Deferred Rendering in XNA 4

Written by Celal Cansin Kayi

Contents

Chapter 1: Deferred Rendering Theory

Traditional Forward Rendering is getting less appealing every day, once mobile (as in phones not laptops) GPU's start supporting Multiple Render Targets I think it'll be almost completely dead. With good reason of course! Deferred Rendering is the future, well considering how ubiquitous it is even right now, I guess it's the present.

What is "Deferred Rendering"?

Deferred Rendering is essentially about decoupling lighting from geometry rendering. It's much more than that however.

Deferred Rendering involves writing all the data needed from the geometry to several render targets and then instead of having to redraw the geometry to access certain data (eg. Position, Normal etc.) you just sample the render targets.

That is of course the heavily simplified explanation.

Are there any downsides to Deferred Rendering?

There are many downsides, which is why you should never just assume that Deferred Rendering is the best way to go. The reasons to NOT choose Deferred Rendering are pretty much as follows:

- Difficult to implement Transparent meshes and even if you do implement it, there is no guarantee it'll work very fast (it probably won't).
- Uses a lot of memory, especially in XNA since it has dropped support for the variety of formats you used to be able to use that made it easier.
- Requires a good GPU, if you're using Deferred Rendering, XNA really isn't the place for it, D3D10/D3D11/OpenGL and even D3D9 is. (Why did I write the tutorial in XNA then? XNA is so simple it would allow anyone to learn Deferred Rendering and apply it in their API of choice later)
- Is very shader intensive
- Slower than Forward Rendering in some cases

What are the upsides to Deferred Rendering?

If you can live with all those downsides, the upsides of Deferred Rendering are as follows:

- Can have multiple lights without blending issues of multi-pass lighting or multiple mesh drawing per light
- The less pixels the light affects, the less lighting computations in general, whereas in forward rendering, every pixel the mesh affects will have to go through lighting computation
- Works well with Post-Processing, such as SSAO

Chapter 2: Making the G-Buffer

Now that you know the basic "gist" of Deferred Rendering let's get to setting up the project shall we?

First off make a new class for the Deferred Renderer and just fill it in as follows:

```
class DeferredRenderer
{
    //Clear Shader
    Effect Clear;

    //GBuffer Shader
    Effect GBuffer;

    //Directional Light Shader
    Effect directionalLight;

    //Point Light Shader
    Effect pointLight;

    //Spot Light Shader
    Effect spotLight;

    //Composition Shader
    Effect compose;

    //LightMap BlendState
    BlendState LightMapBS;

    //GBuffer Targets
    RenderTargetBinding[] GBufferTargets;

    //GBuffer Texture Size
    Vector2 GBufferTextureSize;

    //Light Map Target
    RenderTarget2D LightMap;

    //Fullscreen Quad
    FullscreenQuad fsq;

    //Point Light Geometry
    Model pointLightGeometry;

    //Spot Light Geometry
    Model spotLightGeometry;

    //Get GBuffer
    public RenderTargetBinding[] getGBuffer() { return GBufferTargets; }

    //Constructor
    public DeferredRenderer(GraphicsDevice GraphicsDevice, ContentManager Content,
                            int Width, int Height)
    {
    }
}
```

You'll notice an undefined class there, "FullscreenQuad". Let's go ahead and create that class shall we… (for details on Fullscreen Rendering for XNA, correctly mapping texels, see References)

```
class FullscreenQuad
{
    //Vertex Buffer
    VertexBuffer vb;

    //Index Buffer
    IndexBuffer ib;

    //Constructor
    public FullscreenQuad(GraphicsDevice GraphicsDevice)
    {
        //Vertices
        VertexPositionTexture[] vertices =
        {
            new VertexPositionTexture(new Vector3(1, -1, 0), new Vector2(1, 1)),
            new VertexPositionTexture(new Vector3(-1, -1, 0), new Vector2(0, 1)),
            new VertexPositionTexture(new Vector3(-1, 1, 0), new Vector2(0, 0)),
            new VertexPositionTexture(new Vector3(1, 1, 0), new Vector2(1, 0))
        };

        //Make Vertex Buffer
        vb = new VertexBuffer(GraphicsDevice, VertexPositionTexture.VertexDeclaration,
                              vertices.Length, BufferUsage.None);
        vb.SetData<VertexPositionTexture>(vertices);

        //Indices
        ushort[] indices = { 0, 1, 2, 2, 3, 0 };

        //Make Index Buffer
        ib = new IndexBuffer(GraphicsDevice, IndexElementSize.SixteenBits,
                             indices.Length, BufferUsage.None);
        ib.SetData<ushort>(indices);
    }

    //Draw and Set Buffers
    public void Draw(GraphicsDevice GraphicsDevice)
    {
        //Set Vertex Buffer
        GraphicsDevice.SetVertexBuffer(vb);

        //Set Index Buffer
        GraphicsDevice.Indices = ib;

        //Draw Quad
      GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0, 4, 0, 2);
    }

    //Set Buffers Onto GPU
    public void ReadyBuffers(GraphicsDevice GraphicsDevice)
    {
        //Set Vertex Buffer
        GraphicsDevice.SetVertexBuffer(vb);

        //Set Index Buffer
        GraphicsDevice.Indices = ib;
    }
```

```
    //Draw without Setting Buffers
    public void JustDraw(GraphicsDevice GraphicsDevice)
    {
        //Draw Quad
      GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0, 4, 0, 2);
    }
}
```

This class has “Draw”, “ReadyBuffers” and “JustDraw” methods. “Draw” will set the buffers onto the GPU and draw, “ReadyBuffers” just sets the buffers and “JustDraw” draws without setting the buffers first.

The reason for this is for situations where you know you will be drawing the quad several times in a row so rather than set the buffers each time, you can set it once and draw multiple times without making that redundant call. We’ll make use of this in the Directional Lighting stage of Deferred Rendering.

With FullscreenQuad now properly defined, let’s go fill in the details of the DeferredRenderer’s constructor…

```
//Constructor
public DeferredRenderer(GraphicsDevice GraphicsDevice, ContentManager Content,
                        int Width, int Height)
{
    //Load Clear Shader
    Clear = Content.Load<Effect>("Effects/Clear");
    Clear.CurrentTechnique = Clear.Techniques[0];

    //Load GBuffer Shader
    GBuffer = Content.Load<Effect>("Effects/GBuffer");
    GBuffer.CurrentTechnique = GBuffer.Techniques[0];

    //Load Directional Light Shader
    directionalLight = Content.Load<Effect>("Effects/DirectionalLight");
    directionalLight.CurrentTechnique = directionalLight.Techniques[0];

    //Load Point Light Shader
    pointLight = Content.Load<Effect>("Effects/PointLight");
    pointLight.CurrentTechnique = pointLight.Techniques[0];

    //Load Spot Light Shader
    spotLight = Content.Load<Effect>("Effects/SpotLight");
    spotLight.CurrentTechnique = spotLight.Techniques[0];

    //Load Composition Shader
    compose = Content.Load<Effect>("Effects/Composition");
    compose.CurrentTechnique = compose.Techniques[0];
```

```
    //Create LightMap BlendState
    LightMapBS = new BlendState();
    LightMapBS.ColorSourceBlend = Blend.One;
    LightMapBS.ColorDestinationBlend = Blend.One;
    LightMapBS.ColorBlendFunction = BlendFunction.Add;
    LightMapBS.AlphaSourceBlend = Blend.One;
    LightMapBS.AlphaDestinationBlend = Blend.One;
    LightMapBS.AlphaBlendFunction = BlendFunction.Add;

    //Set GBuffer Texture Size
    GBufferTextureSize = new Vector2(Width, Height);

    //Initialize GBuffer Targets Array
    GBufferTargets = new RenderTargetBinding[3];

    //Intialize Each Target of the GBuffer
    GBufferTargets[0] = new RenderTargetBinding(new RenderTarget2D(GraphicsDevice,
                                                Width, Height, false,
                                                SurfaceFormat.Rgba64,
                                                DepthFormat.Depth24Stencil8));
    GBufferTargets[1] = new RenderTargetBinding(new RenderTarget2D(GraphicsDevice,
                                                Width, Height, false,
                                                SurfaceFormat.Rgba64,
                                                DepthFormat.Depth24Stencil8));
    GBufferTargets[2] = new RenderTargetBinding(new RenderTarget2D(GraphicsDevice,
                                                Width, Height, false,
                                                SurfaceFormat.Vector2,
                                                DepthFormat.Depth24Stencil8));

    //Initialize LightMap
    LightMap = new RenderTarget2D(GraphicsDevice, Width, Height, false,
                                  SurfaceFormat.Color, DepthFormat.Depth24Stencil8);

    //Create Fullscreen Quad
    fsq = new FullscreenQuad(GraphicsDevice);

    //Load Point Light Geometry
    pointLightGeometry = Content.Load<Model>("PointLightGeometry");

    //Load Spot Light Geometry
    spotLightGeometry = Content.Load<Model>("SpotLightGeometry");
}
```

I think all the shader loading doesn't really need an explanation, I'll explain the BlendState when we use it a bit later(Chapter 3) and the geometry will be explained later as well(chapter 4 and on). I will however explain the Multiple Render Targets in the form of the RenderTargetBinding array that is "GBufferTargets" at this point.

Rendering to Multiple Render Targets

The GBuffer is where we are gonna store all the data that you need the geometry for. The data we are storing however is too much for just one render target we could either render to some form of render target that allows more elements than just RGBA(doesn't exist) or just render to separate render targets at the same time.

When you write an ordinary Pixel Shader you set its output semantic to "COLOR0" and its return type as just float4 etc. When rendering to Multiple Render Targets however, you do (at its simplest form) this:

```
//Pixel Output Structure
struct PSO
{
	float4 Target0 : COLOR0;
	float4 Target1 : COLOR1;
	float4 Target2 : COLOR2;
	float4 Target3 : COLOR3;
};

//Pixel Shader
PSO PS()
{
    //Initialize Output
    PSO output;

    //Fill Target0
    output.Target0 = float4(0, 0, 0, 1);

    //Fill Target1
    output.Target1 = float4(1, 0, 0, 1);

    //Fill Target2
    output.Target2 = float4(0, 1, 0, 1);

    //Fill Target3
    output.Target3 = float4(0, 0, 1, 1);

    //Return Output
    return output;
}
```

The other thing you need to know is that under XNA, the render targets can be different formats but the total size of each has to be the same. So for example you can see in our DeferredRendering constructor that the first target in the GBuffer has a format of RGBA64 whereas the last has a format of Vector2, the total of each format is 64 bits, but individual elements get different numbers of bits. Under D3D10/11 and some cards in D3D9 you are not limited by this restriction.

Making the GBuffer

We have everything set up at this point so that we can now create the GBuffer for the current frame.

Add this function to your DeferredRenderer:

```
//GBuffer Creation
void MakeGBuffer(GraphicsDevice GraphicsDevice, List<Model> Models, BaseCamera Camera)
{
    //Set Depth State
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;

    //Set up global GBuffer parameters
    GBuffer.Parameters["View"].SetValue(Camera.View);
    GBuffer.Parameters["Projection"].SetValue(Camera.Projection);

    //Draw Each Model
    foreach (Model model in Models)
    {
        //Get Transforms
        Matrix[] transforms = new Matrix[model.Bones.Count];
        model.CopyAbsoluteBoneTransformsTo(transforms);

        //Draw Each ModelMesh
        foreach(ModelMesh mesh in model.Meshes)
        {
            //Draw Each ModelMeshPart
            foreach(ModelMeshPart part in mesh.MeshParts)
            {
                //Set Vertex Buffer
                GraphicsDevice.SetVertexBuffer(part.VertexBuffer, part.VertexOffset);

                //Set Index Buffer
                GraphicsDevice.Indices = part.IndexBuffer;

                //Set World
              GBuffer.Parameters["World"].SetValue(transforms[mesh.ParentBone.Index]);

                //Set WorldIT
GBuffer.Parameters["WorldViewIT"].SetValue(Matrix.Transpose(Matrix.Invert(transforms[m
esh.ParentBone.Index] * Camera.View)));

                //Set Albedo Texture
GBuffer.Parameters["Texture"].SetValue(part.Effect.Parameters["Texture"].GetValueTextu
re2D());

                //Set Normal Texture
GBuffer.Parameters["NormalMap"].SetValue(part.Effect.Parameters["NormalMap"].GetValueT
exture2D());

                //Set Specular Texture
GBuffer.Parameters["SpecularMap"].SetValue(part.Effect.Parameters["SpecularMap"].GetVa
lueTexture2D());

                //Apply Effect
                GBuffer.CurrentTechnique.Passes[0].Apply();
```

```
                //Draw
                GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                            part.NumVertices, part.StartIndex, part.PrimitiveCount);
            }
        }
    }

    //Set RenderTargets off
    GraphicsDevice.SetRenderTargets(null);
}
```

Obviously there are a few things missing here as the mesh already seems to have the GBuffer set as its Effect and we are referencing some BaseCamera class.

In order for the DeferredRenderer class to render a mesh, the mesh has to have been processed by our special Content Pipeline project. This is so that we have easy access to a meshes textures and to set the GBuffer effect onto it.

For the Camera, just go ahead and copy the Camera classes from the source code.

Before we set up the Content Pipeline we are first going to write up the GBuffer shader.

Writing the GBuffer Shader

The GBuffer shader is actually fairly standard when you think about it, we will be transforming the geometry in the standard way and then just writing out the values. In a more sophisticated system you would want to encode the normals in a more precise way though.

Add a folder in your Content project “Effects” and add a new Effect in there called “GBuffer.fx”, then fill it out like so:

```
//Vertex Shader Constants
float4x4 World;
float4x4 View;
float4x4 Projection;
float4x4 WorldViewIT;

//Color Texture
texture Texture;

//Normal Texture
texture NormalMap;

//Specular Texture
texture SpecularMap;
```

```
//Albedo Sampler
sampler AlbedoSampler = sampler_state
{
	texture = <Texture>;
	MINFILTER = LINEAR;
	MAGFILTER = LINEAR;
	MIPFILTER = LINEAR;
	ADDRESSU = WRAP;
	ADDRESSV = WRAP;
};

//NormalMap Sampler
sampler NormalSampler = sampler_state
{
	texture = <NormalMap>;
	MINFILTER = LINEAR;
	MAGFILTER = LINEAR;
	MIPFILTER = LINEAR;
	ADDRESSU = WRAP;
	ADDRESSV = WRAP;
};

//SpecularMap Sampler
sampler SpecularSampler = sampler_state
{
	texture = <SpecularMap>;
	MINFILTER = LINEAR;
	MAGFILTER = LINEAR;
	MIPFILTER = LINEAR;
	ADDRESSU = WRAP;
	ADDRESSV = WRAP;
};

//Vertex Input Structure
struct VSI
{
	float4 Position : POSITION0;
	float3 Normal : NORMAL0;
	float2 UV : TEXCOORD0;
	float3 Tangent : TANGENT0;
	float3 BiTangent : BINORMAL0;
};

//Vertex Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float2 UV : TEXCOORD0;
	float3 Depth : TEXCOORD1;
	float3x3 TBN : TEXCOORD2;
};
```

```
//Vertex Shader
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Transform Position
	float4 worldPosition = mul(input.Position, World);
	float4 viewPosition = mul(worldPosition, View);
	output.Position = mul(viewPosition, Projection);

	//Pass Depth
	output.Depth.x = output.Position.z;
	output.Depth.y = output.Position.w;
	output.Depth.z = viewPosition.z;

	//Build TBN Matrix
	output.TBN[0] = normalize(mul(input.Tangent, (float3x3)WorldViewIT));
	output.TBN[1] = normalize(mul(input.BiTangent, (float3x3)WorldViewIT));
	output.TBN[2] = normalize(mul(input.Normal, (float3x3)WorldViewIT));

	//Pass UV
	output.UV = input.UV;

	//Return Output
	return output;
}

//Pixel Output Structure
struct PSO
{
	float4 Albedo : COLOR0;
	float4 Normals : COLOR1;
	float4 Depth : COLOR2;
};

//Normal Encoding Function
half3 encode(half3 n)
{
	n = normalize(n);

	n.xyz = 0.5f * (n.xyz + 1.0f);

	return n;
}

//Normal Decoding Function
half3 decode(half4 enc)
{
	return (2.0f * enc.xyz- 1.0f);
}
```

```
//Pixel Shader
PSO PS(VSO input)
{
	//Initialize Output
	PSO output;

	//Pass Albedo from Texture
	output.Albedo = tex2D(AlbedoSampler, input.UV);

	//Pass Extra - Can be whatever you want, in this case will be a Specular Value
	output.Albedo.w = tex2D(SpecularSampler, input.UV).x;

	//Read Normal From Texture
	half3 normal = tex2D(NormalSampler, input.UV).xyz * 2.0f - 1.0f;

	//Transform Normal to WorldViewSpace from TangentSpace
	normal = normalize(mul(normal, input.TBN));

	//Pass Encoded Normal
	output.Normals.xyz = encode(normal);

	//Pass this instead to disable normal mapping
	//output.Normals.xyz = encode(normalize(input.TBN[2]));

	//Pass Extra - Can be whatever you want, in this case will be a Specular Value
	output.Normals.w = tex2D(SpecularSampler, input.UV).y;

	//Pass Depth(Screen Space, for lighting)
	output.Depth = input.Depth.x / input.Depth.y;

	//Pass Depth(View Space, for SSAO)
	output.Depth.g = input.Depth.z;

	//Return Output
	return output;
}

//Technique
technique Default
{
	pass p0
	{
		VertexShader = compile vs_3_0 VS();
		PixelShader = compile ps_3_0 PS();
	}
}
```

I don’t think there is anything too exotic happening in this shader. I made the normals encode and decode through functions so that in the event you would prefer a different encoding scheme you may implement such a thing with ease; If so I would suggest you see the References in this guide for a really good tutorial on normal encoding schemes.

The first target of the GBuffer is storing the Color(or Albedo) and the W component will store the SpecularIntensity as dictated by the SpecularMap.

The second target of the GBuffer is storing the View-Space Normals and the W component will store the SpecularPower.

The third target of the GBuffer is storing the Screen-Space Depth (for lighting) and the View-Space Depth(for SSAO later).

Creating the Content Pipeline Project

Now is the time to create the Content Pipeline Project for the Deferred Renderer as we now actually have a shader to reference.

Add a new Content Pipeline Extension Project to the Solution and name it "DeferredRenderingModel", clear it of any files and copy the "DeferredRenderingModel.cs" file from the provided source material to it.

All this pipeline does, without getting into the specifics, is either set a default texture or find a user specified one and set the GBuffer effect onto the model as well.

If you haven't already, copy all the non-shader Content that I provided in the archive to your content project, you can leave out the shaders if you want cause we'll be writing them anyway in the process of the tutorial.

Now you have to add the Content Pipeline reference to the Content Project to use it, so right click on the "References" of your Content Project(holding your models and effects etc.) and click "Add Reference", go to the "Projects" tab and select "DeferredRenderingModel".

Now open up the "Properties" of the model you want to load and select "DeferredRenderingModel" as its "Content Processor".

You can set up the models to use the provided textures or just leave them on the default if you wish.

Clearing the GBuffer

Clearing the GBuffer requires a special shader to be run on a fullscreen quad with them set. This is so that the Normal's can be correctly set to an encoded (0, 0, 0) or if you have any other values that need to be cleared to a default value as well.

In the Deferred Renderer class add this function to just clear the screen with the fullscreen quad, turning the depth to read only so as not to interrupt with the GBuffer creation later.

```
//Clear GBuffer
void ClearGBuffer(GraphicsDevice GraphicsDevice)
{
    //Set to ReadOnly depth for now...
    GraphicsDevice.DepthStencilState = DepthStencilState.DepthRead;

    //Set GBuffer Render Targets
    GraphicsDevice.SetRenderTargets(GBufferTargets);
```

```
    //Set Clear Effect
    Clear.CurrentTechnique.Passes[0].Apply();

    //Draw
    fsq.Draw(GraphicsDevice);
}
```

Okay now let's write the Clear shader, add a Clear.fx to your Effects folder in your Content Project.

We're just gonna pass the Quad's input straight through to the pixel shader and set all the targets to 0 or their encoded equivalent.

```
//Vertex Shader
float4 VS(float3 Position : POSITION0) : POSITION0
{
	return float4(Position, 1);
}

//Pixel Shader Out
struct PSO
{
	float4 Albedo : COLOR0;
	float4 Normals : COLOR1;
	float4 Depth : COLOR2;
};

//Normal Encoding Function
half3 encode(half3 n)
{
	n = normalize(n);

	n.xyz = 0.5f * (n.xyz + 1.0f);

	return n;
}

//Pixel Shader
PSO PS()
{
	//Initialize Output
	PSO output;

	//Clear Albedo to Transperant Black
	output.Albedo = 0.0f;

//Clear Normals to 0(encoded value is 0.5 but can't use normalize on 0, compile error)
	output.Normals.xyz = 0.5f;
	output.Normals.w = 0.0f;

	//Clear Depth to 1.0f
	output.Depth = 1.0f;

	//Return
	return output;
}
```

```
//Technique
technique Default
{
	pass p0
	{
		VertexShader = compile vs_3_0 VS();
		PixelShader = compile ps_3_0 PS();
	}
}
```

Putting it all together

We have written the meat of the Deferred Renderer at this point so now let's set up the rest of the project to see the fruits of our labour.

First off, rather than call each method in the order they should be called from your base Draw method of your game, we're gonna set up a Draw method for the Deferred Renderer to take care of this for us.

So with that, add the following to your Deferred Renderer:

```
//Draw Scene Deferred
public void Draw(GraphicsDevice GraphicsDevice, List<Model> Models, BaseCamera Camera)
{
    //Set States
    GraphicsDevice.BlendState = BlendState.Opaque;
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;

    //Clear GBuffer
    ClearGBuffer(GraphicsDevice);

    //Make GBuffer
    MakeGBuffer(GraphicsDevice, Models, Camera);
}
```

You'll notice this won't draw anything to screen, instead we are gonna set up a Debug method which we will also add to as the tutorial continues that will draw each buffer to the corner of the screen. It is extremely important to set up debug visualization facilities that are in engine, PIX and PerfHud are good but it's still good to have your own at easy access.

Let's add a Debug method to the Deferred Renderer:

```
//Debug
public void Debug(GraphicsDevice GraphicsDevice, SpriteBatch spriteBatch)
{
    //Begin SpriteBatch
    spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.Opaque,
                      SamplerState.PointClamp, null, null);
```

```
    //Width + Height
    int width = 128;
    int height = 128;

    //Set up Drawing Rectangle
    Rectangle rect = new Rectangle(0, 0, width, height);

    //Draw GBuffer 0
    spriteBatch.Draw((Texture2D)GBufferTargets[0].RenderTarget, rect, Color.White);

    //Draw GBuffer 1
    rect.X += width;
    spriteBatch.Draw((Texture2D)GBufferTargets[1].RenderTarget, rect, Color.White);

    //Draw GBuffer 2
    rect.X += width;
    spriteBatch.Draw((Texture2D)GBufferTargets[2].RenderTarget, rect, Color.White);

    //End SpriteBatch
    spriteBatch.End();
}
```

Just a simple manipulation of the SpriteBatch to draw each GBufferTarget to screen beside each other, you have to use Point sampling as in XNA4 you can't Linear sample vector targets like in XNA3.

Now add the following members to your Game class:

```
//Graphics Device
GraphicsDeviceManager graphics;

//SpriteBatch
SpriteBatch spriteBatch;

//Sprite Font
SpriteFont spriteFont;

//Keyboard State Holders
KeyboardState currentK;
KeyboardState previousK;

//Mouse State Holder
MouseState currentM;

//Camera
FreeCamera Camera;

//Deffered Renderer
DeferredRenderer deferredRenderer;

//List of Models
List<Model> models;
```

Initialize the Camera in your Initialize method of your Game class like so:

```
//Initialize Camera
Camera = new FreeCamera(this, new Vector3(0, 0, 5), new Vector3(0, 0, 0), Vector3.Up);
```

Okay now in your Load method, you're gonna need to add the following:

```
//Create Deferred Renderer
deferredRenderer = new DeferredRenderer(GraphicsDevice, Content,
graphics.PreferredBackBufferWidth, graphics.PreferredBackBufferHeight);

//Initialize Model List
models = new List<Model>();

//Load Models
Model scene = Content.Load<Model>("Scene");
models.Add(scene);
```

Okay now we're gonna set up the Update method so the Camera will actually work and you can exit the game easily (I can hear some developers right now "Why let the player quit the game at all!? Best Idea Ever!" *facepalm*):

```
//Input Housekeeping
previousK = currentK;
currentK = Keyboard.GetState();
currentM = Mouse.GetState();
Mouse.SetPosition(GraphicsDevice.Viewport.Width / 2, GraphicsDevice.Viewport.Height /
2);

//Exit check
if (currentK.IsKeyUp(Keys.Escape) && previousK.IsKeyDown(Keys.Escape)) this.Exit();

//Update Camera
Camera.Update(this, currentK, previousK, currentM);
```

Now all that's left is to Draw using the Deferred Renderer, so in your Game class Draw method add the following:

```
//Draw using Deferred Renderer
deferredRenderer.Draw(GraphicsDevice, models, Camera);

//Debug Deferred Renderer
deferredRenderer.Debug(GraphicsDevice, spriteBatch);
```

Now if you run it you may encounter some errors, because the Deferred Renderer's constructor references shaders that we haven't written yet, just comment out the loading lines that give you the runtime exceptions and everything should be fine.

Chapter 3: Handling Directional Lights

From here on things get a little more complex, especially for those of you who aren't very mathematically competent. Doesn't matter though, as long as you understand the general process everything should be easy enough.

There are plenty of good references to learn more about lighting theory so I won't get into the specifics, but there are two general lighting models, global and local.

Local lighting is the most encountered, especially with beginners as it's fairly simple to understand and implement, all it requires is the position, normal and lighting vector for the pixel being lit. You'll see this implemented here as some simple Phong shading.

Global lighting doesn't really have any easy or simple implementations, it usually requires a comprehensive framework and doesn't render with simple one pass lighting shaders. The reason for this is that while local illumination only approximates a single light bounce with no atmosphere or etc. taken into account, global illumination will attempt to recreate the light bounces, atmospheric condition, light absorption and reflection etc. Though recently it's become more and more viable as screen-space ray-tracing methods (such as with SSAO) become explored further. We will approximate some global illumination effects with SSAO, but really it's not a substitute for a full global illumination framework.

Directional Lighting Shader

We will add Directional Lighting capability to the Deferred Renderer first, because it's not too difficult and we can't add shadowing support for it (more on that in Chapter 4) so we can just focus on the lighting.

With the GBuffer made, the job of the pixel shader for the Directional Light will be to rebuild the data in World-Space and then compute lighting for that pixel. A Directional Light will affect every pixel on the screen so we will treat it like a fullscreen post processing pass.

So add a new shader to your effects folder in your content project, "DirectionalLight.fx" and fill it like so:

```
//Inverse View Projection
float4x4 InverseViewProjection;

//Inverse View
float4x4 inverseView;

//Camera Position
float3 CameraPosition;

//Light Vector
float3 L;

//Light Color
float4 LightColor;
```

```
//Light Intensity
float LightIntensity;

//GBuffer Texture Size
float2 GBufferTextureSize;

//GBuffer Texture0
sampler GBuffer0 : register(s0);

//GBuffer Texture1
sampler GBuffer1 : register(s1);

//GBuffer Texture2
sampler GBuffer2 : register(s2);

//Vertex Input Structure
struct VSI
{
	float3 Position : POSITION0;
	float2 UV : TEXCOORD0;
};

//Vertex Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float2 UV : TEXCOORD0;
};

//Vertex Shader
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Just Straight Pass Position
	output.Position = float4(input.Position, 1);

	//Pass UV too
	output.UV = input.UV - float2(1.0f / GBufferTextureSize.xy);

	//Return
	return output;
}

//Manually Linear Sample
float4 manualSample(sampler Sampler, float2 UV, float2 textureSize)
{
	float2 texelpos = textureSize * UV;
	float2 lerps = frac(texelpos);
	float texelSize = 1.0 / textureSize;

	float4 sourcevals[4];
	sourcevals[0] = tex2D(Sampler, UV);
	sourcevals[1] = tex2D(Sampler, UV + float2(texelSize, 0));
	sourcevals[2] = tex2D(Sampler, UV + float2(0, texelSize));
	sourcevals[3] = tex2D(Sampler, UV + float2(texelSize, texelSize));

	float4 interpolated = lerp(lerp(sourcevals[0], sourcevals[1], lerps.x),
	                           lerp(sourcevals[2], sourcevals[3], lerps.x ), lerps.y);
```

```
	return interpolated;
}

//Phong Shader
float4 Phong(float3 Position, float3 N, float SpecularIntensity, float SpecularPower)
{
	//Calculate Reflection vector
	float3 R = normalize(reflect(L, N));

	//Calculate Eye vector
	float3 E = normalize(CameraPosition - Position.xyz);

	//Calculate N.L
	float NL = dot(N, -L);

	//Calculate Diffuse
	float3 Diffuse = NL * LightColor.xyz;

	//Calculate Specular
	float Specular = SpecularIntensity * pow(saturate(dot(R, E)), SpecularPower);

	//Calculate Final Product
	return LightIntensity * float4(Diffuse.rgb, Specular);
}

//Decoding of GBuffer Normals
float3 decode(float3 enc)
{
	return (2.0f * enc.xyz- 1.0f);
}

//Pixel Shader
float4 PS(VSO input) : COLOR0
{
	//Get All Data from Normal part of the GBuffer
	half4 encodedNormal = tex2D(GBuffer1, input.UV);

	//Decode Normal
	half3 Normal = mul(decode(encodedNormal.xyz), inverseView);

	//Get Specular Intensity from GBuffer
	float SpecularIntensity = tex2D(GBuffer0, input.UV).w;

	//Get Specular Power from GBuffer
	float SpecularPower = encodedNormal.w * 255;

	//Get Depth from GBuffer
	float Depth = manualSample(GBuffer2, input.UV, GBufferTextureSize).x;

	//Calculate Position in Homogenous Space
	float4 Position = 1.0f;

	Position.x = input.UV.x * 2.0f - 1.0f;

	Position.y = -(input.UV.x * 2.0f - 1.0f);

	Position.z = Depth;

	//Transform Position from Homogenous Space to World Space
	Position = mul(Position, InverseViewProjection);

	Position /= Position.w;
```

```
	//Return Phong Shaded Value
	return Phong(Position.xyz, Normal, SpecularIntensity, SpecularPower);
}

//Technique
technique Default
{
	pass p0
	{
		VertexShader = compile vs_3_0 VS();
		PixelShader = compile ps_3_0 PS();
	}
}
```

Quite a bit to take in, so let me give you an overview of the pixel shader:

1. Sample the Normal from the Normal Texture of the GBuffer, then decode it and transform it into World space by multiplying by the inverse of the View matrix.
2. Sample the Specular Intensity and Specular Power from the GBuffer, multiply Specular Power by 255 to get its actual value, you can change that 255 to anything else if you want the maximum Specular Power to be greater (note that the higher the Specular Power the smaller the Specular will be, try it out when you get the chance).
3. Manually Linear Sample the Screen-Space Depth from the GBuffer, the reason for the manual linear sampling is because XNA4 doesn't support Linear Sampling of Vector formats so we have to do it ourselves (check the references section for more information on manual linear sampling a 2D texture, in Chapter 5 we will tackle manual linear sampling of a 3D texture).
4. Use the UV's to calculate the Screen-Space (X, Y) Position of the quad and set its Z to the depth we sampled in step 3. To understand how this works, take any UV of the quad and multiply it by 2 then minus 1.
5. Transform from Screen-Space to World Space by multiplying the Screen-Space Position calculated in step 4 by the inverse of the ViewProjection matrix. Then make sure to divide the Position by the W component.
6. At this point you now have the Normal in World-Space and the Position in World-Space, so you can treat this point and further as though the model was rendered normally really.

It's quite a busy shader as you can see, hence why Deferred Rendering shouldn't be automatically seen as the best way to do lighting.

Understanding Blend States

Now that we have a lighting shader we need to set up the Deferred Renderer to make use of it and create a Lightmap.

The most important aspect of this step is to understand the Blend State, because without this you will be unable to have more than one light.

You may recall that in the constructor for the Deferred Renderer in the previous chapter we made the Light Map Blend State like so:

```
//Create LightMap BlendState
LightMapBS = new BlendState();
LightMapBS.ColorSourceBlend = Blend.One;
LightMapBS.ColorDestinationBlend = Blend.One;
LightMapBS.ColorBlendFunction = BlendFunction.Add;
LightMapBS.AlphaSourceBlend = Blend.One;
LightMapBS.AlphaDestinationBlend = Blend.One;
LightMapBS.AlphaBlendFunction = BlendFunction.Add;
```

To understand how this works, imagine we just lit a pixel using the directional light shader from before (where the xyz components of the pixel are the Diffuse component of the light and the w component is the Specular component of the light) and we have now applied a second light:

pixelColor.xyz = (currentPixelColor.xyz * BlendState.ColorDestinationBlend)
(BlendState.ColorBlendFunction)
(newPixelColor.xyz * BlendState.ColorSourceBlend)

pixelColor.w = (currentPixelColor.w * BlendState.AlphaDestinationBlend)
(BlendState.AlphaBlendFunction)
(newPixelColor.w * BlendState.AlphaSourceBlend)

The currentPixelColor is equal to the lighting value that we have already drawn to screen and the newPixelColor is the lighting value we just calculated in this pass. The BlendFunction for the Color and the Alpha is set to ”Add” so the equation is basically adding the two values together and they are automatically clamped from 0 to 1.

Making the Light-Map

Okay let’s get to making the Light-Map, in the Deferred Renderer add the following function:

```
//Light Map Creation
void MakeLightMap(GraphicsDevice GraphicsDevice, LightManager Lights,
                  BaseCamera Camera)
{
    //Set LightMap Target
    GraphicsDevice.SetRenderTarget(LightMap);

    //Clear to Transperant Black
    GraphicsDevice.Clear(Color.Transparent);
```

```
    //Set States
    GraphicsDevice.BlendState = LightMapBS;
    GraphicsDevice.DepthStencilState = DepthStencilState.DepthRead;

    #region Set Global Samplers
    //GBuffer 1 Sampler
    GraphicsDevice.Textures[0] = GBufferTargets[0].RenderTarget;
    GraphicsDevice.SamplerStates[0] = SamplerState.LinearClamp;

    //GBuffer 2 Sampler
    GraphicsDevice.Textures[1] = GBufferTargets[1].RenderTarget;
    GraphicsDevice.SamplerStates[1] = SamplerState.LinearClamp;

    //GBuffer 3 Sampler
    GraphicsDevice.Textures[2] = GBufferTargets[2].RenderTarget;
    GraphicsDevice.SamplerStates[2] = SamplerState.PointClamp;

    //SpotLight Cookie Sampler
    GraphicsDevice.SamplerStates[3] = SamplerState.LinearClamp;

    //ShadowMap Sampler
    GraphicsDevice.SamplerStates[4] = SamplerState.PointClamp;
    #endregion

    //Calculate InverseView
    Matrix InverseView = Matrix.Invert(Camera.View);

    //Calculate InverseViewProjection
    Matrix InverseViewProjection = Matrix.Invert(Camera.View * Camera.Projection);

    //Set Directional Lights Globals
 directionalLight.Parameters["InverseViewProjection"].SetValue(InverseViewProjection);
    directionalLight.Parameters["inverseView"].SetValue(InverseView);
    directionalLight.Parameters["CameraPosition"].SetValue(Camera.Position);
    directionalLight.Parameters["GBufferTextureSize"].SetValue(GBufferTextureSize);

    //Set the Directional Lights Geometry Buffers
    fsq.ReadyBuffers(GraphicsDevice);

    //Draw Directional Lights
    foreach (Lights.DirectionalLight light in Lights.getDirectionalLights())
    {
        //Set Directional Light Parameters
   directionalLight.Parameters["L"].SetValue(Vector3.Normalize(light.getDirection()));
        directionalLight.Parameters["LightColor"].SetValue(light.getColor());
        directionalLight.Parameters["LightIntensity"].SetValue(light.getIntensity());

        //Apply
        directionalLight.CurrentTechnique.Passes[0].Apply();

        //Draw
        fsq.JustDraw(GraphicsDevice);
    }

    //Set States Off
    GraphicsDevice.BlendState = BlendState.Opaque;
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
}
```

Quite a few things missing there, the LightManager class and a Directional Light class; so let's take care of them shall we?

Add a folder to your project "Lights" and add to that a new class "DirectionalLight" under the namespace "Lights" then fill it like so:

```
class DirectionalLight
{
    //Direction
    Vector3 direction;

    //Color
    Vector4 color;

    //Intensity
    float intensity;

    #region Get Functions
    //Get Direction
    public Vector3 getDirection() { return direction; }

    //Get Color
    public Vector4 getColor() { return color; }

    //Get Intensity
    public float getIntensity() { return intensity; }
    #endregion

    #region Set Functions
    //Set Direction
    public void setDirection(Vector3 dir) { dir.Normalize(); this.direction = dir; }

    //Set Color
    public void setColor(Vector4 color) { this.color = color; }

    //Set Color
    public void setColor(Color color) { this.color = color.ToVector4(); }

    //Set Intensity
    public void setIntensity(float intensity) { this.intensity = intensity; }
    #endregion

    //Constructor
    public DirectionalLight(Vector3 Direction, Vector4 Color, float Intensity)
    {
        setDirection(Direction);
        setColor(Color);
        setIntensity(Intensity);
    }

    //Constructor
    public DirectionalLight(Vector3 Direction, Color Color, float Intensity)
    {
        setDirection(Direction);
        setColor(Color);
        setIntensity(Intensity);
    }
}
```

A Directional Light is really just defined by its Direction, Color and Intensity, so it's simple enough. The LightManager class will just hold all the lights of each type and will be fleshed out further in Chapter 4 and 5, for now just add a "LightManager" under the namespace "Lights" in the "Lights" folder and fill it out like so:

```
/// <summary>
/// A Manager for all the light types
/// </summary>
class LightManager
{
    //Directional Lights
    List<DirectionalLight> directionalLights;

    #region Get Functions
    //Get Directional Lights
    public List<DirectionalLight> getDirectionalLights() { return directionalLights; }
    #endregion

    //Constructor
    public LightManager(ContentManager Content)
    {
        //Initialize Directional Lights
        directionalLights = new List<DirectionalLight>();
    }

    //Add a Directional Light
    public void AddLight(DirectionalLight Light)
    {
        directionalLights.Add(Light);
    }

    //Remove a Directional Light
    public void RemoveLight(DirectionalLight Light)
    {
        directionalLights.Remove(Light);
    }
}
```

Make sure to add the Lights namespace to the usings of the Deferred Renderer and now let's move on to the next part.

Compositing the final image

At this point we're now going to blend everything together to produce the final image, so for that we'll need two things: a shader to composite the final render from the Light-Map and G-Buffer and also a function in the Deferred Renderer to interface with that shader.

Let's start with the shader, add a new effect file to your effects folder "Composition.fx" and fill it out like so:

```
float2 GBufferTextureSize;

sampler Albedo : register(s0);

sampler LightMap : register(s1);
```

```
//Vertex Input Structure
struct VSI
{
	float3 Position : POSITION0;
	float2 UV : TEXCOORD0;
};

//Vertex Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float2 UV : TEXCOORD0;
};

//Vertex Shader
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Pass Position
	output.Position = float4(input.Position, 1);

	//Pass Texcoord's
	output.UV = input.UV - float2(1.0f / GBufferTextureSize.xy);

	//Return
	return output;
}

//Pixel Shader
float4 PS(VSO input) : COLOR0
{
	//Sample Albedo
	float3 Color = tex2D(Albedo, input.UV).xyz;

	//Sample Light Map
	float4 Lighting = tex2D(LightMap, input.UV);

	//Accumulate to Final Color
	float4 output = float4(Color.xyz * Lighting.xyz + Lighting.w, 1);

	//Return
	return output;
}

//Technique
technique Default
{
	pass p0
	{
		VertexShader = compile vs_3_0 VS();
		PixelShader = compile ps_3_0 PS();
	}
}
```

You can now see how the Light-Map works, the Diffuse components of the lighting will multiply against the color texture sampled in the GBuffer creation stage and the Specular

value is merely added to that without any impact on individual colors. This Specular model is obviously incorrect but it's a worthwhile tradeoff.

Now let's get the Deferred Renderer to use this shader by adding the following function to it:

```
//Composition
void MakeFinal(GraphicsDevice GraphicsDevice, RenderTarget2D Output)
{
    //Set Composition Target
    GraphicsDevice.SetRenderTarget(Output);

    //Clear
    GraphicsDevice.Clear(Color.Transparent);

    //Set Textures
    GraphicsDevice.Textures[0] = GBufferTargets[0].RenderTarget;
    GraphicsDevice.SamplerStates[0] = SamplerState.LinearClamp;

    GraphicsDevice.Textures[1] = LightMap;
    GraphicsDevice.SamplerStates[1] = SamplerState.LinearClamp;

    //Set Effect Parameters
    compose.Parameters["GBufferTextureSize"].SetValue(GBufferTextureSize);

    //Apply
    compose.CurrentTechnique.Passes[0].Apply();

    //Draw
    fsq.Draw(GraphicsDevice);
}
```

Nothing too amazing going on there, just setting the textures to be used by the shader; though you will notice the "RenderTarget2D Output" parameter, this will allow easy control over where to render to and will help in building post-processing chains as you'll see later.

Now we must modify the Draw function of the Deferred Renderer for these changes:

```
//Draw Scene Deferred
public void Draw(GraphicsDevice GraphicsDevice, List<Model> Models,
                 LightManager Lights, BaseCamera Camera, RenderTarget2D Output)
{
    //Set States
    GraphicsDevice.BlendState = BlendState.Opaque;
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;

    //Clear GBuffer
    ClearGBuffer(GraphicsDevice);

    //Make GBuffer
    MakeGBuffer(GraphicsDevice, Models, Camera);
```

```
    //Make LightMap
    MakeLightMap(GraphicsDevice, Lights, Camera);

    //Make Final Rendered Scene
    MakeFinal(GraphicsDevice, Output);
}
```

Okay now we add a LightManager to the Game class and let's add a test light in the Load method:

```
//Create Light Manager
lightManager = new LightManager(Content);

//Add a Directional Light
lightManager.AddLight(new Lights.DirectionalLight(Vector3.Down, Color.White, 1.0f));
```

And make sure the Deferred Renderer makes use of it in the Draw method, also making sure to set the Output to NULL(oh wait it's C#, so it's null =P):

```
//Draw using Deferred Renderer
deferredRenderer.Draw(GraphicsDevice, models, lightManager, Camera, null);
```

Run it or move on, either way we're done; on to the next exciting chapter!

Chapter 4: Exponential Shadow Mapping Theory and Spot Light Implementation

Now that we understand the basics of lighting and recreating World-Space data from the G-Buffer, it's time for us to take it up a notch by adding dynamic shadow support. I would advise that you be familiar with the basics of projective texturing before continuing but it's not required.

Dynamic Shadows are pretty much done in two ways generally, though with a third very hacky and horrible way:

1. Stencil Shadowing, which as it implies, involves using the stencil buffer to mark shadowed pixels. Good if you want really really sharp shadows but horrible if you want soft shadows and is kind of inefficient and bothersome in some implementations. Last major game to have used this method was probably Doom 3, which holds up surprisingly well to today's standards.
2. Shadow Mapping, basically you draw the whole scene from the light's perspective and then project that texture onto the scene again comparing the projected textures depth from the current depth, it's easy to set up and easy to apply, looks nice when the Shadow Map resolution is decent and the texture can be blurred or modified in all sorts of ways to produce interesting looking shadows, also fairly cheap in terms of performance. You would have seen this at work in pretty much every SM2.0+ game made since 2004.
3. Drawing the Mesh to be Shadowed with a special "Shadow Matrix" that basically flattens the Mesh when it's transformed by it, looks horrible, only works on flat surfaces, looks horrible, stupid to implement, not that efficient, looks horrible, won't work well with deferred rendering… Should I go on?

Obviously from that list and the name of the chapter we will be implementing Shadow Mapping, but not just ANY shadow mapping; Exponential Shadow Mapping!

What's the difference you ask? Well there isn't much, but basically, exponential shadow mapping produces softer shadows that scale better based on their depth and look fairly nice at low shadow mapping resolutions too, this is because all depth written to the shadow map is done on an exponential rather than a linear curve, I would advise you switch your depth buffering to be on an exponential scale too as that can resolve some issues there too, but that's not gonna be covered here.

Creating the Shadow Maps

A Shadow Map is just a texture of purely depth values taken from a set of meshes drawn in the Light's perspective, so using a View-Projection specially made for a specific type of Light. The light we are dealing with is a Spot Light, this is pretty much the only model by which we can deal with Shadow Mapping as everything else is really just derivative of it.

I know the preceding is kinda confusing to beginners so it's really just best we code it first cause it's actually fairly straightforward.

Before we start making the Shadow Map, we need to define what a Spot Light is, so add a "SpotLight" class to the "Lights" folder under the namespace "Lights" and fill it out like so:

```
/// <summary>
/// A Spot Light with shadows
/// </summary>
class SpotLight
{
    //Position
    Vector3 position;

    //Direction
    Vector3 direction;

    //Color
    Vector4 color;

    //Intensity
    float intensity;

    //NearPlane
    float nearPlane;

    //FarPlane
    float farPlane;

    //FOV
    float FOV;

    //Is this Light with Shadows?
    bool isWithShadows;

    //Shadow Map Resoloution
    int shadowMapResoloution;

    //DepthBias for the Shadowing... (1.0f / 2000.0f)
    float depthBias;

    //World(for geometry in LightMapping phase...)
    Matrix world;

    //View
    Matrix view;

    //Projection
    Matrix projection;
```

```
    //Shadow Map
    RenderTarget2D shadowMap;

    //Attenuation Texture
    Texture2D attenuationTexture;

    #region Get Functions
    //Get Position
    public Vector3 getPosition() { return position; }

    //Get Direction
    public Vector3 getDirection() { return direction; }

    //Get Color
    public Vector4 getColor() { return color; }

    //Get Intensity
    public float getIntensity() { return intensity; }

    //Get NearPlane
    public float getNearPlane() { return nearPlane; }

    //Get FarPlane
    public float getFarPlane() { return farPlane; }

    //Get FOV
    public float getFOV() { return FOV; }

    //Get IsWithShadows
    public bool getIsWithShadows() { return isWithShadows; }

    //Get ShadowMapResoloution
    public int getShadowMapResoloution() { if (shadowMapResoloution < 2048)
                                               return shadowMapResoloution;
                                           else
                                               return 2048; }

    //Get DepthBias
    public float getDepthBias() { return depthBias; }

    //Get World
    public Matrix getWorld() { return world; }

    //Get View
    public Matrix getView() { return view; }

    //Get Projection
    public Matrix getProjection() { return projection; }

    //Get ShadowMap
    public RenderTarget2D getShadowMap() { return shadowMap; }

    //Get Attenuation Texture
    public Texture2D getAttenuationTexture() { return attenuationTexture; }
    #endregion

    #region Set Functions
    //Set Position
    public void setPosition(Vector3 position) { this.position = position; }
```

```
    //Set Direction
    public void setDirection(Vector3 direction) { direction.Normalize();
                                                  this.direction = direction; }

    //Set Color
    public void setColor(Vector4 color) { this.color = color; }

    //Set Color
    public void setColor(Color color) { this.color = color.ToVector4(); }

    //Set Intensity
    public void setIntensity(float intensity) { this.intensity = intensity; }

    //Set isWithShadows
    public void setIsWithShadows(bool iswith) { this.isWithShadows = iswith; }

    //Set DepthBias
    public void setDepthBias(float bias) { this.depthBias = bias; }

    //Set Attenuation Texture
    public void setAttenuationTexture(Texture2D attenuationTexture) {
                                  this.attenuationTexture = attenuationTexture; }
    #endregion

    //Constructor
    public SpotLight(GraphicsDevice GraphicsDevice, Vector3 Position,
                     Vector3 Direction, Vector4 Color, float Intensity,
                     bool isWithShadows, int ShadowMapResoloution,
                     Texture2D AttenuationTexture)
    {
        //Position
        setPosition(Position);

        //Direction
        setDirection(Direction);

        //Color
        setColor(Color);

        //Intensity
        setIntensity(Intensity);

        //NearPlane
        nearPlane = 1.0f;

        //FarPlane
        farPlane = 100.0f;

        //FOV
        FOV = MathHelper.PiOver2;

        //Set whether Is With Shadows
        setIsWithShadows(isWithShadows);

        //Shadow Map Resoloution
        shadowMapResoloution = ShadowMapResoloution;

        //Depth Bias
        depthBias = 1.0f / 2000.0f;
```

```
        //Projection
        projection = Matrix.CreatePerspectiveFieldOfView(FOV, 1.0f, nearPlane,
                                                         farPlane);

        //Shadow Map
        shadowMap = new RenderTarget2D(GraphicsDevice, getShadowMapResoloution(),
                                       getShadowMapResoloution(), false,
                                       SurfaceFormat.Single,
                                       DepthFormat.Depth24Stencil8);

        //Attenuation Texture
        attenuationTexture = AttenuationTexture;

        //Create View and World
        Update();
    }

    //Calculate the Cosine of the LightAngle
    public float LightAngleCos()
    {
        //float ConeAngle = 2 * atanf(Radius / Height);
        float ConeAngle = FOV;

        return (float)Math.Cos((double)ConeAngle);
    }

    //Update
    public void Update()
    {
        //Target
        Vector3 target = (position + direction);

        if (target == Vector3.Zero) target = -Vector3.Up;

        //Up
        Vector3 up = Vector3.Cross(direction, Vector3.Up);

        if (up == Vector3.Zero) up = Vector3.Right;
        else up = Vector3.Up;

        //ReMake View
        view = Matrix.CreateLookAt(position, target, up);

        //Make Scaling Factor
        float radial = (float)Math.Tan((double)FOV / 2.0) * 2 * farPlane;

        //Make Scaling Matrix
        Matrix Scaling = Matrix.CreateScale(radial, radial, farPlane);

        //Make Translation Matrix
    Matrix Translation = Matrix.CreateTranslation(position.X, position.Y, position.Z);

        //Make Inverse View
        Matrix inverseView = Matrix.Invert(view);

        //Make Semi-Product
        Matrix semiProduct = Scaling * inverseView;

        //Decompose Semi-Product
        Vector3 S; Vector3 P; Quaternion Q;
        semiProduct.Decompose(out S, out Q, out P);
```

```
        //Make Rotation
        Matrix Rotation = Matrix.CreateFromQuaternion(Q);

        //Make World
        world = Scaling * Rotation * Translation;
    }
}
```

A lot of code there but all you need to understand from this is that a Spot Light is defined by its direction, position and field of view. You also might have noticed the World, View and Projection matrices contained by this class, the World matrix will be explained when we get to the Light-Mapping stage of the Deferred Renderer later on, but the View and the Projection is because when creating a Shadow Map you basically have to treat the Spot Light as though it's a Camera and as you already know, we emulate a Camera through View and Projection transformations done via Matrices.

Now we revisit the LightManager class, to start off with let's modify it to include SpotLights:

```
/// <summary>
/// A Manager for all the light types
/// </summary>
class LightManager
{
    //Directional Lights
    List<DirectionalLight> directionalLights;

    //Spot Lights
    List<SpotLight> spotLights;

    #region Get Functions
    //Get Directional Lights
    public List<DirectionalLight> getDirectionalLights() { return directionalLights; }

    //Get Spot Lights
    public List<SpotLight> getSpotLights() { return spotLights; }
    #endregion

    //Constructor
    public LightManager(ContentManager Content)
    {
        //Initialize Directional Lights
        directionalLights = new List<DirectionalLight>();

        //Initialize Spot Lights
        spotLights = new List<SpotLight>();
    }

    //Add a Directional Light
    public void AddLight(DirectionalLight Light)
    {
        directionalLights.Add(Light);
    }
```

```
    //Add a Spot Light
    public void AddLight(SpotLight Light)
    {
        spotLights.Add(Light);
    }

    //Remove a Directional Light
    public void RemoveLight(DirectionalLight Light)
    {
        directionalLights.Remove(Light);
    }

    //Remove a Spot Light
    public void RemoveLight(SpotLight Light)
    {
        spotLights.Remove(Light);
    }
}
```

With the boring parts of the class now handled let's get to the Creating of the Shadow Maps!

Add the following function to the LightManager:

```
//Draw Shadow Maps
public void DrawShadowMaps(GraphicsDevice GraphicsDevice, List<Model> Models)
{
    //Set States
    GraphicsDevice.BlendState = BlendState.Opaque;
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;

    //Foreach SpotLight with Shadows
    foreach (SpotLight Light in spotLights)
    {
        //Update it
        Light.Update();

        //Draw it's Shadow Map
        if (Light.getIsWithShadows()) DrawShadowMap(GraphicsDevice, Light, Models);
    }
}
```

This function will take care of updating the SpotLights as well as sending them off to get their Shadow Map made. You'll note that while we're handling Spot Lights with shadows we will also be handling spot lights without in a kind of two-birds-one-stone tactic, though you could get better performance by separating the two.

Now let's flesh out the DrawShadowMap() function, all this is gonna do is set the View, Projection and other shadowing parameters then draw all the models.

```
//Draw a Shadow Map for a Spot Light
void DrawShadowMap(GraphicsDevice GraphicsDevice, SpotLight Light, List<Model> Models)
{
    //Set Light's Target onto the Graphics Device
    GraphicsDevice.SetRenderTarget(Light.getShadowMap());

    //Clear Target
    GraphicsDevice.Clear(Color.Transparent);

    //Set global Effect parameters
    depthWriter.Parameters["View"].SetValue(Light.getView());
    depthWriter.Parameters["Projection"].SetValue(Light.getProjection());
    depthWriter.Parameters["LightPosition"].SetValue(Light.getPosition());
    depthWriter.Parameters["DepthPrecision"].SetValue(Light.getFarPlane());

    //Draw Models
    DrawModels(GraphicsDevice, Models);
}
```

Yes that's right, this function calls yet another function. This may seem stupid but once we deal with Point Light Shadowing it's gonna make sense.

Now Add the DrawModels function:

```
//Draw Models
void DrawModels(GraphicsDevice GraphicsDevice, List<Model> Models)
{
    //Draw Each Model
    foreach (Model model in Models)
    {
        //Get Transforms
        Matrix[] transforms = new Matrix[model.Bones.Count];
        model.CopyAbsoluteBoneTransformsTo(transforms);

        //Draw Each ModelMesh
        foreach (ModelMesh mesh in model.Meshes)
        {
            //Draw Each ModelMeshPart
            foreach (ModelMeshPart part in mesh.MeshParts)
            {
                //Set Vertex Buffer
                GraphicsDevice.SetVertexBuffer(part.VertexBuffer, part.VertexOffset);

                //Set Index Buffer
                GraphicsDevice.Indices = part.IndexBuffer;

                //Set World
          depthWriter.Parameters["World"].SetValue(transforms[mesh.ParentBone.Index]);

                //Apply Effect
                depthWriter.CurrentTechnique.Passes[0].Apply();
```

```
                //Draw
   GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                                        part.NumVertices, part.StartIndex,
                                        part.PrimitiveCount);
            }
        }
    }
}
```

Make sure to add this private member in the LightManager class:

```
//Depth Writing Shader
Effect depthWriter;
```

And make sure to load it's (yet to be made) effect:

```
//Constructor
public LightManager(ContentManager Content)
{
    //Initialize Directional Lights
    directionalLights = new List<DirectionalLight>();

    //Initialize Spot Lights
    spotLights = new List<SpotLight>();

    //Load the Depth Writing Shader
    depthWriter = Content.Load<Effect>("Effects/DepthWriter");
    depthWriter.CurrentTechnique = depthWriter.Techniques[0];
}
```

Creating the Exponential Depth Writing Effect

Now that we have everything set up so that the models can be drawn to shadow map render target using the depth writing effect, let's create the depth writing effect.

Ordinarily when writing depth you just write the length of the vector from the light to the world position, but in the case of Exponential Shadow Mapping you also must encode this value onto an exponential curve.

Add a new Effect file in your Effects folder "DepthWriter.fx":

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

float3 LightPosition;

//This is for modulating the Light's Depth Precision
float DepthPrecision;
```

```
//Input Structure
struct VSI
{
	float4 Position : POSITION0;
};

//Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float4 WorldPosition : TEXCOORD0;
};

//Vertex Shader
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Transform Position
	float4 worldPosition = mul(input.Position, World);
	float4 viewPosition = mul(worldPosition, View);
	output.Position = mul(viewPosition, Projection);

	//Pass World Position
	output.WorldPosition = worldPosition;

	//Return Output
	return output;
}

//Pixel Shader
float4 PS(VSO input) : COLOR0
{
	//Fix World Position
	input.WorldPosition /= input.WorldPosition.w;

	//Calculate Depth from Light
float depth = max(0.01f, length(LightPosition - input.WorldPosition))/DepthPrecision;

	//Return Exponential of Depth
	return exp((DepthPrecision * 0.5f) * depth);
}

//Technique
technique Default
{
	pass p0
	{
		VertexShader = compile vs_3_0 VS();
		PixelShader = compile ps_3_0 PS();
	}
}
```

The Depth Precision variable there is important and is very dependent on your cameras farclip so if you are working with either really small or large scenes you may have to play around with that before you see any shadows.

Drawing a Spot Light onto the Light-Map

At this point we now have our Shadow-Map made so we turn our attention to drawing the SpotLight onto the Light-Map.

Remember how in the previous chapter we treated the Directional Light as a fullscreen post-processing pass? That's because a purely Directional Light will affect all the pixels on the screen as it doesn't really have any way of being attenuated, it's just a direction without an emanating position.

A SpotLight on the other hand does have an emanating position and thus attenuates based on that position and a special attenuation cookie texture that we sample with it. So if you want a Directional Light that doesn't affect all the pixels just use a special square attenuation texture with the SpotLight. I'm jumping the gun a bit though as unless you have dealt with lighting before you would have no idea what I'm talking about.

Basically from all that crazy rambling I'm trying to say that treating a SpotLight as a fullscreen post-processing pass would be ridiculous and inefficient so instead what we will do is draw some special cone geometry to screen so that only the pixels that will be affected by the SpotLight will go through all the processing.

In your DeferredRenderer classes MakeLightMap function (The highlighted part is the new part of the function):

```
//Light Map Creation
void MakeLightMap(GraphicsDevice GraphicsDevice, LightManager Lights, BaseCamera
Camera)
{
    //Set LightMap Target
    GraphicsDevice.SetRenderTarget(LightMap);

    //Clear to Transperant Black
    GraphicsDevice.Clear(Color.Transparent);

    //Set States
    GraphicsDevice.BlendState = LightMapBS;
    GraphicsDevice.DepthStencilState = DepthStencilState.DepthRead;

    #region Set Global Samplers
    //GBuffer 1 Sampler
    GraphicsDevice.Textures[0] = GBufferTargets[0].RenderTarget;
    GraphicsDevice.SamplerStates[0] = SamplerState.LinearClamp;

    //GBuffer 2 Sampler
    GraphicsDevice.Textures[1] = GBufferTargets[1].RenderTarget;
    GraphicsDevice.SamplerStates[1] = SamplerState.LinearClamp;

    //GBuffer 3 Sampler
    GraphicsDevice.Textures[2] = GBufferTargets[2].RenderTarget;
    GraphicsDevice.SamplerStates[2] = SamplerState.PointClamp;

    //SpotLight Cookie Sampler
    GraphicsDevice.SamplerStates[3] = SamplerState.LinearClamp;
```

```
    //ShadowMap Sampler
    GraphicsDevice.SamplerStates[4] = SamplerState.PointClamp;
    #endregion

    //Calculate InverseView
    Matrix InverseView = Matrix.Invert(Camera.View);

    //Calculate InverseViewProjection
    Matrix InverseViewProjection = Matrix.Invert(Camera.View * Camera.Projection);

    //Set Directional Lights Globals
directionalLight.Parameters["InverseViewProjection"].SetValue(InverseViewProjection);
    directionalLight.Parameters["inverseView"].SetValue(InverseView);
    directionalLight.Parameters["CameraPosition"].SetValue(Camera.Position);
    directionalLight.Parameters["GBufferTextureSize"].SetValue(GBufferTextureSize);

    //Set the Directional Lights Geometry Buffers
    fsq.ReadyBuffers(GraphicsDevice);

    //Draw Directional Lights
    foreach (Lights.DirectionalLight light in Lights.getDirectionalLights())
    {
        //Set Directional Light Parameters
  directionalLight.Parameters["L"].SetValue(Vector3.Normalize(light.getDirection()));
        directionalLight.Parameters["LightColor"].SetValue(light.getColor());
        directionalLight.Parameters["LightIntensity"].SetValue(light.getIntensity());

        //Apply
        directionalLight.CurrentTechnique.Passes[0].Apply();

        //Draw
        fsq.JustDraw(GraphicsDevice);
    }

    //Set Spot Lights Globals
    spotLight.Parameters["View"].SetValue(Camera.View);
    spotLight.Parameters["inverseView"].SetValue(InverseView);
    spotLight.Parameters["Projection"].SetValue(Camera.Projection);
    spotLight.Parameters["InverseViewProjection"].SetValue(InverseViewProjection);
    spotLight.Parameters["CameraPosition"].SetValue(Camera.Position);
    spotLight.Parameters["GBufferTextureSize"].SetValue(GBufferTextureSize);

    //Set Spot Lights Geometry Buffers
GraphicsDevice.SetVertexBuffer(spotLightGeometry.Meshes[0].MeshParts[0].VertexBuffer,
                               spotLightGeometry.Meshes[0].MeshParts[0].VertexOffset);
    GraphicsDevice.Indices = spotLightGeometry.Meshes[0].MeshParts[0].IndexBuffer;

    //Draw Spot Lights
    foreach (Lights.SpotLight light in Lights.getSpotLights())
    {
        //Set Attenuation Cookie Texture and SamplerState
        GraphicsDevice.Textures[3] = light.getAttenuationTexture();

        //Set ShadowMap and SamplerState
        GraphicsDevice.Textures[4] = light.getShadowMap();

        //Set Spot Light Parameters
        spotLight.Parameters["World"].SetValue(light.getWorld());
        spotLight.Parameters["LightViewProjection"].SetValue(light.getView()*
                                                             light.getProjection());
        spotLight.Parameters["LightPosition"].SetValue(light.getPosition());
```

```
        spotLight.Parameters["LightColor"].SetValue(light.getColor());
        spotLight.Parameters["LightIntensity"].SetValue(light.getIntensity());
        spotLight.Parameters["S"].SetValue(light.getDirection());
        spotLight.Parameters["LightAngleCos"].SetValue(light.LightAngleCos());
        spotLight.Parameters["LightHeight"].SetValue(light.getFarPlane());
        spotLight.Parameters["Shadows"].SetValue(light.getIsWithShadows());
      spotLight.Parameters["shadowMapSize"].SetValue(light.getShadowMapResoloution());
        spotLight.Parameters["DepthPrecision"].SetValue(light.getFarPlane());
        spotLight.Parameters["DepthBias"].SetValue(light.getDepthBias());

        #region Set Cull Mode
        //Calculate L
        Vector3 L = Camera.Position - light.getPosition();

        //Calculate S.L
        float SL = Math.Abs(Vector3.Dot(L, light.getDirection()));

        //Check if SL is within the LightAngle, if so then draw the BackFaces, if not
        //then draw the FrontFaces
        if (SL < light.LightAngleCos())
              GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
        else
              GraphicsDevice.RasterizerState = RasterizerState.CullClockwise;
        #endregion

        //Apply
        spotLight.CurrentTechnique.Passes[0].Apply();

        //Draw
        GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                                spotLightGeometry.Meshes[0].MeshParts[0].NumVertices,
                                 spotLightGeometry.Meshes[0].MeshParts[0].StartIndex,
                               spotLightGeometry.Meshes[0].MeshParts[0].PrimitiveCount);
    }

    //Set States Off
    GraphicsDevice.BlendState = BlendState.Opaque;
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
}
```

You might be bewildered at the sight of the “Set Cull Mode” region, the reason we need to set the cull mode for each individual light is because if we set it to Cull None then each pixel will be accumulate the lighting twice, if we never set the cull then nothing will be drawn when we are inside the light so it is necessary that we set the cull mode based on the dot product of the light direction and the camera to light vector.

Now we just need to write the SpotLight shader whose job it is to draw the geometry and light the pixels.

Add a new Effect to your effects folder “SpotLight.fx” and fill it like so:

```
//World
float4x4 World;

//View
float4x4 View;
```

```
//Inverse View
float4x4 inverseView;

//Projection
float4x4 Projection;

//Inverse View Projection
float4x4 InverseViewProjection;

//Camera Position
float3 CameraPosition;

//Light ViewProjection
float4x4 LightViewProjection;

//Light Position
float3 LightPosition;

//Light Color
float4 LightColor;

//Light Intensity
float LightIntensity;

//SpotLight Direction Vector
float3 S;

//Light Angle Cos
float LightAngleCos;

//The Height of the Light(AKA It's FarPlane)
float LightHeight;

//GBuffer Texture Size
float2 GBufferTextureSize;

//Shadows?
bool Shadows;

//ShadowMap size
float shadowMapSize;

//This is for modulating the Light's Depth Precision (100)
float DepthPrecision;

//DepthBias for the Shadowing... (1.0f / 2000.0f)
float DepthBias;

//GBuffer Texture0
sampler GBuffer0 : register(s0);

//GBuffer Texture1
sampler GBuffer1 : register(s1);

//GBuffer Texture2
sampler GBuffer2 : register(s2);

//Attenuation Cookie Sampler
sampler Cookie : register(s3);
```

```
//ShadowMap
sampler ShadowMap : register(s4);

//Vertex Input Structure
struct VSI
{
	float4 Position : POSITION0;
};

//Vertex Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float4 ScreenPosition : TEXCOORD0;
};

//Vertex Shader
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Transform Position
	float4 worldPosition = mul(input.Position, World);
	float4 viewPosition = mul(worldPosition, View);
	output.Position = mul(viewPosition, Projection);

	//Pass to ScreenPosition
	output.ScreenPosition = output.Position;

	//Return
	return output;
}

//Manually Linear Sample
float4 manualSample(sampler Sampler, float2 UV, float2 textureSize)
{
	float2 texelpos = textureSize * UV;
	float2 lerps = frac(texelpos);
	float texelSize = 1.0 / textureSize;

	float4 sourcevals[4];
	sourcevals[0] = tex2D(Sampler, UV);
	sourcevals[1] = tex2D(Sampler, UV + float2(texelSize, 0));
	sourcevals[2] = tex2D(Sampler, UV + float2(0, texelSize));
	sourcevals[3] = tex2D(Sampler, UV + float2(texelSize, texelSize));

	float4 interpolated = lerp(lerp(sourcevals[0], sourcevals[1], lerps.x),
lerp(sourcevals[2], sourcevals[3], lerps.x ), lerps.y);

	return interpolated;
}



//Phong Shader
float4 Phong(float3 Position, float3 N, float radialAttenuation,
		float SpecularIntensity, float SpecularPower)
{
	//Calculate Light vector
    float3 L = LightPosition.xyz - Position.xyz;
```

```
	//Calculate height Attenuation
	float heightAttenuation = 1.0f - saturate(length(L) - (LightHeight / 2));

	//Calculate total Attenuation
	float Attenuation = min(radialAttenuation, heightAttenuation);

	//Now Normalize the Light
	L = normalize(L);

	//Calculate L.S
	float SL = dot(L, S);

	//No asymmetrical returns in HLSL, so work around with this
	float4 Shading = 0;

	//If this pixel is in the SpotLights Cone
	if(SL <= LightAngleCos)
	{
		//Calculate Reflection Vector
		float3 R = normalize(reflect(-L, N));

		//Calculate Eye Vector
		float3 E = normalize(CameraPosition - Position.xyz);

		//Calculate N.L
		float NL = dot(N, L);

		//Calculate Diffuse
		float3 Diffuse = NL * LightColor.xyz;

		//Calculate Specular
		float Specular = SpecularIntensity * pow(saturate(dot(R, E)),
                                                 SpecularPower);

		//Calculate Final Product
		Shading = Attenuation * LightIntensity * float4(Diffuse.rgb, Specular);
	}

	//Return Shading Value
	return Shading;
}

//Decoding of GBuffer Normals
float3 decode(float3 enc)
{
	return (2.0f * enc.xyz- 1.0f);
}

//Decode Color Vector to Float Value for shadowMap
float RGBADecode(float4 value)
{
    const float4 bits = float4(1.0 / (256.0 * 256.0 * 256.0),
                               1.0 / (256.0 * 256.0),
                               1.0 / 256.0,
                               1);

    return dot(value.xyzw , bits);
}
```

```
//Pixel Shader
float4 PS(VSO input) : COLOR0
{
	//Get Screen Position
	input.ScreenPosition.xy /= input.ScreenPosition.w;

	//Calculate UV from ScreenPosition
	float2 UV = 0.5f * (float2(input.ScreenPosition.x, -input.ScreenPosition.y) +
                                        1) - float2(1.0f / GBufferTextureSize.xy);

	//Get All Data from Normal part of the GBuffer
	half4 encodedNormal = tex2D(GBuffer1, UV);

	//Decode Normal
	half3 Normal = mul(decode(encodedNormal.xyz), inverseView);

	//Get Specular Intensity from GBuffer
	float SpecularIntensity = tex2D(GBuffer0, UV).w;

	//Get Specular Power from GBuffer
	float SpecularPower = encodedNormal.w * 255;

	//Get Depth from GBuffer
	float Depth = manualSample(GBuffer2, UV, GBufferTextureSize).x;

	//Make Position in Homogenous Space using current ScreenSpace coordinates and
	//the Depth from the GBuffer
	float4 Position = 1.0f;

	Position.xy = input.ScreenPosition.xy;

	Position.z = Depth;

	//Transform Position from Homogenous Space to World Space
	Position = mul(Position, InverseViewProjection);

	Position /= Position.w;

	//Calculate Homogenous Position with respect to light
	float4 LightScreenPos = mul(Position, LightViewProjection);

	LightScreenPos /= LightScreenPos.w;

	//Calculate Projected UV from Light POV
	float2 LUV = 0.5f * (float2(LightScreenPos.x, -LightScreenPos.y) + 1);

	//Load the Projected Depth from the Shadow Map, do manual linear filtering
	float lZ = manualSample(ShadowMap, LUV, shadowMapSize);

	//Get Attenuation factor from cookie
	float Attenuation = tex2D(Cookie, LUV).r;

	//Assymetric Workaround...
	float ShadowFactor = 1;

	//If Shadowing is on then get the Shadow Factor
	if(Shadows)
	{
		// Calculate distance to the light
	float len = max(0.01f, length(LightPosition - Position)) / DepthPrecision;
```

```
            //Calculate the Shadow Factor
            ShadowFactor = (lZ * exp(-(DepthPrecision * 0.5f) * (len - DepthBias)));
      }

      //Return Phong Shaded Value Modulated by Shadows if Shadowing is on
return ShadowFactor * Phong(Position.xyz, Normal, Attenuation, SpecularIntensity,
                                                                  SpecularPower);
}

//Technique
technique Default
{
      pass p0
      {
            VertexShader = compile vs_3_0 VS();
            PixelShader = compile ps_3_0 PS();
      }
}
```

Okay so a couple of things need to be clarified here I think. First off you'll notice that the shadowing doesn't have any Boolean outcome like you would see in regular shadow-mapping, instead we just multiply the shadow-map depth by the current depth. That may seem ridiculous but it works, for more information on that see the references but really ESM theory is simple(just a simple exponential equation as seen here) but it's made unnecessarily difficult by the prose of all the text covering it.

You may also notice that we have two separate attenuation values, height and radial. The reason for this is that the light must attenuate based on distance from the cone's point, just like with a point light, but it also must attenuate based on the cones angle, which is where the radial attenuation comes in. You can do radial attenuation through some equations but when you involve shadow mapping it's way better to do it via texture.

And last of all, you'll notice that the way we recreate the world position differently from the directional light. If you just follow the equations it should be simple enough but basically all it's doing is taking the screen position and using the sampled depth then transforming back to world space as before. We set up the UV by just transforming from screen-space (x, y) which goes from (-1, -1) to (1, 1) to UV-space which goes from (0, 0) to (1, 1), which is pretty much the same as when we encoded the normals in the GBuffer stage; so just like with the shadow-map we're essentially projecting the texture to screen only with the default pov instead of the light's.

Putting it all together

Pretty much everything is done now, all that's left to do is set up the Game class to use the LightManager properly, so add the highlighted areas:

```
namespace TheCansinDeferredRenderer

{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
```

```
        protected override void LoadContent()
        {
            // Create a new SpriteBatch, which can be used to draw textures.
            spriteBatch = new SpriteBatch(GraphicsDevice);

            //Create Deferred Renderer
           deferredRenderer = new DeferredRenderer(GraphicsDevice, Content,
                                                   graphics.PreferredBackBufferWidth,
                                                   graphics.PreferredBackBufferHeight);

            //Create Light Manager
            lightManager = new LightManager(Content);

            //Load SpotLight Cookies
            Texture2D spotCookie = Content.Load<Texture2D>("SpotCookie");
            Texture2D squareCookie = Content.Load<Texture2D>("SquareCookie");

            //Add a Directional Light
            lightManager.AddLight(new Lights.DirectionalLight(Vector3.Down,
                                                              Color.White, 0.10f));

            //Add a Spot Light
lightManager.AddLight(new Lights.SpotLight(GraphicsDevice, new Vector3(0, 15.0f, 0),
                                           new Vector3(0, -1, 0),
                                           Color.White.ToVector4(), 1.0f, true, 2048,
                                           spotCookie));

            //Initialize Model List
            models = new List<Model>();

            //Load Models
            Model scene = Content.Load<Model>("Scene");
            models.Add(scene);
        }

        protected override void Draw(GameTime gameTime)
        {
            //Clear
            GraphicsDevice.Clear(Color.CornflowerBlue);

            //Draw Shadow Maps
            lightManager.DrawShadowMaps(GraphicsDevice, models);

            //Draw using Deferred Renderer
            deferredRenderer.Draw(GraphicsDevice, models, lightManager, Camera, null);

            //Debug Deferred Renderer
            deferredRenderer.Debug(GraphicsDevice, spriteBatch);

            //Base Drawing
            base.Draw(gameTime);
        }
    }
}
```

Now run it and bask in the glory of Exponential Shadow Mapping!

… That's enough basking, on to the next chapter!

Chapter 5: Exponential Shadow Mapping and Point Light Implementation

There is a surprising lack of material on Point Light Shadow-Mapping, it seems like it would be kinda hard to implement when just glancing at the theory. Actually it's very simple and surprisingly not very error prone or code heavy as I originally thought.

Basically all Point Light Shadow Mapping is, is just rendering from 6 different Views to create a Cube Shadow Map. Which is as simple as just drawing the models 6 times (culling algorithms are very scene graph architecture dependant so if I implemented such a system here, even a simple one, it would convolute everything far too much, it's up to you to implement such things, however simple they may be) to 6 different render targets that will be read by the pixel shader as a cube map. Fortunately this is perhaps the only way that XNA4 shines.

Before we get to the nitty gritty, we need to, just as in the previous chapter, define a Point Light. A Point Light is just a point in space that emanates light in all directions to within a certain radius, know this let us now create the Point Light class.

Add a new class to your Lights namespace "PointLight" and fill it like so:

```
/// <summary>
/// A Point Light
/// </summary>
class PointLight
{
    //Position
    Vector3 position;

    //Radius
    float radius;

    //Color
    Vector4 color;

    //Intensity
    float intensity;

    //ShadowMap
    RenderTargetCube shadowMap;

    //Is this Light with Shadows?
    bool isWithShadows;

    //Shadow Map Resoloution
    int shadowMapResoloution;

    #region Get Functions
    //Get Position
    public Vector3 getPosition() { return position; }
```

```
    //Get Radius
    public float getRadius() { return radius; }

    //Get Color
    public Vector4 getColor() { return color; }

    //Get Intensity
    public float getIntensity() { return intensity; }

    //Get IsWithShadows
    public bool getIsWithShadows() { return isWithShadows; }

    //Get ShadowMapResoloution
    public int getShadowMapResoloution()
    {
       if (shadowMapResoloution < 2048)  return shadowMapResoloution;
       else return 2048;
    }

    //Get DepthBias
    public float getDepthBias() { return (1.0f / (20 * radius)); }

    //Get ShadowMap
    public RenderTargetCube getShadowMap() { return shadowMap; }
    #endregion

    #region Set Functions
    //Set Position
    public void setPosition(Vector3 position) { this.position = position; }

    //Set Radius
    public void setRadius(float radius) { this.radius = radius; }

    //Set Color
    public void setColor(Color color) { this.color = color.ToVector4(); }

    //Set Color
    public void setColor(Vector4 color) { this.color = color; }

    //Set Intensity
    public void setIntensity(float intensity) { this.intensity = intensity; }

    //Set isWithShadows
    public void setIsWithShadows(bool shadows) { this.isWithShadows = shadows; }
    #endregion

    //Constructor
    public PointLight(GraphicsDevice GraphicsDevice, Vector3 Position, float Radius,
                      Vector4 Color, float Intensity, bool isWithShadows,
                      int shadowMapResoloution)
    {
        //Set Position
        setPosition(Position);

        //Set Radius
        setRadius(Radius);

        //Set Color
        setColor(Color);

        //Set Intensity
        setIntensity(Intensity);
```

```
        //Set isWithShadows
        this.isWithShadows = isWithShadows;

        //Set shadowMapResoloution
        this.shadowMapResoloution = shadowMapResoloution;

        //Make ShadowMap
        shadowMap = new RenderTargetCube(GraphicsDevice, getShadowMapResoloution(),
                                         false, SurfaceFormat.Single,
                                         DepthFormat.Depth24Stencil8);
    }

    //Create World Matrix for Deferred Rendering Geometry
    public Matrix World()
    {
        //Make Scaling Matrix
        Matrix scale = Matrix.CreateScale(radius / 100.0f);

        //Make Translation Matrix
        Matrix translation = Matrix.CreateTranslation(position);

        //Return World Transform
        return (scale * translation);
    }
}
```

The Point Light class is simpler than the Spot Light when you think about it, as for the dividing the radius by 100 thing for the scale, that's my bad; I didn't bother to scale the point light geometry in maya by 100 before exporting, but for the purposes of this implementation it should be fine.

With our Point Light defined now let's head over to the LightManager to add it to the minor functions:

```
/// <summary>
/// A Manager for all the light types
/// </summary>
class LightManager
{
    //Directional Lights
    List<DirectionalLight> directionalLights;

    //Spot Lights
    List<SpotLight> spotLights;

    //Point Lights
    List<PointLight> pointLights;

    //Depth Writing Shader
    Effect depthWriter;

    #region Get Functions
    //Get Directional Lights
    public List<DirectionalLight> getDirectionalLights() { return directionalLights; }
    //Get Spot Lights
```

```
    public List<SpotLight> getSpotLights() { return spotLights; }

    //Get Point Lights
    public List<PointLight> getPointLights() { return pointLights; }
    #endregion

    //Constructor
    public LightManager(ContentManager Content)
    {
        //Initialize Directional Lights
        directionalLights = new List<DirectionalLight>();

        //Initialize Spot Lights
        spotLights = new List<SpotLight>();

        //Initialize Point Lights
        pointLights = new List<PointLight>();

        //Load the Depth Writing Shader
        depthWriter = Content.Load<Effect>("Effects/DepthWriter");
        depthWriter.CurrentTechnique = depthWriter.Techniques[0];
    }

    //Add a Point Light
    public void AddLight(PointLight Light)
    {
        pointLights.Add(Light);
    }

    //Remove a Point Light
    public void RemoveLight(PointLight Light)
    {
        pointLights.Remove(Light);
    }
}
```

Creating a Cube Shadow Map

Finally the good part right? As before we handle the Shadow Map creation as three functions, one that will call the shadow map creation function for each light, one that will set up the render targets and global effect parameters and call the model drawing function for each render target and finally the model drawing function. The model drawing function remains the same as the previous chapter though.

We need to call the new, yet to be defined DrawShadowMap function for each Point Light in the DrawShadowMaps function:

```
//Draw Shadow Maps
public void DrawShadowMaps(GraphicsDevice GraphicsDevice, List<Model> Models)
{
    //Set States
    GraphicsDevice.BlendState = BlendState.Opaque;
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;

    //Foreach SpotLight with Shadows
```

```
    foreach (SpotLight Light in spotLights)
    {
        //Update it
        Light.Update();

        //Draw it's Shadow Map
        if (Light.getIsWithShadows()) DrawShadowMap(GraphicsDevice, Light, Models);
    }

    //Foreach PointLight with Shadows
    foreach (PointLight Light in pointLights)
    {
        //Draw it's Shadow Map
        if (Light.getIsWithShadows()) DrawShadowMap(GraphicsDevice, Light, Models);
    }
}
```

Now we get to the actual creation of the Cube Shadow Map:

```
//Draw a Shadow Map for a Point Light
void DrawShadowMap(GraphicsDevice GraphicsDevice, PointLight Light, List<Model>
Models)
{
    //Initialize View Matrices Array
    Matrix[] views = new Matrix[6];

    //Create View Matrices
    views[0] = Matrix.CreateLookAt(Light.getPosition(), Light.getPosition() +
                                                        Vector3.Forward, Vector3.Up);

    views[1] = Matrix.CreateLookAt(Light.getPosition(), Light.getPosition() +
                                                        Vector3.Backward, Vector3.Up);

    views[2] = Matrix.CreateLookAt(Light.getPosition(), Light.getPosition() +
                                                        Vector3.Left, Vector3.Up);

    views[3] = Matrix.CreateLookAt(Light.getPosition(), Light.getPosition() +
                                                        Vector3.Right, Vector3.Up);

    views[4] = Matrix.CreateLookAt(Light.getPosition(), Light.getPosition() +
                                                        Vector3.Down, Vector3.Forward);

    views[5] = Matrix.CreateLookAt(Light.getPosition(), Light.getPosition() +
                                                        Vector3.Up, Vector3.Backward);

    //Create Projection Matrix
Matrix projection = Matrix.CreatePerspectiveFieldOfView(MathHelper.ToRadians(90.0f),
                                                        1.0f, 1.0f, Light.getRadius());

    //Set Global Effect Values
    depthWriter.Parameters["Projection"].SetValue(projection);
    depthWriter.Parameters["LightPosition"].SetValue(Light.getPosition());
    depthWriter.Parameters["DepthPrecision"].SetValue(Light.getRadius());
```

```
#region Forward
GraphicsDevice.SetRenderTarget(Light.getShadowMap(), CubeMapFace.PositiveZ);

//Clear Target
GraphicsDevice.Clear(Color.Transparent);

//Set global Effect parameters
depthWriter.Parameters["View"].SetValue(views[0]);

//Draw Models
DrawModels(GraphicsDevice, Models);
#endregion

#region Backward
GraphicsDevice.SetRenderTarget(Light.getShadowMap(), CubeMapFace.NegativeZ);

//Clear Target
GraphicsDevice.Clear(Color.Transparent);

//Set global Effect parameters
depthWriter.Parameters["View"].SetValue(views[1]);

//Draw Models
DrawModels(GraphicsDevice, Models);
#endregion

#region Left
GraphicsDevice.SetRenderTarget(Light.getShadowMap(), CubeMapFace.NegativeX);

//Clear Target
GraphicsDevice.Clear(Color.Transparent);

//Set global Effect parameters
depthWriter.Parameters["View"].SetValue(views[2]);

//Draw Models
DrawModels(GraphicsDevice, Models);
#endregion

#region Right
GraphicsDevice.SetRenderTarget(Light.getShadowMap(), CubeMapFace.PositiveX);

//Clear Target
GraphicsDevice.Clear(Color.Transparent);

//Set global Effect parameters
depthWriter.Parameters["View"].SetValue(views[3]);

//Draw Models
DrawModels(GraphicsDevice, Models);
#endregion

#region Down
GraphicsDevice.SetRenderTarget(Light.getShadowMap(), CubeMapFace.NegativeY);

//Clear Target
GraphicsDevice.Clear(Color.Transparent);

//Set global Effect parameters
depthWriter.Parameters["View"].SetValue(views[4]);
```

```
    //Draw Models
    DrawModels(GraphicsDevice, Models);
    #endregion

    #region Up
    GraphicsDevice.SetRenderTarget(Light.getShadowMap(), CubeMapFace.PositiveY);

    //Clear Target
    GraphicsDevice.Clear(Color.Transparent);

    //Set global Effect parameters
    depthWriter.Parameters["View"].SetValue(views[5]);

    //Draw Models
    DrawModels(GraphicsDevice, Models);
    #endregion
}
```

Let me explain this function in simple terms, as it is actually fairly simple:

1. Create a View matrix for each face of the Cube map, the faces consist of:
    1. Forward
    2. Backward
    3. Left
    4. Right
    5. Down (with Forward as the Up vector)
    6. Up (with Backward as the Up vector)
2. Create a single Projection matrix for all the lights, has to be 90 in FOV and 1 as aspect ratio
3. Set the effect parameters that are the same for all the lights
4. Draw Models from each separate View to each separate face of the CubeMap, setting the appropriate face in the SetRenderTarget call

That's all there is to it really, the next few bits should just be a cruise through familiar territory!

Drawing a Point Light onto the Light-Map

Now that we have a Cube Shadow Map, how do we use it? Before we get to that let's set up the Deferred Renderer's Light Map creation to now handle Point Lights, pretty much the same as with Spot Lights only the culling operation is a bit easier:

```
//Light Map Creation
void MakeLightMap(GraphicsDevice GraphicsDevice, LightManager Lights,
                  BaseCamera Camera)
{
    //Set LightMap Target
    GraphicsDevice.SetRenderTarget(LightMap);

    //Clear to Transperant Black
    GraphicsDevice.Clear(Color.Transparent);
```

```
    //Set States
    GraphicsDevice.BlendState = LightMapBS;
    GraphicsDevice.DepthStencilState = DepthStencilState.DepthRead;

    #region Set Global Samplers
    //GBuffer 1 Sampler
    GraphicsDevice.Textures[0] = GBufferTargets[0].RenderTarget;
    GraphicsDevice.SamplerStates[0] = SamplerState.LinearClamp;

    //GBuffer 2 Sampler
    GraphicsDevice.Textures[1] = GBufferTargets[1].RenderTarget;
    GraphicsDevice.SamplerStates[1] = SamplerState.LinearClamp;

    //GBuffer 3 Sampler
    GraphicsDevice.Textures[2] = GBufferTargets[2].RenderTarget;
    GraphicsDevice.SamplerStates[2] = SamplerState.PointClamp;

    //SpotLight Cookie Sampler
    GraphicsDevice.SamplerStates[3] = SamplerState.LinearClamp;

    //ShadowMap Sampler
    GraphicsDevice.SamplerStates[4] = SamplerState.PointClamp;
    #endregion

    //Calculate InverseView
    Matrix InverseView = Matrix.Invert(Camera.View);

    //Calculate InverseViewProjection
    Matrix InverseViewProjection = Matrix.Invert(Camera.View * Camera.Projection);

    //Set Directional Lights Globals
directionalLight.Parameters["InverseViewProjection"].SetValue(InverseViewProjection);
    directionalLight.Parameters["inverseView"].SetValue(InverseView);
    directionalLight.Parameters["CameraPosition"].SetValue(Camera.Position);
    directionalLight.Parameters["GBufferTextureSize"].SetValue(GBufferTextureSize);

    //Set the Directional Lights Geometry Buffers
    fsq.ReadyBuffers(GraphicsDevice);

    //Draw Directional Lights
    foreach (Lights.DirectionalLight light in Lights.getDirectionalLights())
    {
        //Set Directional Light Parameters
   directionalLight.Parameters["L"].SetValue(Vector3.Normalize(light.getDirection()));
        directionalLight.Parameters["LightColor"].SetValue(light.getColor());
        directionalLight.Parameters["LightIntensity"].SetValue(light.getIntensity());

        //Apply
        directionalLight.CurrentTechnique.Passes[0].Apply();

        //Draw
        fsq.JustDraw(GraphicsDevice);
    }

    //Set Spot Lights Globals
    spotLight.Parameters["View"].SetValue(Camera.View);
    spotLight.Parameters["inverseView"].SetValue(InverseView);
    spotLight.Parameters["Projection"].SetValue(Camera.Projection);
    spotLight.Parameters["InverseViewProjection"].SetValue(InverseViewProjection);
    spotLight.Parameters["CameraPosition"].SetValue(Camera.Position);
    spotLight.Parameters["GBufferTextureSize"].SetValue(GBufferTextureSize);
```

```
    //Set Spot Lights Geometry Buffers
GraphicsDevice.SetVertexBuffer(spotLightGeometry.Meshes[0].MeshParts[0].VertexBuffer,
                               spotLightGeometry.Meshes[0].MeshParts[0].VertexOffset);
    GraphicsDevice.Indices = spotLightGeometry.Meshes[0].MeshParts[0].IndexBuffer;

    //Draw Spot Lights
    foreach (Lights.SpotLight light in Lights.getSpotLights())
    {
        //Set Attenuation Cookie Texture and SamplerState
        GraphicsDevice.Textures[3] = light.getAttenuationTexture();

        //Set ShadowMap and SamplerState
        GraphicsDevice.Textures[4] = light.getShadowMap();

        //Set Spot Light Parameters
        spotLight.Parameters["World"].SetValue(light.getWorld());
        spotLight.Parameters["LightViewProjection"].SetValue(light.getView() *
                                                             light.getProjection());
        spotLight.Parameters["LightPosition"].SetValue(light.getPosition());
        spotLight.Parameters["LightColor"].SetValue(light.getColor());
        spotLight.Parameters["LightIntensity"].SetValue(light.getIntensity());
        spotLight.Parameters["S"].SetValue(light.getDirection());
        spotLight.Parameters["LightAngleCos"].SetValue(light.LightAngleCos());
        spotLight.Parameters["LightHeight"].SetValue(light.getFarPlane());
        spotLight.Parameters["Shadows"].SetValue(light.getIsWithShadows());
      spotLight.Parameters["shadowMapSize"].SetValue(light.getShadowMapResoloution());
        spotLight.Parameters["DepthPrecision"].SetValue(light.getFarPlane());
        spotLight.Parameters["DepthBias"].SetValue(light.getDepthBias());

        #region Set Cull Mode
        //Calculate L
        Vector3 L = Camera.Position - light.getPosition();

        //Calculate S.L
        float SL = Math.Abs(Vector3.Dot(L, light.getDirection()));

        //Check if SL is within the LightAngle, if so then draw the BackFaces, if not
        //then draw the FrontFaces
        if (SL < light.LightAngleCos())
              GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
        else
              GraphicsDevice.RasterizerState = RasterizerState.CullClockwise;
        #endregion

        //Apply
        spotLight.CurrentTechnique.Passes[0].Apply();

        //Draw
        GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                             spotLightGeometry.Meshes[0].MeshParts[0].NumVertices,
                             spotLightGeometry.Meshes[0].MeshParts[0].StartIndex,
                             spotLightGeometry.Meshes[0].MeshParts[0].PrimitiveCount);
    }
```

```
    //Set Point Lights Geometry Buffers
GraphicsDevice.SetVertexBuffer(pointLightGeometry.Meshes[0].MeshParts[0].VertexBuffer,
                               pointLightGeometry.Meshes[0].MeshParts[0].VertexOffset);
    GraphicsDevice.Indices = pointLightGeometry.Meshes[0].MeshParts[0].IndexBuffer;

    //Set Point Lights Globals
    pointLight.Parameters["inverseView"].SetValue(InverseView);
    pointLight.Parameters["View"].SetValue(Camera.View);
    pointLight.Parameters["Projection"].SetValue(Camera.Projection);
    pointLight.Parameters["InverseViewProjection"].SetValue(InverseViewProjection);
    pointLight.Parameters["CameraPosition"].SetValue(Camera.Position);
    pointLight.Parameters["GBufferTextureSize"].SetValue(GBufferTextureSize);

    //Draw Point Lights without Shadows
    foreach (Lights.PointLight light in Lights.getPointLights())
    {
        //Set Point Light Sampler
        GraphicsDevice.Textures[4] = light.getShadowMap();
        GraphicsDevice.SamplerStates[4] = SamplerState.PointWrap;

        //Set Point Light Parameters
        pointLight.Parameters["World"].SetValue(light.World());
        pointLight.Parameters["LightPosition"].SetValue(light.getPosition());
        pointLight.Parameters["LightRadius"].SetValue(light.getRadius());
        pointLight.Parameters["LightColor"].SetValue(light.getColor());
        pointLight.Parameters["LightIntensity"].SetValue(light.getIntensity()); ;
        pointLight.Parameters["Shadows"].SetValue(light.getIsWithShadows());
        pointLight.Parameters["DepthPrecision"].SetValue(light.getRadius());
        pointLight.Parameters["DepthBias"].SetValue(light.getDepthBias());
     pointLight.Parameters["shadowMapSize"].SetValue(light.getShadowMapResoloution());

        //Set Cull Mode
        Vector3 diff = Camera.Position - light.getPosition();

     float CameraToLight = (float)Math.Sqrt((float)Vector3.Dot(diff, diff)) / 100.0f;

        //If the Camera is in the light, render the backfaces, if it's out of the
        //light, render the frontfaces
        if (CameraToLight <= light.getRadius())
              GraphicsDevice.RasterizerState = RasterizerState.CullClockwise;
        else if (CameraToLight > light.getRadius())
              GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;

        //Apply
        pointLight.CurrentTechnique.Passes[0].Apply();

        //Draw
        GraphicsDevice.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0,
                               pointLightGeometry.Meshes[0].MeshParts[0].NumVertices,
                               pointLightGeometry.Meshes[0].MeshParts[0].StartIndex,
                             pointLightGeometry.Meshes[0].MeshParts[0].PrimitiveCount);
    }

    //Set States Off
    GraphicsDevice.BlendState = BlendState.Opaque;
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
}
```

With that taken care of, noting the difference between the cull check of the Spot Light and Point Light, we will turn our attention to the Point Light Shader.

Creating the Point Light Shader

The Point Light shader is interesting in two ways, first that it will show you how to sample a Cube Map and second in that it will show you how to manually linear sample and apply a small blur to a Cube map (see the references for more info on that…)

Add a new Effect to your Effects folder "PointLight.fx":

```
//World
float4x4 World;

//View
float4x4 View;

//Inverse View
float4x4 inverseView;

//Projection
float4x4 Projection;

//Inverse View Projection
float4x4 InverseViewProjection;

//Camera Position
float3 CameraPosition;

//Light Position
float3 LightPosition;

//Light Radius
float LightRadius;

//Light Color
float4 LightColor;

//Light Intensity
float LightIntensity;

//GBuffer Texture Size
float2 GBufferTextureSize;

//Shadows?
bool Shadows;

//This is for modulating the Light's Depth Precision (100)
float DepthPrecision;

//DepthBias for the Shadowing... (1.0f / 2000.0f)
float DepthBias;

//ShadowMap size
float shadowMapSize;

//GBuffer Texture0
sampler GBuffer0 : register(s0);
```

```
//GBuffer Texture1
sampler GBuffer1 : register(s1);

//GBuffer Texture2
sampler GBuffer2 : register(s2);

//ShadowMap
sampler ShadowMap : register(s4);

//Vertex Input Structure
struct VSI
{
	float4 Position : POSITION0;
};

//Vertex Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float4 ScreenPosition : TEXCOORD0;
};

//Vertex Shader
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Transform Position
	float4 worldPosition = mul(input.Position, World);
	float4 viewPosition = mul(worldPosition, View);
	output.Position = mul(viewPosition, Projection);

	//Pass to ScreenPosition
	output.ScreenPosition = output.Position;

	//Return
	return output;
}

//Manually Linear Sample
float4 manualSample(sampler Sampler, float2 UV, float2 textureSize)
{
	float2 texelpos = textureSize * UV;
	float2 lerps = frac(texelpos);
	float texelSize = 1.0 / textureSize;

	float4 sourcevals[4];
	sourcevals[0] = tex2D(Sampler, UV);
	sourcevals[1] = tex2D(Sampler, UV + float2(texelSize, 0));
	sourcevals[2] = tex2D(Sampler, UV + float2(0, texelSize));
	sourcevals[3] = tex2D(Sampler, UV + float2(texelSize, texelSize));

	float4 interpolated = lerp(lerp(sourcevals[0], sourcevals[1], lerps.x),
				lerp(sourcevals[2], sourcevals[3], lerps.x ),
				lerps.y);

	return interpolated;
}
```

```
//Manually Linear Sample a Cube Map
float4 manualSampleCUBE(sampler Sampler, float3 UVW, float3 textureSize)
{
	//Calculate the reciprocal
	float3 textureSizeDiv = 1 / textureSize;

	//Multiply coordinates by the texture size
	float3 texPos = UVW * textureSize;

	//Compute first integer coordinates
	float3 texPos0 = floor(texPos + 0.5f);

	//Compute second integer coordinates
	float3 texPos1 = texPos0 + 1.0f;

	//Perform division on integer coordinates
	texPos0 = texPos0 * textureSizeDiv;
	texPos1 = texPos1 * textureSizeDiv;

	//Compute contributions for each coordinate
	float3 blend = frac(texPos + 0.5f);

	//Construct 8 new coordinates
	float3 texPos000 = texPos0;
	float3 texPos001 = float3(texPos0.x, texPos0.y, texPos1.z);
	float3 texPos010 = float3(texPos0.x, texPos1.y, texPos0.z);
	float3 texPos011 = float3(texPos0.x, texPos1.y, texPos1.z);
	float3 texPos100 = float3(texPos1.x, texPos0.y, texPos0.z);
	float3 texPos101 = float3(texPos1.x, texPos0.y, texPos1.z);
	float3 texPos110 = float3(texPos1.x, texPos1.y, texPos0.z);
	float3 texPos111 = texPos1;

	//Sample Cube Map
	float3 C000 = texCUBE(Sampler, texPos000);
	float3 C001 = texCUBE(Sampler, texPos001);
	float3 C010 = texCUBE(Sampler, texPos010);
	float3 C011 = texCUBE(Sampler, texPos011);
	float3 C100 = texCUBE(Sampler, texPos100);
	float3 C101 = texCUBE(Sampler, texPos101);
	float3 C110 = texCUBE(Sampler, texPos110);
	float3 C111 = texCUBE(Sampler, texPos111);

	//Compute final value by lerping everything
  float3 C = lerp(lerp(lerp(C000, C010, blend.y), lerp(C100, C110, blend.y), blend.x),
                  lerp( lerp(C001, C011, blend.y), lerp(C101, C111, blend.y),blend.x),
                  blend.z);

	//Return
	return float4(C, 1);
}

//Phong Shader
float4 Phong(float3 Position, float3 N, float SpecularIntensity, float SpecularPower)
{
	//Calculate Light Vector
	float3 L = LightPosition.xyz - Position.xyz;

	//Calculate Linear Attenuation
	float Attenuation = saturate(1.0f  - max(.01f, length(L)) / (LightRadius / 2));

	//Normalize Light Vector
	L = normalize(L);
```

```
	//Calculate Reflection vector
	float3 R = normalize(reflect(-L, N));

	//Calculate Eye vector
	float3 E = normalize(CameraPosition - Position.xyz);

	//Calculate N.L
	float NL = dot(N, L);

	//Calculate Diffuse
	float3 Diffuse = NL * LightColor.xyz;

	//Calculate Specular
	float Specular = SpecularIntensity * pow(saturate(dot(R, E)), SpecularPower);

	//Get Light-Z from Manually Sampled ShadowMap
	float lZ = manualSampleCUBE(ShadowMap, float3(-L.xy, L.z), shadowMapSize).r;

	//Assymetric Workaround...
	float ShadowFactor = 1;

	//If Shadowing is on then get the Shadow Factor
	if(Shadows)
	{
		// Calculate distance to the light
	     float len = max(0.01f, length(LightPosition - Position)) / DepthPrecision;

		//Calculate the Shadow Factor
		ShadowFactor = (lZ * exp(-(DepthPrecision * 0.5f) * (len - DepthBias)));
	}

	//Calculate Final Product
   return ShadowFactor * Attenuation * LightIntensity * float4(Diffuse.rgb, Specular);
}

//Decoding of GBuffer Normals
float3 decode(float3 enc)
{
	return (2.0f * enc.xyz- 1.0f);
}

//Pixel Shader
float4 PS(VSO input) : COLOR0
{
	//Get Screen Position
	input.ScreenPosition.xy /= input.ScreenPosition.w;

	//Calculate UV from ScreenPosition
    float2 UV = 0.5f * (float2(input.ScreenPosition.x, -input.ScreenPosition.y) + 1) -
                                                  float2(1.0f / GBufferTextureSize.xy);

	//Get All Data from Normal part of the GBuffer
	half4 encodedNormal = tex2D(GBuffer1, UV);

	//Decode Normal
	half3 Normal = mul(decode(encodedNormal.xyz), inverseView);

	//Get Specular Intensity from GBuffer
	float SpecularIntensity = tex2D(GBuffer0, UV).w;
```

```
        //Get Specular Power from GBuffer
        float SpecularPower = encodedNormal.w * 255;

        //Get Depth from GBuffer
        float Depth = manualSample(GBuffer2, UV, GBufferTextureSize).x;

        //Make Position in Homogenous Space using current ScreenSpace coordinates and
        //the Depth from the GBuffer
        float4 Position = 1.0f;

        Position.xy = input.ScreenPosition.xy;

        Position.z = Depth;

        //Transform Position from Homogenous Space to World Space
        Position = mul(Position, InverseViewProjection);

        Position /= Position.w;

        //Return Phong Shaded Value
        return Phong(Position.xyz, Normal, SpecularIntensity, SpecularPower);
}

//Technique
technique Default
{
        pass p0
        {
                VertexShader = compile vs_3_0 VS();
                PixelShader = compile ps_3_0 PS();
        }
}
```

Putting it all together

Since we haven't really added any new functions or etc. this step is far more simple than previous chapters, all we're gonna do is add a Point Light to the game!

Set up your lights like so in the Load function of your Game:

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    //Create Deferred Renderer
    deferredRenderer = new DeferredRenderer(GraphicsDevice, Content,
                                            graphics.PreferredBackBufferWidth,
                                            graphics.PreferredBackBufferHeight);

    //Create Light Manager
    lightManager = new LightManager(Content);

    //Load SpotLight Cookies
    Texture2D spotCookie = Content.Load<Texture2D>("SpotCookie");
    Texture2D squareCookie = Content.Load<Texture2D>("SquareCookie");

    //Add a Directional Light
```

```
 lightManager.AddLight(new Lights.DirectionalLight(Vector3.Down, Color.White, 0.05f));

    //Add a Spot Light
  lightManager.AddLight(new Lights.SpotLight(GraphicsDevice, new Vector3(0, 15.0f, 0),
                                             new Vector3(0, -1, 0),
                                             Color.White.ToVector4(), 0.10f, true,
                                             2048, spotCookie));

    //Add a Point Light
  lightManager.AddLight(new Lights.PointLight(GraphicsDevice, new Vector3(0, 5.0f, 0),
                                              20.0f, Color.White.ToVector4(), 0.80f,
                                              true, 256));

    //Initialize Model List
    models = new List<Model>();

    //Load Models
    Model scene = Content.Load<Model>("Scene");
    models.Add(scene);
}
```

Now see the awesomeness of three different styles of light, Directional, Spot Lights with shadows and Point Lights with shadows, by the way you can always toggle the shadows on and off as well as the position and direction of all these lights with a simple custom function that you can implement if you wish.

Now on to the last chapter, SSAO with Normals!

Chapter 6: Screen-Space Ambient Occlusion Theory and Implementation

SSAO can be very intimidating to newcomers, it's like a really simple ray tracer done through a pixel shader so it's not as simple as something like HDR or etc. where you're just manipulating colour with a simple filter or such.

This particular implementation is pretty much a modified version of Alex Urbano Alvarez's work (see references) which takes into account Normals however it comes at the cost of being half the samples, so rather than 16 samples per pixel, we only take 8. Still looks alright enough though.

The Screen-Space Ambient Occlusion Shader

I always like to explain the theory after showing you the code first, so add a new Effect to your Effects folder "SSAO.fx":

```
#define NUMSAMPLES 8

//Projection matrix
float4x4 Projection;

//Corner Fustrum
float3 cornerFustrum;

//Sample Radius
float sampleRadius;

//Distance Scale
float distanceScale;

//GBuffer Texture Size
float2 GBufferTextureSize;

//Samplers
sampler GBuffer1 : register(s1);
sampler GBuffer2 : register(s2);
sampler RandNormal : register(s3);

//Vertex Input Structure
struct VSI
{
	float3 Position : POSITION0;
	float2 UV : TEXCOORD0;
};

//Vertex Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float2 UV : TEXCOORD0;
	float3 ViewDirection : TEXCOORD1;
};
```

```
//Vertex Shader
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Just Straight Pass Position
	output.Position = float4(input.Position, 1);

	//Set up UV's
	output.UV = input.UV - float2(1.0f / GBufferTextureSize.xy);

	//Set up ViewDirection vector
	output.ViewDirection = float3(-cornerFustrum.x * input.Position.x,
	                              cornerFustrum.y * input.Position.y,
	                              cornerFustrum.z);

	//Return
	return output;
}

//Normal Decoding Function
float3 decode(float3 enc)
{
	return (2.0f * enc.xyz- 1.0f);
}

//Pixel Shader
float4 PS(VSO input) : COLOR0
{
	//Sample Vectors
	float4 samples[8] =
	{
		float4(0.355512,     -0.709318,    -0.102371,    0.0 ),
		float4(0.534186,     0.71511,      -0.115167,    0.0 ),
		float4(-0.87866,     0.157139,     -0.115167,    0.0 ),
		float4(0.140679,     -0.475516,    -0.0639818,   0.0 ),
		float4(-0.207641,    0.414286,     0.187755,     0.0 ),
		float4(-0.277332,    -0.371262,    0.187755,     0.0 ),
		float4(0.63864,      -0.114214,    0.262857,     0.0 ),
		float4(-0.184051,    0.622119,     0.262857,     0.0 )
	};

	//Normalize the input ViewDirection
	float3 ViewDirection = normalize(input.ViewDirection);

	//Sample the depth
	float depth = tex2D(GBuffer2, input.UV).g;

	//Calculate the depth at this pixel along the view direction
	float3 se = depth * ViewDirection;

	//Sample a random normal vector
	float3 randNormal = tex2D(RandNormal, input.UV * 200.0f).xyz;

	//Sample the Normal for this pixel
	float3 normal = decode(tex2D(GBuffer1, input.UV).xyz);

	//No assymetry in HLSL, workaround
	float finalColor = 0.0f;
```

```
	//SSAO loop
	for (int i = 0; i < NUMSAMPLES; i++)
	{
		//Calculate the Reflection Ray
		float3 ray = reflect(samples[i].xyz, randNormal) * sampleRadius;

		//Test the Reflection Ray against the surface normal
		if(dot(ray, normal) < 0) ray += normal * sampleRadius;

		//Calculate the Sample vector
		float4 sample = float4(se + ray, 1.0f);

		//Project the Sample vector into ScreenSpace
		float4 ss = mul(sample, Projection);

		//Convert SS into UV space
		float2 sampleTexCoord = 0.5f * ss.xy / ss.w + float2(0.5f, 0.5f);

		//Sample the Depth along the ray
		float sampleDepth = tex2D(GBuffer2, sampleTexCoord).g;

		//Check the sampled depth value
		if (sampleDepth == 1.0)
		{
			//Non-Occluded sample
			finalColor++;
		}
		else
		{
			//Calculate Occlusion
			float occlusion = distanceScale * max(sampleDepth - depth, 0.0f);

			//Accumulate to finalColor
			finalColor += 1.0f / (1.0f + occlusion * occlusion * 0.1);
		}
	}

	//Output the Average of finalColor
	return float4(finalColor / NUMSAMPLES,
			finalColor / NUMSAMPLES,
			finalColor / NUMSAMPLES,
			1.0f);
}

//Technique
technique Default
{
	pass p0
	{
		VertexShader = compile vs_3_0 VS();
		PixelShader = compile ps_3_0 PS();
	}
}
```

Let me try and break the pixel shader down for you:

1. Rebuild Position in View-Space, ordinarily this would involve transforming the sampled clip-space depth by the inverse-projection matrix, this would be too costly so instead we multiply the View Direction by the sampled Depth, the View Direction is built in the Vertex Shader by using the Position and the Corner Frustum.
2. Sample a Random Normal from the Random Normal texture
3. Sample the View-Space Normal from the G-Buffer and decode it from unsigned to signed space.
4. Now for each Sample:
    i. Calculate the Reflection Ray of the particular Sample Vector and Random Normal and make it reach out to the Sample Radius.
    ii. Check the Dot Product of that Ray and the Normal, if it's not perpendicular and the angle is greater than 90 degrees, then add the Sample Radius reaching Normal to the Ray
    iii. Add that Ray to the Position in View-Space to get the Sample Vector
    iv. Transform the Sample Vector from View-Space to Screen-Space by multiplying the Sample Vector by the Projection matrix.
    v. Transform Sample Vector from Screen-Space to UV-Space
    vi. Sample the Depth from the G-Buffer with the UV-Space'd Sample Vector
    vii. If the Sample Depth is 1.0, then this sample is not occluded, else it is and the occlusion factor is calculated
    viii. Accumulate the Occlusion Factor(whether nothing or something)
5. Average the Accumulated Occlusion Factor to get the final return value

Even in simple form it's still fairly complex, I actually think that since XNA4 SSAO has become unfeasible due to its inability to sample floating point textures linearly but it's still possible.

The SSAO Class

I think we have the most difficult portion of this chapter done, now we just set up the class to interface with the shader before.

Add a new Class to your project "SSAO.cs":

```
class SSAO
{
    //SSAO effect
    Effect ssao;

    //SSAO Blur Effect
    Effect ssaoBlur;

    //SSAO Composition effect
    Effect composer;
```

```
    //Random Normal Texture
    Texture2D randomNormals;

    //Sample Radius
    float sampleRadius;

    //Distance Scale
    float distanceScale;

    //SSAO Target
    RenderTarget2D SSAOTarget;

    //Blue Target
    RenderTarget2D BlurTarget;

    //FSQ
    FullscreenQuad fsq;

    #region Get Methods
    //Get Sample Radius
    float getSampleRadius() { return sampleRadius; }

    //Get Distance Scale
    float getDistanceScale() { return distanceScale; }
    #endregion

    #region Set Methods
    //Set Sample Radius
    void setSampleRadius(float radius) { this.sampleRadius = radius; }

    //Set Distance Scale
    void setDistanceScale(float scale) { this.distanceScale = scale; }
    #endregion

    //Constructor
    public SSAO(GraphicsDevice GraphicsDevice, ContentManager Content,
                int Width, int Height)
    {
        //Load SSAO effect
        ssao = Content.Load<Effect>("Effects/SSAO");
        ssao.CurrentTechnique = ssao.Techniques[0];

        //Load SSAO Blur effect
        ssaoBlur = Content.Load<Effect>("Effects/SSAOBlur");
        ssaoBlur.CurrentTechnique = ssaoBlur.Techniques[0];

        //Load SSAO composition effect
        composer = Content.Load<Effect>("Effects/SSAOFinal");
        composer.CurrentTechnique = composer.Techniques[0];

        //Create SSAO Target
        SSAOTarget = new RenderTarget2D(GraphicsDevice, Width, Height, false,
                                        SurfaceFormat.Color, DepthFormat.None);

        //Create SSAO Blur Target
        BlurTarget = new RenderTarget2D(GraphicsDevice, Width, Height, false,
                                        SurfaceFormat.Color, DepthFormat.None);

        //Create FSQ
        fsq = new FullscreenQuad(GraphicsDevice);
```

```
        //Load Random Normal Texture
        randomNormals = Content.Load<Texture2D>("RandomNormals");

        //Set Sample Radius to Default
        sampleRadius = 0;

        //Set Distance Scale to Default
        distanceScale = 0;
    }
}
```

Now let's get to the actual SSAO drawing, add a Draw function to the SSAO class:

```
//Draw
public void Draw(GraphicsDevice GraphicsDevice, RenderTargetBinding[] GBuffer,
                 RenderTarget2D Scene, BaseCamera Camera, RenderTarget2D Output)
{
    //Set States
    GraphicsDevice.BlendState = BlendState.Opaque;
    GraphicsDevice.DepthStencilState = DepthStencilState.Default;
    GraphicsDevice.RasterizerState = RasterizerState.CullCounterClockwise;

    //Render SSAO
    RenderSSAO(GraphicsDevice, GBuffer, Camera);

    //Blur SSAO
    BlurSSAO(GraphicsDevice);

    //Compose final
    Compose(GraphicsDevice, Scene, Output, true);
}
```

As you might have guessed from the function, the SSAO process is basically to draw the SSAO, blur that SSAO and then combine it with the Scene image to produce the final image.

So now we define the RenderSSAO method, which will pretty much just interface with the SSAO shader as a fullscreen post-process:

```
//Render SSAO
void RenderSSAO(GraphicsDevice GraphicsDevice, RenderTargetBinding[] GBuffer,
                BaseCamera Camera)
{
    //Set SSAO Target
    GraphicsDevice.SetRenderTarget(SSAOTarget);

    //Clear
    GraphicsDevice.Clear(Color.White);

    //Set Samplers
    GraphicsDevice.Textures[1] = GBuffer[1].RenderTarget;
    GraphicsDevice.SamplerStates[1] = SamplerState.LinearClamp;

    GraphicsDevice.Textures[2] = GBuffer[1].RenderTarget;
    GraphicsDevice.SamplerStates[2] = SamplerState.PointClamp;
```

```
    GraphicsDevice.Textures[3] = randomNormals;
    GraphicsDevice.SamplerStates[3] = SamplerState.LinearWrap;

    //Calculate Frustum Corner of the Camera
    Vector3 cornerFrustum = Vector3.Zero;
    cornerFrustum.Y = (float)Math.Tan(Math.PI / 3.0 / 2.0) * Camera.FarClip;
    cornerFrustum.X = cornerFrustum.Y * Camera.AspectRatio;
    cornerFrustum.Z = Camera.FarClip;

    //Set SSAO parameters
    ssao.Parameters["Projection"].SetValue(Camera.Projection);
    ssao.Parameters["cornerFustrum"].SetValue(cornerFrustum);
    ssao.Parameters["sampleRadius"].SetValue(sampleRadius);
    ssao.Parameters["distanceScale"].SetValue(distanceScale);
    ssao.Parameters["GBufferTextureSize"].SetValue(new Vector2(SSAOTarget.Width,
                                                               SSAOTarget.Height));

    //Apply
    ssao.CurrentTechnique.Passes[0].Apply();

    //Draw
    fsq.Draw(GraphicsDevice);
}
```

Blurring the Screen-Space Ambient Occlusion Texture

Blurring the SSAO texture is a bit more complicated than a simple gaussian blur; the best way is to apply a special bilateral filter.

Add a new Effect to your Effects folder “SSAOBlur.fx”:

```
float2 blurDirection;

float2 targetSize;

sampler GBuffer1 : register(s1);

sampler GBuffer2 : register(s2);

sampler SSAO : register(s3);

//Vertex Input Structure
struct VSI
{
	float3 Position : POSITION0;
	float2 UV : TEXCOORD0;
};

//Vertex Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float2 UV : TEXCOORD0;
};



//Vertex Shader
```

```
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Just Straight Pass Position
	output.Position = float4(input.Position, 1);

	//Pass UV
	output.UV = input.UV - float2(1.0f / targetSize.xy);

	//Return
	return output;
}

//Manually Linear Sample
float4 manualSample(sampler Sampler, float2 UV, float2 textureSize)
{
	float2 texelpos = textureSize * UV;
	float2 lerps = frac(texelpos);
	float texelSize = 1.0 / textureSize;

	float4 sourcevals[4];
	sourcevals[0] = tex2D(Sampler, UV);
	sourcevals[1] = tex2D(Sampler, UV + float2(texelSize, 0));
	sourcevals[2] = tex2D(Sampler, UV + float2(0, texelSize));
	sourcevals[3] = tex2D(Sampler, UV + float2(texelSize, texelSize));

	float4 interpolated = lerp(lerp(sourcevals[0], sourcevals[1], lerps.x),
				lerp(sourcevals[2], sourcevals[3], lerps.x),
				lerps.y);

	return interpolated;
}

//Normal Decoding Function
float3 decode(float3 enc)
{
	return (2.0f * enc.xyz- 1.0f);
}

//Pixel Shader
float4 PS(float2 UV :TEXCOORD0) : COLOR0
{
	//Sample Depth
	float depth = manualSample(GBuffer2, UV, targetSize).y;

	//Sample Normal
	float3 normal = decode(tex2D(GBuffer1, UV).xyz);

	//Sample SSAO
	float ssao = tex2D(SSAO, UV).x;

	//Color Normalizer
	float ssaoNormalizer = 1;

	//Blur Samples to be done
	int blurSamples = 8;
```

```
	//From the negative half of blurSamples to the positive half; almost like
	//gaussian blur
	for(int i = -blurSamples / 2; i <= blurSamples / 2; i++)
	{
		//Calculate newUV as the current UV offset by the current sample
		float2 newUV = float2(UV.xy + i * blurDirection.xy);

		//Sample SSAO
		float sample = manualSample(SSAO, newUV, targetSize).y;

		//Sample Normal
		float3 samplenormal = decode(tex2D(GBuffer1, newUV).xyz);

		//Check Angle Between SampleNormal and Normal
		if (dot(samplenormal, normal) > 0.99)
		{
			//Calculate this samples contribution
			float contribution = blurSamples / 2 - abs(i);

			//Accumulate to normalizer
			ssaoNormalizer += (blurSamples / 2 - abs(i));

			//Accumulate to SSAO
			ssao += sample * contribution;
		}
	}

	//Return Averaged Samples
	return ssao / ssaoNormalizer;
}

//Technique
technique Default
{
	pass p0
	{
		VertexShader = compile vs_3_0 VS();
		PixelShader = compile ps_3_0 PS();
	}
}
```

Basically all that's happening is that if the sample normal is parallel than we don't consider the sample as contributing and we average the accumulated contributions at the end to get the blurred result.

Now we go back to our SSAO class to add a function to interface with the blurring shader:

```
//Blur SSAO
void BlurSSAO(GraphicsDevice GraphicsDevice)
{
    //Set Blur Target
    GraphicsDevice.SetRenderTarget(BlurTarget);

    //Clear
    GraphicsDevice.Clear(Color.White);

    //Set Samplers, GBuffer was set before so no need to reset...
    GraphicsDevice.Textures[3] = SSAOTarget;
```

```
    GraphicsDevice.SamplerStates[3] = SamplerState.LinearClamp;

    //Set SSAO parameters
    ssaoBlur.Parameters["blurDirection"].SetValue(Vector2.One);
    ssaoBlur.Parameters["targetSize"].SetValue(new Vector2(SSAOTarget.Width,
                                                           SSAOTarget.Height));

    //Apply
    ssaoBlur.CurrentTechnique.Passes[0].Apply();

    //Draw
    fsq.Draw(GraphicsDevice);
}
```

Just a simple fullscreen post-process so it shouldn't seem anything exotic.

Compositing the Final Image

We will just multiply the Deferred Rendering output by the SSAO to get the final value. Add a new Effect to your Effects folder "SSAOFinal.fx":

```
float2 halfPixel;

sampler Scene : register(s0);

sampler SSAO : register(s1);

//Vertex Input Structure
struct VSI
{
	float3 Position : POSITION0;
	float2 UV : TEXCOORD0;
};

//Vertex Output Structure
struct VSO
{
	float4 Position : POSITION0;
	float2 UV : TEXCOORD0;
};

//Vertex Shader
VSO VS(VSI input)
{
	//Initialize Output
	VSO output;

	//Pass Position
	output.Position = float4(input.Position, 1);

	//Pass Texcoord's
	output.UV = input.UV - halfPixel;

	//Return
	return output;
}
//Pixel Shader
float4 PS(VSO input) : COLOR0
{
```

```
	//Sample Scene
	float4 scene = tex2D(Scene, input.UV);

	//Sample SSAO
	float4 ssao = tex2D(SSAO, input.UV);

	//Return
	return (scene * ssao);
}

//Technique
technique Default
{
	pass p0
	{
		VertexShader = compile vs_3_0 VS();
		PixelShader = compile ps_3_0 PS();
	}
}
```

And now we add a function in the SSAO class to composite the final image:

```
//Compose
void Compose(GraphicsDevice GraphicsDevice, RenderTarget2D Scene, RenderTarget2D
Output, bool useBlurredSSAO)
{
    //Set Output Target
    GraphicsDevice.SetRenderTarget(Output);

    //Clear
    GraphicsDevice.Clear(Color.White);

    //Set Samplers
    GraphicsDevice.Textures[0] = Scene;
    GraphicsDevice.SamplerStates[0] = SamplerState.LinearClamp;

    if (useBlurredSSAO) GraphicsDevice.Textures[1] = BlurTarget;
    else GraphicsDevice.Textures[1] = SSAOTarget;
    GraphicsDevice.SamplerStates[1] = SamplerState.LinearClamp;

    //Set Effect Parameters
    composer.Parameters["halfPixel"].SetValue(new Vector2(1.0f / SSAOTarget.Width,
                                                          1.0f / SSAOTarget.Height));

    //Apply
    composer.CurrentTechnique.Passes[0].Apply();

    //Draw
    fsq.Draw(GraphicsDevice);
}
```

Putting it all together

Now ordinarily I would say that's a wrap but this class you'll notice has two very important variables: SampleRadius and DistanceScale. So it is very very necessary that we have a way to modify those values and to display exactly what they are at.

We will split those to two separate functions, Modify and Debug. Modify will handle the changing of the values; you can call this Update if you prefer. Debug will handle the drawing of the values and the SSAO buffer.

Add both of these functions to your SSAO class:

```
//Modify
public void Modify(KeyboardState Current)
{
    float speed = 0.01f;

    if (Current.IsKeyDown(Keys.Z)) sampleRadius -= speed;
    if (Current.IsKeyDown(Keys.X)) sampleRadius += speed;

    if (Current.IsKeyDown(Keys.C)) distanceScale -= speed;
    if (Current.IsKeyDown(Keys.V)) distanceScale += speed;
}

//Debug Values
public void Debug(SpriteBatch spriteBatch, SpriteFont spriteFont)
{
    //Width + Height
    int width = 128;
    int height = 128;

    //Set up Drawing Rectangle
    Rectangle rect = new Rectangle(384, 0, width, height);

    //Begin SpriteBatch for Buffer
    spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.Opaque,
                      SamplerState.LinearClamp, null, null);

    //Draw SSAO buffer
    spriteBatch.Draw((Texture2D)SSAOTarget, rect, Color.White);

    //Draw SSAO Blurred
    rect.X += 128;
    spriteBatch.Draw((Texture2D)BlurTarget, rect, Color.White);

    //End SpriteBatch
    spriteBatch.End();

    //Begin SpriteBatch for Text
    spriteBatch.Begin(SpriteSortMode.Immediate, BlendState.AlphaBlend);

    //Draw sampleRadius
    spriteBatch.DrawString(spriteFont, "Sample Radius: " + sampleRadius.ToString(),
                           new Vector2(0, 128), Color.Red);

    //Draw distanceScale
    spriteBatch.DrawString(spriteFont, "Distance Scale: " + distanceScale.ToString(),
                           new Vector2(0, 148), Color.Blue);
```

```
    //End SpriteBatch
    spriteBatch.End();
}
```

At this point our SSAO class is complete so we're just gonna make the following changes to our Game class to use it with our Deferred Renderer:

```
//Graphics Device
GraphicsDeviceManager graphics;

//SpriteBatch
SpriteBatch spriteBatch;

//Sprite Font
SpriteFont spriteFont;

//Keyboard State Holders
KeyboardState currentK;
KeyboardState previousK;

//Mouse State Holder
MouseState currentM;

//Camera
FreeCamera Camera;

//Light Manager
LightManager lightManager;

//Deffered Renderer
DeferredRenderer deferredRenderer;

//SSAO
SSAO ssao;

//Scene
RenderTarget2D Scene;

//List of Models
List<Model> models;

public Game1()
{
    graphics = new GraphicsDeviceManager(this);
    Content.RootDirectory = "Content";

    graphics.PreferredBackBufferWidth = 800;
    graphics.PreferredBackBufferHeight = 600;
}

protected override void Initialize()
{
    //Initialize Camera
    Camera = new FreeCamera(this, new Vector3(0, 0, 5), new Vector3(0, 0, 0),
                            Vector3.Up);

    base.Initialize();
}
```

```
protected override void LoadContent()
{
    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    //Create spriteFont
    spriteFont = Content.Load<SpriteFont>("DefaultFont");

    //Create Deferred Renderer
    deferredRenderer = new DeferredRenderer(GraphicsDevice, Content,
                                            graphics.PreferredBackBufferWidth,
                                            graphics.PreferredBackBufferHeight);

    //Create SSAO
    ssao = new SSAO(GraphicsDevice, Content, graphics.PreferredBackBufferWidth,
                    graphics.PreferredBackBufferHeight);

    //Create Scene Render Target
    Scene = new RenderTarget2D(GraphicsDevice, graphics.PreferredBackBufferWidth,
                               graphics.PreferredBackBufferHeight, false,
                               SurfaceFormat.Color, DepthFormat.Depth24Stencil8);

    //Create Light Manager
    lightManager = new LightManager(Content);

    //Load SpotLight Cookies
    Texture2D spotCookie = Content.Load<Texture2D>("SpotCookie");
    Texture2D squareCookie = Content.Load<Texture2D>("SquareCookie");

    //Add a Directional Light
 lightManager.AddLight(new Lights.DirectionalLight(Vector3.Down, Color.White, 0.05f));

    //Add a Spot Light
 lightManager.AddLight(new Lights.SpotLight(GraphicsDevice, new Vector3(0, 15.0f, 0),
                                            new Vector3(0, -1, 0),
                                            Color.White.ToVector4(), 0.10f, true,
                                            2048, spotCookie));

    //Add a Point Light
  lightManager.AddLight(new Lights.PointLight(GraphicsDevice, new Vector3(0, 5.0f, 0),
                                              20.0f, Color.White.ToVector4(), 0.80f,
                                              true, 256));

    //Initialize Model List
    models = new List<Model>();

    //Load Models
    Model scene = Content.Load<Model>("Scene");
    models.Add(scene);
}

protected override void UnloadContent()
{
}

protected override void Update(GameTime gameTime)
{
    //Input Housekeeping
    previousK = currentK;
    currentK = Keyboard.GetState();
    currentM = Mouse.GetState();
```

```
    Mouse.SetPosition(GraphicsDevice.Viewport.Width / 2,
                      GraphicsDevice.Viewport.Height / 2);

    //Exit check
    if (currentK.IsKeyUp(Keys.Escape) && previousK.IsKeyDown(Keys.Escape))
       this.Exit();

    //Update Camera
    Camera.Update(this, currentK, previousK, currentM);

    //Modify
    ssao.Modify(currentK);

    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    //Clear
    GraphicsDevice.Clear(Color.CornflowerBlue);

    //Draw Shadow Maps
    lightManager.DrawShadowMaps(GraphicsDevice, models);

    //Draw with SSAO unless F1 is down
    if (currentK.IsKeyDown(Keys.F1))
    {
        //Draw using Deferred Renderer straight to BackBuffer
        deferredRenderer.Draw(GraphicsDevice, models, lightManager, Camera, null);
    }
    else
    {
        //Draw using Deferred Renderer
        deferredRenderer.Draw(GraphicsDevice, models, lightManager, Camera, Scene);

        //Draw non blurred ssao
        ssao.Draw(GraphicsDevice, deferredRenderer.getGBuffer(), Scene, Camera, null);
    }

    //Debug Deferred Renderer
    deferredRenderer.Debug(GraphicsDevice, spriteBatch);

    //Debug SSAO
    ssao.Debug(spriteBatch, spriteFont);

    //Base Drawing
    base.Draw(gameTime);
}
```

That’s all there is to it! Deferred Rendering with Directional Lights, Spot Lights with ESM, Point Lights with ESM and SSAO with Normals. Not as hard as it seemed right?

Recommended Reading and References

For a survey of Normal encoding methods:

http://aras-p.info/texts/CompactNormalStorage.html

For an excellent tutorial on Deferred Rendering in XNA 3:

http://www.catalinzima.com/tutorials/deferred-rendering-in-xna/

And a ported XNA 4 version:

http://roy-t.nl/index.php/2010/12/28/deferred-rendering-in-xna4-0-source-code/

For information on a Logarithmic/Exponential Depth Buffer:

http://www.gamasutra.com/blogs/BranoKemen/20090812/2725/Logarithmic_Depth_Buffer.php

For more information on Depth Buffers:

http://www.sjbaker.org/steve/omniv/love_your_z_buffer.html

For information on Exponential Shadow Mapping:

http://www.thomasannen.com/pub/gi2008esm.pdf

For more information on Exponential Shadow Mapping:

http://pixelstoomany.wordpress.com/category/shadows/exponential-shadow-maps/

For a very good survey of SSAO techniques:

http://meshula.net/wordpress/?p=145

For Alex Urbano Alvarez's SSAO implementation, of which this tutorial is based:

http://xnacommunity.codeplex.com/wikipage?title=SSAO

For a detailed paper on SSAO theory:

http://developer.download.nvidia.com/presentations/2008/GDC/GDC08_Ambient_Occlusion.pdf

For more Deferred Rendering theory:

http://www.garrywilliams.fr/projects