 Normal             : TEXCOORD1;
    float3 ViewDirection      : TEXCOORD2;
};

VS OUTPUT Transform(VS INPUT Input) {
    VS OUTPUT Output;

    float4 worldPosition      = mul(Input.Position, World);
    float4 viewPosition       = mul(worldPosition, View);

    Output.Position           = mul(viewPosition, Projection);
    Output.Texcoord           = Input.Texcoord;
    Output.Normal             = mul(Input.Normal, World);
    Output.ViewDirection      = EyePosition - worldPosition;

    return Output;
}
```

```
struct PS_INPUT {
    float2 Texcoord           : TEXCOORD0;
    float3 Normal             : TEXCOORD1;
    float3 ViewDirection      : TEXCOORD2;
};

float4 BasicShader(PS_INPUT Input) : COLOR0 {
    float3 Normal             = normalize(Input.Normal);
    float3 ViewDirection      = normalize(Input.ViewDirection);
    float3 NormalDirection    = normalize(LightDirection);

    float EdgeComponent       = dot(Normal, ViewDirection);
    float3 TotalAmbient       = saturate(AmbientColor * EdgeComponent);

// Diffuse Calculations
    float NDotL               = dot(Normal, NormalDirection);
    float3 TotalDiffuse       = saturate(LightColor * NDotL);
// End Diffuse

// Specular Calculations
    float3 Reflection         = normalize(2.0f * Normal * NDotL - NormalDirection);
    float RDotV               = max(0.0f, dot(Reflection, ViewDirection));
    float3 TotalSpecular
    = saturate(LightColor * pow(RDotV, SpecularPower) * tex2D(SpecularSampler, Input.Texcoord));
// End Specular

    float4 TextureColor       = tex2D(DiffuseSampler, Input.Texcoord);
    float4 LightingColor      = float4(TotalAmbient + TotalDiffuse + TotalSpecular, 1.0f);

    return TextureColor * LightingColor;
    return tex2D(SpecularSampler, Input.Texcoord);
};

technique TransformBasicShader {
    pass P0 {
        VertexShader = compile vs_2_0 Transform();
        PixelShader = compile ps_2_0 BasicShader();
    }
}
```

Screenshots

For anyone who is looking to compare their output directly to my images, this is the keyboard handler I used to get the camera in the same place every time (just press R after the demo loads):

```
private static void keyboard_OnKeyReleased(Collection<Keys> keys) {
    if (keys.Contains(Keys.F)) {
        game.ToggleFullScreen();
    }
    if (keys.Contains(Keys.Escape)) {
        game.Exit();
    }
    if (keys.Contains(Keys.R)) {
        HMCameraManager.ActiveCamera.Position = new Vector3(0, 0, 3);
        HMCameraManager.ActiveCamera.Revolve(new Vector3(1, 0, 0), -20 * 0.01f);
        HMCameraManager.ActiveCamera.RevolveGlobal(new Vector3(0, 1, 0), 70 * 0.01f);
    }
}
```