ATLS 4519/5519 LAB 11[1]
JK Bennett
Spring 2013

This lab dives a little bit deeper into HLSL effects, particularly as they relate to lighting and shading. We will begin by reviewing some basic 3D principles, and then move on to more advanced topics.

You will need to load several files to complete this lab. Download and unzip this file to obtain them. If for some reason this does not work, here is the hard link:
http://redwood.colorado.edu/jkb/atls-4519-S13/labs/Lab11_Files.zip.

We will start with code a code file named Lab11_Game1.cs. This code loads the effect file, positions the camera, and clears the window and Z buffer in the Draw method

Create a new Windows 4.0 game project, and replace the contents of Game1.cs with the contents of Lab11_Game1.cs (located in the directory that you just downloaded). Change the namespace declaration in Program.cs to ATLS_4519_Lab11.

Add the Lab11_effects.fx file to your content project from the Lab11_Files folder: This file contains the techniques we will start with in this lab. We will shortly create our own effects file from scratch.

Compile and run the code. You should see a single triangle on a dark blue background.

You might ask why is it important to learn to code in HLSL. The answer is that, although HLSL does not directly improve gameplay, enhancing the quality of the final image can significantly increase overall enjoyment (plus it's always nice to know more about what's under the hood).

When this lab is complete, every vertex that is drawn will pass through your vertex shader, and even every pixel drawn will have passed through your pixel shader. You will learn that shaders can perform almost any manipulation upon the data that they process. HLSL represents the link between XNA code and what you see on the screen. The BasicEffects provided by XNA are in fact created by HLSL code written by Microsoft developers (you can download this code if you like). The best games on the market typically distinguish themselves with their graphical special effects.

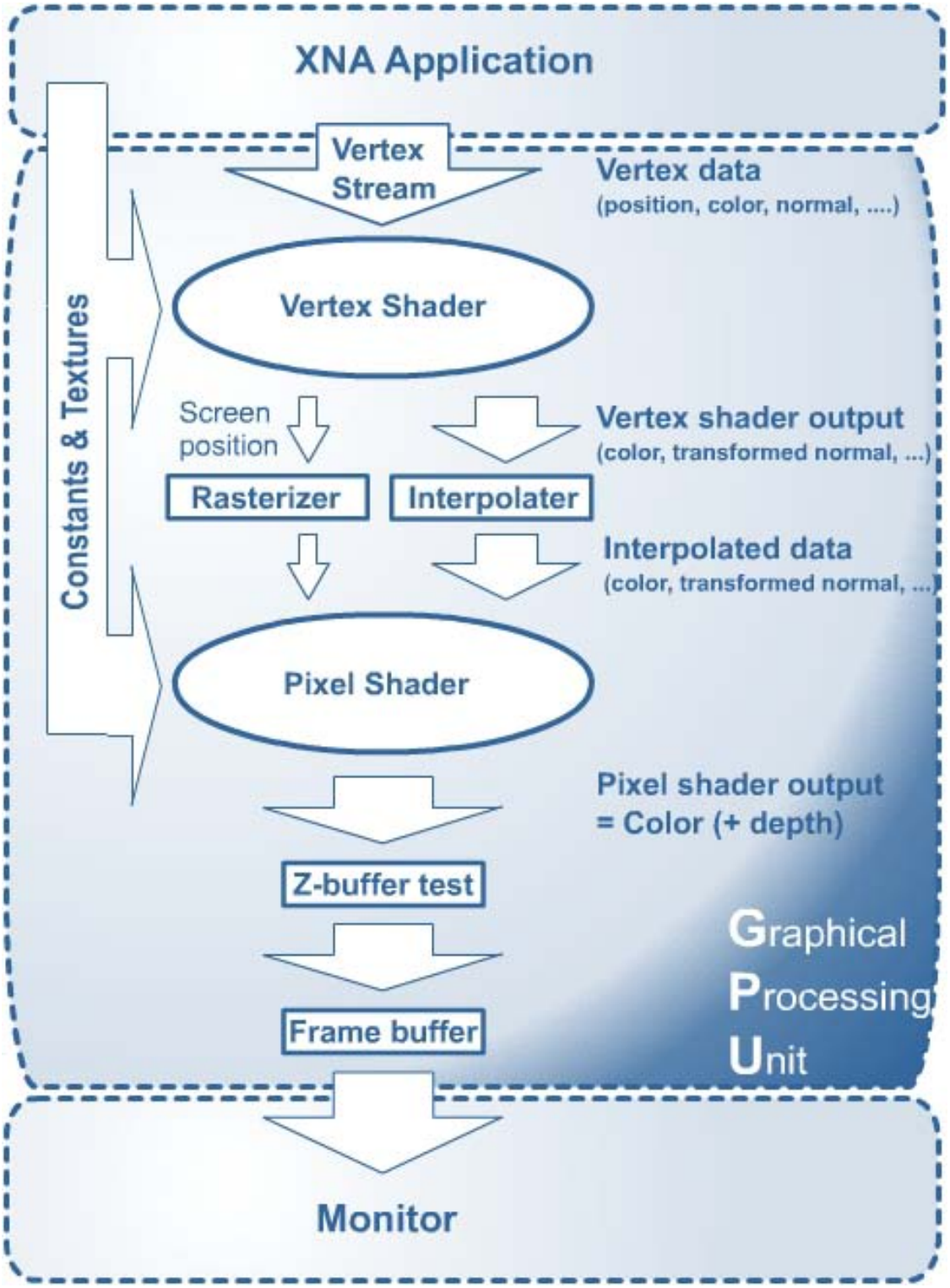
The figure below depicts where the shader code fits into the big picture. The big arrow from the XNA application to the vertex shader represents the flow of vertex data from the XNA application to the vertex shader on the graphics card. This "vertex stream" is sent every time we issue some kind of draw command in XNA. When we pass the vertex data from our XNA application to the vertex shader, we need to pass information indicating what kind of data is contained in the vertex stream. When writing the vertex shader, we need to specify exactly what information can be found in the stream, and where.

So what we need is:

- a structure that can hold the necessary data for each vertex, and
- a definition of the data, so the vertex shader knows what data is included with every vertex.

In the starting code, we use the VertexPositionColor struct, which satisfies both requirements. However, now we will define a new struct, MyOwnVertexFormat, which will be exactly the same as the VertexPositionColor struct. The purpose in duplicating something that is already provided is to learn exactly how things work, and to provide a foundation for more advanced manipulation later.

[1] This lab derives significantly from a series of excellent on-line tutorials created by Riemer Grootjans for XNA 3.1. Be sure to visit his web site at http://www.riemers.net/.



To satisfy the two requirements, in the example of a simple colored triangle, we need our vertices to hold 3D Position data as well as Color data. Therefore, we need to define a new struct (put this at the top of our code):

```
struct MyOwnVertexFormat
   {
       private Vector3 position;
       private Color color;
```

```
        public MyOwnVertexFormat(Vector3 position, Color color)
        {
            this.position = position;
            this.color = color;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Color,
                    VertexElementUsage.Color, 0)
        );

    }
```

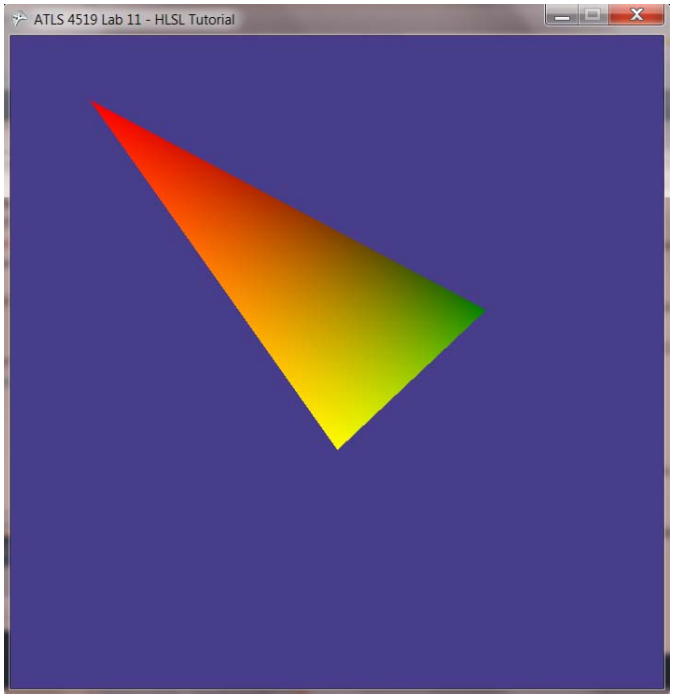
Our new struct will hold a position and a color. We define a constructor, so we can create and fill a new instance of this struct in one line. We also defined a VertexDeclaration that will tell the graphics card how to process vertex information, and the VertexElements that comprise these vertices. For each type of data, we define how many bytes it occupies, what it is used for, and where it can be found. The first argument of a VertexElement is the offset in bytes of the where the VertexElement data can be found in the vertex stream. For example, the first element, the position, starts at offset 0. The next element indicates the format of the data, e.g., Vector3. A position is composed of three floats. The next argument describes the how the information in the VertexElement is to be interpreted, e.g., as position, color, texture coordinate, etc. This information is needed so that XNA can automatically map the data from the vertex stream to the correct variables in our vertex shaders. Although we gave the elements meaningful names (position, color), the graphics card needs to be told explicitly what data it will receive. The final VertexElement parameter is called the UsageIndex, and will be always be zero if we are only using the data for one purpose. However, if we want to, for example, add two textures to a triangle, we would have to pass two sets of texture coordinates with each vector. The UsageIndex allows us to do this. Since we are only passing one position and one color for each vertex, the UsageIndex will be 0 for both VertexElements.

Now let's change our code to use MyOwnVertexFormat instead of the VertexPositionColor struct. Change our SetUpVertices as follows:

```
 private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[3];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-2, 2, 0), Color.Red);
            vertices[1] = new MyOwnVertexFormat(new Vector3(2, -2, -2), Color.Green);
            vertices[2] = new MyOwnVertexFormat(new Vector3(0, 0, 2), Color.Yellow);

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
                                    vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }
}
```

Note that we have also adjusted the vertexBuffer declaration to use MyOwnVertexFormat.

OK, we have recreated the XNA VertexPositionColor struct. When you run the code (do this now), you should see the same triangle (below). Only this time, we know what the VertexDeclaration contains and, more importantly, we know how to extend a vertex format.



Here is our code so far:

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Color color;

        public MyOwnVertexFormat(Vector3 position, Color color)
        {
            this.position = position;
            this.color = color;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Color, VertexElementUsage.Color,
0)
        );

    }
```

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

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

        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Lab11_effects");
            SetUpVertices();
            SetUpCamera();
        }

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[3];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-2, 2, 0), Color.Red);
            vertices[1] = new MyOwnVertexFormat(new Vector3(2, -2, -2), Color.Green);
            vertices[2] = new MyOwnVertexFormat(new Vector3(0, 0, 2), Color.Yellow);

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }

        private void SetUpCamera()
        {
            cameraPos = new Vector3(0, 5, 6);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 0, 1), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        protected override void UnloadContent()
        {
        }
```

```
        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.DarkSlateBlue, 1.0f, 0);

            effect.CurrentTechnique = effect.Techniques["ColoredNoShading"];
            effect.Parameters["xView"].SetValue(viewMatrix);
            effect.Parameters["xProjection"].SetValue(projectionMatrix);
            effect.Parameters["xWorld"].SetValue(Matrix.Identity);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleList, 0, vertexBuffer.VertexCount / 3);
            }

            base.Draw(gameTime);
        }
    }
}
```

Creating a Custom HLSL Shader

In this section, we will write our first HLSL code from scratch, together with our first vertex shader.

Take another look at our flowchart above. Notice the big arrow from the XNA application toward the vertex shader. At this point, we have the vertex stream (the position and color of our 3 vertices), as well as the metadata, which describes the contents of this vertex stream and the memory used by each vertex.

Create a new effect file (right click on the content project in the Solution Explorer and select Add ➢ New Item ➢ Effect File). Name your new file Effect1.fx.

You should see the file added to the content project. Open this file by double clicking on it. You will see that some default effect code has been provided. Delete it all.

Although HLSL is not C# code, you should have no problem reading and writing the this C-like code. As we have seen in previous labs, an effect file describes one or more techniques. A technique has one or more passes.

Let's create a new technique. Add this code to the empty Effect1.fx file:

```
technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = NULL;
    }
}
```

This defines a technique, SimpleTexture, which has one pass. This pass has a vertex shader (STVertexShader), but no pixel shader. This indicates our vertex shader will pass its output data to the default pixel shader.

The main task of a 3D vertex shader is to accept a vertex with a 3D position, and to process these data into 2D screen coordinates. Optionally, the vertex shader can also produce extra data, such as the color or texture coordinate.

Before we start coding our vertex shader, we need to define a structure to hold the data our vertex shader will send to the default pixel shader. The vertex shader method we will create, STVertexShader, will simply transform the 3D positions in the vertices it receives from our XNA application to 2D screen pixel coordinates, and send them together with the color to the pixel shader. Add this code at the top of the file:

```
struct VertexToPixel
{
    float4 Position     : POSITION;
    float4 Color        : COLOR0;
};
```

This again looks very much like C#, except for the :POSITION and :COLOR0. These are called "semantics," which indicate to the GPU how it should use the data. To create a STVertexShader method, add this method between the structure definition and our technique definition:

```
VertexToPixel STVertexShader(float4 inPos : POSITION)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xViewProjection);
    Output.Color = 1.0f;

    return Output;
}
```

The first line indicates our method (our vertex shader) will generate a filled VertexToPixel structure. It also indicates the vertices that XNA sends to your vertex shader should contain position data, as indicated by the POSITION semantic. This semantic is critical (leave it out and the HLSL compiler will complain): it links the data inside our vertex stream (as defined in our VertexDeclaration) to our HLSL code.

Keep in mind that this method is called for every vertex in your vertex stream. The first line in the method creates an empty output structure. The second line takes the 3D coordinates of the vertex, and transforms them to 2D screen coordinates by multiplying them by the combination of the View and Projection matrices. Then we fill the Color member of the output structure. When you examine the definition of the output structure, you will see this has to be a float4: one float for each of the three color components, and an extra float for the alpha (transparency) value. We could fill this color by using the following code (do not do this):

```
Output.Color.r = 1.0f;
Output.Color.g = 0.0f;
Output.Color.b = 1.0f;
Output.Color.a = 1.0f;
```

This would indicate purple, as you combine red and blue. The following code does exactly the same thing:

```
Output.Color.rba = 1.0f;
Output.Color.g = 0.0f;
```

Instead of rgba, we can also use xyzw. The rgba nomenclature is usually used when working with colors, while xyzw is used in combination with coordinates, but they do the same thing. You can also use indices, as follows:

```
Output.Color[0] = 1.0f;
Output.Color[1] = 0.0f;
Output.Color[2] = 1.0f;
Output.Color[3] = 1.0f;
```

In our example vertex shader above, we set Output.Color = 1.0f, which means the four components of the color are all set to 1.0f, corresponding to white. Thus, our vertex shader will transform each 3D vertex to 2D screen coordinates, and pass these coordinates together with the color white to the default pixel shader. In our case of our one triangle, the pixel shader will draw a solid white triangle to the window.

However, we still need to define xViewProjection, the matrix used in the vertex shader to transform the 3D coordinate to the 2D screen coordinate. This matrix depends on the View and Projection matrices of the camera, which are the same for all vertices rendered in one frame. This matrix needs to be declared, so add this declaration at the very top of our HLSL code:

```
float4x4 xViewProjection;
```

This defines xViewProjection is a matrix with 4 rows and 4 columns, so it can hold a standard XNA transformation matrix. Our XNA application will fill in this matrix in the next section.

That's it for our basic HLSL code. We still need to call the technique from our XNA application, as well as set the xViewProjection matrix. We will do that in the next section.

Here is our HLSL code so far:

```
float4x4 xViewProjection;

struct VertexToPixel
{
    float4 Position     : POSITION;
    float4 Color        : COLOR0;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};


VertexToPixel STVertexShader(float4 inPos : POSITION)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xViewProjection);
    Output.Color = 1.0f;

    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = NULL;
    }
}
```

So far, we have a vertex buffer, filled with three vertices that define a single triangle. We also have the metadata: the VertexDeclaration, which describes what kind of data is contained in the vertex stream, together with the offsets to that data. We also have a very simple vertex shader. From our vertex stream, the shader extracts only the position data. For each vertex, this 3D position is transformed into 2D screen coordinates, and passed on to the pixel shader (which does not yet exist). To perform this transformation, we multiply each vertex with the matrix that is the combination of the camera's View and Projection matrices, which at this point are not yet specified by our XNA code.

In our XNA application, we need to load our new effect file, and set the transformation matrix. To do this, add the following code to the LoadContent method:

```
effect = Content.Load<Effect>("Effect1");
```

In order to draw our triangle using this technique, we need to specify the technique and set its parameters. In our case, the only parameter we have to set is xViewProjection, which is the combination of the viewMatrix and the projectionMatrix. So, in the Draw method replace the existing code (between the device.Clear line and the foreach line) with this code:

```
effect.CurrentTechnique = effect.Techniques["SimpleTexture"];
effect.Parameters["xViewProjection"].SetValue(viewMatrix * projectionMatrix);
```

Remove the lines where other parameters such as xView, xWorld, are set because these do not exist (yet) in our effect file.

If you were to try to run the program at this point, you will get an error stating "Both a vertex shader and pixel shader must be set on the device before any draw operations may be performed." This is because although our technique contains a vertex shader, it doesn't yet contain a valid pixel shader. So let's write one.

The pixel shader is called for each pixel of the screen that needs to be drawn. In most cases, the pixel shader only needs to calculate the correct color (although we will see later that even this calculation can be quite complex).

The pixel shader receives its input (position and color, in our case) from the vertex shader, and needs to output only color. First, define its output structure at the top of our effect file:

```
struct PixelToFrame
{
    float4 Color        : COLOR0;
};
```

Our first pixel shader will be a very simple method, as follows:

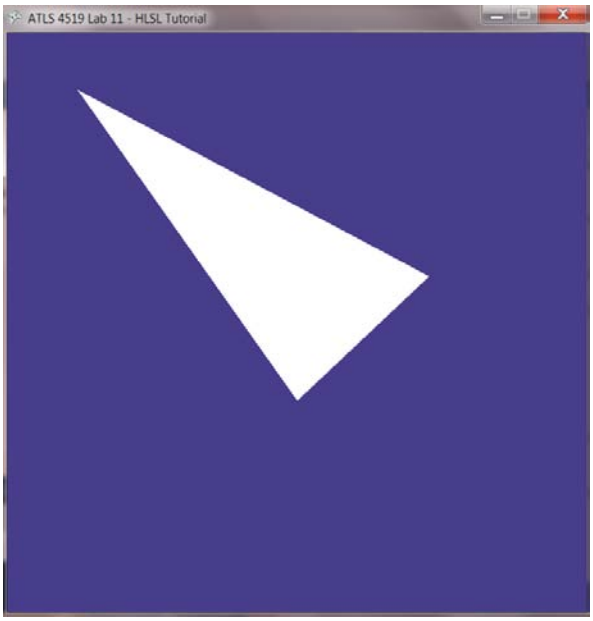
```
PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

    Output.Color = PSIn.Color;

    return Output;
}
```

First, an output structure is created, and the color received from the vertex shader is put in the output structure. That's all the pixel shader does at this point.

Now we still need to set this method as pixel shader for our technique, at the bottom of the file:

```
PixelShader = compile ps_2_0 STPixelShader();
```

Run the code at this point. You should see a white triangle, as shown below:

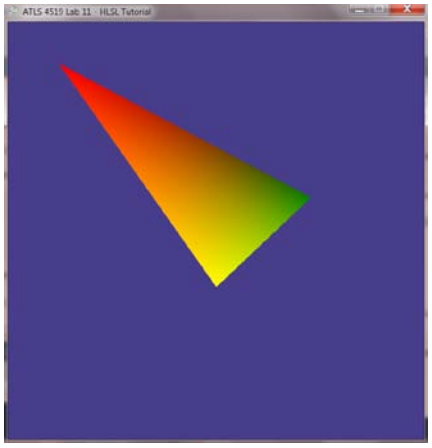


The triangle is white because we coded our vertex shader to draw every vertex white. Let's add some color. Change the vertex shader as follows:

```
VertexToPixel STVertexShader( float4 inPos : POSITION, float4 inColor : COLOR0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position =mul(inPos, xViewProjection);
    Output.Color = inColor;

    return Output;
}
```

Note the changes in the first line. Now, our vertex shader also expects the vertices to carry color information. This is OK, since the MyOwnVertexFormat contains both position and color information. In the interior of the shader, we now route the color we get from our XNA application to the output of the vertex shader, instead of making the vertex white.

Run the code again. You should see the same colored triangle as before. Our vertex shader simply transforms the 3D coordinates to 2D screen coordinates, and passes these coordinates together with the correct color to the pixel shader. The pixel shader passes this color directly on to the screen.



Optional - test your knowledge:

- Adjust the vertex shader so the triangle is rendered in solid yellow.
- Make the colors in the vertices dependant on their 3D position.

Here is our code so far:

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Color color;

        public MyOwnVertexFormat(Vector3 position, Color color)
        {
            this.position = position;
            this.color = color;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Color, VertexElementUsage.Color,
0)
        );

    }

    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

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

        protected override void Initialize()
```

```
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            SetUpVertices();
            SetUpCamera();
        }

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[3];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-2, 2, 0), Color.Red);
            vertices[1] = new MyOwnVertexFormat(new Vector3(2, -2, -2), Color.Green);
            vertices[2] = new MyOwnVertexFormat(new Vector3(0, 0, 2), Color.Yellow);

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }

        private void SetUpCamera()
        {
            cameraPos = new Vector3(0, 5, 6);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 0, 1), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.DarkSlateBlue, 1.0f, 0);

            effect.CurrentTechnique = effect.Techniques["SimpleTexture"];
            effect.Parameters["xViewProjection"].SetValue(viewMatrix * projectionMatrix);


            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
```

```
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleList, 0, vertexBuffer.VertexCount / 3);
            }

            base.Draw(gameTime);
        }
    }
}
```

Effect1.cs

```
float4x4 xViewProjection;

struct VertexToPixel
{
    float4 Position     : POSITION;
    float4 Color        : COLOR0;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION, float4 inColor : COLOR0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position =mul(inPos, xViewProjection);
    Output.Color = inColor;

    return Output;
}


PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

    Output.Color = PSIn.Color;

    return Output;
}


technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}
```

Per-Pixel Color

Our XNA program passes vertices to our vertex shader, which transforms their coordinates to 2D screen coordinates and sends these coordinates together with the color to the pixel shader, which takes the color and draws the pixel on the screen.

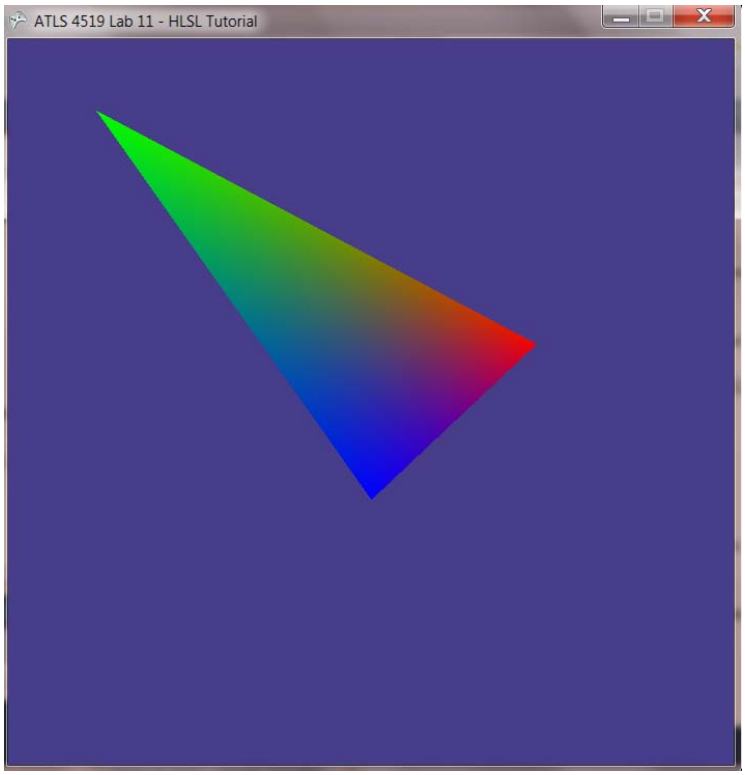
Looking at our flowchart again, notice the rasterizer between the vertex and the pixel shader. This rasterizer determines which pixels on the screen are part our triangle, and makes sure that these pixels are also sent to the pixel shader. Without this rasterizer, only the three points corresponding to the vertices would be sent to the pixel shader. So how does the pixel shader determine the color of the pixels at the interior of the triangle, which are sent to the pixel shader receives at its input? The interpolator, shown next to the rasterizer, is responsible for calculating these values, by interpolating the color value of the corner points. This means that a pixel exactly in the middle between a blue and a red corner point will get the color that is exactly in the middle of these colors on the color scale. In our case, this means that, for all pixels the triangle occupies, the colors will be shaded from corner to corner.

We can also adjust the color using the vertex shader, which we have done before (we made our whole triangle white), and which we will do again now as an exercise. For this example, we will discard the color information provided to us by the vertex stream, and define our own colors. Suppose we want our vertex shader to make the red color component indicate the X coordinate of each vertex, the green component the Y coordinate, and the blue color component indicate the Z coordinate. To do this, insert this line of code into our vertex shader:

```
Output.Color.rgb = inPos.xyz;
```

Note that for any color, a value of zero or below means “none of this color”, and a value of one means “all of this color.” Right now, the coordinates of our vertexes in the XNA application are in the [-2,2] range, which was chosen specifically for this example. Thus, for every color, we expect that the color will not be present in the [-2,0] region, then shaded from “none” to “all” in the [0,1] region, and remain at “all” in the [1,2] region. For example, the point (0,0,0), which is part of our triangle, should be drawn completely black (zero red component, zero green component and zero blue component).

Try to run the code with the new line instead of the `Output.Color = inColor;` line in the vertex shader. The colors will be different from the previous section: this time, the corner towards you (positive Z coordinate) will be blue, the upper corner (positive Y coordinate) will be green and the point to the right (positive X coordinate) will be red, as shown below:



This looks nice, but it is not what we wanted. For example, we wanted the (0,0,0) point in the middle of our triangle to be black, which it is not. So what is happening? Before the colors are passed to the interpolator, the three color values of the three vertices are being clipped to the [0,1] region. For example, the (-2,-2,2) vertex should have -2, -2 and 2 as RGB color values, but instead it gets 0, 0 and 1 as its color values. Next, the interpolator simply gives every pixel in the triangle the color that is the interpolation of the three clipped color values in the vertices. So, for example, the (0,0,0) point gets a color value that is an interpolation of color values between the [0,1] region, and thus will never be completely 0,0,0 (=black). To summarize, when using only a vertex shader, the colors of a triangle can only change linearly, and then only with maximum and minimum values in the [0,1] range.

If we take a look at the flowchart, we notice that there are two arrows going from the vertex shader to the pixel shader. The left one is necessary, as it provides the position of the pixel in 2D screen coordinates. The rasterizer and interpolator, as well as the pixel shader, need this information to do their work. Note that we do not HAVE TO use this position as input to our pixel shader. Thus, the bigger arrow is not necessary, but we will almost always want to use it, since it contains the data the pixel shader uses as input. This data can include, interpolated color information, interpolated normal information, interpolated texture coordinates, etc. Note the emphasis on the word "interpolated." We can also pass a copy of the 2D screen position in this arrow, so our pixel shader can use this information as input.

When we look at the flowchart, we see that the pixel shader eventually sends its output to the frame buffer. This is where multiple renderings get blended in case of alpha blending.

Now for a pixel shader example. We are going to demonstrate per-pixel coloring. We are going to program the color of each pixels individually. We will use the same example as before, but this time using a pixel shader. Recall that our objective is to have each of the three color channels take the value of each of the three 3D-coordinates of the pixel. Because we will derive the color from the 3D position, we need the 3D position as input to our pixel shader. The only coordinate currently being passed to the pixel shader is the 2D screen position (we cannot use this value in our pixel shader, since it is the left arrow in our flowchart). So what we need to do is to pass the 3D coordinate from our vertex shader to our pixel shader. To do this, we first redefine the output structure of our vertex shader:

```
struct VertexToPixel
{
    float4 Position     : POSITION;
    float4 Color        : COLOR0;
    float3 Position3D    : TEXCOORD0;
};
```

We have added a structure member to store the 3D position, using the TEXCOORD0 semantic. We can use TEXCOORD0 to TEXCOORD16 to pass float4 values from our vertex shader to our pixel shader.

Next we update our vertex shader so that it routes the 3D position it receives from the vertex stream to its output. To do this, we add the following line to our vertex shader:

```
Output.Position3D = inPos;
```

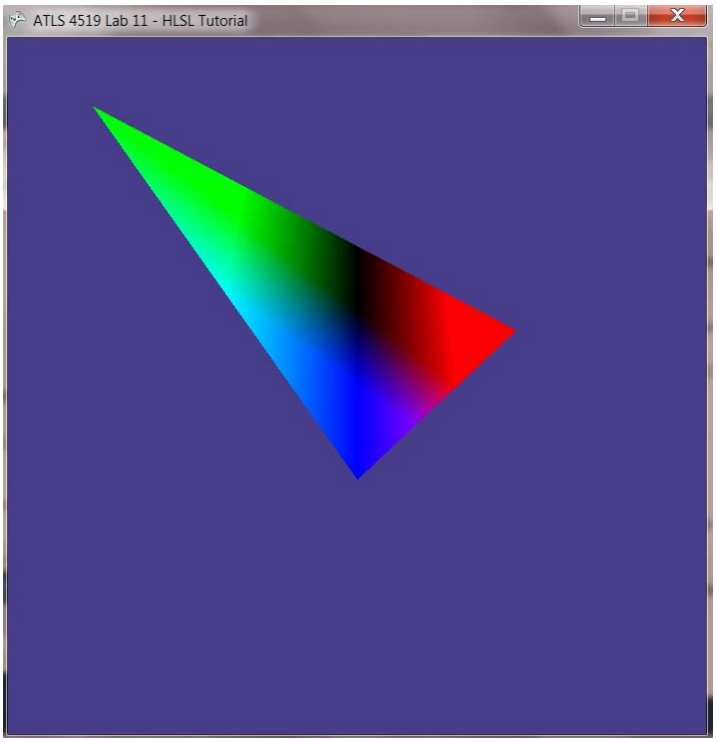
This will have our 3D position sent to the interpolator, which will interpolate the correct 3D position of every pixel in the triangle. For each pixel, this interpolated 3D coordinate will be sent to the pixel shader. The position we receive in the pixel shader will be exact for each pixel, for two reasons:

1. Only colors are clipped to the [0,1] range, TEXCOORD0 semantics can have any value
2. The interpolator performs linear interpolation, and our triangle is linear (since it is flat).

Now we will have our pixel shader set the color components to the value of this 3D coordinate. In our **pixel shader**, add this line to provide the color definition:

```
Output.Color.rgb = PSIn.Position3D.xyz;
```

This line sets the value of the X coordinate as the red color component, and so on. Now run the code. You should see the image below: a simple triangle, with pixels that display color. But, in this case, we have programmed the color of every pixel individually. How cool is that? As a quick check, you can see that now the (0,0,0) middle point of our triangle is black. Woo hoo!



We have not made any changes to our C# XNA code, only the HLSL part has been changed. This means our CPU still has the same workload. Only the GPU will have to work a little bit harder, but no worries, it has lots of horsepower left.

Optional - test your knowledge:

- In your pixel shader, add 1 to the Red color component of each pixel.

Here is our HLSL code so far:

```
float4x4 xViewProjection;

struct VertexToPixel
{
    float4 Position     : POSITION;
    float4 Color        : COLOR0;
    float3 Position3D    : TEXCOORD0;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};
```

```
VertexToPixel STVertexShader( float4 inPos : POSITION, float4 inColor : COLOR0)
{
    VertexToPixel Output = (VertexToPixel)0;
    Output.Position =mul(inPos, xViewProjection);
       Output.Color.rgb = inPos.xyz;
       Output.Position3D = inPos;

    return Output;
}

PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       Output.Color.rgb = PSIn.Position3D.xyz;

    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}
```

Adding Texture

If you take another look at the large flowchart, you will see we have already worked through most of it, from the vertex stream input, to the output of the pixel shader. We have also set a shader constant, xViewProjection, from within the XNA application. This means we have implemented the Colored technique from the Lab11_effects.fx file. How cool is that?

Now we will load a texture from within the XNA application, and have the pixel shader sample the correct color for each pixel. First, we will load the texture in XNA, and update the vertex stream and VertexDeclaration, so that they send texture coordinate information to the vertex shader.

If you have not already done so, add streettexture.dds to the content project. Add the following instance variable to Game1.cs:

```
Texture2D streetTexture;
```

And add this line to the LoadContent method:

```
streetTexture = Content.Load<Texture2D>("streettexture");
```

The next thing we need to do is update the MyOwnVertexFormat structure so it can handle texture coordinates. We will remove the Color entry, as it will no longer be needed:

```
 struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord)
        {
            this.position = position;
            this.texCoord = texCoord;
```

```
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
                    VertexElementUsage.TextureCoordinate, 0)
        );
    }
```

To specify the position in a texture, we need an X and a Y coordinate, so we will store a Vector2. Now each vertex can now hold a position as well as a texture coordinate, so let's update them in our SetUpVertices method:

```
vertices[0] = new MyOwnVertexFormat(new Vector3(-2, 2, 0), new Vector2(0.0f, 0.0f));
vertices[1] = new MyOwnVertexFormat(new Vector3(2, -2, -2), new Vector2(0.125f, 1.0f));
vertices[2] = new MyOwnVertexFormat(new Vector3(0, 0, 2), new Vector2(0.25f, 0.0f));
```

This defines the 3D position as well as the 2D texture coordinate of our three vertices. We have replaced the Color entry with a TextureCoordinate entry, which is stored as a Vector2. The second argument remained the same, as the texture coordinate can still be found at the same position as where the color was, immediately after the positional data.

That's it for the XNA part, so let's turn to our HLSL file again. Add what is called the "texture sampler" to the effect file with these lines at the top of the file:

```
Texture xTexture;

sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                          magfilter = LINEAR;
                                          minfilter = LINEAR;
                                          mipfilter=LINEAR;
                                          AddressU = mirror;
                                          AddressV = mirror;};
```

The first line defines a variable (xTexture) that will hold our texture. We will fill this variable from the XNA side of things. The second line sets up the sampler. A sampler is linked to a texture, and describes how the texture should be processed. We set the min- and magfilters, together with the mipfilter, to linear, so we will always get smoothly shaded colors, even when the camera is very close to the triangle.

We set the texture coordinate states to mirror, which means that, for example, texture coordinate (2.2f, 1.4f) will be automatically mapped to the [0,1] region and will thus be replaced by (0.2f, 0.6f).

Next, we will instruct our vertex shader to route the texture coordinates from its input to its output. Therefore, we first need to adjust its output structure so the vertex shader will generate a texture coordinate instead of a color:

```
struct VertexToPixel
{
    float4 Position     : POSITION;
    float2 TexCoords    : TEXCOORD0;
};
```

Once again, we are using the TEXCOORD0 semantic to pass additional data from the vertex shader to the pixel shader. Although we are only passing two floats instead of the maximum four, this is the correct choice. Now update the vertex shader to this:

```
VertexToPixel STVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0)
{
```

```
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position =mul(inPos, xViewProjection);
    Output.TexCoords = inTexCoords;

    return Output;
}
```

We have made two major changes:

- The first line indicates that the shader expects the vertices to carry TEXCOORD0 information
- This information is immediately routed to the output of the vertex shader.

The pixel shader receives the interpolated 2D screen position and the interpolated 2D texture coordinate. Using these data, the pixel shader needs to output the pixel color, which will be sampled from the texture at the correct position. To do this, change the output line in the pixel shader to this:

```
Output.Color = tex2D(TextureSampler, PSIn.TexCoords);
```

This code retrieves the color of the pixel in the xTexture image corresponding to the 2D coordinate in PSIn.TexCoords. All we have left to do is to set the xTexture from within XNA. To do this, add the following line to our Draw method:

```
effect.Parameters["xTexture"].SetValue(streetTexture);
```

Run the code at this point. You should see the textured triangle below.

Optional - test your knowledge:

- In your vertex shader, override the texture coordinates so the xy coordinates of the 3D position are stored as xy texture coordinates. This should add a lot of copies of the texture.

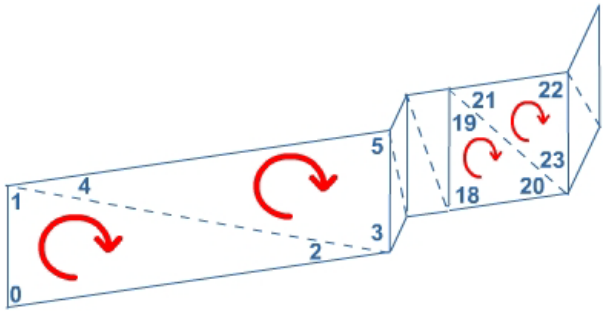
Here is our effect code so far:

```
float4x4 xViewProjection;

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                          magfilter = LINEAR;
                                          minfilter = LINEAR;
```

```
                                        mipfilter=LINEAR;
                                        AddressU = mirror;
                                        AddressV = mirror;};

struct VertexToPixel
{
    float4 Position     : POSITION;
    float2 TexCoords    : TEXCOORD0;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position =mul(inPos, xViewProjection);
    Output.TexCoords = inTexCoords;
    return Output;
}

PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

    Output.Color = tex2D(TextureSampler, PSIn.TexCoords);
    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}
```

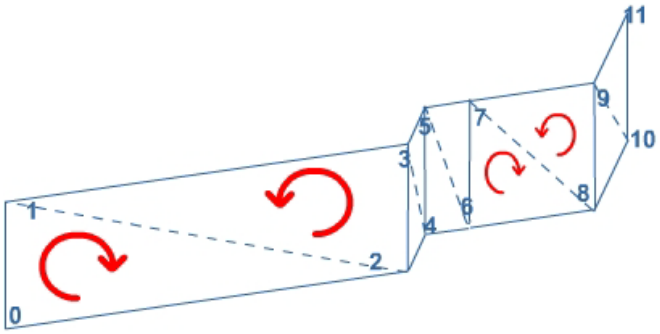
Triangle Strips

Until now, we have only drawn a single triangle. In this section, we will expand our scene, so it will look a bit more like a street. The scene is divided into five parts: the road, the two sides of the pavement border, the pavement itself and the wall. This is a total of five textured quads, which require ten textured triangles to be drawn. We could simply define thirty vertices to draw these ten triangles , each vertex holding a 3D position and a 2D texture coordinate, as depicted below:



Only a few vertex numbers are shown. Every triangle has three vertices, and the vertices are all declared in a clockwise manner relative to the camera (to prevent backface culling). As you can see, much of this vertex information is redundant,

since most vertices are shared by several triangles. We can reduce the amount of redundant vertex information being sent to the graphics card by using what are called TriangleStrips. This idea is illustrated below:

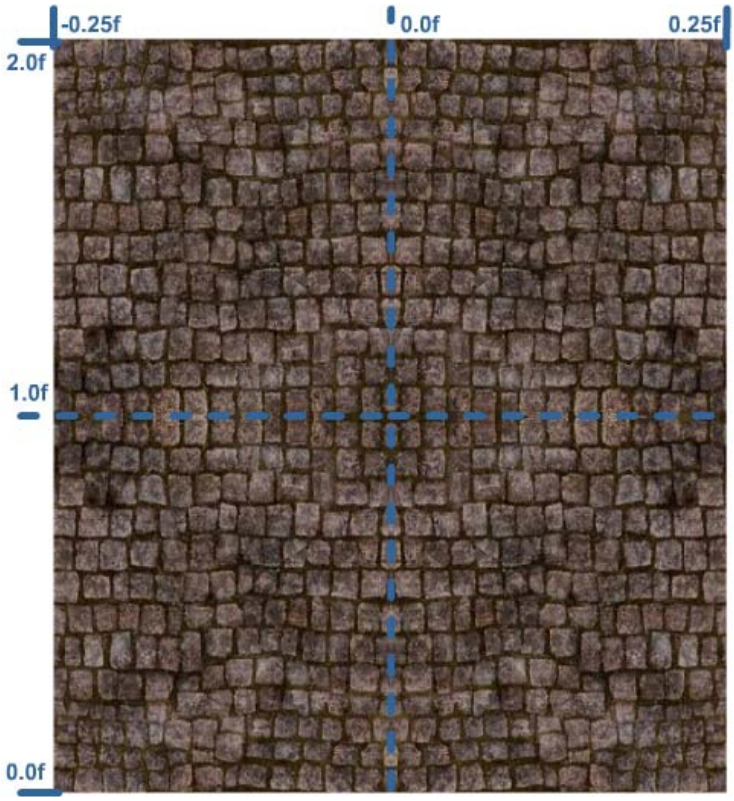


The idea behind TriangleStrips is that we should define each vertex only once. So for the first triangle, we define vertices 0,1 and 2. To create the second triangle, we only have to add vertex 3. Using TriangleStrips, XNA will always use the last three vertices to draw the triangle, so the second triangle uses vertices 1, 2 and 3, the third triangle uses vertices 2,3, and 4, etc. The n-th triangle is defined by the (n-1)-th, the n-th and the (n+1)-th vertex (n starts at 1). TriangleStrips significantly reduce the total number of vertices sent to the graphics card, allowing for much higher frame rates in complex scenes. These is a small remaining problem, however. Notice the red arrows in the TriangleStrip image above. As we have defined a TriangleStrip, it is not possible to define all vertices in a clockwise manner for every triangle. This will always be the case, so when using a TriangleStrip, the rule is we alternate our vertex definition from clockwise to counter clockwise every triangle.

To use TriangleStrips, change the vertex definitions as follows:

```
MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[12];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f));
            vertices[6] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f));
            vertices[7] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f));
            vertices[8] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f));
            vertices[9] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f));
            vertices[10] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f));
            vertices[11] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f));
```

Only twelve vertices are needed to define ten triangles. Notice that we have used horizontal texture coordinates that are outside the [0,1] range, such as -0.25f and 1.25f. This is OK, since we set the shader code AddressU and AddressV states to Mirror, which results in these points being mapped to 0.25f and 0.75f, respectively. This creates a mirrored view, as you can see in the image below. The same trick was used with the vertical coordinates, where we have mirrored the texture twenty-five times. If we had not mirrored the texture image, that small image would have been stretched over the whole street.



Using TriangleStrips, the kind of information contained in our vertices has not changed; we are still sending position and texture information for each vertex. Therefore, we do not need to change our VertexDeclaration. We will, however, change the camera position, to get a better view. To do this, change the contents of the SetUpCamera method as follows:

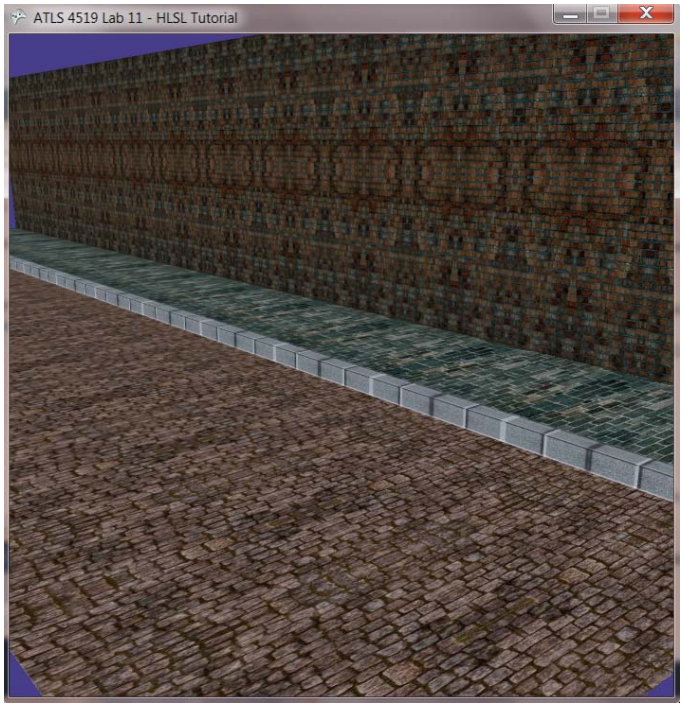
```
private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
                                    device.Viewport.AspectRatio, 1.0f, 200.0f);
        }
```

Finally, we need to tell the Draw method how many triangles we want to draw (ten), and that we have defined these triangles in a strip, instead of a list:

```
device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 10);
```

Run the code, and you should see the image below. If you want to get rid of the purple, you can change the background default to black, as follows:

```
device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);
```



Here is our code so far:

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord)
        {
            this.position = position;
            this.texCoord = texCoord;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0)
        );
    }

    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
```

```
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[12];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f));
            vertices[6] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f));
            vertices[7] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f));
            vertices[8] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f));
            vertices[9] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f));
            vertices[10] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f));
            vertices[11] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f));


            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }

        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;
```

```
            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");
            SetUpVertices();
            SetUpCamera();
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            effect.CurrentTechnique = effect.Techniques["SimpleTexture"];
            effect.Parameters["xViewProjection"].SetValue(viewMatrix * projectionMatrix);
            effect.Parameters["xTexture"].SetValue(streetTexture);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 10);
            }

            base.Draw(gameTime);
        }
    }
}
```

Adding Models to the Scene - the World Transform

In this section we will add two cars and two lamp posts to our scene. If you have not already done so, add the models lamppost.x and car.x to your content project. Now declare the instance variables that will allow is to load these models:

```
Model lamppostModel;
Matrix[] LampModelTransforms;

Model carModel;
Texture2D[] carTextures;
Matrix car1Matrix;
Matrix car2Matrix;
Matrix[] CarModelTransforms;
```

The car model contains positions and textures for all of its vertices, but the lamppost model contains only position and normal information. As a result, we will have to handle each model differently. Add the following code to load the car:

```
private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes
            int i = 0;
```

```
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
                                Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
                                Matrix.CreateTranslation(-28, 0, -1.9f);
        }
```

And the following code to load the lamppost:

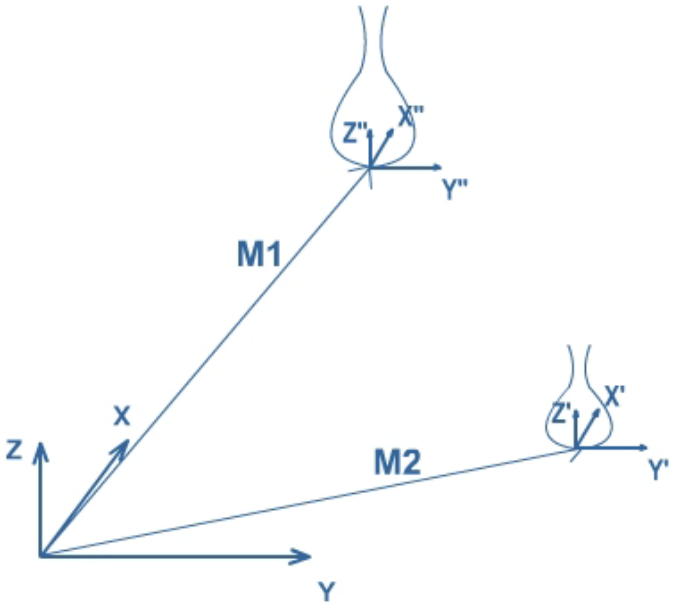
```
private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }
```

Finally, we need to call these methods in LoadContent:

```
// Load the cars
LoadCar();

//Load the Lamp Posts
LoadLamp();
```

With both models loaded, it's time to draw them. To draw an object in our scene, we have to first set the world transform. If we did not, all objects would be drawn relative to the origin, the (0,0,0) point. Imagine that we want to draw two objects from the same mesh, like our two lampposts. If we did not set a different world transform before drawing each lamppost, both objects would be drawn at the same place, and we would see only one of them. Instead, what we want to do is tell XNA to "draw the first object three units to the left and two units up, and draw the second object three units to the right, rotated around the Y axis by 40 degrees and twice as big as the first one". These transformations are called world transforms, which are stored in a matrix called the "World Matrix".

This idea is depicted in the figure below: the large axes represent our World axes, with its origin at the World (0,0,0) point. Suppose we would like to draw the first object from a mesh. First, we would have to tell XNA to create a new axis, with an origin where we would like the center of the object to be drawn. This new location, as well as its rotation and its scaling, are stored in matrix M1, the World Matrix for that object drawn in that location with those transformations. When we draw the object, its mesh will be drawn around its new axes, as shown.



The same story for the second object: we have to create M2 as the World transform for object 2, so it will be drawn around the correct axes. To make this all happen, we have to multiply our vertices by these World matrices in the vertex shader. However, we are not done. The World Matrix only places the object correctly with respect to world coordinates. We also have to create the view as seen by the camera (the View Matrix), and the camera's view as projected onto the 2D screen (the Projection Matrix). Multiplying each vertex by these matrices in the correct order of multiplication (World, View, then Projection) will result in the scene being correctly rendered. For now, we will perform the matrix multiplication in our XNA application, and pass the result to our HLSL shader, so we need to perform only the multiplication of our vertices with this combined matrix in the vertex shader. To pass the multiplied transform matrix in, we need to add a variable to our effect file. To do this, change both instances of "xViewProjection" to "xWorldViewProjection":

```
float4x4 xWorldViewProjection;
```

And

```
Output.Position =mul(inPos, xWorldViewProjection);
```

The first line tells the HLSL code to expect the XNA application to set this matrix, and the second line (in the vertex shader) multiplies every vertex by this combined matrix. The result is that the position of each vertex is first transformed to its proper local axis, and then transformed to 2D screen coordinates.

We still need to fill this matrix. Go to the Draw method in our XNA code, and replace the line where you fill the xViewProjection matrix by this line:

```
effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix * projectionMatrix);
```

Because our TriangleStrip containing the street actually needs to be drawn around the (0,0,0) World origin, we don't need a World matrix for the street. In this case, we specify Matrix.Identity, which is the unity element in matrix math: multiplying matrix M by the identity matrix gives M. More applicable for us: multiplying position P with the identity matrix, gives P again.

When you run this code, you should see the same image as in the last section. Now we are going to add objects. The DrawCar method will render a car at any position, since we can specify its World matrix as an argument. We also specify the technique to use, so later in the lab we will be able to choose different techniques at different stages. The DrawLampPost method allows us to specify both a scale and translation; the appropriate World Matrix is created in the method itself.

```
private void DrawCar(Matrix wMatrix, string technique)
{
    int i = 0;
    foreach (ModelMesh mesh in carModel.Meshes)
    {
        Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
            currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
                        projectionMatrix);
            currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
        }
        mesh.Draw();
    }
}


private void DrawLampPost(float scale, Vector3 translation, string technique)
{
    Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);
    foreach (ModelMesh mesh in lamppostModel.Meshes)
    {
        Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
            currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
                        projectionMatrix);
        }
        mesh.Draw();
    }
}
```

Because the lamppost model only contains position and normal information, we have to create a new shader technique (we will call it SimpleNormal) to render the lamppost. Add the following new technique to our effect file:

```
//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position     : POSITION;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
```

```
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color

       Output.Color = baseColor;

   return Output;
}


technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}
```

Using the DrawCar method, it's very easy to render a car to our scene. Add these lines in our Draw method:

```
DrawCar(car1Matrix, "SimpleTexture");

DrawCar(car2Matrix, "SimpleTexture");
```

Since the correct matrix for each car was created in the LoadCar method, all we have to do is pass these matrices as arguments. In the case of the two lampposts, we provide the scaling and translation information as arguments to the DrawLampPost method:

```
DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");

DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");
```

Run the code at this point, you should see two textured cars, and two dark gray lampposts, as the SimpleNormal pixel shader sets the lamppost color to a dark gray. However, if you look closely, you will see that the car textures are a bit confused. This is because the texture coordinates used in the car model are inside the [1,2] region instead of in the [0,1] region. Since we are using the Mirror texture addressing mode, this will result in the textures being applied mirrored on the car. To compensate for this, add the following line to the beginning of our pixel shader:

```
PSIn.TexCoords.y--;
```

Now when you run the code, all should be well, and look like the figure below. The scene looks a bit flat, because only ambient lighting has been used. In the next section, we will add normals to our vertices. These are needed before we can move on to our first light.

Optional - test your knowledge:

- Add another car to your scene, in the middle of the street.
- Rotate this third car.

Here is our code so far:

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord)
        {
            this.position = position;
            this.texCoord = texCoord;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0)
        );
    }
```

```
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[12];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f));
            vertices[6] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f));
            vertices[7] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f));
            vertices[8] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f));
            vertices[9] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f));
            vertices[10] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f));
            vertices[11] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f));


            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }

        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void LoadCar()
```

```
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }


        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

private void DrawCar(Matrix wMatrix, string technique)
{
    int i = 0;
    foreach (ModelMesh mesh in carModel.Meshes)
    {
        Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
            currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
projectionMatrix);
            currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
        }
        mesh.Draw();
    }
}


private void DrawLampPost(float scale, Vector3 translation, string technique)
{
    Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);

    foreach (ModelMesh mesh in lamppostModel.Meshes)
    {
        Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
            currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
projectionMatrix);
```

```
        }
        mesh.Draw();
    }
}



        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();


            SetUpVertices();
            SetUpCamera();
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            effect.CurrentTechnique = effect.Techniques["SimpleTexture"];
            effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
            effect.Parameters["xTexture"].SetValue(streetTexture);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 10);
```

```
            }
            DrawCar(car1Matrix, "SimpleTexture");

            DrawCar(car2Matrix, "SimpleTexture");

            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");

            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");


            base.Draw(gameTime);
        }
    }
}
```

Effect1.fx:

```
// Global Inputs

float4x4 xWorldViewProjection;

//Texture Samplers

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = mirror;
                                                    AddressV = mirror;};

// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
{
    float4 Position     : POSITION;
    float2 TexCoords    : TEXCOORD0;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position =mul(inPos, xWorldViewProjection);
    Output.TexCoords = inTexCoords;

    return Output;
}

PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       PSIn.TexCoords.y--;
```

```
       Output.Color = tex2D(TextureSampler, PSIn.TexCoords);

    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position         : POSITION;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position =mul(inPos, xWorldViewProjection);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

    Output.Color = float4(.1, .1, .1, 1); // pick a dark gray color

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}
```

Adding Normals

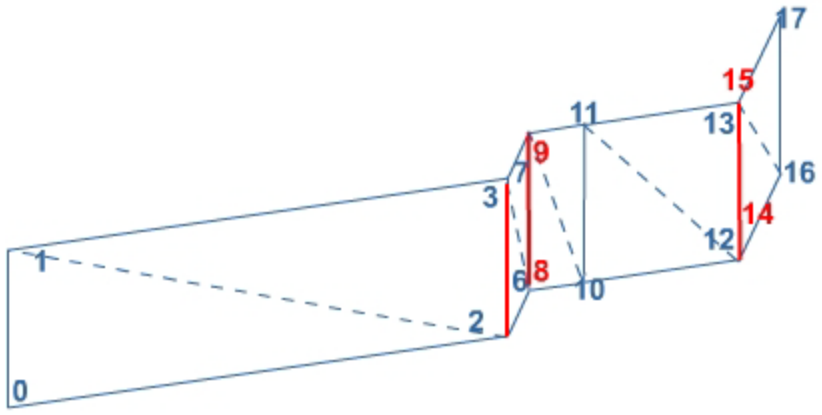
Before we can start defining our own lights, we need to add normals to every vertex. The normal has to be included in the vertex stream, so we first need to redefine the MyownVertexFormat structure:

```
struct MyOwnVertexFormat
   {
       private Vector3 position;
       private Vector2 texCoord;
       private Vector3 normal;

       public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
       {
           this.position = position;
           this.texCoord = texCoord;
           this.normal = normal;
       }

       public static VertexDeclaration VertexDeclaration = new VertexDeclaration
       (
           new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
           new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
                   VertexElementUsage.TextureCoordinate, 0),
           new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
                   VertexElementUsage.Normal, 0)
       );
   }
```

The last line defines the vertex element for normal data. This indicates the normal data can be found after five floats (three of the position, two of the texture coordinates). A normal is stored as three floats, and we want to bind it the NORMAL semantic in of our vertex shader's input.

Now we need to expand our vertices with normal data. This is quite easy: the normals of the horizontal quads (such as the street) are pointing upward, and the normals of the vertical quads (such as the wall) are pointing to the left (the negative X direction in our case). But we have a problem: because we are using a triangle strip, some vertices are shared by two triangles that should have a different normal. This is an intrinsic problem with triangle strips. One solution would be to give the shared vertices an interpolated normal. Since we want to clearly see the edges between perpendicular surfaces, this is not a good solution in this instance.

A better approach is to add 'ghost triangles'. The idea is that we add two new vertices at the places where two triangles with different normals share a side. In the image below, these sides are indicated by a red line; the extra vertices are also red (vertices 4 and 5 should also be drawn in red, but they are not shown in the figure). Note that the coordinates of the red vertices are exactly the same as those of the previous blue vertices; only the normals should be different.



With this approach, we will define eighteen vertices, defining ten "real" triangles and six "ghost" triangles. Even in this almost-worst-case example, we still have to store twelve vertices less than if they would have been stored in a Triangle List.

Here is the array containing all vertices, with normal data added. Replace SetUpVertices as shown:

```
private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
                              new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
                              new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f),
                              new Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f),
                              new Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f),
                              new Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f),
                              new Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f),
                              new Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f),
                              new Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f),
                              new Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f),
                              new Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f),
                              new Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f),
                              new Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f),
                              new Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
                              new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f),
                              new Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
                              new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
                              new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
                              new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
                              vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }
```

Finally, we need to update the number of triangles to draw in the Draw method:

```
device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
```

Our models already have normal data included, we just have not yet used this information. So there's nothing we have to change to the rest of the code. We have not yet modified our shader code to make use of normal data; we will do that in the next section.

When you run this code, you should see the same image as last section. This time, however, our vertex stream includes the correct normal data, so we are finally ready to define our first light.

Here is our code so far (the effect file is unchanged, so only the C# code is shown):

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }


    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
```

```
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
```

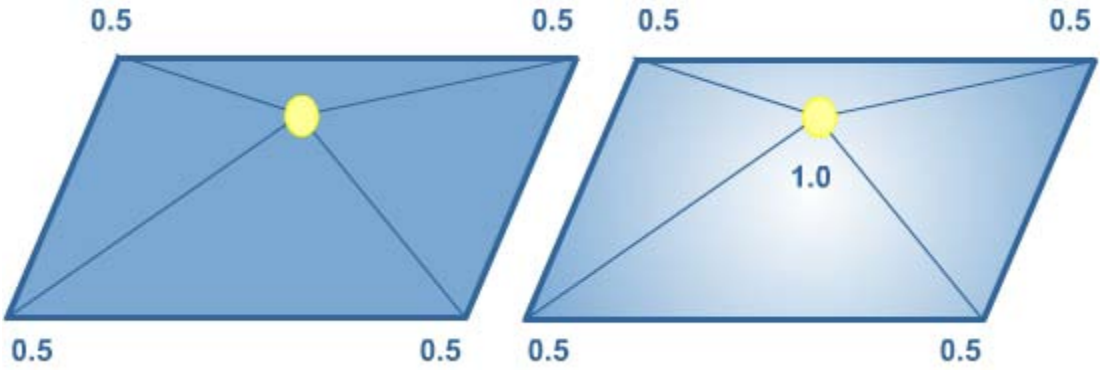
```
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }


        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

private void DrawCar(Matrix wMatrix, string technique)
{
    int i = 0;
    foreach (ModelMesh mesh in carModel.Meshes)
    {
        Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
            currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
projectionMatrix);
            currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
        }
        mesh.Draw();
    }
}


private void DrawLampPost(float scale, Vector3 translation, string technique)
{
    Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);

    foreach (ModelMesh mesh in lamppostModel.Meshes)
    {
        Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
```

```
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
            currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
projectionMatrix);
        }
        mesh.Draw();
    }
}



        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();


            SetUpVertices();
            SetUpCamera();
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            effect.CurrentTechnique = effect.Techniques["SimpleTexture"];
            effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
            effect.Parameters["xTexture"].SetValue(streetTexture);
```

```
            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
            DrawCar(car1Matrix, "SimpleTexture");

            DrawCar(car2Matrix, "SimpleTexture");

            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");

            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");


            base.Draw(gameTime);
        }
    }
}
```

Per-pixel Lighting

In this section we will create a point light. A point light describes a source of light that emanates in every direction from a point in 3D World space. Every object is lit by an amount of light that depends upon the angle between the object's normal and the direction of the light. As you may recall, this angle can be determined by taking the dot product between each of the object's normals and the direction of the incoming light. Now that we have normal data included in our vertex stream, we can implement a point light in our HLSL shader code. Before we do that however, let's consider how (and where) we should perform the required computation. The figure below depicts two quads in World space, each lit by a point light. Imagine that the light is exactly above the centre of the each quad.



The left quad represents the situation where the dot product for every vertex is calculated in the vertex shader. In this case, the angle between the direction (the thin blue lines) of the light and the plane is the same in every vertex. For example, if the dot product between this light direction and the normal is 0.5 for all four vertices, 0.5 would be the output of the vertex shader for each of the four vertices. Now the interpolator comes into play. For each pixel of the left quad, the dot product value sent to the pixel shader is interpolated from the dot product values of each of the four vertices, which in this case, is 0.5 for each pixel. This will result in every pixel in our quad being lit the same way, which is wrong.

The right quad illustrates what should happen. For each pixel, the dot product has to be calculated separately. For example, if the corner points again all have the value 0.5, the pixel exactly below the light will get a value of 1.0 (since the direction of the normal is exactly along the direction of the light) and all off the pixels in between will have some value between 0.5 and 1. This is the reason that we have to perform the dot product in the pixel shader.

Let's begin by creating a method in our shader code that calculates the dot product, given it the 3D position of the light, the 3D position of the pixel and the normal in that pixel. This method will be called by all of our pixel shader methods, so put it before the Techniques, but after the Texture Samplers.

```
// Subroutines

float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
    return dot(-lightDir, normal);
}
```

First the normalized direction of the light towards the pixel is calculated, this is the vector between the 3D position of the light and the 3D position of the pixel. Then we calculate the dot product between this light direction and the normal in the pixel, which is what the method returns. Note that the direction of the light needs to be negated, as this direction and the normal have opposite directions.

We will call the DotProduct from within our pixel shaders, as discussed above. Since this method requires the 3D position as well as the normal to be available to the pixel shader, we need to update our VertexToPixel structure in both SimpleTexture and SimpleNormal:

In SimpleTexture:

```
struct VertexToPixel
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};
```

In SimpleNormal:

```
struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
};
```

We are using the TEXCOORDn semantics to pass values from the vertex shader to the pixel shader. Now we need to change our vertex shader methods so they actually use the normal data we are sending in the vertex stream:

In SimpleTexture:

```
VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

       Output.Position = mul(inPos, xWorldViewProjection);
       Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}
```

In SimpleNormal:

```
SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

       Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}
```

Computation of the normal might benefit from a little explanation. Think of our meshes: they contain normal data. However, before we actually draw the meshes, we have rotated, scaled and translated them. This means that we also need to rotate the normals. We don't want to scale the normals (since they should still have a length of one), and we definitely don't want to translate them (because if we translate a normal over ten units to the left, all of the normals of the model will points to the left; not at all what we want), but if we rotate the mesh, we want the normals to rotate with it. However, since the length of a normal is one, a rotated normal will also be of length one.

Multiplying the normal by the 3x3 World matrix (containing only rotation data) will give us the rotated version of the normal. In order to get the 4x4 World matrix (which contains rotation, translation and scaling information) to only contain rotation information, we cast the 4x4 matrix to a 3x3 matrix. For reasons explained in class (related to our discussion of homogenous transformation matrices), this operation will remove the translation information. We then normalize, to ensure that the normal length remains equal to one.

The 3D position of the vertices are passed straight on to the interpolator, which interpolates the 3D position for each pixel. These position need to be transformed by the World matrix, in order to place each model in the correct location and orientation in world space.

We currently only pass the WorldViewProjection matrix to our effect. We also need to pass the World matrix, so add this global constant to our effect file.

```
float4x4 xWorld;
```

Before we move to the pixel shader, let's declare three more global constants that we will need. The first holds the position of the light, the second one allows the XNA application to set the strength of the light and the third one will set the ambient light, present in each and every pixel of our 3D scene. Since these variables are the same for all vertices and pixel of one frame, they should be specified by XNA through XNA-to-HLSL variables:

```
float3 xLightPos;
float xLightPower;
float xAmbient;
```

Now we can update our pixel shaders, as follows:

SimpleTexture:

```
PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       PSIn.TexCoords.y--;
```

```
        float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
        Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}
```

SimpleNormal:

```
SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

        float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
        diffuseLightingFactor = saturate(diffuseLightingFactor);
        diffuseLightingFactor *= xLightPower;

        float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
        Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}
```

The second line in each pixel shader calls our DotProduct method for the current pixel. As a result, we obtain a value that indicates the amount of light that is "caught", and thus reflected, by the current pixel. Because the result of a dot product is always within the [-1, 1] range (if both vectors are normalized), we need to map it to the [0,1] range using the saturate HLSL intrinsic function. Next, we multiply the value by the strength of the light. Now we find the base color. In the case of the street and car, this color has to be sampled from the texture (in SimpleTexture). In the case of the lamp post, this value is a dark gray hard coded into the SimpleNormal pixel shader. Once we know the base color, we multiply it by the combination of the light that will be different in each pixel, and the ambient light that is present in our entire scene, and output this value as the color for that pixel.

If you run your code at this point (do this) you should get no errors, but your screen will look disappointingly black, because we still have to set all the XNA-to-HLSL lighting variables from within XNA. Let's do this now.

Back in our C# XNA code, add three instance variables to our code:

```
Vector3 lightPos;
float lightPower;
float ambientPower;
```

Next, add a small method that sets these values:

```
 private void UpdateLightData()
        {
            lightPos = new Vector3(-10, 4, -2);
            lightPower = 2.2f;
            ambientPower = 0.2f;
        }
```

And call this method from our Update method:

```
UpdateLightData();
```

Although this method currently sets three values that remain constant, we can change them change every frame, should we later, for example, make the second car (and thus its lights) move . We need to pass these values, together with the World matrix to our effect file. Add the following code to our Draw method:

```
effect.Parameters["xWorld"].SetValue(Matrix.Identity);
effect.Parameters["xLightPos"].SetValue(lightPos);
effect.Parameters["xLightPower"].SetValue(lightPower);
effect.Parameters["xAmbient"].SetValue(ambientPower);
```

And, in the DrawCar and DrawLampPost methods, add the following code:

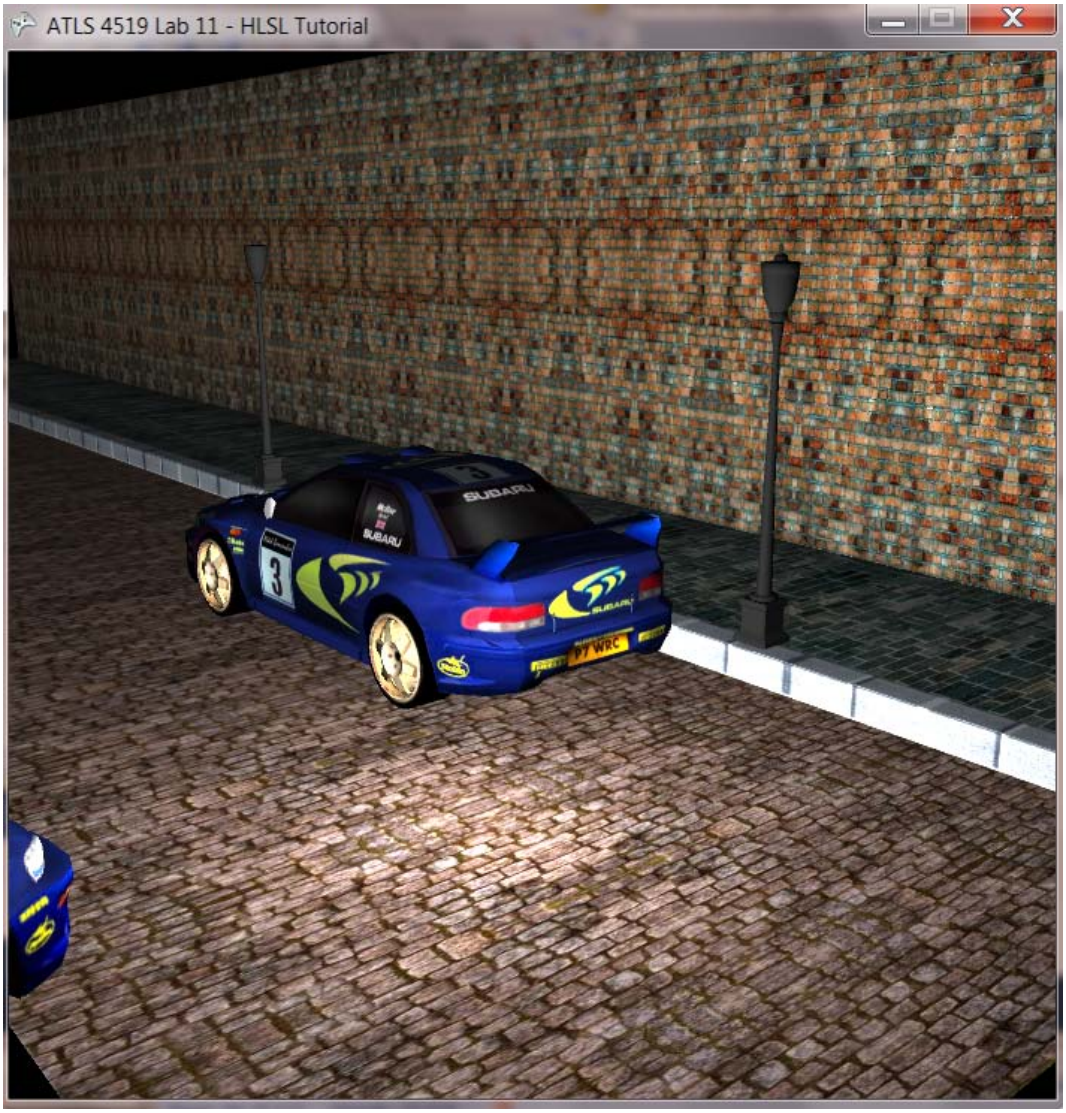
```
currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
currentEffect.Parameters["xLightPos"].SetValue(lightPos);
currentEffect.Parameters["xLightPower"].SetValue(lightPower);
currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
```

Run the code at this point. You should see the lighted image below. You can see clearly where the light is positioned. Note that we are not yet taking the distance into account; the only reason why the end of the wall gets less illumination is because the angle to the light is getting sharper.

Now that we have the basics of lighting, we will move on to shadowing.

Optional - test your knowledge:

- Play around with the position and strength of the light, as well as with the ambient light setting.



Here is our code so far:

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }


    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
```

```
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float lightPower;
        float ambientPower;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
```

```
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
        {
            lightPos = new Vector3(-10, 4, -2);
            lightPower = 2.2f;
            ambientPower = 0.2f;
        }


        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }


        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

private void DrawCar(Matrix wMatrix, string technique)
{
    int i = 0;
    foreach (ModelMesh mesh in carModel.Meshes)
    {
        Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
            currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
projectionMatrix);
            currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
            currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
            currentEffect.Parameters["xLightPos"].SetValue(lightPos);
            currentEffect.Parameters["xLightPower"].SetValue(lightPower);
            currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
```

```
        }
        mesh.Draw();
    }
}


private void DrawLampPost(float scale, Vector3 translation, string technique)
{
    Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);

    foreach (ModelMesh mesh in lamppostModel.Meshes)
    {
        Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
        foreach (Effect currentEffect in mesh.Effects)
        {
            currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
            currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
projectionMatrix);
            currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
            currentEffect.Parameters["xLightPos"].SetValue(lightPos);
            currentEffect.Parameters["xLightPower"].SetValue(lightPower);
            currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
        }
        mesh.Draw();
    }
}



        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();


            SetUpVertices();
            SetUpCamera();
        }

        protected override void UnloadContent()
        {
        }
```

```
        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            effect.CurrentTechnique = effect.Techniques["SimpleTexture"];
            effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
            effect.Parameters["xTexture"].SetValue(streetTexture);
            effect.Parameters["xWorld"].SetValue(Matrix.Identity);
            effect.Parameters["xLightPos"].SetValue(lightPos);
            effect.Parameters["xLightPower"].SetValue(lightPower);
            effect.Parameters["xAmbient"].SetValue(ambientPower);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
            DrawCar(car1Matrix, "SimpleTexture");

            DrawCar(car2Matrix, "SimpleTexture");

            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");

            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");


            base.Draw(gameTime);
        }
    }
}
```

Effect1.fx

```
// Global Inputs

float4x4 xWorldViewProjection;
float4x4 xWorld;
float3 xLightPos;
float xLightPower;
float xAmbient;


//Texture Samplers

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                                magfilter = LINEAR;
                                                minfilter = LINEAR;
                                                mipfilter = LINEAR;
```

```
                                                    AddressU = mirror;
                                                    AddressV = mirror;};

// Subroutines

float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
    return dot(-lightDir, normal);
}



// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}


PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       PSIn.TexCoords.y--;

       float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleTexture
{
```

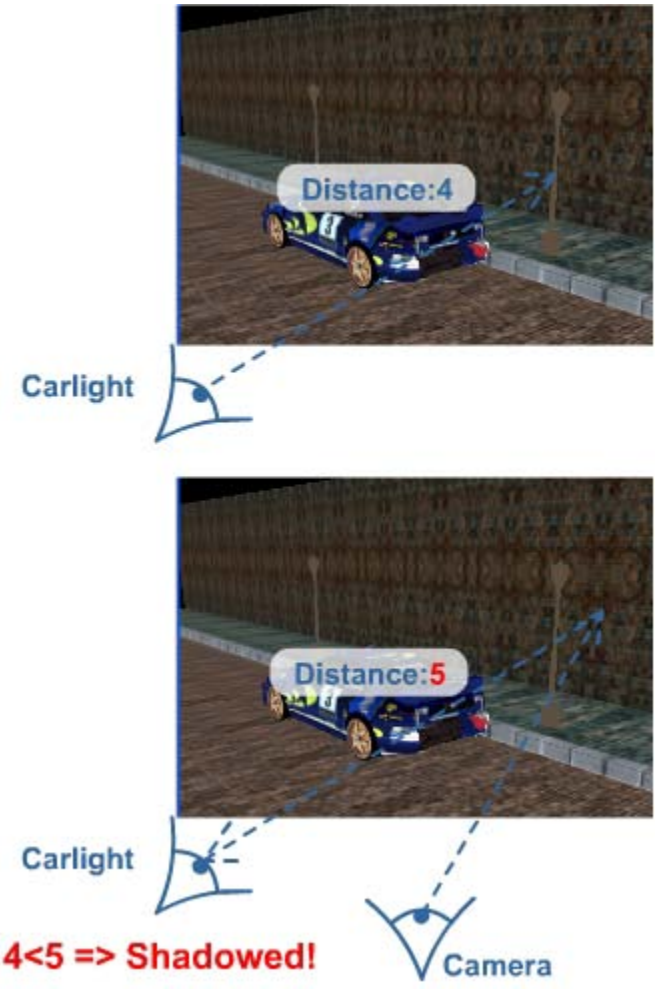
```
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}
```

Creating a Shadow Map

Creating shadows is a multi-step process. As we will see, the first step is creating what is called a shadow map. Before we do so, let's discuss what is going on. Consider the car at the bottom left corner of the screen. Its headlights are shining

light towards the right side of our scene. So the light hits the lampposts and the other car, which should cast shadows on the wall. The question is: how do we know which pixels on the wall should be shadowed? The answer is: we first draw the scene, as seen by the headlights. This means we have to move our camera perspective to the position of the headlights (for simplicity we assume that the headlights are a point source of light halfway between the two headlights on the car). Using this point of view, we are only interested in the distance of every pixel to the headlights. For example, the rightmost lamppost might be four meters away from the headlights, while the wall behind the lamppost (that we want to be in shadow) is 5.5 meters from the headlights. The pixels of the wall that are behind lamppost, as viewed from the perspective of the headlights, are in shadow, and are thus not seen by the headlights. So, at the location of these pixels on the wall, four meters should be stored as the distance to the headlights, not the actual distance from the headlights to the wall. We store this distance information for each pixel in the entire scene in what is called a "shadow map" or "depth map".

During the second phase, we draw the scene the usual way; that is, from the camera's point of view. Only this time, for each pixel we first calculate the distance from the pixel in question to the headlights, and compare this depth to the depth stored in the shadow map. For most pixels, both distances will be the same. The pixels that define our lamppost, for example, will still be four meters away from the headlights. However, when we calculate the distance for the pixels of the wall behind the lamppost, we will find that the distance to the headlights is 5.5 meters, but the value stored in the shadow map is four meters. For each pixel in which this occurs, we know that the pixel is in shadow, and thus should not be lit by the headlights. This situation is depicted in the two figures below.



Now that we understand how it works, we can create a shadow map. We will use a new HLSL technique to do this, called ShadowMap. The ShadowMap technique will have a pixel shader that renders the depth of the scene to an area of video memory called a "rendertarget." Let's append the new technique to the end of our effect file:

```
//------- Technique: ShadowMap --------
```

```
struct SMapVertexToPixel
{
    float4 Position     : POSITION;
    float4 Position2D    : TEXCOORD0;
};

struct SMapPixelToFrame
{
    float4 Color : COLOR0;
};

SMapVertexToPixel ShadowMapVertexShader( float4 inPos : POSITION)
{
    SMapVertexToPixel Output = (SMapVertexToPixel)0;

    Output.Position = mul(inPos, xLightsWorldViewProjection);
    Output.Position2D = Output.Position;

    return Output;
}

SMapPixelToFrame ShadowMapPixelShader(SMapVertexToPixel PSIn)
{
    SMapPixelToFrame Output = (SMapPixelToFrame)0;

    Output.Color = PSIn.Position2D.z/PSIn.Position2D.w;

    return Output;
}


technique ShadowMap
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowMapVertexShader();
        PixelShader = compile ps_2_0 ShadowMapPixelShader();
    }
}
```

The vertex shader for the ShadowMap technique is fairly simple. As we have done before, we supply the interpolator and pixel shader with the 2D screen coordinates. Since these screen coordinate also contains the distance between the vertex and the camera (or, in our case, the car lights), we need to pass this information to our pixel shader using one of the TEXCOORDn semantics. Notice that we are using the light's xLightsWorldViewProjection instead of the camera's xWorldViewProjection, because for the shadow map we need to look at the scene as seen by the headlights, instead of as by our camera. The WorldViewProjection matrix is the combination of three matrices (World, View and Projection matrix). The LightsWorldViewProjection matrix is also a combination of also three matrices: the same World matrix, but a View and Projection matrix specific to the car lights. This is a new matrix, so we need to declare it at the top of our HLSL code:

```
float4x4 xLightsWorldViewProjection;
```

Moving to the pixel shader, we again only have to calculate the color, however in this case the "color" is the result of the depth calculation, as follows: The X and Y coordinates of the PSIn.Position contain the X and Y values of the screen coordinate of the current pixel, but the Z coordinate contains the distance between the car lights "camera" and the pixel. However, this vector is the result of a multiplication of a vector and a 4x4 matrix, which happened in the vertex shader. The result of such a multiplication has four components: X,Y,Z and W. To be able to use the X,Y or Z components (and,

in particular, the Z component, we must first divide them by the W component. Math Alert: The W component is called the "homogeneous" component. Before dividing by W, the points X,Y,Z and W are called "homogenous coordinates." We use these as a convenience when working in a 3D space that can, at least in principle, stretch to infinity. A given point (x, y, z) in Euclidean space is identified by three ratios (X/W, Y/W, and Z/W), so the point (x, y, z) corresponds to the Vector4 (X, Y, Z, W) = (xW, yW, zW) where $Z \neq 0$. Such a vector is a set of homogeneous coordinates for the point (x, y, z). Note that, since ratios are used, multiplying the three homogeneous coordinates by a common, non-zero factor does not change the point represented. Thus, unlike Euclidean coordinates, a single point can be represented by infinitely many homogeneous coordinates. End of Math Alert

After dividing the Z component by the homogeneous component, the result will be between 0 and 1, where 0 corresponds to pixel at the near clipping plane (of the viewing frustum) and 1 to pixels at the far clipping plane of the viewing frustum. The viewing frustum was defined when we created the Projection matrix (using the CreatePerspectiveFieldOfView method).

That's it for the shader code; let's move on to the C# code, where we need to pass the required variables to our ShadowMap effect. Let's start with the xLightsWorldViewProjection variable. Because this depends on the position of the light, we need to update it in the UpdateLightData method. First we need to add a new instance variable:

```
Matrix lightsViewProjectionMatrix;
```

Now change the contents of the UpdateLightData method, where we change the position of our light to the front of the car, and lower its intensity:

```
 private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);
            lightPower = 1.0f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10),
                                    new Vector3(0, 1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
                                         10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;
        }
```

In order to draw the shadow map, we need to draw the scene as seen by the headlights. To do this, we need to create a ViewProjection matrix that looks at the scene from that perspective. This is almost identical to setting up these matrices for the camera, except:

- The aspect ratio of the projection matrix needs to equal the aspect ratio of our screen. In this case, we used 1f as aspect ratio, indicating a square viewport.
- The distance of the near and far clipping planes will correspond to black and white in our distance map.

We need to pass the lightsViewProjectionMatrix matrix to our graphics card before drawing the triangle strip, and before drawing our models. Add this line to our Draw method:

```
effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
              lightsViewProjectionMatrix);
```

Note that the ViewProjection matrix is multiplied with the World matrix (the Identity matrix in the case of the street) to obtain the WorldViewProjection matrix. Similarly, add this line to the DrawCar and DrawLampPost methods:

```
currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
                    lightsViewProjectionMatrix);
```

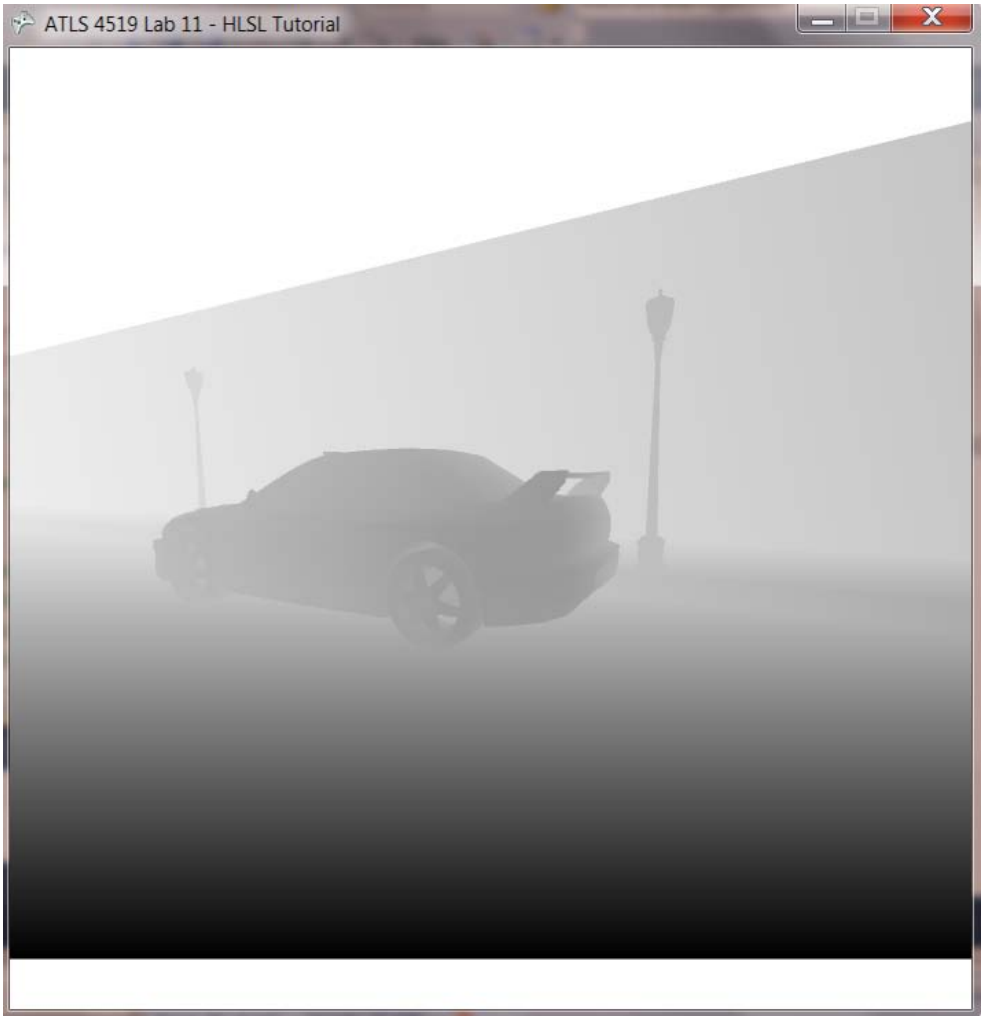
Now we still need to select the proper technique to render our scene. Since we will need to render our scene a second time with a later technique later on, now would be a nice time to put most of the contents of our Draw method in another method, DrawScene, as follows:

```
 private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
                    projectionMatrix);
            effect.Parameters["xTexture"].SetValue(streetTexture);
            effect.Parameters["xWorld"].SetValue(Matrix.Identity);
            effect.Parameters["xLightPos"].SetValue(lightPos);
            effect.Parameters["xLightPower"].SetValue(lightPower);
            effect.Parameters["xAmbient"].SetValue(ambientPower);
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
                    lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }
```

In order to actually draw the shadow map, we need to set the technique to ShadowMap in DrawScene, DrawCar and DrawLampPost. We do this in what is left of our Draw method, as follows (for now leave the second set of draws commented out):

```
protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            //device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            //DrawScene("SimpleTexture");
            //DrawCar(car1Matrix, "SimpleTexture");
            //DrawCar(car2Matrix, "SimpleTexture");
            //DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            //DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
```

Run the code at this point; you should see the image below. This image shows you what the shadow map actually looks like, as seen by the headlights of the car. The more white the pixel, the farther away it is from the headlights.



Here is our code so far.

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;
```

```
        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }


    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float lightPower;
        float ambientPower;

        Matrix lightsViewProjectionMatrix;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
```

```
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);
            lightPower = 1.0f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;
        }
```

```
        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }


        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

        private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix
* projectionMatrix);
                    currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                    currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                    currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                    currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                    currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }


        private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);
```

```
            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
                Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix
* projectionMatrix);
                    currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                    currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                    currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                    currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }

        private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
            effect.Parameters["xTexture"].SetValue(streetTexture);
            effect.Parameters["xWorld"].SetValue(Matrix.Identity);
            effect.Parameters["xLightPos"].SetValue(lightPos);
            effect.Parameters["xLightPower"].SetValue(lightPower);
            effect.Parameters["xAmbient"].SetValue(ambientPower);
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }


        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");

            // Load the cars
            LoadCar();
```

```
            //Load the Lamp Posts
            LoadLamp();


            SetUpVertices();
            SetUpCamera();
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            //device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            //DrawScene("SimpleTexture");
            //DrawCar(car1Matrix, "SimpleTexture");
            //DrawCar(car2Matrix, "SimpleTexture");
            //DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            //DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
    }

}
```

Effect1.fx

```
// Global Inputs

float4x4 xWorldViewProjection;
float4x4 xWorld;
float4x4 xLightsWorldViewProjection;
float3 xLightPos;
float xLightPower;
float xAmbient;


//Texture Samplers

Texture xTexture;
```

```
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = mirror;
                                                    AddressV = mirror;};

// Subroutines

float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
    return dot(-lightDir, normal);
}



// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}


PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       PSIn.TexCoords.y--;

       float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
```

```
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}

//------- Technique: ShadowMap --------

struct SMapVertexToPixel
```

```
{
    float4 Position     : POSITION;
    float4 Position2D    : TEXCOORD0;
};

struct SMapPixelToFrame
{
    float4 Color : COLOR0;
};

SMapVertexToPixel ShadowMapVertexShader( float4 inPos : POSITION)
{
    SMapVertexToPixel Output = (SMapVertexToPixel)0;

    Output.Position = mul(inPos, xLightsWorldViewProjection);
    Output.Position2D = Output.Position;

    return Output;
}

SMapPixelToFrame ShadowMapPixelShader(SMapVertexToPixel PSIn)
{
    SMapPixelToFrame Output = (SMapPixelToFrame)0;

    Output.Color = PSIn.Position2D.z/PSIn.Position2D.w;

    return Output;
}


technique ShadowMap
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowMapVertexShader();
        PixelShader = compile ps_2_0 ShadowMapPixelShader();
    }
}
```

Rendering the Shadow Map to a Texture

The second step of the shadow mapping algorithm requires comparing scene depth values with shadow map depth values. To do this, we need to preserve the shadow map that we have created as a texture, so we can use the data in this texture to do the required comparison. XNA provides a convenient way to do this, called "render targets." Instead of writing to the display, we instruct the graphics card to write to a portion of its memory set aside for this purpose (this is the render target). After rendering, we copy the render target to a texture, and we are good to go. Render targets have a large number of useful applications. For example, we used render targets in Lab 8 to enable the realistic implementation of water reflection and refraction.

We begin by creating instance variables for the render target and shadow map:

```
RenderTarget2D renderTarget;
Texture2D shadowMap;
```

Now we need to initialize the renderTarget, so add this code to the LoadContent method:

```
PresentationParameters pp = device.PresentationParameters;
```

```
renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
                    pp.BackBufferFormat, pp.DepthStencilFormat);
```

This code creates a renderTarget of exactly the same size and data format as our original target, the backbuffer to the screen. Now we can modify the Draw method to draw the shadow map to the renderTarget before drawing the scene normally. Add this line to the beginning of the Draw method:

```
device.SetRenderTarget(renderTarget);
```

In the next line, the renderTarget will get cleared to white, and our shadow map will be drawn to it. After we draw the shadow map, we need to store the contents of our custom render target into the texture. Before we can access its contents, however, we need to de-activate the custom render target. Add these lines after the shadow map draws to do both:

```
device.SetRenderTarget(null);
shadowMap = renderTarget;
```

Run your code at this point (to make sure there are no errors), but all you will see is a blank purple screen, since we never draw anything to the screen. We will correct this omission in the next section.

Our shader code has not changed; here is the C# code at this point:

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }


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

```
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float lightPower;
        float ambientPower;

        Matrix lightsViewProjectionMatrix;

        RenderTarget2D renderTarget;
        Texture2D shadowMap;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
```

```
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);
            lightPower = 1.0f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;
        }



        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
```

```
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }


        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

        private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix
* projectionMatrix);
                    currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                    currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                    currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                    currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                    currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }


        private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);

            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
                Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix
* projectionMatrix);
                    currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                    currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                    currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                    currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
```

```
        }

        private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
            effect.Parameters["xTexture"].SetValue(streetTexture);
            effect.Parameters["xWorld"].SetValue(Matrix.Identity);
            effect.Parameters["xLightPos"].SetValue(lightPos);
            effect.Parameters["xLightPower"].SetValue(lightPower);
            effect.Parameters["xAmbient"].SetValue(ambientPower);
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }


        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();

            SetUpVertices();
            SetUpCamera();

            PresentationParameters pp = device.PresentationParameters;
            renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
pp.BackBufferFormat, pp.DepthStencilFormat);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
```

```
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.SetRenderTarget(renderTarget);

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            device.SetRenderTarget(null);
            shadowMap = renderTarget;

            //device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            //DrawScene("SimpleTexture");
            //DrawCar(car1Matrix, "SimpleTexture");
            //DrawCar(car2Matrix, "SimpleTexture");
            //DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            //DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
    }

}
```

Drawing the Shadowed Scene

Now that we have a Shadow Map, we can draw the scene as seen by the "real" camera, and check whether the pixels in the scene should be shadowed or lit. As we have described, we will do this by sampling our shadow pap at each location.

We will therefore need to draw our scene using two different techniques: the first one will generate our shadow map, which we will store in a texture, the second one will draw the scene to the screen using the information in the shadow map to determine what pixels should be in shadow. For the second step we need to create a new shader technique, which we will call ShadowedScene. Here is that technique:

```
//------- Technique: ShadowedScene --------

struct SSceneVertexToPixel
{
    float4 Position             : POSITION;
    float4 Pos2DAsSeenByLight    : TEXCOORD0;
};

struct SScenePixelToFrame
{
    float4 Color : COLOR0;
};

SSceneVertexToPixel ShadowedSceneVertexShader( float4 inPos : POSITION)
```

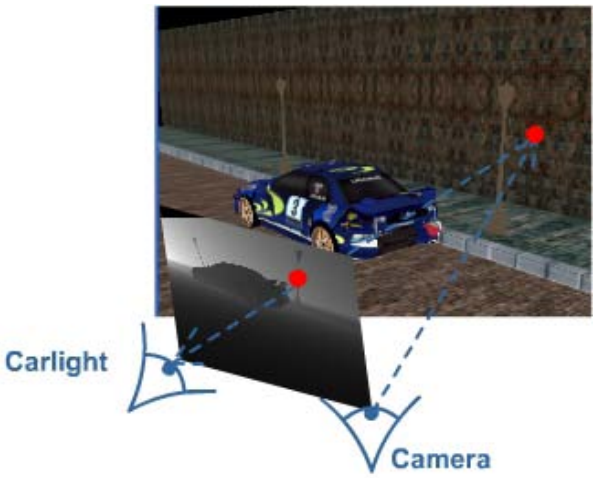
```
{
    SSceneVertexToPixel Output = (SSceneVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.Pos2DAsSeenByLight= mul(inPos, xLightsWorldViewProjection);
    return Output;
}

SScenePixelToFrame ShadowedScenePixelShader(SSceneVertexToPixel PSIn)
{
    SScenePixelToFrame Output = (SScenePixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords.x = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords.y = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

    Output.Color = tex2D(ShadowMapSampler, ProjectedTexCoords);

    return Output;
}

technique ShadowedScene
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowedSceneVertexShader();
        PixelShader = compile ps_2_0 ShadowedScenePixelShader();
    }
}
```

This code first defines new structs to hold the output data of the new vertex and pixel shaders. As you can see, the vertex shader will pass the (required) 2D position, as seen by the camera to the pixel shader, as well as the 2D position as seen by the car lights. From the latter, we can retrieve the position of the shadow map that corresponds to the current pixel that is being drawn. The pixel shader will only output the pixel's color.

We also need a texture sampler for the shadow map, so add one to Effect1.fx:

```
Texture xShadowMap;
sampler ShadowMapSampler = sampler_state { texture = <xShadowMap>;
                                           magfilter = LINEAR;
                                           minfilter = LINEAR;
                                           mipfilter = LINEAR;
                                           AddressU = clamp;
                                           AddressV = clamp;};
```

Looking at the vertex shader, recall that at this point we have already drawn the shadow map using its technique, and that our shadow map was drawn as seen by the car headlights. So now, for every pixel seen by the camera, we have to find which part of shadow map corresponds to this pixel. In other words, we need the 2D screen coordinates of the shadow map that correspond with every pixel being drawn. This idea is depicted below.



The red dot in the figure represents a pixel on the wall that is behind a lamppost, and therefore perhaps in shadow. We want to find the color of this pixel of the wall. To do this, we need to know the 2D coordinate of the corresponding pixel in our shadow map. Our first step is to project the 3D coordinates of the red-dotted pixel of the wall into the 2D camera space of the car lights. This is done using the same method we used for the shadowmap technique: by transforming the 3D positions with the WorldViewProjection matrix of the car lights. The resulting 2D coordinate is stored in Output.Pos2DAsSeenByLight. This gives our pixel shader access to the homogeneous screen coordinates (stored in PSIn.Pos2DAsSeenByLight) of each pixel's position, as seen by the car lights. Recall that to change the homogenous coordinates to X,Y and Z coordinates that we can use, we have to divide them by the W component. Because screen coordinates are 2D, we will not use the Z component.

When we divide the X and Y coordinates by W, we get 2D screen coordinates, with the X and Y component values in the [-1, 1] region. The coordinate (-1,-1) would represent the pixel in upper left corner, and coordinate (0,1) would represent the pixel in the middle of the right side of the screen. This is a problem, since texture coordinates have to be in the [0, 1] range. To fix this, we need to map the range [-1,1] to the range [0,1]. For example, point (-1,-1) has to become (0,0), and point (1,1) has to stay (1,1). A little thought should lead you to conclude that the following formulae are what we need:

$$X_{texture} = X_{screen}/2 + 0.5$$

$$Y_{texture} = Y_{screen}/2 + 0.5$$

This mapping is directly coded into our pixel shader. Having computed the correct coordinates, we sample the value of our shadow map at that position and set this as output color for the pixel (for now).

That's it for the HLSL code, so let's switch over to the XNA C# code. We have to draw our scene twice, once for the shadow map and once for everything else. Uncomment the second set of draw calls in the Draw method to accomplish this:

```
protected override void Draw(GameTime gameTime)
       {
           device.SetRenderTarget(renderTarget);

           device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

           DrawScene("ShadowMap");
           DrawCar(car1Matrix, "ShadowMap");
           DrawCar(car2Matrix, "ShadowMap");
           DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
           DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

           device.SetRenderTarget(null);
           shadowMap = renderTarget;
```

```
            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            DrawScene("ShadowedScene");
            DrawCar(car1Matrix, "ShadowedScene");
            DrawCar(car2Matrix, "ShadowedScene");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
```

First, the custom render target is activated and cleared. The scene is rendered into this target using the ShadowMap technique, as seen by the headlights of the car. This target is deactivated by re-activating the backbuffer for the screen and its contents is saved into the shadowMap texture. Next, the backbuffer for the screen is cleared and the whole scene is rendered again, this time using the ShadowedScene technique as seen by our camera.

If we were to run the code at this point, you will see a black screen, because we have not yet set the xShadowMap variable of the effects in DrawScene, DrawCar and DrawLampPost . At the same time, let's clean things up a bit so that we aren't sending data to the graphics card that is not going to be used (when we are drawing the shadow map. Make the following changes to DrawScene, DrawCar and DrawLampPost:

```
private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
                                  viewMatrix * projectionMatrix);
                        currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
                                  lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }


        private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);

            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
                Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
```

```
                if (technique != "ShadowMap")
                {
                    currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                    currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
                              viewMatrix * projectionMatrix);
                    currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                    currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                    currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                    currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                }
                currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
                              lightsViewProjectionMatrix);
            }
            mesh.Draw();
        }
    }

    private void DrawScene(string technique)
    {
        effect.CurrentTechnique = effect.Techniques[technique];
        if (technique != "ShadowMap")
        {
            effect.Parameters["xShadowMap"].SetValue(shadowMap);
            effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
                              projectionMatrix);
            effect.Parameters["xTexture"].SetValue(streetTexture);
            effect.Parameters["xWorld"].SetValue(Matrix.Identity);
            effect.Parameters["xLightPos"].SetValue(lightPos);
            effect.Parameters["xLightPower"].SetValue(lightPower);
            effect.Parameters["xAmbient"].SetValue(ambientPower);

        }
        effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
                              lightsViewProjectionMatrix);

        foreach (EffectPass pass in effect.CurrentTechnique.Passes)
        {
            pass.Apply();
            device.SetVertexBuffer(vertexBuffer);
            device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
        }
    }
```

Now when you run the code, you should see something like the image below. It looks like our normal scene, except for the colors: these are taken from the shadow map (except for the lamppost, where the color is hardcoded into the pixel shader).

Here is our code so far.

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
```

```
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }


    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float lightPower;
        float ambientPower;

        Matrix lightsViewProjectionMatrix;

        RenderTarget2D renderTarget;
        Texture2D shadowMap;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
```

```
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);
            lightPower = 2.0f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;
        }

        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
```

```
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }

        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

        private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
viewMatrix * projectionMatrix);
                        currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }


        private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);

            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
                Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
```

```
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
viewMatrix * projectionMatrix);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }

        private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            if (technique != "ShadowMap")
            {
                effect.Parameters["xShadowMap"].SetValue(shadowMap);
                effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
                effect.Parameters["xTexture"].SetValue(streetTexture);
                effect.Parameters["xWorld"].SetValue(Matrix.Identity);
                effect.Parameters["xLightPos"].SetValue(lightPos);
                effect.Parameters["xLightPower"].SetValue(lightPower);
                effect.Parameters["xAmbient"].SetValue(ambientPower);

            }
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }


        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");
```

```
            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();

            SetUpVertices();
            SetUpCamera();

            PresentationParameters pp = device.PresentationParameters;
            renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
pp.BackBufferFormat, pp.DepthStencilFormat);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.SetRenderTarget(renderTarget);

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            device.SetRenderTarget(null);
            shadowMap = renderTarget;

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            DrawScene("ShadowedScene");
            DrawCar(car1Matrix, "ShadowedScene");
            DrawCar(car2Matrix, "ShadowedScene");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
    }

}
```

Effect1.fx

```
// Global Inputs
```

```
float4x4 xWorldViewProjection;
float4x4 xWorld;
float4x4 xLightsWorldViewProjection;
float3 xLightPos;
float xLightPower;
float xAmbient;


//Texture Samplers

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = mirror;
                                                    AddressV = mirror;};

Texture xShadowMap;
sampler ShadowMapSampler = sampler_state {      texture = <xShadowMap>;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter=LINEAR;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};


// Subroutines

float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
    return dot(-lightDir, normal);
}



// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
```

```
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}


PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       PSIn.TexCoords.y--;

       float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
```

```
       diffuseLightingFactor *= xLightPower;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}

//------- Technique: ShadowMap --------

struct SMapVertexToPixel
{
    float4 Position     : POSITION;
    float4 Position2D    : TEXCOORD0;
};

struct SMapPixelToFrame
{
    float4 Color : COLOR0;
};

SMapVertexToPixel ShadowMapVertexShader( float4 inPos : POSITION)
{
    SMapVertexToPixel Output = (SMapVertexToPixel)0;

    Output.Position = mul(inPos, xLightsWorldViewProjection);
    Output.Position2D = Output.Position;

    return Output;
}

SMapPixelToFrame ShadowMapPixelShader(SMapVertexToPixel PSIn)
{
    SMapPixelToFrame Output = (SMapPixelToFrame)0;

    Output.Color = PSIn.Position2D.z/PSIn.Position2D.w;

    return Output;
}


technique ShadowMap
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowMapVertexShader();
        PixelShader = compile ps_2_0 ShadowMapPixelShader();
    }
}

//------- Technique: ShadowedScene --------

struct SSceneVertexToPixel
```

```
{
    float4 Position             : POSITION;
    float4 Pos2DAsSeenByLight    : TEXCOORD0;
};

struct SScenePixelToFrame
{
    float4 Color : COLOR0;
};

SSceneVertexToPixel ShadowedSceneVertexShader( float4 inPos : POSITION)
{
    SSceneVertexToPixel Output = (SSceneVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.Pos2DAsSeenByLight= mul(inPos, xLightsWorldViewProjection);
    return Output;
}

SScenePixelToFrame ShadowedScenePixelShader(SSceneVertexToPixel PSIn)
{
    SScenePixelToFrame Output = (SScenePixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

    Output.Color = tex2D(ShadowMapSampler, ProjectedTexCoords);

    return Output;
}


technique ShadowedScene
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowedSceneVertexShader();
        PixelShader = compile ps_2_0 ShadowedScenePixelShader();
    }
}
```

Drawing the Textured Scene with Shadowing

Now that we can sample the correct position in the shadow map for each pixel in our scene, we have all the ingredients we need to create our shadows. We will draw our scene with real colors and lighting like we did before, so our vertex shader will need to pass the texture coordinates, the normal and the 3D position of every vertex to the interpolator and pixel shader. We need to add these three variables to our SSceneVertexToPixel struct, as follows:

```
struct SSceneVertexToPixel
{
       float4 Position              : POSITION;
       float4 Pos2DAsSeenByLight    : TEXCOORD0;
       float2 TexCoords             : TEXCOORD1;
       float3 Normal                : TEXCOORD2;
       float4 Position3D            : TEXCOORD3;
};
```

We have already seen how to fill generate these values in the vertex shader, as follows:

```
SSceneVertexToPixel ShadowedSceneVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0,
float3 inNormal : NORMAL)
{
    SSceneVertexToPixel Output = (SSceneVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);
    Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
    Output.Position3D = mul(inPos, xWorld);
    Output.TexCoords = inTexCoords;

    return Output;
}
```

In the pixel shader, we need to compare the real distance between the pixel and the light against the value found in the shadow map. As before, when we created the shadow map, this distance can be found from the 2D position as seen by the light, by dividing the Z component by the homogeneous W component. Our pixel shader already has access to this 2D position stored in the Pos2DasSeenByLight variable.

Since we only want to draw the pixels that are being lit by the headlights, we must check if the pixels are in view of the headlights. In other words: if the projected x and y coordinates are within the [0, 1] range. For this, we can use the HLSL 'saturate' method, which clips any value to this range. So if the value after clipping is not the same as the value before clipping, we know the value was not in the [0, 1] range, and thus is not in the view of our headlights. Let's get our pixel shader in front of us to continue discussion of its operation:

```
SScenePixelToFrame ShadowedScenePixelShader(SSceneVertexToPixel PSIn)
{
    SScenePixelToFrame Output = (SScenePixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

    float diffuseLightingFactor = 0;
    if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y ==
                    ProjectedTexCoords.y))
    {
        float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
        float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
        if ((realDistance - 0.01f) <= depthStoredInShadowMap)
        {
            diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
            diffuseLightingFactor = saturate(diffuseLightingFactor);
            diffuseLightingFactor *= xLightPower;
        }
    }

    PSIn.TexCoords.y--;

    float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);

    Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}
```

After computing the coordinates from the perspective of the car lights in the same way as before, we initialize the impact of the light to zero. Then we check whether the current pixel is in sight of the light, as discussed above. If this is true, we change the impact of the light. If the pixel is not in the sight of the light (i.e., in shadow), the impact of the light will

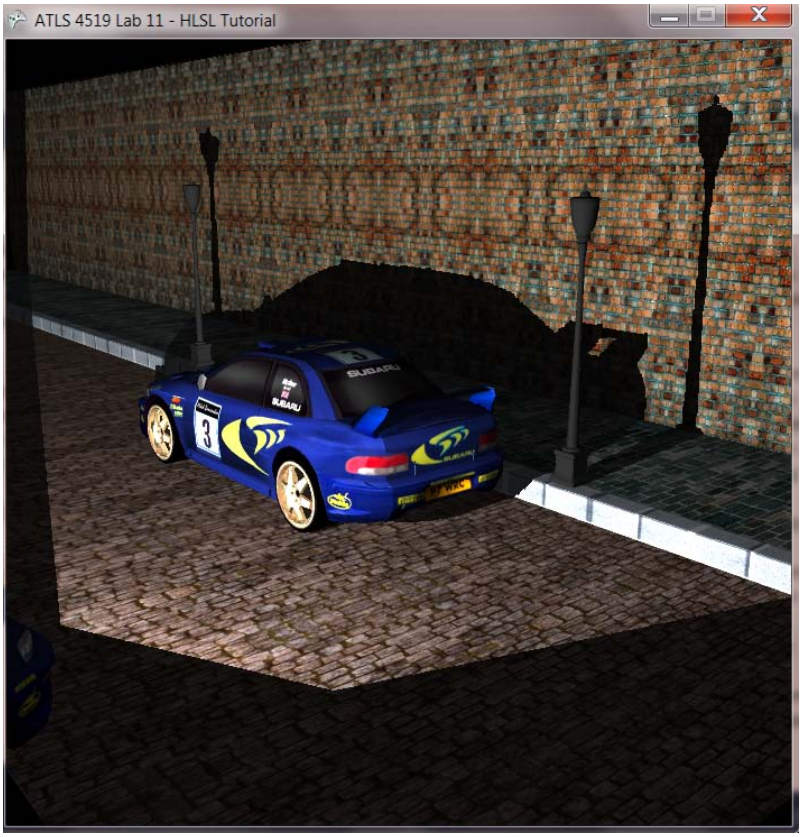
remain zero and the scene will be rendered with only the ambient light.

If the pixel is not in shadow (i.e., if the saturate test fails), it should be lit by the light. However, we first need to check that the pixel is not shadowed by some other object. To do this, we first retrieve the distance between pixel and light, as stored in the shadow map. Next, we calculate the real distance between the pixel and the light, which is done exactly as before. Now we check to ensure that the real distance is not larger than the value stored in the shadow map. This is where we perform a little sleight of hand, as follows.

Before we compare the realDistance with the depthStoredInShadowMap, we subtract a small "depth bias" of 0.01f. Here is why: we store depth information as a color in our shadow map. Each color component of the shadow map is stored in only eight bits, so the smallest difference between depths is 1/256 of the maximum depth. Suppose the maximum distance in our scene is 40.0f, so the smallest difference is 40/256 = 0.156f. This means that, in even the best case, all points with distances 0.156f to 0.234f will be stored as 0.156f. To compensate for this resolution problem, we need to subtract a small value of our real distance before performing the comparison (try removing this subtraction to see what can happen). Even if we were to increase the resolution of our depth buffer (this is pretty easy to do this, see below) we would still need a little bit of depth bias. It is possible to get fancy, for example we could have a depth bias that varied with slope (areas of high slope are where the resolution of the depth buffer matters most), but for our purposes, a constant depth bias is fine.

Moving on, if the real distance equals the distance stored in the shadow map, the pixel should be lit. This is done by changing the diffuseLightingFactor value, as we have done before: by calculating the dot-product-factor and the power of the light.

We do not need to make any changes to our C# XNA code. Running this code should give you something like the image below.



To increase the resolution of our depth buffer, make the following changes.

In LoadContent:

```
PresentationParameters pp = device.PresentationParameters;

renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
                    pp.BackBufferFormat, pp.DepthStencilFormat);
renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
                    SurfaceFormat.Single, DepthFormat.Depth24);
```

In Draw:

```
protected override void Draw(GameTime gameTime)
        {
            device.SamplerStates[0] = SamplerState.PointClamp;

            device.SetRenderTarget(renderTarget);
...

shadowMap = renderTarget;
// Reset 3D Defaults
GraphicsDevice.BlendState = BlendState.Opaque;
GraphicsDevice.DepthStencilState = DepthStencilState.Default;
GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;

...
```

In the shadow map texture sampler in Effect1.fx:

```
Texture xShadowMap;
sampler ShadowMapSampler = sampler_state {      texture = <xShadowMap>;
                                        magfilter = POINT;
                                        minfilter = POINT;
                                        mipfilter = POINT;
                                        AddressU = clamp;
                                        AddressV = clamp;};
```

After making these changes, you will not see a large difference from the current camera position, but if you move your camera closer to the shadows you should see a large improvement. You will also note that the left lamppost, which should be in the car's shadow, appears to stand out unnaturally. This is because we are drawing the lamppost using the SimpleNormal technique instead of the ShadowedScene technique, and the SimpleNormal technique doesn't know anything about shadow maps. There are two ways to fix this: (a) modify SimpleNormal to use the shadow map, or (b) modify ShadowedScene to be able to draw object that do not have texture information. We will choose Option (a), as follows:

```
//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
    float4 Pos2DAsSeenByLight    : TEXCOORD2;
};

struct SNPixelToFrame
```

```
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

       Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);
       Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       float2 ProjectedTexCoords;
       ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
       ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       float diffuseLightingFactor = 0;

       if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y
                   == ProjectedTexCoords.y))
       {
        float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
        float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
        if ((realDistance - 0.01f) <= depthStoredInShadowMap)
        {
                   diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
                   diffuseLightingFactor = saturate(diffuseLightingFactor);
                   diffuseLightingFactor *= xLightPower;
        }
       }
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}
```

Now when you run the code, the lamppost should be appropriately shadowed, although the car lights still have a squared-off, unnatural shape because we can see the corners of its view frustum. We will address this issue in the next section.

Optional - test your knowledge:

- Save the shadow map image to a file, and open this image in paint (or some other image editor), and add some gray areas. Load this image in XNA and look at the impact of your changes.
- Experiment with different depth bias values. What is the smallest value that appears to produce a good image?

Here is our code so far.

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }


    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;
```

```
        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float lightPower;
        float ambientPower;

        Matrix lightsViewProjectionMatrix;

        RenderTarget2D renderTarget;
        Texture2D shadowMap;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
```

```
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);
            lightPower = 2.0f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;
        }

        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }

        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
```

```
                meshPart.Effect = effect.Clone();
        }

        private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
viewMatrix * projectionMatrix);
                        currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }


        private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);

            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
                Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
viewMatrix * projectionMatrix);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }

        private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            if (technique != "ShadowMap")
```

```
            {
                effect.Parameters["xShadowMap"].SetValue(shadowMap);
                effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
                effect.Parameters["xTexture"].SetValue(streetTexture);
                effect.Parameters["xWorld"].SetValue(Matrix.Identity);
                effect.Parameters["xLightPos"].SetValue(lightPos);
                effect.Parameters["xLightPower"].SetValue(lightPower);
                effect.Parameters["xAmbient"].SetValue(ambientPower);

            }
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }


        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();

            SetUpVertices();
            SetUpCamera();

            PresentationParameters pp = device.PresentationParameters;
            //renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
pp.BackBufferFormat, pp.DepthStencilFormat);
            renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
SurfaceFormat.Single, DepthFormat.Depth24);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
```

```
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.SamplerStates[0] = SamplerState.PointClamp;

            device.SetRenderTarget(renderTarget);

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            device.SetRenderTarget(null);
            shadowMap = renderTarget;

            // Reset 3D Defaults
            GraphicsDevice.BlendState = BlendState.Opaque;
            GraphicsDevice.DepthStencilState = DepthStencilState.Default;
            GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            DrawScene("ShadowedScene");
            DrawCar(car1Matrix, "ShadowedScene");
            DrawCar(car2Matrix, "ShadowedScene");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
    }

}
```

Effect1.fx

```
// Global Inputs

float4x4 xWorldViewProjection;
float4x4 xWorld;
float4x4 xLightsWorldViewProjection;
float3 xLightPos;
float xLightPower;
float xAmbient;


//Texture Samplers

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
```

```
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = mirror;
                                                    AddressV = mirror;};

Texture xShadowMap;
sampler ShadowMapSampler = sampler_state {      texture = <xShadowMap>;
                                                    magfilter = POINT;
                                                    minfilter = POINT;
                                                    mipfilter = POINT;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};


// Subroutines

float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
    return dot(-lightDir, normal);
}



// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}


PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
```

```
       diffuseLightingFactor *= xLightPower;

       PSIn.TexCoords.y--;

       float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
    float4 Pos2DAsSeenByLight    : TEXCOORD2;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);
       Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       float diffuseLightingFactor = 0;

       if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y
                    == ProjectedTexCoords.y))
       {
                    float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
                    float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
```

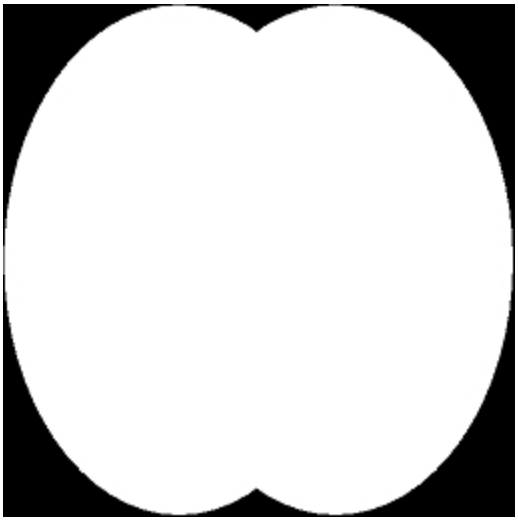
```
                    if ((realDistance - 0.01f) <= depthStoredInShadowMap)
                        {
                            diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D,
PSIn.Normal);
                            diffuseLightingFactor = saturate(diffuseLightingFactor);
                            diffuseLightingFactor *= xLightPower;
                        }
        }

        Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}

//------- Technique: ShadowMap --------

struct SMapVertexToPixel
{
    float4 Position     : POSITION;
    float4 Position2D    : TEXCOORD0;
};

struct SMapPixelToFrame
{
    float4 Color : COLOR0;
};

SMapVertexToPixel ShadowMapVertexShader( float4 inPos : POSITION)
{
    SMapVertexToPixel Output = (SMapVertexToPixel)0;

    Output.Position = mul(inPos, xLightsWorldViewProjection);
    Output.Position2D = Output.Position;

    return Output;
}

SMapPixelToFrame ShadowMapPixelShader(SMapVertexToPixel PSIn)
{
    SMapPixelToFrame Output = (SMapPixelToFrame)0;

    Output.Color = PSIn.Position2D.z/PSIn.Position2D.w;

    return Output;
}


technique ShadowMap
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowMapVertexShader();
        PixelShader = compile ps_2_0 ShadowMapPixelShader();
```

```
    }
}

//------- Technique: ShadowedScene --------

struct SSceneVertexToPixel
{
    float4 Position              : POSITION;
    float4 Pos2DAsSeenByLight    : TEXCOORD0;
    float2 TexCoords             : TEXCOORD1;
       float3 Normal                : TEXCOORD2;
       float4 Position3D            : TEXCOORD3;
};


struct SScenePixelToFrame
{
    float4 Color : COLOR0;
};

SSceneVertexToPixel ShadowedSceneVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0,
float3 inNormal : NORMAL)
{
    SSceneVertexToPixel Output = (SSceneVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);
    Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
    Output.Position3D = mul(inPos, xWorld);
    Output.TexCoords = inTexCoords;

    return Output;
}


SScenePixelToFrame ShadowedScenePixelShader(SSceneVertexToPixel PSIn)
{
    SScenePixelToFrame Output = (SScenePixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

    float diffuseLightingFactor = 0;
    if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y ==
ProjectedTexCoords.y))
    {
        float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
        float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
        if ((realDistance - 0.01f) <= depthStoredInShadowMap)
        {
            diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
            diffuseLightingFactor = saturate(diffuseLightingFactor);
            diffuseLightingFactor *= xLightPower;
        }
    }

       PSIn.TexCoords.y--;

    float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);

    Output.Color = baseColor*(diffuseLightingFactor + xAmbient);
```

```
    return Output;
}

technique ShadowedScene
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowedSceneVertexShader();
        PixelShader = compile ps_2_0 ShadowedScenePixelShader();
    }
}
```

Shaping the Light

Right now we have a square light casting shadows on our scene, but most real lights, and in particular car lights, do not project a square image. How can we make our car light projection look more like a real light? We could certainly find a set of equations that could represent what we want, but in this case, a much easier method is to use another texture to tell the pixel shader what to do. Consider the following image:



If we project this image across our scene from the perspective of the car lights, only the white part will be lit. If you have not already done so, add carlight.jpg to your content project.

Now we need to declare an instance variable to refer to our image:

```
Texture2D carLight;
```

And load the image file to LoadContent:

```
carLight = Content.Load<Texture2D>("carlight");
```

Finally, we need to pass the texture to our effect file. In DrawScene, add this line to the list of XNA-to-HLSL parameters we pass to the graphics device:

```
effect.Parameters["xCarLightTexture"].SetValue(carLight);
```

And add this line to DrawCar and DrawLampPost:

```
currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
```

Now we need to modify our shader code to make use of this information. First, add the texture sampler to Effect1.fx:

```
Texture xCarLightTexture;
sampler CarLightSampler = sampler_state { texture = <xCarLightTexture>;
                                          magfilter = LINEAR;
                                          minfilter = LINEAR;
                                          mipfilter = LINEAR;
                                          AddressU = clamp;
                                          AddressV = clamp;};
```

Now we need to sample the color value of this image, which will gives us a value between 0 and 1. We will then to multiply our final color by this value in the pixel shader, as follows:

```
{
diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
diffuseLightingFactor = saturate(diffuseLightingFactor);
diffuseLightingFactor *= xLightPower;

float lightTextureFactor = tex2D(CarLightSampler, ProjectedTexCoords).r;
diffuseLightingFactor *= lightTextureFactor;
}
```

Note that we do not need to modify the SimpleNormal shader code to achieve the desired effect.
Run the code at this point. You should see the image below.

Optional - test your knowledge:

- You can also use shaping images with other values than completely white/black. Grey pixels will be lit with less light than white pixels. Edit the carlight.jpg image to experiment with this effect.

Here is our code so far:

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }
```

```
        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }


    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float lightPower;
        float ambientPower;

        Matrix lightsViewProjectionMatrix;

        RenderTarget2D renderTarget;
        Texture2D shadowMap;
        Texture2D carLight;

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
```

```
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);
            lightPower = 2.0f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;
        }

        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
```

```
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }

        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

        private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
viewMatrix * projectionMatrix);
                        currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                        currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }


        private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);

            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
```

```
                Matrix worldMatrix = LampModelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
viewMatrix * projectionMatrix);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }

        private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            if (technique != "ShadowMap")
            {
                effect.Parameters["xShadowMap"].SetValue(shadowMap);
                effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
                effect.Parameters["xTexture"].SetValue(streetTexture);
                effect.Parameters["xWorld"].SetValue(Matrix.Identity);
                effect.Parameters["xLightPos"].SetValue(lightPos);
                effect.Parameters["xLightPower"].SetValue(lightPower);
                effect.Parameters["xAmbient"].SetValue(ambientPower);
                effect.Parameters["xCarLightTexture"].SetValue(carLight);

            }
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }


        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
```

```
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");
            carLight = Content.Load<Texture2D>("carlight");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();

            SetUpVertices();
            SetUpCamera();

            PresentationParameters pp = device.PresentationParameters;
            //renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
pp.BackBufferFormat, pp.DepthStencilFormat);
            renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
SurfaceFormat.Single, DepthFormat.Depth24);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.SamplerStates[0] = SamplerState.PointClamp;

            device.SetRenderTarget(renderTarget);

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            device.SetRenderTarget(null);
            shadowMap = renderTarget;

            // Reset 3D Defaults
            GraphicsDevice.BlendState = BlendState.Opaque;
            GraphicsDevice.DepthStencilState = DepthStencilState.Default;
            GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            DrawScene("ShadowedScene");
            DrawCar(car1Matrix, "ShadowedScene");
```

```
            DrawCar(car2Matrix, "ShadowedScene");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
    }

}
```

Effect1.fx

```
// Global Inputs

float4x4 xWorldViewProjection;
float4x4 xWorld;
float4x4 xLightsWorldViewProjection;
float3 xLightPos;
float xLightPower;
float xAmbient;

//Texture Samplers

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = mirror;
                                                    AddressV = mirror;};

Texture xShadowMap;
sampler ShadowMapSampler = sampler_state {      texture = <xShadowMap>;
                                                    magfilter = POINT;
                                                    minfilter = POINT;
                                                    mipfilter = POINT;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};

Texture xCarLightTexture;
sampler CarLightSampler = sampler_state {texture = <xCarLightTexture>;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};

// Subroutines

float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
    return dot(-lightDir, normal);
}



// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
```

```
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}

PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       PSIn.TexCoords.y--;

       float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
    float4 Pos2DAsSeenByLight    : TEXCOORD2;
};

struct SNPixelToFrame
{
```

```
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);
       Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       float diffuseLightingFactor = 0;

       if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y
                    == ProjectedTexCoords.y))
       {
                    float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
                    float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
                    if ((realDistance - 0.01f) <= depthStoredInShadowMap)
                           {
                                  diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D,
PSIn.Normal);
                                  diffuseLightingFactor = saturate(diffuseLightingFactor);
                                  diffuseLightingFactor *= xLightPower;
                           }
       }

       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}

//------- Technique: ShadowMap --------

struct SMapVertexToPixel
{
    float4 Position     : POSITION;
    float4 Position2D    : TEXCOORD0;
};
```

```
struct SMapPixelToFrame
{
    float4 Color : COLOR0;
};

SMapVertexToPixel ShadowMapVertexShader( float4 inPos : POSITION)
{
    SMapVertexToPixel Output = (SMapVertexToPixel)0;

    Output.Position = mul(inPos, xLightsWorldViewProjection);
    Output.Position2D = Output.Position;

    return Output;
}

SMapPixelToFrame ShadowMapPixelShader(SMapVertexToPixel PSIn)
{
    SMapPixelToFrame Output = (SMapPixelToFrame)0;

    Output.Color = PSIn.Position2D.z/PSIn.Position2D.w;

    return Output;
}


technique ShadowMap
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowMapVertexShader();
        PixelShader = compile ps_2_0 ShadowMapPixelShader();
    }
}

//------- Technique: ShadowedScene --------

struct SSceneVertexToPixel
{
    float4 Position              : POSITION;
    float4 Pos2DAsSeenByLight    : TEXCOORD0;
    float2 TexCoords             : TEXCOORD1;
       float3 Normal                : TEXCOORD2;
       float4 Position3D            : TEXCOORD3;
};

struct SScenePixelToFrame
{
    float4 Color : COLOR0;
};

SSceneVertexToPixel ShadowedSceneVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0,
float3 inNormal : NORMAL)
{
    SSceneVertexToPixel Output = (SSceneVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);
    Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
    Output.Position3D = mul(inPos, xWorld);
    Output.TexCoords = inTexCoords;

    return Output;
```

```
}

SScenePixelToFrame ShadowedScenePixelShader(SSceneVertexToPixel PSIn)
{
    SScenePixelToFrame Output = (SScenePixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

    float diffuseLightingFactor = 0;
    if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y ==
ProjectedTexCoords.y))
    {
        float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
        float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
        if ((realDistance - 0.01f) <= depthStoredInShadowMap)
        {
            diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
            diffuseLightingFactor = saturate(diffuseLightingFactor);
            diffuseLightingFactor *= xLightPower;

                    float lightTextureFactor = tex2D(CarLightSampler, ProjectedTexCoords).r;
                    diffuseLightingFactor *= lightTextureFactor;
        }
    }

       PSIn.TexCoords.y--;

    float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);

    Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique ShadowedScene
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowedSceneVertexShader();
        PixelShader = compile ps_2_0 ShadowedScenePixelShader();
    }
}
```

Emissive Lighting

In this section we are going to make the lamps more realistic. We will do this in three ways:

- we will add a glow around the light bulbs in the lamps that diminishes rapidly with distance;
- we will make the light emanating from the lampposts add illumination to the scene in proportion to distance; and
- we will add a bit of light to the shadowed image of the lamps.

There are several ways that we might accomplish these effects, most significantly by using billboards. However, since this lab is all about doing things with HLSL, we will add these effects in our shader code. Depending upon the effect that we are trying to achieve, we will modify either the ShadowedScene technique (to add light to everything but the lampposts), or the SimpleNormal technique (to add light to the lampposts themselves).

The majority of lighting techniques work as follows: we first calculate the 2D screen position of the light origin. Then we calculate the distance between the 2D screen position of the light and the 2D screen position of the current pixel. If this distance is smaller than a threshold value related to the particular technique, we add some white to the final color value for

that pixel. In some cases we can reduce the amount of computation by performing this comparison using untransformed world coordinates.

In order to incorporate our lighting techniques, our HLSL code needs to know the position of the lights, the position of the camera, and the camera's ViewProjection Matrix. Let's add the three globals in our shader code file to hold this information:

```
float4x4 xViewProjection;
float3 xCameraPos;
float3 xLamppostPos[4];
```

The xViewProjection matrix is required to transform 3D positions to 2D screen coordinates. We will need the position of the camera to calculate the desired radius of the lamps' glow (lights farther away from the camera will have smaller radii. We of course also need the 3D positions of both lampposts. Note that in this case we are passing an four-element array as an XNA-to-HLSL variable. The first to elements of this array will hold the world coordinates of the light bulbs in each lamp post. The second two elements will hold the world coordinates of the bulbs in the projected shadows of the lamp posts.

We need a few other globals to represent constants that we do not want to bury in our shader code. These will allow us to experiment with our lighting effects from the C# side of the XNA-to-HLSL boundary.

```
float xLightToCamRadius0;
float xLightToCamRadius1;
float xLampGlowFactor0;
float xLampGlowFactor1;
float xLightAttenuation;
float xLightFallOff;
```

We first pass the actual distance from the camera to each light (we will compute these distances on the C# side of things to reduce redundant computations in the shader code). The last four parameters will allow us to characterize the lighting in interesting ways, as we will see in a bit.

Now to the lighting. We will first we will add a glow around the light bulbs in the lamps that diminishes rapidly with distance. Add the following code to the ShadowedScene pixel shader:

```
...

    Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    // Lamp emissive light code (this handles the "aura" of both real and shadow lights)

    float4 screenPos = mul(PSIn.Position3D, xViewProjection);
    screenPos /= screenPos.w;

    for (int CurrentLight=0; CurrentLight<4; CurrentLight++)
       {
          float4 lightScreenPos = mul(float4(xLamppostPos[CurrentLight],1),xViewProjection);
          lightScreenPos /= lightScreenPos.w;

          float dist = distance(screenPos.xy, lightScreenPos.xy);
          float radius = 5.0f/distance(xCameraPos, xLamppostPos[CurrentLight]);
          if (dist < radius)
               {
                     Output.Color.rgb += (radius-dist)*4.0f;
               }
          }

    return Output;
}
```

We first calculate the 2D screen pos of the current pixel, as we have done before: by multiplying the 3D position with the WorldViewProjection matrix of the camera, and then dividing the result by its homogeneous coordinate, leaving the X and Y coordinate in the [-1,1] region. We are multiply by the ViewProjection matrix instead of the WorldViewProjection matrix because PSIn.Position3D was created in the vertex shader by multiplying the 3D position in the vertices by the World matrix. Thus, the resulting screenPos value is the original 3D position, transformed by the World matrix, and then by the ViewProjection matrix. Next, we determine the screen positions of the two lights, and the two "shadow lights." The for loop allows to iterate through each of the four elements of the xLamppostPos array. In each pass of the for loop we perform the following computation:

1. We will calculate the 2D screen coordinates of the light or shadow light by again multiplying by the ViewProjection matrix. In this case, the position of the lights, although not transformed by the World matrix in the vertex shader, (since it comes directly from our C# XNA code), have been provided in World coordinates. Since the coordinates are already in World space, we do not have to transform them using the World matrix.

2. Now that we know both the screen coordinate of the current pixel and of the lamppost (both in the [-1,1] region), we calculate the distance between them (on the screen).

3. Then we define a maximum radius for the light's glow. We want this radius to get smaller as the distance between the camera and lamppost gets larger. We perform this calculation in world space by dividing the distance just calculated into a constant that will need to be changed if the world space orientation changes. We can also adjust this constant (now set at 5.0) to make the glow larger or smaller.

4. With the radius defined, we check if the current pixel is within the threshold distance to lamppost. If so, we add some white to the pixel color. The closer to the lamppost, the more white we add.

Now we will make the light emanating from the lampposts add illumination to the scene in proportion to distance. This will also be done in the ShadedScene pixel shader, as follows:

```
...

        Output.Color = baseColor * (diffuseLightingFactor + xAmbient);

        //Lamp point light code - we do this for both lamps

        float3 lightDir0 = normalize(xLamppostPos[0] - PSIn.Position3D);
        float3 lightDir1 = normalize(xLamppostPos[1] - PSIn.Position3D);
        float diffuse0 = saturate(dot(normalize(PSIn.Normal), lightDir0));
        float diffuse1 = saturate(dot(normalize(PSIn.Normal), lightDir1));

        float d0 = distance(xLamppostPos[0], PSIn.Position3D);
        float d1 = distance(xLamppostPos[1], PSIn.Position3D);

        float att0 = 1 - pow(clamp(d0 / xLightAttenuation, 0, 1), xLightFallOff);
        float att1 = 1 - pow(clamp(d1 / xLightAttenuation, 0, 1), xLightFallOff);

        Output.Color +=  (diffuse0 * att0) + (diffuse1 * att1);

        // Lamp emissive light code (this handles the "aura" of both real and shadow lights)
...
```

This technique is called "point lighting" because the image is made to appear as if the light source is a point. In this case, our lamps are located at two points in the scene. Although the car lights are much brighter than the lamp post light, they should add some light to make the scene realistic.

For each pixel in the scene, we proceed as follows:

1. We first determine the direction of the light relative to the pixel, and normalize this distance.

2. We then determine the diffuse (basic) impact of the light on this pixel by taking the dot product of the pixel normal value and the light direction (as we have done before many times).

3. We then use the saturate HLSL intrinsic function to clamp the lighting value to be added to this pixel in the range [0,1].

4. We then compute the distance between each light and the current pixel.

5. The we compute the rate at which each light will fall off with respect to distance using the formula:

   $$K_{att} = 1 - (d/a)^f$$

   In this equation, K_{att} is the brightness factor which we will use as a multiplier for the existing pixel value, d is the distance between the pixel and the light source, a is the distance at which the light should stop affecting objects, and f is the falloff exponent that determines the shape of the exponential falloff curve. Both `a` and `f` are passed in from the C# XNA side to allow us to vary the lighting effect from XNA.

6. Finally, we add the diffuse and attention factors for each light to the scene.

Our last modifications to `Effect1.fx` will be in the `SimpleNormal` pixel shader. Here we will add emissive lighting that affects the lamppost mesh itself (and its shadow) for both lampposts drawn in the scene. Here is the modified code:

```
...

   float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color

   Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

   // Add some point emissive lighting for the lamps
   // No reason not to do the computation in world space

   float dist0 = distance(PSIn.Position3D, xLamppostPos[0]);
   float dist1 = distance(PSIn.Position3D, xLamppostPos[1]);

   //We want a different level of emissive light for each lamp
   if (dist0 < 1)
   {
       Output.Color.rgb += (xLightToCamRadius0 * xLampGlowFactor0 * (1-dist0));
   }

   if (dist1 < 1)
   {
       Output.Color.rgb += (xLightToCamRadius1 * xLampGlowFactor1 * (1-dist1));
   }

   return Output;

...
```

All of the calculations in this code are performed using world coordinates. For each pixel, we perform the following computations:

1. We first compute the distance between the current pixel and each lamp.

2. For each lamp, if this value is less than 1, we add a bit of white to the current pixel color using the following formula:

```
Output.Color.rgb += (xLightToCamRadius0 * xLampGlowFactor0 * (1-dist0));
```

The distance from the camera to each light is computed on the XNA side and passed into our shader code. This value is multiplied times the relevant LampGlowFactor times one minus the relevant distance between the current pixel and each lamp. This causes the light to fall off with distance linearly.

Finally, we need to upgrade the version of the pixel shader compiler to version 3.0, as follows:

```
technique ShadowedScene
{
    pass Pass0
    {
        VertexShader = compile vs_3_0 ShadowedSceneVertexShader();
        PixelShader = compile ps_3_0 ShadowedScenePixelShader();
    }
}
```

The upgrade is required because we have made the ShadowedScene pixel shader too complex for the Version 2 HLSL compiler.

That's it for our HLSL code. Now we need to modify our C# XNA code to take advantage of the new features of our shader code.

We need a new array instance variable to hold the light positions, as follows:

```
Vector3[] lamppostPositions = new Vector3[4];
```

And we need to set these positions in our UpdateLightData method:

```
//These are the "real" lights
lamppostPositions[0] = new Vector3(3f, 11.5f, -35f);
lamppostPositions[1] = new Vector3(3f, 11.5f, -5f);

//These are the shadows of the real lights
lamppostPositions[2] = new Vector3(11f, 14.5f, -5f);
lamppostPositions[3] = new Vector3(4.5f, 14.0f, -35f);
```

Now we need to set the XNA-to-HLSL parameters in DrawScene:

```
effect.Parameters["xCameraPos"].SetValue(cameraPos);
effect.Parameters["xViewProjection"].SetValue(viewMatrix * projectionMatrix);
effect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
effect.Parameters["xLightAttenuation"].SetValue(4000);
effect.Parameters["xLightFallOff"].SetValue(0.027f);
```

And in DrawCar:

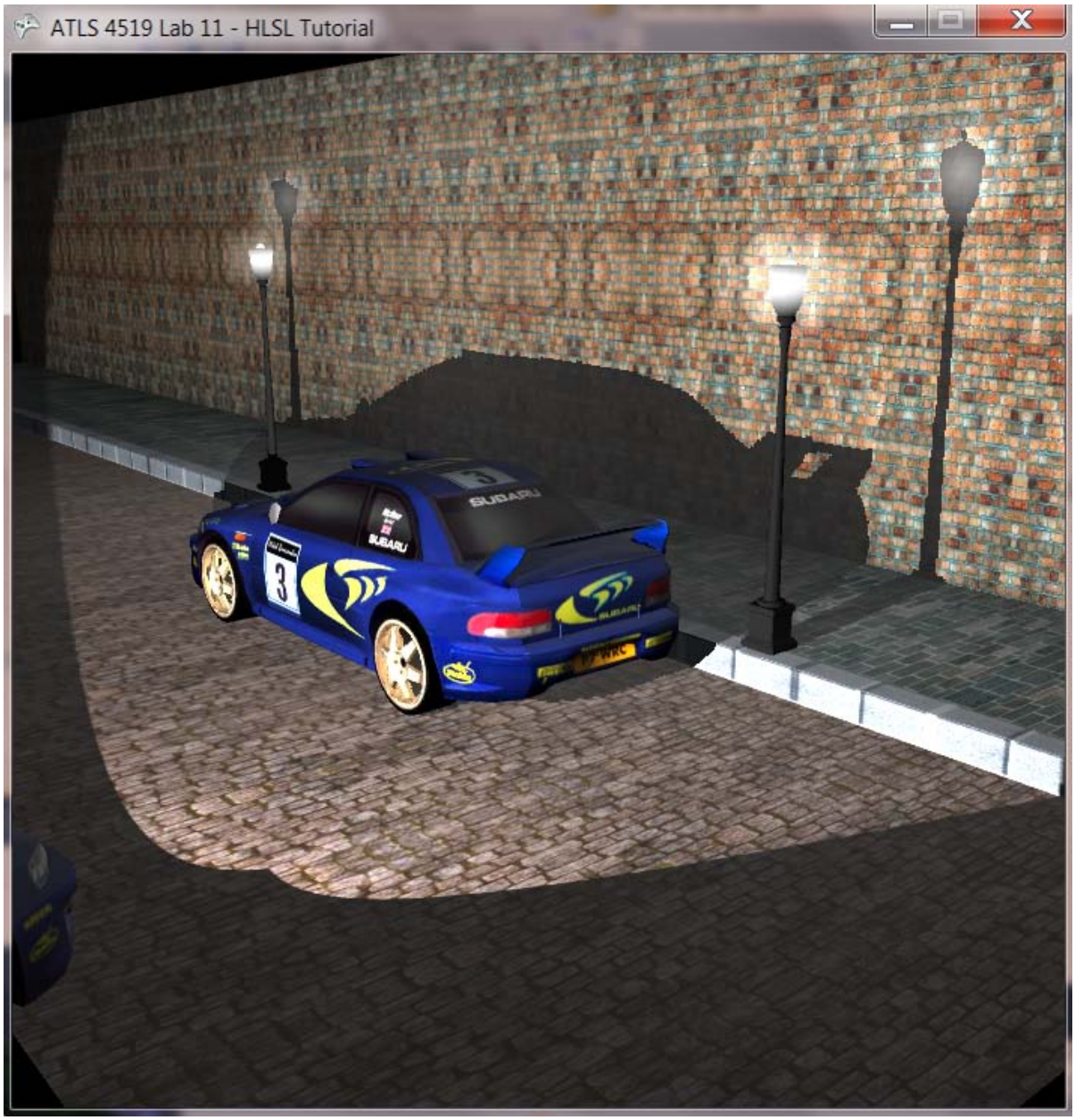
```
currentEffect.Parameters["xCameraPos"].SetValue(cameraPos);
currentEffect.Parameters["xViewProjection"].SetValue(viewMatrix * projectionMatrix);
currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
currentEffect.Parameters["xLightAttenuation"].SetValue(4000);
currentEffect.Parameters["xLightFallOff"].SetValue(0.027f);
```

We have several changes to make in DrawLampPost, so to avoid confusion, here is the entire modified method, with additions shown in yellow:

```
private void DrawLampPost(float scale, Vector3 translation, string technique)
```

```
{
        float camrad0 = 0;
        float camrad1 = 0;

        Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);
        Matrix[] modelTransforms = new Matrix[lamppostModel.Bones.Count];
        lamppostModel.CopyAbsoluteBoneTransformsTo(modelTransforms);
        if (technique != "ShadowMap")
                {
                // Compute the distance between each lamp and the camera
                Vector3 camdist0 = cameraPos - lamppostPositions[0];
                camrad0 = camdist0.Length();
                Vector3 camdist1 = cameraPos - lamppostPositions[1];
                camrad1 = camdist1.Length();
                }

       foreach (ModelMesh mesh in lamppostModel.Meshes)
       {
           Matrix worldMatrix = modelTransforms[mesh.ParentBone.Index] * lampMatrix;
           foreach (Effect currentEffect in mesh.Effects)
           {
               currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
               if (technique != "ShadowMap")
               {
                   currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
                      viewMatrix * projectionMatrix);
                   currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                   currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                   currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                   currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                   currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                   currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                   currentEffect.Parameters["xCameraPos"].SetValue(cameraPos);
                   currentEffect.Parameters["xViewProjection"].SetValue(viewMatrix *
                             projectionMatrix);
                   currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                   currentEffect.Parameters["xLightToCamRadius0"].SetValue(camrad0);
                   currentEffect.Parameters["xLightToCamRadius1"].SetValue(camrad1);
                   currentEffect.Parameters["xLampGlowFactor0"].SetValue(0.034f);
                   currentEffect.Parameters["xLampGlowFactor1"].SetValue(0.060f);
               }
               currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
                             lightsViewProjectionMatrix);
           }
          mesh.Draw();
       }
}
```

Run the code. You should see the final image below:



Experiment with the various constants that tune the lighting of the scene to understand how each effects the lighting, then tune the scene to your liking.

Here is our code at this point (with a little bit of clean-up):

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
```

```
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }


    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float lightPower;
        float ambientPower;

        Matrix lightsViewProjectionMatrix;

        RenderTarget2D renderTarget;
        Texture2D shadowMap;
        Texture2D carLight;

        Vector3[] lamppostPositions = new Vector3[4];

        public Game1()
```

```
        {
            graphics = new GraphicsDeviceManager(this);
            Content.RootDirectory = "Content";
        }

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
```

```
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);
            lightPower = 2.0f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;

            //These are the "real" lights
            lamppostPositions[0] = new Vector3(3f, 11.5f, -35f);
            lamppostPositions[1] = new Vector3(3f, 11.5f, -5f);

            //These are the shadows of the real lights
            lamppostPositions[2] = new Vector3(11f, 14.5f, -5f);
            lamppostPositions[3] = new Vector3(4.5f, 14.0f, -35f);

        }

        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }

        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

        private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
```

```
            {
                currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                if (technique != "ShadowMap")
                {
                    currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                    currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
viewMatrix * projectionMatrix);
                    currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                    currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                    currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                    currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                    currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                    currentEffect.Parameters["xCameraPos"].SetValue(cameraPos);
                    currentEffect.Parameters["xViewProjection"].SetValue(viewMatrix *
projectionMatrix);
                    currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                    currentEffect.Parameters["xLightAttenuation"].SetValue(4000);
                    currentEffect.Parameters["xLightFallOff"].SetValue(0.027f);
                }
                currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
            }
            mesh.Draw();
        }
    }


    private void DrawLampPost(float scale, Vector3 translation, string technique)
    {
        float camrad0 = 0;
        float camrad1 = 0;

        Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);
        Matrix[] modelTransforms = new Matrix[lamppostModel.Bones.Count];
        lamppostModel.CopyAbsoluteBoneTransformsTo(modelTransforms);
        if (technique != "ShadowMap")
        {
            // Compute the distance between each lamp and the camera
            Vector3 camdist0 = cameraPos - lamppostPositions[0];
            camrad0 = camdist0.Length();
            Vector3 camdist1 = cameraPos - lamppostPositions[1];
            camrad1 = camdist1.Length();
        }

        foreach (ModelMesh mesh in lamppostModel.Meshes)
        {
            Matrix worldMatrix = modelTransforms[mesh.ParentBone.Index] * lampMatrix;
            foreach (Effect currentEffect in mesh.Effects)
            {
                currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                if (technique != "ShadowMap")
                {
                    currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
viewMatrix * projectionMatrix);
                    currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                    currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                    currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                    currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                    currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                    currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                    currentEffect.Parameters["xCameraPos"].SetValue(cameraPos);
```

```
                        currentEffect.Parameters["xViewProjection"].SetValue(viewMatrix *
projectionMatrix);
                        currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                        currentEffect.Parameters["xLightToCamRadius0"].SetValue(camrad0);
                        currentEffect.Parameters["xLightToCamRadius1"].SetValue(camrad1);
                        currentEffect.Parameters["xLampGlowFactor0"].SetValue(0.034f);
                        currentEffect.Parameters["xLampGlowFactor1"].SetValue(0.060f);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }


        private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            if (technique != "ShadowMap")
            {
                effect.Parameters["xShadowMap"].SetValue(shadowMap);
                effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix *
projectionMatrix);
                effect.Parameters["xTexture"].SetValue(streetTexture);
                effect.Parameters["xWorld"].SetValue(Matrix.Identity);
                effect.Parameters["xLightPos"].SetValue(lightPos);
                effect.Parameters["xLightPower"].SetValue(lightPower);
                effect.Parameters["xAmbient"].SetValue(ambientPower);
                effect.Parameters["xCarLightTexture"].SetValue(carLight);
                effect.Parameters["xCameraPos"].SetValue(cameraPos);
                effect.Parameters["xViewProjection"].SetValue(viewMatrix * projectionMatrix);
                effect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                effect.Parameters["xLightAttenuation"].SetValue(4000);
                effect.Parameters["xLightFallOff"].SetValue(0.027f);
            }
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }


        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;
```

```
            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");
            carLight = Content.Load<Texture2D>("carlight");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();

            SetUpVertices();
            SetUpCamera();

            PresentationParameters pp = device.PresentationParameters;
            //renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
pp.BackBufferFormat, pp.DepthStencilFormat);
            renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
SurfaceFormat.Single, DepthFormat.Depth24);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.SamplerStates[0] = SamplerState.PointClamp;

            device.SetRenderTarget(renderTarget);

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            device.SetRenderTarget(null);
            shadowMap = renderTarget;

            // Reset 3D Defaults
            GraphicsDevice.BlendState = BlendState.Opaque;
            GraphicsDevice.DepthStencilState = DepthStencilState.Default;
            GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            DrawScene("ShadowedScene");
            DrawCar(car1Matrix, "ShadowedScene");
            DrawCar(car2Matrix, "ShadowedScene");
```

```
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
    }

}
```

Effect1.fx

```
// Global Inputs

float4x4 xWorldViewProjection;
float4x4 xWorld;
float4x4 xLightsWorldViewProjection;
float3 xLightPos;
float xLightPower;
float xAmbient;
float4x4 xViewProjection;
float3 xCameraPos;
float3 xLamppostPos[4];
float xLightToCamRadius0;
float xLightToCamRadius1;
float xLampGlowFactor0;
float xLampGlowFactor1;
float xLightAttenuation;
float xLightFallOff;



//Texture Samplers

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = mirror;
                                                    AddressV = mirror;};

Texture xShadowMap;
sampler ShadowMapSampler = sampler_state {      texture = <xShadowMap>;
                                                    magfilter = POINT;
                                                    minfilter = POINT;
                                                    mipfilter = POINT;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};

Texture xCarLightTexture;
sampler CarLightSampler = sampler_state {texture = <xCarLightTexture>;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};

// Subroutines

float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
```

```
    return dot(-lightDir, normal);
}



// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}

PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower;

       PSIn.TexCoords.y--;

       float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------
```

```
struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
    float4 Pos2DAsSeenByLight    : TEXCOORD2;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);
       Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       float diffuseLightingFactor = 0;

       if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y
                     == ProjectedTexCoords.y))
       {
                     float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
                     float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
                     if ((realDistance - 0.01f) <= depthStoredInShadowMap)
                            {
                                   diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D,
PSIn.Normal);
                                   diffuseLightingFactor = saturate(diffuseLightingFactor);
                                   diffuseLightingFactor *= xLightPower;
                            }
       }

       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

       // Add some point emissive lighting for the lamps
       // No reason not to do the computation in world space

       float dist0 = distance(PSIn.Position3D, xLamppostPos[0]);
       float dist1 = distance(PSIn.Position3D, xLamppostPos[1]);

       // We want a different level of emissive light for each lamp
       if (dist0 < 1)
              {
                     Output.Color.rgb += (xLightToCamRadius0 * xLampGlowFactor0 * (1-dist0));
```

```
            }

       if (dist1 < 1)
            {
                Output.Color.rgb += (xLightToCamRadius1 * xLampGlowFactor1 * (1-dist1));
            }

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}

//------- Technique: ShadowMap --------

struct SMapVertexToPixel
{
    float4 Position     : POSITION;
    float4 Position2D    : TEXCOORD0;
};

struct SMapPixelToFrame
{
    float4 Color : COLOR0;
};

SMapVertexToPixel ShadowMapVertexShader( float4 inPos : POSITION)
{
    SMapVertexToPixel Output = (SMapVertexToPixel)0;

    Output.Position = mul(inPos, xLightsWorldViewProjection);
    Output.Position2D = Output.Position;

    return Output;
}

SMapPixelToFrame ShadowMapPixelShader(SMapVertexToPixel PSIn)
{
    SMapPixelToFrame Output = (SMapPixelToFrame)0;

    Output.Color = PSIn.Position2D.z/PSIn.Position2D.w;

    return Output;
}


technique ShadowMap
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowMapVertexShader();
        PixelShader = compile ps_2_0 ShadowMapPixelShader();
    }
}

//------- Technique: ShadowedScene --------
```

```
struct SSceneVertexToPixel
{
    float4 Position              : POSITION;
    float4 Pos2DAsSeenByLight    : TEXCOORD0;
    float2 TexCoords             : TEXCOORD1;
       float3 Normal                : TEXCOORD2;
       float4 Position3D            : TEXCOORD3;
};

struct SScenePixelToFrame
{
    float4 Color : COLOR0;
};

SSceneVertexToPixel ShadowedSceneVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0,
float3 inNormal : NORMAL)
{
    SSceneVertexToPixel Output = (SSceneVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);
    Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
    Output.Position3D = mul(inPos, xWorld);
    Output.TexCoords = inTexCoords;

    return Output;
}

SScenePixelToFrame ShadowedScenePixelShader(SSceneVertexToPixel PSIn)
{
    SScenePixelToFrame Output = (SScenePixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

    float diffuseLightingFactor = 0;
    if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y ==
ProjectedTexCoords.y))
    {
        float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
        float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
        if ((realDistance - 0.01f) <= depthStoredInShadowMap)
        {
            diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
            diffuseLightingFactor = saturate(diffuseLightingFactor);
            diffuseLightingFactor *= xLightPower;

                    float lightTextureFactor = tex2D(CarLightSampler, ProjectedTexCoords).r;
                    diffuseLightingFactor *= lightTextureFactor;
        }
    }

       PSIn.TexCoords.y--;

    float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);

    Output.Color = baseColor * (diffuseLightingFactor + xAmbient);

       //Lamp point light code - we do this for both lamps
```

```
        float3 lightDir0 = normalize(xLamppostPos[0] - PSIn.Position3D);
        float3 lightDir1 = normalize(xLamppostPos[1] - PSIn.Position3D);
        float diffuse0 = saturate(dot(normalize(PSIn.Normal), lightDir0));
        float diffuse1 = saturate(dot(normalize(PSIn.Normal), lightDir1));

        float d0 = distance(xLamppostPos[0], PSIn.Position3D);
        float d1 = distance(xLamppostPos[1], PSIn.Position3D);

        float att0 = 1 - pow(clamp(d0 / xLightAttenuation, 0, 1), xLightFallOff);
        float att1 = 1 - pow(clamp(d1 / xLightAttenuation, 0, 1), xLightFallOff);

        Output.Color +=  (diffuse0 * att0) + (diffuse1 * att1);

    // Lamp emissive light code (this handles the "aura" of both real and shadow lights)

    float4 screenPos = mul(PSIn.Position3D, xViewProjection);
    screenPos /= screenPos.w;

    for (int CurrentLight=0; CurrentLight<4; CurrentLight++)
       {
         float4 lightScreenPos = mul(float4(xLamppostPos[CurrentLight],1),xViewProjection);
         lightScreenPos /= lightScreenPos.w;

               float dist = distance(screenPos.xy, lightScreenPos.xy);
         float radius = 5.0f/distance(xCameraPos, xLamppostPos[CurrentLight]);
         if (dist < radius)
              {
              Output.Color.rgb += (radius-dist)*4.0f;
              }
       }

    return Output;

}

technique ShadowedScene
{
    pass Pass0
    {
        VertexShader = compile vs_3_0 ShadowedSceneVertexShader();
        PixelShader = compile ps_3_0 ShadowedScenePixelShader();
    }
}
```

HLSL Optimization

HLSL optimization is sometimes unnecessary. If you are drawing at sixty frames per second, and plan no significant changes to your code, you probably do not need to optimize. But, of course, we always want to add some new feature to our game, so learning to think about improving shader performance is a useful exercise. There are several commercial tools that can help you optimize your shader code, most notably the free nVidia FX Composer, which can be downloaded here (you need not do this).

When we optimize HLSL code, we need to remember that:

- sending things to the graphics card is relatively slow, **but**
- the graphics card is faster than the CPU, **but**
- we execute the vertex shader for every vertex, **and**
- we execute the pixel shader for every pixel.

These factors interact in interesting ways to guide our thinking about optimization. For example, if there is a computation that is performed in the or vertex or pixel shaders that we can do once somewhere else, we need to find a way to move the computation, even if this requires us to sent more data to the graphics card. As we will see below, there are two ways to do this. Similarly, if we are passing redundant information to the graphics card, we should try to eliminate the redundancy, even if it moves a computation to the shader. This works because the HLSL compiler is smart enough to notice computations that only need to be performed once, and move these computations back to the CPU in what is called a "preshader." In this section, we will introduce a view optimizations to our code. They will not change the image quality of the final result, but they will cause the image to be rendered more efficiently. This approach is depicted in the following diagram:

Let's start by inspecting our shader code. The following computations (highlighted in blue) are unnecessarily performed for every pixel, when in fact they only need to be performed once:

In SimpleNormal:

```
Output.Color.rgb += (xLightToCamRadius0 * xLampGlowFactor0 * (1-dist0));
```

In ShadowedScene:

```
float4 lightScreenPos = mul(float4(xLamppostPos[CurrentLight],1),xViewProjection);
lightScreenPos /= lightScreenPos.w;

float dist = distance(screenPos.xy, lightScreenPos.xy);
float radius = 5.0f/distance(xCameraPos, xLamppostPos[CurrentLight]);
```

We will modify our C# code to perform these computations, and pass in the pre-computed values to our shader code. First, we need some new Game1 instance variables, as follow:

```
        float lightPower;
        float [] lightPower = new float [2];
        Vector4[] ltscrPositions = new Vector4[4];
        Matrix CameraViewProjection;
        Vector3[] camdists = new Vector3[4];
        float[] camrads = new float[4];
        float[] CamLtTimesGlow = new float[4];
        float[] LampGlowFactor = new float[4];
```

We are replacing the single lightPower value with a two-element array. This will allow us to have one value for pixels in full light, and another value for pixels in partial shadow. Now we need to modify the UpdateLightData method, as follows:

```
private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);
            lightPower = 2.0f;
            // using an array for light power; [0] is in light, [1] is in shadow
            lightPower [0] = 2.0f;
            lightPower [1] = 1.6f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
                1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
                10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;

            //These are the "real" lights
```

```
            lamppostPositions[0] = new Vector3(3f, 11.5f, -35f);
            lamppostPositions[1] = new Vector3(3f, 11.5f, -5f);

            //These are the shadows of the real lights
            lamppostPositions[2] = new Vector3(11f, 14.5f, -5f);
            lamppostPositions[3] = new Vector3(4.5f, 14.0f, -35f);

            //compute light screen positions
            CameraViewProjection = viewMatrix * projectionMatrix;
            for (int i = 0; i < 4; i++)
                {
                    ltscrPositions[i] = Vector4.Transform(lamppostPositions[i], CameraViewProjection);
                    ltscrPositions[i] /= ltscrPositions[i].W;
                    // Compute the distance between each lamp and the camera
                    camdists[i] = cameraPos - lamppostPositions[i];
                    camrads [i] = 5.0f / camdists[i].Length();
                    CamLtTimesGlow[i] = camdists[i].Length() * LampGlowFactor[i];
                }
        }
```

We first provide values for our new two-valued lightPower variable. The code at the end replaces the blue-highlighted pixel shader code above. Note the use of the Vector4.Transform method, which allow us to compute the screen position of each light (or shadow light) by applying the camera's ViewProjection transform. We also use the Length method, which computes the distance between two points in 3D space.

Now to use these values we make modifications as shown below.

In `Initialize`:

```
LampGlowFactor[0] = 0.034f;
LampGlowFactor[1] = 0.060f;
LampGlowFactor[2] = 0.0f;
LampGlowFactor[3] = 0.0f;
```

In `DrawCar`:

```
currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
//currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * viewMatrix *
projectionMatrix);
currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * CameraViewProjection);
currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
currentEffect.Parameters["xLightPos"].SetValue(lightPos);
currentEffect.Parameters["xLightPower"].SetValue(lightPower);
currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
//currentEffect.Parameters["xCameraPos"].SetValue(cameraPos);
//currentEffect.Parameters["xViewProjection"].SetValue(viewMatrix * projectionMatrix);
currentEffect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
currentEffect.Parameters["xLightAttenuation"].SetValue(4000);
currentEffect.Parameters["xLightFallOff"].SetValue(0.027f);
currentEffect.Parameters["xLtscrPositions"].SetValue(ltscrPositions);
currentEffect.Parameters["xCamRads"].SetValue(camrads);
```

In `DrawLampPost`:

```
private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
```

```
            float camrad0 = 0;
            float camrad1 = 0;

            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);
            Matrix[] modelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(modelTransforms);

            if (technique != "ShadowMap")
            {
                // Compute the distance between each lamp and the camera
                Vector3 camdist0 = cameraPos - lamppostPositions[0];
                camrad0 = camdist0.Length();
                Vector3 camdist1 = cameraPos - lamppostPositions[1];
                camrad1 = camdist1.Length();
            }

            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
                Matrix worldMatrix = modelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        //currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
                                    viewMatrix * projectionMatrix);
                     currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
                                CameraViewProjection);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                        //currentEffect.Parameters["xCameraPos"].SetValue(cameraPos);
                        //currentEffect.Parameters["xViewProjection"].SetValue(viewMatrix *
                                projectionMatrix);
                        currentEffect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
                        currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                        //currentEffect.Parameters["xLightToCamRadius0"].SetValue(camrad0);
                        //currentEffect.Parameters["xLightToCamRadius1"].SetValue(camrad1);
                        //currentEffect.Parameters["xLampGlowFactor0"].SetValue(0.034f);
                        //currentEffect.Parameters["xLampGlowFactor1"].SetValue(0.060f);
                        currentEffect.Parameters["xCamLtTimesGlow"].SetValue(CamLtTimesGlow);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
                                lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }
```

In DrawScene:

```
effect.Parameters["xShadowMap"].SetValue(shadowMap);
//effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * viewMatrix * projectionMatrix);
effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * CameraViewProjection);
effect.Parameters["xTexture"].SetValue(streetTexture);
effect.Parameters["xWorld"].SetValue(Matrix.Identity);
effect.Parameters["xLightPos"].SetValue(lightPos);
effect.Parameters["xLightPower"].SetValue(lightPower);
```

```
effect.Parameters["xAmbient"].SetValue(ambientPower);
effect.Parameters["xCarLightTexture"].SetValue(carLight);
//effect.Parameters["xCameraPos"].SetValue(cameraPos);
//effect.Parameters["xViewProjection"].SetValue(viewMatrix * projectionMatrix);
effect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
effect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
effect.Parameters["xLtscrPositions"].SetValue(ltscrPositions);
effect.Parameters["xCamRads"].SetValue(camrads);
effect.Parameters["xLightAttenuation"].SetValue(4000);
effect.Parameters["xLightFallOff"].SetValue(0.027f);
```

In our HLSL code:

```
// Global Inputs
float4x4 xWorldViewProjection;
float4x4 xWorld;
float4x4 xLightsWorldViewProjection;
float3 xLightPos;
//float xLightPower;
float xLightPower [2];
float xAmbient;
float4x4 xViewProjection;
//float3 xCameraPos;
float3 xLamppostPos[4];
//float xLightToCamRadius0;
//float xLightToCamRadius1;
//float xLampGlowFactor0;
//float xLampGlowFactor1;
float xLightAttenuation;
float xLightFallOff;
float xCamRads [4];
float xCamLtTimesGlow [4];
float4 xLtscrPositions [4];
```

In SimpleTexture:

```
diffuseLightingFactor *= xLightPower;
diffuseLightingFactor *= xLightPower [0];
```

In SimpleNormal:

```
...

            float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
            diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
            diffuseLightingFactor = saturate(diffuseLightingFactor);

            if ((realDistance - 0.01f) <= depthStoredInShadowMap)
                  {
                        diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D,
                              PSIn.Normal);
                        diffuseLightingFactor = saturate(diffuseLightingFactor);
                        diffuseLightingFactor *= xLightPower;
                        diffuseLightingFactor *= xLightPower [0];

                  }
            else diffuseLightingFactor *= xLightPower [1];

      ...
```

```
	// We want a different level of emissive light for each lamp
	if (dist0 < 1)
		{
			Output.Color.rgb += (xLightToCamRadius0 * xLampGlowFactor0 * (1-dist0));
			Output.Color.rgb += (xCamLtTimesGlow [0] * (1-dist0));
		}

	if (dist1 < 1)
		{
			Output.Color.rgb += (xLightToCamRadius1 * xLampGlowFactor1 * (1-dist1));
			Output.Color.rgb += (xCamLtTimesGlow [1] * (1-dist1));
		}

    ...
```

In ShadowedScene:

```
...
	diffuseLightingFactor = saturate(diffuseLightingFactor);
	diffuseLightingFactor *= xLightPower;
	diffuseLightingFactor *= xLightPower [0];

...

for (int CurrentLight=0; CurrentLight<4; CurrentLight++)
	{
	  float4 lightScreenPos = mul(float4(xLamppostPos[CurrentLight],1),xViewProjection);
	  lightScreenPos /= lightScreenPos.w;

	  float dist = distance(screenPos.xy, lightScreenPos.xy);
	  float dist = distance(screenPos.xy, xLtscrPositions[CurrentLight].xy);

	  float radius = 5.0f/distance(xCameraPos, xLamppostPos[CurrentLight]);
	  if (dist < radius)
		{
			Output.Color.rgb += (radius-dist)*4.0f;
		}
	  if (dist < xCamRads[CurrentLight])
		{
			Output.Color.rgb += (xCamRads[CurrentLight]-dist)*4.0f;
		}

	}
```

Run the code. You should see the same image as before, but it is being rendered more efficiently.

Here is our code at this point.

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
```

```
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }

    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float[] lightPower = new float[2];
        Vector4[] ltscrPositions = new Vector4[4];
        Matrix CameraViewProjection;
        Vector3[] camdists = new Vector3[4];
        float[] camrads = new float[4];
        float[] CamLtTimesGlow = new float[4];
        float[] LampGlowFactor = new float[4];

        float ambientPower;
```

```
        Matrix lightsViewProjectionMatrix;

        RenderTarget2D renderTarget;
        Texture2D shadowMap;
        Texture2D carLight;

        Vector3[] lamppostPositions = new Vector3[4];

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }

        private void SetUpCamera()
```

```
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);

            // using an array for light power; [0] is in light, [1] is in shadow
            lightPower[0] = 2.0f;
            lightPower[1] = 1.6f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;

            //These are the "real" lights
            lamppostPositions[0] = new Vector3(3f, 11.5f, -35f);
            lamppostPositions[1] = new Vector3(3f, 11.5f, -5f);

            //These are the shadows of the real lights
            lamppostPositions[2] = new Vector3(11f, 14.5f, -5f);
            lamppostPositions[3] = new Vector3(4.5f, 14.0f, -35f);

            //compute light screen positions
            CameraViewProjection = viewMatrix * projectionMatrix;
            for (int i = 0; i < 4; i++)
            {
                ltscrPositions[i] = Vector4.Transform(lamppostPositions[i], CameraViewProjection);
                ltscrPositions[i] /= ltscrPositions[i].W;
                // Compute the distance between each lamp and the camera
                camdists[i] = cameraPos - lamppostPositions[i];
                camrads[i] = 5.0f / camdists[i].Length();
                CamLtTimesGlow[i] = camdists[i].Length() * LampGlowFactor[i];
            }
        }

        private void LoadCar()
        {
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
```

```
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }

        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

        private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
CameraViewProjection);
                        currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                        currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                        currentEffect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
                        currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                        currentEffect.Parameters["xLightAttenuation"].SetValue(4000);
                        currentEffect.Parameters["xLightFallOff"].SetValue(0.027f);
                        currentEffect.Parameters["xLtscrPositions"].SetValue(ltscrPositions);
                        currentEffect.Parameters["xCamRads"].SetValue(camrads);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }

        private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);
            Matrix[] modelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(modelTransforms);

            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
                Matrix worldMatrix = modelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
```

```
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix *
CameraViewProjection);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                        currentEffect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
                        currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                        currentEffect.Parameters["xCamLtTimesGlow"].SetValue(CamLtTimesGlow);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }

        private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            if (technique != "ShadowMap")
            {
                effect.Parameters["xShadowMap"].SetValue(shadowMap);
                effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity *
CameraViewProjection);
                effect.Parameters["xTexture"].SetValue(streetTexture);
                effect.Parameters["xWorld"].SetValue(Matrix.Identity);
                effect.Parameters["xLightPos"].SetValue(lightPos);
                effect.Parameters["xLightPower"].SetValue(lightPower);
                effect.Parameters["xAmbient"].SetValue(ambientPower);
                effect.Parameters["xCarLightTexture"].SetValue(carLight);
                effect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
                effect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                effect.Parameters["xLtscrPositions"].SetValue(ltscrPositions);
                effect.Parameters["xCamRads"].SetValue(camrads);
                effect.Parameters["xLightAttenuation"].SetValue(4000);
                effect.Parameters["xLightFallOff"].SetValue(0.027f);
            }
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }

        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            LampGlowFactor[0] = 0.034f;
```

```
            LampGlowFactor[1] = 0.060f;
            LampGlowFactor[2] = 0.0f;
            LampGlowFactor[3] = 0.0f;

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");
            carLight = Content.Load<Texture2D>("carlight");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();

            SetUpVertices();
            SetUpCamera();

            PresentationParameters pp = device.PresentationParameters;
            renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
SurfaceFormat.Single, DepthFormat.Depth24);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.SamplerStates[0] = SamplerState.PointClamp;

            device.SetRenderTarget(renderTarget);

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            device.SetRenderTarget(null);
            shadowMap = renderTarget;

            // Reset 3D Defaults
            GraphicsDevice.BlendState = BlendState.Opaque;
```

```
            GraphicsDevice.DepthStencilState = DepthStencilState.Default;
            GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            DrawScene("ShadowedScene");
            DrawCar(car1Matrix, "ShadowedScene");
            DrawCar(car2Matrix, "ShadowedScene");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
    }
}
```

Effect1.fx

```
// Global Inputs

float4x4 xWorldViewProjection;
float4x4 xWorld;
float4x4 xLightsWorldViewProjection;
float3 xLightPos;
float xLightPower [2];
float xAmbient;
float4x4 xViewProjection;
float3 xLamppostPos[4];
float xLightAttenuation;
float xLightFallOff;
float xCamRads [4];
float xCamLtTimesGlow [4];
float4 xLtscrPositions [4];

//Texture Samplers

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = mirror;
                                                    AddressV = mirror;};

Texture xShadowMap;
sampler ShadowMapSampler = sampler_state {      texture = <xShadowMap>;
                                                    magfilter = POINT;
                                                    minfilter = POINT;
                                                    mipfilter = POINT;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};

Texture xCarLightTexture;
sampler CarLightSampler = sampler_state {texture = <xCarLightTexture>;
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};

// Subroutines
```

```
float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
    return dot(-lightDir, normal);
}

// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}

PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower [0];

       PSIn.TexCoords.y--;

       float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}

technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------
```

```
struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
    float4 Pos2DAsSeenByLight    : TEXCOORD2;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);
       Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       float diffuseLightingFactor = 0;

       if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y
                    == ProjectedTexCoords.y))
       {
                    float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
                    float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
                    diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
                    diffuseLightingFactor = saturate(diffuseLightingFactor);
                    if ((realDistance - 0.01f) <= depthStoredInShadowMap)
                           {
                                  diffuseLightingFactor *= xLightPower [0];
                           }
                    else diffuseLightingFactor *= xLightPower [1];
       }

       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

       // Add some point emissive lighting for the lamps
       // No reason not to do the computation in world space

       float dist0 = distance(PSIn.Position3D, xLamppostPos[0]);
       float dist1 = distance(PSIn.Position3D, xLamppostPos[1]);

       // We want a different level of emissive light for each lamp
       if (dist0 < 1)
              {
```

```
                    Output.Color.rgb += (xCamLtTimesGlow [0] * (1-dist0));
            }

       if (dist1 < 1)
            {
                    Output.Color.rgb += (xCamLtTimesGlow [1] * (1-dist1));
            }

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}

//------- Technique: ShadowMap --------

struct SMapVertexToPixel
{
    float4 Position     : POSITION;
    float4 Position2D    : TEXCOORD0;
};

struct SMapPixelToFrame
{
    float4 Color : COLOR0;
};

SMapVertexToPixel ShadowMapVertexShader( float4 inPos : POSITION)
{
    SMapVertexToPixel Output = (SMapVertexToPixel)0;

    Output.Position = mul(inPos, xLightsWorldViewProjection);
    Output.Position2D = Output.Position;

    return Output;
}

SMapPixelToFrame ShadowMapPixelShader(SMapVertexToPixel PSIn)
{
    SMapPixelToFrame Output = (SMapPixelToFrame)0;

    Output.Color = PSIn.Position2D.z/PSIn.Position2D.w;

    return Output;
}


technique ShadowMap
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowMapVertexShader();
        PixelShader = compile ps_2_0 ShadowMapPixelShader();
    }
}
```

```
//------- Technique: ShadowedScene --------

struct SSceneVertexToPixel
{
    float4 Position              : POSITION;
    float4 Pos2DAsSeenByLight    : TEXCOORD0;
    float2 TexCoords             : TEXCOORD1;
       float3 Normal                : TEXCOORD2;
       float4 Position3D            : TEXCOORD3;
};

struct SScenePixelToFrame
{
    float4 Color : COLOR0;
};

SSceneVertexToPixel ShadowedSceneVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0,
float3 inNormal : NORMAL)
{
    SSceneVertexToPixel Output = (SSceneVertexToPixel)0;

    Output.Position = mul(inPos, xWorldViewProjection);
    Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);
    Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
    Output.Position3D = mul(inPos, xWorld);
    Output.TexCoords = inTexCoords;

    return Output;
}

SScenePixelToFrame ShadowedScenePixelShader(SSceneVertexToPixel PSIn)
{
    SScenePixelToFrame Output = (SScenePixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

    float diffuseLightingFactor = 0;
    if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y ==
ProjectedTexCoords.y))
    {
        float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
        float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
        if ((realDistance - 0.01f) <= depthStoredInShadowMap)
        {
            diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
            diffuseLightingFactor = saturate(diffuseLightingFactor);
            diffuseLightingFactor *= xLightPower [0];

                    float lightTextureFactor = tex2D(CarLightSampler, ProjectedTexCoords).r;
                    diffuseLightingFactor *= lightTextureFactor;
        }
    }

       PSIn.TexCoords.y--;

    float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);

    Output.Color = baseColor * (diffuseLightingFactor + xAmbient);

       //Lamp point light code - we do this for both lamps
```

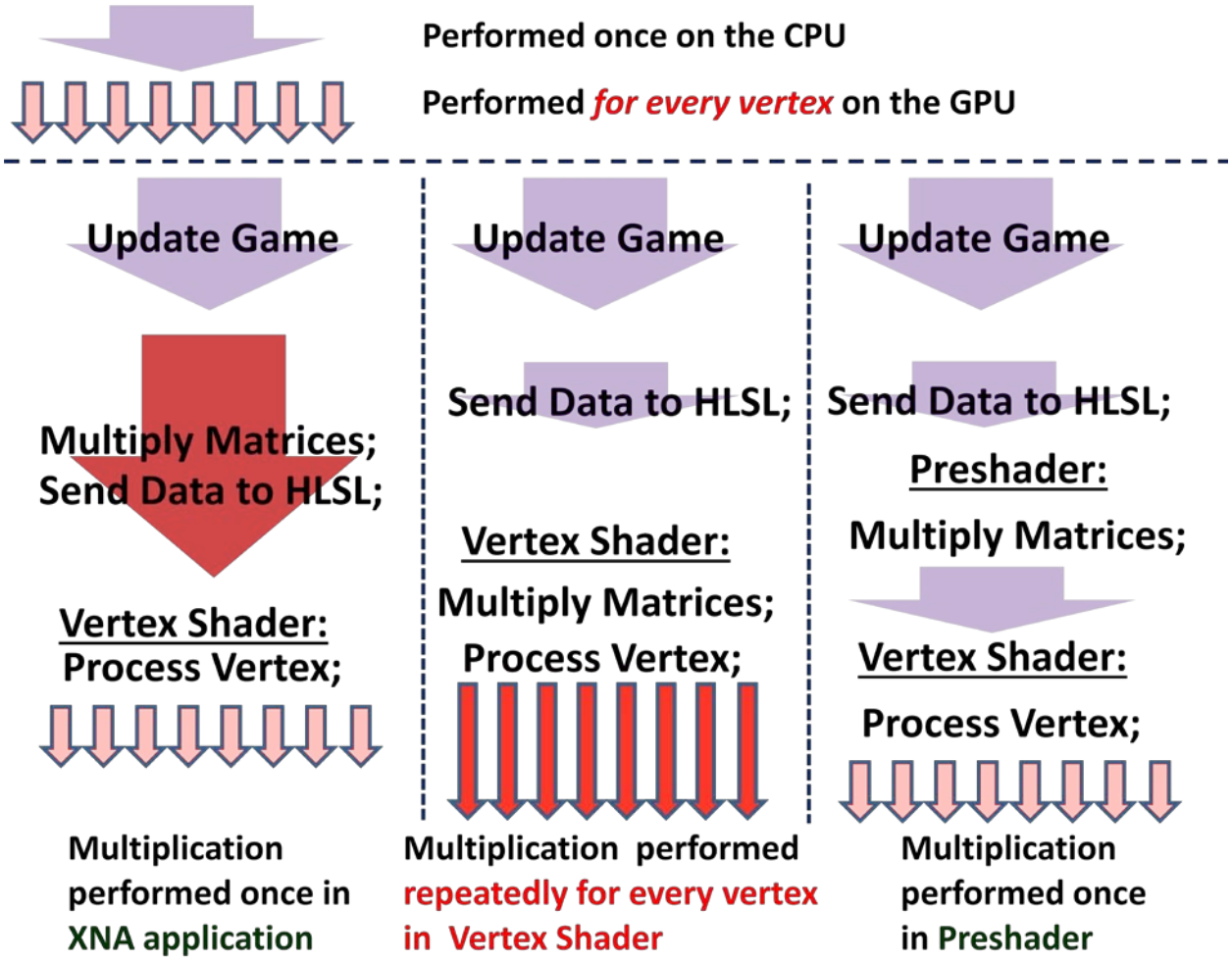
```
        float3 lightDir0 = normalize(xLamppostPos[0] - PSIn.Position3D);
        float3 lightDir1 = normalize(xLamppostPos[1] - PSIn.Position3D);
        float diffuse0 = saturate(dot(normalize(PSIn.Normal), lightDir0));
        float diffuse1 = saturate(dot(normalize(PSIn.Normal), lightDir1));

        float d0 = distance(xLamppostPos[0], PSIn.Position3D);
        float d1 = distance(xLamppostPos[1], PSIn.Position3D);

        float att0 = 1 - pow(clamp(d0 / xLightAttenuation, 0, 1), xLightFallOff);
        float att1 = 1 - pow(clamp(d1 / xLightAttenuation, 0, 1), xLightFallOff);

        Output.Color +=  (diffuse0 * att0) + (diffuse1 * att1);

    // Lamp emissive light code (this handles the "aura" of both real and shadow lights)

    float4 screenPos = mul(PSIn.Position3D, xViewProjection);
    screenPos /= screenPos.w;

    for (int CurrentLight=0; CurrentLight<4; CurrentLight++)
       {
         float dist = distance(screenPos.xy, xLtscrPositions[CurrentLight].xy);
               if (dist < xCamRads[CurrentLight])
              {
                           Output.Color.rgb += (xCamRads[CurrentLight]-dist)*4.0f;
              }
       }

    return Output;

}

technique ShadowedScene
{
    pass Pass0
    {
        VertexShader = compile vs_3_0 ShadowedSceneVertexShader();
        PixelShader = compile ps_3_0 ShadowedScenePixelShader();
    }
}
```

Preshaders

Instead of moving non-varying computations to our XNA C# code, we can rely on the HLSL compiler to do this for us, by creating what is known as a preshader. In this section, we will demonstrate how to use preshaders.

Consider the figure below. Three approaches are shown for multiplying matrices (imagine, for example, that this is the multiplication that computes the WorldViewProjection transform). So far, we have moved computations directly into the XNA application (the leftmost approach), and we have tried to avoid the middle approach (where we perform repeatedly calculations that only need to happen once).

Preshaders allow us to take the rightmost approach: when the HLSL code is being compiled, the compiler checks for code that will be the same for every vertex (or every pixel, in case of the pixel shader). If we place calculations that only need to be performed once inside the vertex shader, the HLSL compiler will pull that part of the HLSL code out of the vertex shader, and put into a “preshader”. This preshader is executed on the **CPU** before the vertex shader is actually called, and the resulting constants are passed to the HLSL code behind the scenes. This way, we can code the multiplications in our vertex shader (often the most convenient place to put them), yet they will be processed only once on the **CPU**, as shown on the rightmost portion of the figure.



To demonstrate this approach, we will move the computation of the WorldViewProjection matrix to a preshader. We already pass the World matrix, and the camera's ViewProjection matrix to the shader code, so all we have to do is multiply them. In each of the relevant techniques: SimpleTexture, SimpleNormal, and ShadowedScene, make the following changes:

```
float4x4 preWorldViewProjection = mul (xWorld, xViewProjection);

~~Output.Position = mul(inPos, xWorldViewProjection);~~

Output.Position = mul(inPos, preWorldViewProjection);
```

Since we are no longer using it, delete the xWorldViewProjection global constant.

```
~~float4x4 xWorldViewProjection;~~
```

Note that we could also compute the LightsWorldViewProjection matrix in a preshader, but we will not do that in this example.

Finally, we need to delete the line in DrawCar, DrawLampPost, and DrawScene that sends xWorldViewProjection to the shader code:

In DrawCar:

```
~~currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * CameraViewProjection);~~
```

In DrawLampPost:

```
currentEffect.Parameters["xWorldViewProjection"].SetValue(worldMatrix * CameraViewProjection);
```

In DrawScene:

```
effect.Parameters["xWorldViewProjection"].SetValue(Matrix.Identity * CameraViewProjection);
```

When you compile and run this code, you should get the same result as before, only this time, we are benefiting from the use of a preshader, and things are much more efficient. How do we know this? One way is to inspect the assembly code that is actually generated using the shader compiler from the command line. It is possible to see the assembly code generated by the HLSL compiler. There are commercial tools to help do this , such as FX Composer, but you can also just download the DirectX SDK from Microsoft, and run the fxc HLSL compiler from the command line.

After installing the DirectX SDK, copy your `Effect1.fx` file to the desktop and open a Windows command prompt window. Then execute the following commands:

```
cd Desktop
fxc /Tfx_2_0 Effect1.fx /Fc:output1.fxc
```

The first command changes the working directory to your desktop (where `Effect1.fx` is located). The second command compiles this code and writes the resulting assembler code in the file `output1.fxc`. By default, the fxc compiler will enable preshaders.

If you examine `output1.fxc` in a text editor, it should look something like this:

```
//listing of all techniques and passes with embedded asm listings

technique SimpleTexture
{
    pass Pass0
    {
        vertexshader =
            asm {
            //
            // Generated by Microsoft (R) D3DX9 Shader Compiler
            //
            // Parameters:
            //
            //   float4x4 xViewProjection;
            //   float4x4 xWorld;
            //
            //
            // Registers:
            //
            //   Name            Reg   Size
            //   --------------- ----- ----
            //   xWorld          c0       4
            //   xViewProjection c4       4
            //

                preshader
                mul r0, c4.x, c0
                mul r1, c4.y, c1
                ... more lines of assembly code ...
                add c3, r1, r0

            // approximately 28 instructions used
            //
```

```
            // Generated by Microsoft (R) D3DX9 Shader Compiler
            //
            // Parameters:
            //
            //   float4x4 xWorld;
            //
            //
            // Registers:
            //
            //   Name         Reg   Size
            //   ------------ ----- ----
            //   xWorld       c4       3
            //

                vs_2_0
                dcl_position v0
                dcl_normal v1
                ... more lines of assembly code ...
                mov oT0.xy, v2

            // approximately 14 instruction slots used
            };
```

You can see that the SimpleTexture technique preshader uses 28 **CPU** instructions (done once), and the SimpleTexture technique vertex shader uses 14 **GPU** instructions (done once for every vertex). To see the value of the preshader, let's now compile our effect file with preshaders disabled. We disable preshaders by using the /Op parameter, as follows:

`fxc /Op /Tfx_2_0 Effect1.fx /Fc:output2.fxc`

Now examine the second output file in a text editor. It should look like this:

```
//listing of all techniques and passes with embedded asm listings

technique SimpleTexture
{
    pass Pass0
    {
        vertexshader =
            asm {
            //
            // Generated by Microsoft (R) D3DX9 Shader Compiler
            //
            // Parameters:
            //
            //   float4x4 xViewProjection;
            //   float4x4 xWorld;
            //
            //
            // Registers:
            //
            //   Name            Reg   Size
            //   --------------- ----- ----
            //   xWorld          c0       4
            //   xViewProjection c4       4
            //

                vs_2_0
                dcl_position v0
                dcl_normal v1
                ... more lines of assembly code ...
                mov oT0.xy, v2
```

```
// approximately 34 instruction slots used
};
```

You can see that the SimpleTexture technique vertex shader uses 34 **GPU** instructions (done once for every vertex), as opposed to the 14 **GPU** instructions with preshading. That's a pretty big difference. The 28 **CPU** instuctions executed only once in the preshader have essentially no impact on performance. If we compare the output for all of our techniques, we see that preshaders can have varying impact, but that this impact is often significant:

Technique	Shader	No. of Insts. or Inst. Slots	No. of Insts. or Inst. Slots (no preshader)
SimpleTexture			
	vertexshader		
	preshader	28	N/A
	vs_2_0	14	34
	pixelshader		
	ps_2_0	11	11
SimpleNormal			
	vertexshader		
	preshader	28	N/A
	vs_2_0	17	37
	pixelshader		
	ps_2_0	46	46
ShadowMap			
	vertexshader		
	vs_2_0	6	6
	pixelshader		
	ps_2_0	3	3
ShadowedScene			
	vertexshader		
	preshader	28	N/A
	vs_3_0	19	39
	pixelshader		
	preshader	1	N/A
	ps_3_0	94	95

Note that it is also possible to extract the partiular shader (e.g., ShadowedScenePixelShader) from the file Effect1.fx, and compile it using a specific version of the shader compiler (e.g., Version 3.0 of the HLSL compiler's pixel shader), as follows:

```
fxc /Tps_3_0 Effect1.fx /EShadowedScenePixelShader /Fc:output-withOp.fxc
```

If you want to disable preshaders, you can tell the compiler to do this by using the /Op parameter, as before:

```
fxc /Op /Tps_3_0 Effect1.fx /EShadowedScenePixelShader /Fc:output-NoOp.fxc
```

That concludes Lab11. We have covered a lot of material, which should put you in a pretty good position to look at more advanced material. There are a host of online resources that can help you; here are two:

Here is our final code.

Game1.cs

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Net;
using Microsoft.Xna.Framework.Storage;

namespace ATLS_4519_Lab11
{
    struct MyOwnVertexFormat
    {
        private Vector3 position;
        private Vector2 texCoord;
        private Vector3 normal;

        public MyOwnVertexFormat(Vector3 position, Vector2 texCoord, Vector3 normal)
        {
            this.position = position;
            this.texCoord = texCoord;
            this.normal = normal;
        }

        public static VertexDeclaration VertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Vector2,
VertexElementUsage.TextureCoordinate, 0),
            new VertexElement(sizeof(float) * 5, VertexElementFormat.Vector3,
VertexElementUsage.Normal, 0)
        );
    }

    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        GraphicsDevice device;

        Effect effect;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        VertexBuffer vertexBuffer;
        Vector3 cameraPos;

        Texture2D streetTexture;

        Model lamppostModel;
        Matrix[] LampModelTransforms;

        Model carModel;
        Texture2D[] carTextures;
        Matrix car1Matrix;
```

```
        Matrix car2Matrix;
        Matrix[] CarModelTransforms;
        Vector3 lightPos;
        float [] lightPower = new float [2];
        float ambientPower;

        Matrix lightsViewProjectionMatrix;

        RenderTarget2D renderTarget;
        Texture2D shadowMap;
        Texture2D carLight;

        Vector3[] lamppostPositions = new Vector3[4];

        Vector4[] ltscrPositions = new Vector4[4];
        Matrix CameraViewProjection;
        Vector3[] camdists = new Vector3[4];
        float[] camrads = new float[4];
        float[] CamLtTimesGlow = new float[4];
        float[] LampGlowFactor = new float[4];

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

        private void SetUpVertices()
        {
            MyOwnVertexFormat[] vertices = new MyOwnVertexFormat[18];

            vertices[0] = new MyOwnVertexFormat(new Vector3(-20, 0, 10), new Vector2(-0.25f, 25.0f),
new Vector3(0, 1, 0));
            vertices[1] = new MyOwnVertexFormat(new Vector3(-20, 0, -100), new Vector2(-0.25f, 0.0f),
new Vector3(0, 1, 0));
            vertices[2] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(0, 1, 0));
            vertices[3] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(0, 1, 0));
            vertices[4] = new MyOwnVertexFormat(new Vector3(2, 0, 10), new Vector2(0.25f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[5] = new MyOwnVertexFormat(new Vector3(2, 0, -100), new Vector2(0.25f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[6] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(-1, 0, 0));
            vertices[7] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(-1, 0, 0));
            vertices[8] = new MyOwnVertexFormat(new Vector3(2, 1, 10), new Vector2(0.375f, 25.0f), new
Vector3(0, 1, 0));
            vertices[9] = new MyOwnVertexFormat(new Vector3(2, 1, -100), new Vector2(0.375f, 0.0f), new
Vector3(0, 1, 0));
            vertices[10] = new MyOwnVertexFormat(new Vector3(3, 1, 10), new Vector2(0.5f, 25.0f), new
Vector3(0, 1, 0));
            vertices[11] = new MyOwnVertexFormat(new Vector3(3, 1, -100), new Vector2(0.5f, 0.0f), new
Vector3(0, 1, 0));
            vertices[12] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(0, 1, 0));
            vertices[13] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(0, 1, 0));
            vertices[14] = new MyOwnVertexFormat(new Vector3(13, 1, 10), new Vector2(0.75f, 25.0f), new
Vector3(-1, 0, 0));
```

```
            vertices[15] = new MyOwnVertexFormat(new Vector3(13, 1, -100), new Vector2(0.75f, 0.0f),
new Vector3(-1, 0, 0));
            vertices[16] = new MyOwnVertexFormat(new Vector3(13, 21, 10), new Vector2(1.25f, 25.0f),
new Vector3(-1, 0, 0));
            vertices[17] = new MyOwnVertexFormat(new Vector3(13, 21, -100), new Vector2(1.25f, 0.0f),
new Vector3(-1, 0, 0));

            vertexBuffer = new VertexBuffer(device, MyOwnVertexFormat.VertexDeclaration,
vertices.Length, BufferUsage.WriteOnly);

            vertexBuffer.SetData(vertices);
        }


        private void SetUpCamera()
        {
            cameraPos = new Vector3(-25, 13, 18);
            viewMatrix = Matrix.CreateLookAt(cameraPos, new Vector3(0, 2, -12), new Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
device.Viewport.AspectRatio, 1.0f, 200.0f);
        }

        private void UpdateLightData()
        {
            ambientPower = 0.2f;

            lightPos = new Vector3(-18, 5, -2);

            // using an array for light power; 0 is in light, 1 is in shadow
            lightPower [0] = 2.0f;
            lightPower [1] = 1.6f;

            Matrix lightsView = Matrix.CreateLookAt(lightPos, new Vector3(-2, 3, -10), new Vector3(0,
1, 0));
            Matrix lightsProjection = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver2, 1f, 5f,
10000f);

            lightsViewProjectionMatrix = lightsView * lightsProjection;

            //These are the "real" lights
            lamppostPositions[0] = new Vector3(3f, 11.5f, -35f);
            lamppostPositions[1] = new Vector3(3f, 11.5f, -5f);

            //These are the shadows of the real lights
            lamppostPositions[2] = new Vector3(11f, 14.5f, -5f);
            lamppostPositions[3] = new Vector3(4.5f, 14.0f, -35f);

            //compute light screen positions
            CameraViewProjection = viewMatrix * projectionMatrix;
            for (int i = 0; i < 4; i++)
            {
                ltscrPositions[i] = Vector4.Transform(lamppostPositions[i], CameraViewProjection);
                ltscrPositions[i] /= ltscrPositions[i].W;
                // Compute the distance between each lamp and the camera
                camdists[i] = cameraPos - lamppostPositions[i];
                camrads [i] = 5.0f / camdists[i].Length();
                CamLtTimesGlow[i] = camdists[i].Length() * LampGlowFactor[i];
            }
        }

        private void LoadCar()
        {
```

```
            carModel = Content.Load<Model>("car");
            carTextures = new Texture2D[carModel.Meshes.Count + 2]; // More effects than meshes in this
model
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (BasicEffect currentEffect in mesh.Effects)
                    carTextures[i++] = currentEffect.Texture;

            foreach (ModelMesh mesh in carModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();

            CarModelTransforms = new Matrix[carModel.Bones.Count];
            carModel.CopyAbsoluteBoneTransformsTo(CarModelTransforms);
            car1Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi) *
Matrix.CreateTranslation(-3, 0, -15);
            car2Matrix = Matrix.CreateScale(4f) * Matrix.CreateRotationY(MathHelper.Pi * 5.0f / 8.0f) *
Matrix.CreateTranslation(-28, 0, -1.9f);
        }

        private void LoadLamp()
        {
            lamppostModel = Content.Load<Model>("lamppost");
            LampModelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(LampModelTransforms);
            foreach (ModelMesh mesh in lamppostModel.Meshes)
                foreach (ModelMeshPart meshPart in mesh.MeshParts)
                    meshPart.Effect = effect.Clone();
        }

        private void DrawCar(Matrix wMatrix, string technique)
        {
            int i = 0;
            foreach (ModelMesh mesh in carModel.Meshes)
            {
                Matrix worldMatrix = CarModelTransforms[mesh.ParentBone.Index] * wMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xTexture"].SetValue(carTextures[i++]);
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                        currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                        currentEffect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
                        currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                        currentEffect.Parameters["xLightAttenuation"].SetValue(4000);
                        currentEffect.Parameters["xLightFallOff"].SetValue(0.027f);
                        currentEffect.Parameters["xLtscrPositions"].SetValue(ltscrPositions);
                        currentEffect.Parameters["xCamRads"].SetValue(camrads);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }
```

```
        private void DrawLampPost(float scale, Vector3 translation, string technique)
        {
            Matrix lampMatrix = Matrix.CreateScale(scale) * Matrix.CreateTranslation(translation);
            Matrix[] modelTransforms = new Matrix[lamppostModel.Bones.Count];
            lamppostModel.CopyAbsoluteBoneTransformsTo(modelTransforms);

            foreach (ModelMesh mesh in lamppostModel.Meshes)
            {
                Matrix worldMatrix = modelTransforms[mesh.ParentBone.Index] * lampMatrix;
                foreach (Effect currentEffect in mesh.Effects)
                {
                    currentEffect.CurrentTechnique = currentEffect.Techniques[technique];
                    if (technique != "ShadowMap")
                    {
                        currentEffect.Parameters["xWorld"].SetValue(worldMatrix);
                        currentEffect.Parameters["xLightPos"].SetValue(lightPos);
                        currentEffect.Parameters["xLightPower"].SetValue(lightPower);
                        currentEffect.Parameters["xAmbient"].SetValue(ambientPower);
                        currentEffect.Parameters["xShadowMap"].SetValue(shadowMap);
                        currentEffect.Parameters["xCarLightTexture"].SetValue(carLight);
                        currentEffect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
                        currentEffect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                        currentEffect.Parameters["xCamLtTimesGlow"].SetValue(CamLtTimesGlow);
                    }
                    currentEffect.Parameters["xLightsWorldViewProjection"].SetValue(worldMatrix *
lightsViewProjectionMatrix);
                }
                mesh.Draw();
            }
        }

        private void DrawScene(string technique)
        {
            effect.CurrentTechnique = effect.Techniques[technique];
            if (technique != "ShadowMap")
            {
                effect.Parameters["xShadowMap"].SetValue(shadowMap);
                effect.Parameters["xTexture"].SetValue(streetTexture);
                effect.Parameters["xWorld"].SetValue(Matrix.Identity);
                effect.Parameters["xLightPos"].SetValue(lightPos);
                effect.Parameters["xLightPower"].SetValue(lightPower);
                effect.Parameters["xAmbient"].SetValue(ambientPower);
                effect.Parameters["xCarLightTexture"].SetValue(carLight);
                effect.Parameters["xViewProjection"].SetValue(CameraViewProjection);
                effect.Parameters["xLamppostPos"].SetValue(lamppostPositions);
                effect.Parameters["xLtscrPositions"].SetValue(ltscrPositions);
                effect.Parameters["xCamRads"].SetValue(camrads);
                effect.Parameters["xLightAttenuation"].SetValue(4000);
                effect.Parameters["xLightFallOff"].SetValue(0.027f);
            }
            effect.Parameters["xLightsWorldViewProjection"].SetValue(Matrix.Identity *
lightsViewProjectionMatrix);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();
                device.SetVertexBuffer(vertexBuffer);
                device.DrawPrimitives(PrimitiveType.TriangleStrip, 0, 18);
            }
        }

        protected override void Initialize()
```

```
        {
            graphics.PreferredBackBufferWidth = 700;
            graphics.PreferredBackBufferHeight = 700;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 11 - HLSL Tutorial";

            LampGlowFactor[0] = 0.034f;
            LampGlowFactor[1] = 0.060f;
            LampGlowFactor[2] = 0.0f;
            LampGlowFactor[3] = 0.0f;

            base.Initialize();
        }

        protected override void LoadContent()
        {
            device = GraphicsDevice;

            effect = Content.Load<Effect>("Effect1");
            streetTexture = Content.Load<Texture2D>("streettexture");
            carLight = Content.Load<Texture2D>("carlight");

            // Load the cars
            LoadCar();

            //Load the Lamp Posts
            LoadLamp();

            SetUpVertices();
            SetUpCamera();

            PresentationParameters pp = device.PresentationParameters;
            renderTarget = new RenderTarget2D(device, pp.BackBufferWidth, pp.BackBufferHeight, true,
SurfaceFormat.Single, DepthFormat.Depth24);
        }

        protected override void UnloadContent()
        {
        }

        protected override void Update(GameTime gameTime)
        {
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            UpdateLightData();

            base.Update(gameTime);
        }

        protected override void Draw(GameTime gameTime)
        {
            device.SamplerStates[0] = SamplerState.PointClamp;

            device.SetRenderTarget(renderTarget);

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.White, 1.0f, 0);

            DrawScene("ShadowMap");
            DrawCar(car1Matrix, "ShadowMap");
            DrawCar(car2Matrix, "ShadowMap");
```

```
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "ShadowMap");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "ShadowMap");

            device.SetRenderTarget(null);
            shadowMap = renderTarget;

            // Reset 3D Defaults
            GraphicsDevice.BlendState = BlendState.Opaque;
            GraphicsDevice.DepthStencilState = DepthStencilState.Default;
            GraphicsDevice.SamplerStates[0] = SamplerState.LinearWrap;

            device.Clear(ClearOptions.Target | ClearOptions.DepthBuffer, Color.Black, 1.0f, 0);

            DrawScene("ShadowedScene");
            DrawCar(car1Matrix, "ShadowedScene");
            DrawCar(car2Matrix, "ShadowedScene");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -35f), "SimpleNormal");
            DrawLampPost(0.05f, new Vector3(4.0f, 1f, -5f), "SimpleNormal");

            base.Draw(gameTime);
        }
    }

}
```

Effect1.fx

```
// Global Inputs

float4x4 xWorld;
float4x4 xLightsWorldViewProjection;
float3 xLightPos;
float xLightPower [2];
float xAmbient;
float4x4 xViewProjection;
float3 xLamppostPos[4];
float xLightAttenuation;
float xLightFallOff;
float xCamRads [4];
float xCamLtTimesGlow [4];
float4 xLtscrPositions [4];

//Texture Samplers

Texture xTexture;
sampler TextureSampler = sampler_state {  texture = <xTexture> ;
                                          magfilter = LINEAR;
                                          minfilter = LINEAR;
                                          mipfilter = LINEAR;
                                          AddressU = mirror;
                                          AddressV = mirror;};

Texture xShadowMap;
sampler ShadowMapSampler = sampler_state {      texture = <xShadowMap>;
                                          magfilter = POINT;
                                          minfilter = POINT;
                                          mipfilter = POINT;
                                          AddressU = clamp;
                                          AddressV = clamp;};

Texture xCarLightTexture;
sampler CarLightSampler = sampler_state {texture = <xCarLightTexture>;
```

```
                                                    magfilter = LINEAR;
                                                    minfilter = LINEAR;
                                                    mipfilter = LINEAR;
                                                    AddressU = clamp;
                                                    AddressV = clamp;};

// Subroutines

float DotProduct(float3 lightPos, float3 pos3D, float3 normal)
{
    float3 lightDir = normalize(pos3D - lightPos);
    return dot(-lightDir, normal);
}

// Techniques

//------- Technique: SimpleTexture --------

struct VertexToPixel
{
    float4 Position      : POSITION;
    float2 TexCoords     : TEXCOORD0;
    float3 Normal        : TEXCOORD1;
    float3 Position3D    : TEXCOORD2;
};

struct PixelToFrame
{
    float4 Color        : COLOR0;
};

VertexToPixel STVertexShader( float4 inPos : POSITION0, float3 inNormal: NORMAL0, float2 inTexCoords :
TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;

    float4x4 preWorldViewProjection = mul (xWorld, xViewProjection);

       Output.Position = mul(inPos, preWorldViewProjection);
    Output.TexCoords = inTexCoords;
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);

    return Output;
}

PixelToFrame STPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;

       float diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
       diffuseLightingFactor = saturate(diffuseLightingFactor);
       diffuseLightingFactor *= xLightPower [0];

       PSIn.TexCoords.y--;

       float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

    return Output;
}
```

```
technique SimpleTexture
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 STVertexShader();
        PixelShader = compile ps_2_0 STPixelShader();
    }
}

//------- Technique: SimpleNormal --------

struct SNVertexToPixel
{
    float4 Position     : POSITION;
    float3 Normal        : TEXCOORD0;
    float3 Position3D    : TEXCOORD1;
    float4 Pos2DAsSeenByLight    : TEXCOORD2;
};

struct SNPixelToFrame
{
    float4 Color        : COLOR0;
};

SNVertexToPixel SNVertexShader(float4 inPos : POSITION, float3 inNormal: NORMAL)
{
    SNVertexToPixel Output = (SNVertexToPixel)0;

    float4x4 preWorldViewProjection = mul (xWorld, xViewProjection);

       Output.Position = mul(inPos, preWorldViewProjection);
       Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
       Output.Position3D = mul(inPos, xWorld);
       Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);

    return Output;
}

SNPixelToFrame SNPixelShader(SNVertexToPixel PSIn)
{
    SNPixelToFrame Output = (SNPixelToFrame)0;

       float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

       float4 baseColor = float4(.1, .1, .1, 1); // pick a dark gray color
       float diffuseLightingFactor = 0;

       if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y
                    == ProjectedTexCoords.y))
       {
                    float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
                    float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
                    diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
                    diffuseLightingFactor = saturate(diffuseLightingFactor);
                    if ((realDistance - 0.01f) <= depthStoredInShadowMap)
                           {
                                  diffuseLightingFactor *= xLightPower [0];
                           }
                    else diffuseLightingFactor *= xLightPower [1];
       }
```

```
       Output.Color = baseColor*(diffuseLightingFactor + xAmbient);

       // Add some point emisssive lighting for the lamps
       // No reason not to do the computation in world space

       float dist0 = distance(PSIn.Position3D, xLamppostPos[0]);
       float dist1 = distance(PSIn.Position3D, xLamppostPos[1]);

       // We want a different level of emissive light for each lamp
       if (dist0 < 1)
              {
                     Output.Color.rgb += (xCamLtTimesGlow [0] * (1-dist0));
              }

       if (dist1 < 1)
              {
                     Output.Color.rgb += (xCamLtTimesGlow [1] * (1-dist1));
              }

    return Output;
}

technique SimpleNormal
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 SNVertexShader();
        PixelShader = compile ps_2_0 SNPixelShader();
    }
}

//------- Technique: ShadowMap --------

struct SMapVertexToPixel
{
    float4 Position     : POSITION;
    float4 Position2D    : TEXCOORD0;
};

struct SMapPixelToFrame
{
    float4 Color : COLOR0;
};

SMapVertexToPixel ShadowMapVertexShader( float4 inPos : POSITION)
{
    SMapVertexToPixel Output = (SMapVertexToPixel)0;

    Output.Position = mul(inPos, xLightsWorldViewProjection);
    Output.Position2D = Output.Position;

    return Output;
}

SMapPixelToFrame ShadowMapPixelShader(SMapVertexToPixel PSIn)
{
    SMapPixelToFrame Output = (SMapPixelToFrame)0;

    Output.Color = PSIn.Position2D.z/PSIn.Position2D.w;

    return Output;
```

```
}


technique ShadowMap
{
    pass Pass0
    {
        VertexShader = compile vs_2_0 ShadowMapVertexShader();
        PixelShader = compile ps_2_0 ShadowMapPixelShader();
    }
}

//------- Technique: ShadowedScene --------

struct SSceneVertexToPixel
{
    float4 Position              : POSITION;
    float4 Pos2DAsSeenByLight    : TEXCOORD0;
    float2 TexCoords             : TEXCOORD1;
       float3 Normal                : TEXCOORD2;
       float4 Position3D            : TEXCOORD3;
};

struct SScenePixelToFrame
{
    float4 Color : COLOR0;
};

SSceneVertexToPixel ShadowedSceneVertexShader( float4 inPos : POSITION, float2 inTexCoords : TEXCOORD0,
float3 inNormal : NORMAL)
{
    SSceneVertexToPixel Output = (SSceneVertexToPixel)0;

    float4x4 preWorldViewProjection = mul (xWorld, xViewProjection);

       Output.Position = mul(inPos, preWorldViewProjection);
    Output.Pos2DAsSeenByLight = mul(inPos, xLightsWorldViewProjection);
    Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
    Output.Position3D = mul(inPos, xWorld);
    Output.TexCoords = inTexCoords;

    return Output;
}

SScenePixelToFrame ShadowedScenePixelShader(SSceneVertexToPixel PSIn)
{
    SScenePixelToFrame Output = (SScenePixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords[0] = PSIn.Pos2DAsSeenByLight.x/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;
    ProjectedTexCoords[1] = -PSIn.Pos2DAsSeenByLight.y/PSIn.Pos2DAsSeenByLight.w/2.0f +0.5f;

    float diffuseLightingFactor = 0;
    if ((saturate(ProjectedTexCoords).x == ProjectedTexCoords.x) && (saturate(ProjectedTexCoords).y ==
ProjectedTexCoords.y))
    {
        float depthStoredInShadowMap = tex2D(ShadowMapSampler, ProjectedTexCoords).r;
        float realDistance = PSIn.Pos2DAsSeenByLight.z/PSIn.Pos2DAsSeenByLight.w;
        if ((realDistance - 0.01f) <= depthStoredInShadowMap)
        {
            diffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D, PSIn.Normal);
            diffuseLightingFactor = saturate(diffuseLightingFactor);
```

```
            diffuseLightingFactor *= xLightPower [0];

                    float lightTextureFactor = tex2D(CarLightSampler, ProjectedTexCoords).r;
                    diffuseLightingFactor *= lightTextureFactor;
         }
    }

       PSIn.TexCoords.y--;

    float4 baseColor = tex2D(TextureSampler, PSIn.TexCoords);

    Output.Color = baseColor * (diffuseLightingFactor + xAmbient);

       //Lamp point light code - we do this for both lamps

       float3 lightDir0 = normalize(xLamppostPos[0] - PSIn.Position3D);
       float3 lightDir1 = normalize(xLamppostPos[1] - PSIn.Position3D);
       float diffuse0 = saturate(dot(normalize(PSIn.Normal), lightDir0));
       float diffuse1 = saturate(dot(normalize(PSIn.Normal), lightDir1));

       float d0 = distance(xLamppostPos[0], PSIn.Position3D);
       float d1 = distance(xLamppostPos[1], PSIn.Position3D);

       float att0 = 1 - pow(clamp(d0 / xLightAttenuation, 0, 1), xLightFallOff);
       float att1 = 1 - pow(clamp(d1 / xLightAttenuation, 0, 1), xLightFallOff);

       Output.Color +=  (diffuse0 * att0) + (diffuse1 * att1);

    // Lamp emissive light code (this handles the "aura" of both real and shadow lights)

    float4 screenPos = mul(PSIn.Position3D, xViewProjection);
    screenPos /= screenPos.w;

    for (int CurrentLight=0; CurrentLight<4; CurrentLight++)
       {
         float dist = distance(screenPos.xy, xLtscrPositions[CurrentLight].xy);
         if (dist < xCamRads[CurrentLight])
              {
              Output.Color.rgb += (xCamRads[CurrentLight]-dist)*4.0f;
              }
       }

    return Output;

}

technique ShadowedScene
{
    pass Pass0
    {
        VertexShader = compile vs_3_0 ShadowedSceneVertexShader();
        PixelShader = compile ps_3_0 ShadowedScenePixelShader();
    }
}
```