ATLS 4519/5519 LAB 7[1]
JK Bennett
Spring 2013

This lab will introduce you to how to create terrain. We will first review some basic principles of 3D graphics, and will gradually add complexity until we have a reasonably sophisticated representation of our desired terrain. In particular, we will learn how HLSL (High Level Shader Language) effect files work. Lab 8 will further develop these ideas to allow us to create even more sophisticated terrain.

You will need to load two files to complete this lab. Download and unzip this file to obtain them. If this link does not work for some reason, here is the hard link: http://redwood.colorado.edu/jkb/atls-4519-s13/Labs/Lab7_Content.zip.

Getting Started

Begin by creating a new Windows 4.0 game project (named `ATLS_4519_Lab7`), as you have done in previous labs. Recall that a 'device' is a direct link to the graphics hardware on your computer. This variable is made available in XNA as an instance of GraphicsDevice. Declare this variable by adding this line to the top of your class, just above the Game1() method:

```
GraphicsDevice device;
```

Now initialize this variable by adding the following line to the `LoadContent` method:

```
device = graphics.GraphicsDevice;
```

Let's put it to use, by making the first line in our `Draw` method a tiny bit shorter:

```
device.Clear(Color.CornflowerBlue);
```

Next, we're going to specify some extra stuff related to our window such as its size and title. Add this code to the Initialize method:

```
graphics.PreferredBackBufferWidth = 500;
graphics.PreferredBackBufferHeight = 500;
graphics.IsFullScreen = false;
graphics.ApplyChanges();
Window.Title = "ATLS 4519 Lab7 - Terrain Tutorial";
```

The first line sets the size of our backbuffer, which will contain what will be drawn to the screen. We also indicate that we want our program to run in a window (as opposed to full screen), after which we apply the changes. The last line sets the title of our window.

When you run this code, you should see a window of 500x500 pixels, with the title as shown.

Using a Custom Effect

XNA uses an effect for everything it draws. So what exactly is an effect?

In virtually all 3D programming, all objects are represented using triangles. Any shape can be represented using triangles, if you use enough of them. An effect is some code that instructs your hardware (the graphics card) how it should display these triangles. An effect file contains one or more **techniques**, for example technique A and technique B. Drawing triangles using technique A might draw them semi-transparent, while drawing them using technique B might draw all objects using only blue-gray colors, as seen in some horror movies.

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

XNA needs an effect file to draw even the simplest triangles. Import effects.fx, which contains some very basic techniques. To import the file into our content project, right-click on the Content project, Select Add->Existing Item and browse to effects.fx file. After you've clicked the OK button, the effects.fx file should be added to your Content entry. You can also just drag and drop the new content to the Content Project.

Next, we will link this effect file to a variable, so we can actually use it in our code. We need to declare a new Effect object, so put this at the beginning of the Game1 class code:

```
Effect effect;
```

In your LoadContent method, add this line to have XNA load the .fx file into the effect variable:

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

The "effects" name refers to the part of the file before the .fx extension.

With all the necessary variables loaded, we can concentrate on the Draw method. You'll notice the first line starts with a Clear command. This line clears the buffer of our window and fills it with a specified color. Let's set this to DarkSlateBlue, just for fun:

```
device.Clear(Color.DarkSlateBlue);
```

XNA draws to a buffer, instead of drawing directly to the window. At the end of the Draw method, the contents of the buffer are drawn on the screen all at once. This way, the screen will not flicker as it might if we were to draw each part of our scene separately to the screen.

To draw something we first need to specify a technique from an Effect object. To activate an effect from our .fx file, add this line to your Draw method:

```
effect.CurrentTechnique = effect.Techniques["Pretransformed"];
```

Open the Effects.fx file in your project and look for the implementation of the Pretransformed technique. A technique can be made up of multiple passes, so we need to iterate through them. Add this code below the code that you just entered:

```
foreach (EffectPass pass in effect.CurrentTechnique.Passes)
{
    pass.Apply();

}
```

All drawing code must be put after the call to pass.Apply().

With all of this code set up, we're finally ready to start drawing things on the screen, which is what we will do next. Every object drawn in 3D is drawn using triangles, and a triangle is defined by 3 points. Every point is defined by a vector, specifying the X, Y and Z coordinates of the point. However, just knowing the coordinates of a point might not be enough. For example, we might want to define a color for the given point as well. We will use the term "vertex" to refer to the list of properties of a given point, including the position, color and so on.

XNA has a structure that fits perfectly to hold our vertex information: the VertexPositionColor struct. A vertex of this type can hold a position and a color, which is all we need for now. To define a triangle, we will need three vertices, which we will store in an array. So let's declare this variable at the top of our class (in Game1.cs, below "Effect effect"):

```
VertexPositionColor[] vertices;
```

Next, we will add a small method to our code, SetUpVertices, which will fill this array with the three vertices:

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

     vertices[0].Position = new Vector3(-0.5f, -0.5f, 0f);
     vertices[0].Color = Color.Red;
     vertices[1].Position = new Vector3(0, 0.5f, 0f);
     vertices[1].Color = Color.Green;
     vertices[2].Position = new Vector3(0.5f, -0.5f, 0f);
     vertices[2].Color = Color.Yellow;
 }
```

The array is initialized to hold 3 vertices, after which it is filled. For now, we are using coordinates that are relative to the screen: the (0,0,0) point would be the middle of our screen, the (-1, -1, 0) point bottom-left and the (1, 1, 0) point top-right. So in the example above, the first point is halfway to the bottom left of the window, and the second point is halfway to the top in the middle of our screen. (As these are not really 3D coordinates, they do not need to be transformed to 2D coordinates. Hence, the name of the technique: 'Pretransformed')

As you can see, we have set each of the vertices to different colors. We now need to call our SetUpVertices method. As it uses the device, call it at the end of the LoadContent method:

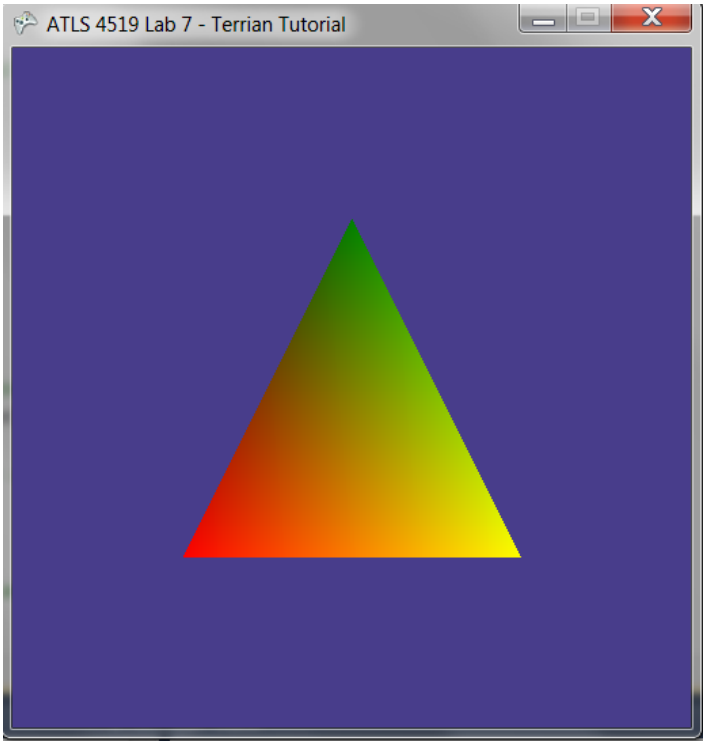
```
SetUpVertices();
```

All we have to do now is tell the device to draw the triangle! Go to our Draw method, where we want to draw the triangle after the call to pass.Apply:

```
device.DrawUserPrimitives(PrimitiveType.TriangleList, vertices, 0, 1,
       VertexPositionColor.VertexDeclaration);
```

This line actually tells the graphics card to draw the triangle: we want to draw one triangle from the vertices array, starting at vertex 0. TriangleList indicates that our vertices array contains a list of triangles (in this case, a list of only one triangle). If we wanted to draw four triangles, we would need an array of 12 vertices. Another possibility is to use a TriangleStrip, which can perform a lot faster, but is only useful to draw triangles that are connected to each other. The last argument specifies the VertexDeclaration, which tells the graphics card the format of the vertices we are sending it. We have stored the position and color data for three vertices inside an array. When we instruct XNA to render a triangle based on this data, XNA put these data in a byte stream and sends it to the graphics card. Our graphics card receives the byte stream, and interprets that byte stream based upon the VertexDeclaration that defines the vertices in the byte stream. By specifying VertexPositionColor as the kind of VertexDeclaration we are using, we are telling the graphics card that there are vertices coming its way that contain Position and Color data. We will see how to create our own vertex types later.

Running this should produce the following image (you can safely ignore the warning text "Warning ... effects.fx: warning X3576: semantics in type overridden by variable/function or enclosing type"):



Here is the code so far:

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
    /// <summary>
    /// This is the main type for your game
    /// </summary>
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;

        GraphicsDevice device;
        Effect effect;
        VertexPositionColor[] vertices;

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

            vertices[0].Position = new Vector3(-0.5f, -0.5f, 0f);
            vertices[0].Color = Color.Red;
            vertices[1].Position = new Vector3(0, 0.5f, 0f);
            vertices[1].Color = Color.Green;
            vertices[2].Position = new Vector3(0.5f, -0.5f, 0f);
```

```
    vertices[2].Color = Color.Yellow;
}

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

/// <summary>
/// Allows the game to perform any initialization it needs to before starting to run.
/// This is where it can query for any required services and load any non-graphic
/// related content.  Calling base.Initialize will enumerate through any components
/// and initialize them as well.
/// </summary>
protected override void Initialize()
{
    // TODO: Add your initialization logic here

    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();

    Window.Title = "ATLS 4519 Lab 7 - Terrain Tutorial";

    base.Initialize();
}

/// <summary>
/// LoadContent will be called once per game and is the place to load
/// all of your content.
/// </summary>
protected override void LoadContent()
{
    device = graphics.GraphicsDevice;
    effect = Content.Load<Effect>("effects");

    SetUpVertices();

    // Create a new SpriteBatch, which can be used to draw textures.
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // TODO: use this.Content to load your game content here
}

/// <summary>
/// UnloadContent will be called once per game and is the place to unload
/// all content.
/// </summary>
protected override void UnloadContent()
{
    // TODO: Unload any non ContentManager content here
}

/// <summary>
/// Allows the game to run logic such as updating the world,
/// checking for collisions, gathering input, and playing audio.
/// </summary>
/// <param name="gameTime">Provides a snapshot of timing values.</param>
protected override void Update(GameTime gameTime)
{
```

```
            // Allows the game to exit
            if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
                this.Exit();

            // TODO: Add your update logic here

            base.Update(gameTime);
        }

        /// <summary>
        /// This is called when the game should draw itself.
        /// </summary>
        /// <param name="gameTime">Provides a snapshot of timing values.</param>
        protected override void Draw(GameTime gameTime)
        {
            device.Clear(Color.DarkSlateBlue);

            effect.CurrentTechnique = effect.Techniques["Pretransformed"];

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                // All your drawing code must be put after the call to pass.Apply()
                pass.Apply();
                device.DrawUserPrimitives(PrimitiveType.TriangleList,
                    vertices, 0, 1, VertexPositionColor.VertexDeclaration);

            }



            // TODO: Add your drawing code here

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

Optional - Test your knowledge:

1. Make the bottom-left corner of the triangle collide with the bottom-left corner of your window.
2. Render 2 triangles, covering your whole window.

Using a Custom Effect and World Coordinates

We just drew a triangle, using 'pretransformed' coordinates. These coordinates allowed us to directly specify their position on the screen. However, we will usually use the untransformed coordinates, the so-called “World Space” coordinates, which we specify in 3D coordinates. These allow us to create a whole scene using simple 3D coordinates, and, importantly, to position a camera through which the user will look at the scene.

Let’s start by redefining our triangle coordinates in 3D world space. Replace the code in the SetUpVertices method with this code:

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

    vertices[0].Position = new Vector3(0f, 0f, 0f);
```

```
    vertices[0].Color = Color.Red;
    vertices[1].Position = new Vector3(10f, 10f, 0f);
    vertices[1].Color = Color.Yellow;
    vertices[2].Position = new Vector3(10f, 0f, -5f);
    vertices[2].Color = Color.Green;
 }
```

As you can see, from here on we will be using the Z-coordinate as well. Because we are no longer using pretransformed screen coordinates (where the x and y coordinates are in the [-1, 1] region), we need to use a different technique from our effect file. Let's create one called 'ColoredNoShading', to reflect that we will be rendering a Colored 3D image, but will not yet specify any lighting or shading information.

Open the effects.fx file and insert the following code at the end of the file:

```
//------- Technique: ColoredNoShading --------
// No lighting or shading, so no normal info passed in

VertexToPixel ColoredNoShadingVS( float4 inPos : POSITION, float4 inColor: COLOR)
{
	VertexToPixel Output = (VertexToPixel)0;
	float4x4 preViewProjection = mul (xView, xProjection);
	float4x4 preWorldViewProjection = mul (xWorld, preViewProjection);

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

	return Output;
}

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

	Output.Color = PSIn.Color;

	return Output;
}

technique ColoredNoShading
{
	pass Pass0
	{
		VertexShader = compile vs_2_0 ColoredNoShadingVS();
		PixelShader  = compile ps_2_0 ColoredNoShadingPS();
	}
}
```

A shader (or effect) is a program that is executed on the GPU (Graphical Processing Unit). The GPU is located on the graphics card and usually contains several cores that can execute shader program parts concurrently. This allows the GPU to have high throughput (number of triangles that can be processed per second). Shaders (GPU programs) are specialized into 3 different types: vertex shaders, geometry shaders and pixel shaders. Vertex shaders are executed once per vertex. They transform 3D positions in space to 2D coordinates on the screen, and they manipulate position, color, and texture coordinates, but they cannot add new vertices. Geometry shaders can add or remove vertices from a mesh, procedurally generate geometry, or add detail to shapes. Pixel shaders (sometimes called fragment shaders) calculate the color of individual pixels. They are used to implement lighting, texturing, bump mapping, etc., and are applied (executed) once per pixel per geometric primitive. The C# XNA game code sends to the GPU the vertices and textures that represent the input for the vertex and pixel shader.

HLSL is a C-like language used to write effects in the XNA Game Studio. Defining an effect in HLSL requires (at least) three parts: VertexToPixel, PixelToFrame, and the actual technique declaration (in our case, ColoredNoShading).

Once we have created our effect we need to tell the program to use it. Make this adjustment in our Draw method:

```
effect.CurrentTechnique = effect.Techniques["ColoredNoShading"];
```

Try to run the code. Your triangle has disappeared again. Why? Because we have not told XNA where to position the camera in our 3D World, and where to point it! To position our camera, we need to define some matrices. Our triangle vertices are points in 3D space. Because our screen is 2D, our 3D points somehow need to be 'transformed' to 2D space. For our purposes, a matrix can be thought as a mathematical element that holds a transformation. If you multiply a 3D position by a matrix, you get the transformed position. Because there are a lot of properties that need to be defined when transforming our points from 3D world space to our 2D screen, the transformation is split in two steps, so we need two matrices, one that represents the view, and one that represents the projection. This will make more sense after we do it. So, first add these variables to the top of your class:

```
Matrix viewMatrix;
Matrix projectionMatrix;
```

Now we need to initialize these matrices, so add this code to our program:

```
private void SetUpCamera()
 {
     viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, 50),
              new Vector3(0, 0, 0), new Vector3(0, 1, 0));
     projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
              device.Viewport.AspectRatio, 1.0f, 300.0f);
 }
```

This code effectively defines the position and the lens settings of our camera. The first line creates a matrix that stores the position and orientation of the camera, through which we look at the scene. The first argument to Matrix.CreateLookAt defines the position of the camera. We position it 50 units on the positive Z axis. The next parameter sets the target point of the camera. We will be looking at (0,0,0), the 3D origin. After defining the viewing axis of our camera, we need to define it's rotation, i.e., which vector will be considered as 'up'.

The second line creates a matrix that stores how the camera looks at the scene, much like defining the lens settings of a real camera. The first argument sets the view angle, 45 degrees (Pi/4) in our case. Then we set the view aspect ratio, i.e., the ratio between the width and height of our screen. In our case (a square 500x500 window), this will equal 1, but the ratio will be different for other resolutions. The last two parameters define the viewing frustum. Any objects closer to the camera than the first argument (1.0), or farther away than the second argument (300.0), will not be displayed. These distances are called the near and the far clipping planes, since all objects not between these planes will be clipped (i.e., not drawn).

Now that we have created these matrices, we need to pass them to our technique, where they will be combined. This is accomplished by the following lines of code, which we need to add to our Draw method:

```
effect.Parameters["xView"].SetValue(viewMatrix);
effect.Parameters["xProjection"].SetValue(projectionMatrix);
effect.Parameters["xWorld"].SetValue(Matrix.Identity);
```

Finally, we need to call this method from within the LoadContent method:

```
SetUpCamera();
```

Now run the code. You should see a triangle, of which the bottom-right corner is not exactly below the top-right corner. This is because you have assigned the bottom-right corner a negative Z coordinate, positioning it a bit further away from

the camera than the other corners.

One important thing you should notice before you start experimenting: you'll see the green corner of the triangle is on the right side of the window, which seems normal because we defined it on the positive x-axis. So, if we position our camera on the negative z-axis, as follows:

```
viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, -50), new Vector3(0, 0, 0), new Vector3(0, 1, 0));
```

we would expect to see the green point in the left half of the window. Try this now. Where did our triangle go?

Recall that XNA culls (only draws triangles that are facing the camera) by default. XNA specifies that triangles facing the camera should be defined clockwise relative to the camera. If we position the camera on the negative z-axis, the corner points of the triangle in our vertices array will be defined counter-clockwise relative to the camera, and thus will not be drawn! Culling greatly improves performance, as it reduces the number of triangles to be drawn. However, when designing an application, it's better to turn culling off by putting these lines of code at the beginning of the Draw method:

```
RasterizerState rs = new RasterizerState();
rs.CullMode = CullMode.None;
device.RasterizerState = rs;
```

This will cause the GPU to draw all triangles, even those not facing the camera. Try this code now.

NOTE: It is unlikely that we would ever want to disable culling in a final product, because this would slow down the drawing process, thus degrading performance.

Now put the camera back to the positive part of the Z axis.

Rotating Triangle

Now we will make our triangle rotate. Since we are using world space coordinates, this is very easy to do. First, add a variable 'angle' to our class to store the current rotation angle:

```
private float angle = 0f;
```

Now, we will increase the rotation angle by 0.005f every frame. Modify the Update method as follows:

```
angle += 0.005f;
```

With our angle increasing automatically, all we have to do is to rotate the world coordinates, i.e., specify the rotation axis and the rotation angle. The rest is done by XNA! The rotation is stored in what is called the World matrix. In the Draw method, replace the line where we set our xWorld parameter with these two lines:

```
Matrix worldMatrix = Matrix.CreateRotationY(3 * angle);
effect.Parameters["xWorld"].SetValue(worldMatrix);
```

The first line creates the World matrix, which we create as a rotation around the Y axis. The second line passes this World matrix to the effect. From now on, everything we draw will be rotated along the Y axis by the amount currently stored in 'angle'!

Run this code now. When we run the application, you will see that the triangle is spinning around its (0,0,0) point! This is because the Y axis runs through this point, so the (0,0,0) point is the only point of our triangle that remains the same. Now imagine we would like to spin the triangle around the center of the triangle. One possibility is to redefine the triangle so the (0,0,0) would be in the center of our triangle. As easier solution is to first move (translate) the triangle a bit to the left and down, and then rotate it. To do this, we simply multiply our World matrix by a translation matrix in the Draw method, as follows:

```
Matrix worldMatrix = Matrix.CreateTranslation(-20.0f/3.0f, -10.0f / 3.0f, 0) *
             Matrix.CreateRotationZ(angle);
```

This will move the triangle so that its center point is at our (0,0,0) 3D World origin. Next, our triangle is rotated around this point, along the Z axis, giving us the desired result. Run this code.

Note the order in which these transformations are applied. If you place the translation AFTER the rotation, you will see a triangle rotating around one point, moved to the left and down. This is because matrix multiplications are NOT commutative, i.e., M1*M2 is not the same as M2*M1!

We can easily change the code to make the triangle rotate around the X or Y axis. Remember that one point of our triangle has a Z coordinate of -5, which explains why the triangle does not appear to rotate symmetrically.

We can also use Matrix CreateFromAxisAngle to specify a rotation about an arbitrary axis, as follows:

```
Vector3 rotAxis = new Vector3(3*angle, angle, 2*angle);
rotAxis.Normalize();
Matrix worldMatrix = Matrix.CreateTranslation(-20.0f/3.0f, -10.0f / 3.0f, 0) *
             Matrix.CreateFromAxisAngle(rotAxis, angle);
```

This code will make our triangle spin around an axis that continues to change. The first line defines this axis (which is changed every frame since it depends on the angle variable). The second line normalizes this axis, which is needed to make the CreateFromAxisAngle method work properly (Normalize() changes the coordinates of the vector, so the distance between the vector and the (0, 0, 0) point is exactly 1).

Here is our code so far:

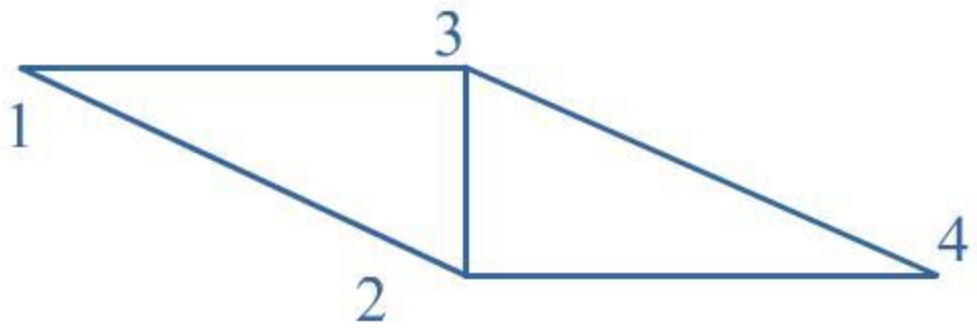
```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Audio;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.GamerServices;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;
using Microsoft.Xna.Framework.Media;

namespace ATLS_4519_Lab7
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        GraphicsDevice device;

        Effect effect;
        VertexPositionColor[] vertices;
        Matrix viewMatrix;
        Matrix projectionMatrix;
        private float angle = 0f;

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

        protected override void Initialize()
```

```
{
    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();
    Window.Title = "ATLS 4519 Lab 7 - Terrain Tutorial";

    base.Initialize();
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    device = graphics.GraphicsDevice;

    effect = Content.Load<Effect>("effects");
    SetUpCamera();

    SetUpVertices();
}

protected override void UnloadContent()
{
}

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

    vertices[0].Position = new Vector3(0f, 0f, 0f);
    vertices[0].Color = Color.Red;
    vertices[1].Position = new Vector3(10f, 10f, 0f);
    vertices[1].Color = Color.Yellow;
    vertices[2].Position = new Vector3(10f, 0f, -5f);
    vertices[2].Color = Color.Green;
}

private void SetUpCamera()
{
    // Camera on positive Z axis
    viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, 50), new Vector3(0, 0, 0), new
            Vector3(0, 1, 0));
    // Camera on negative Z axis
    // viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, -50), new Vector3(0, 0, 0), new
            Vector3(0, 1, 0));

    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
            device.Viewport.AspectRatio, 1.0f, 300.0f);
}

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

    angle += 0.005f;

    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
```

```
        {
            device.Clear(Color.DarkSlateBlue);

            // Uncomment the Code below to show triangles facing away from the camera
            RasterizerState rs = new RasterizerState();
            rs.CullMode = CullMode.None;
            device.RasterizerState = rs;

            // Create a rotation
            // Matrix worldMatrix = Matrix.CreateRotationY(3 * angle);

            // Create a translation and rotation
            // Matrix worldMatrix = Matrix.CreateTranslation(-20.0f / 3.0f, -10.0f / 3.0f, 0) *
                    Matrix.CreateRotationZ(3 * angle);

            // Rotate about a changing arbitrary axis
            Vector3 rotAxis = new Vector3(3 * angle, angle, 2 * angle);
            rotAxis.Normalize();
            Matrix worldMatrix = Matrix.CreateTranslation(-20.0f / 3.0f, -10.0f / 3.0f, 0) *
                    Matrix.CreateFromAxisAngle(rotAxis, angle);

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

            effect.Parameters["xWorld"].SetValue(worldMatrix);
            // effect.Parameters["xWorld"].SetValue(Matrix.Identity);
            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();

                device.DrawUserPrimitives(PrimitiveType.TriangleList, vertices, 0, 1,
                    VertexPositionColor.VertexDeclaration);
            }

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

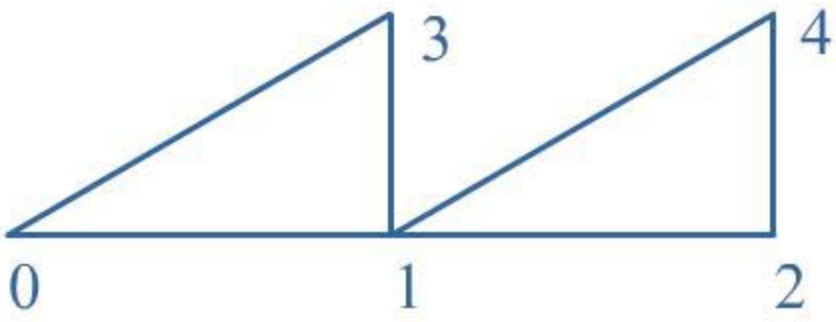
From One to Many Triangles - Using Indices

Now that we know how to draw one triangle, we will learn how to draw a lot of triangles. Each triangle requires us, at least in principle, to specify three vertices. Consider the following example:



In this example, only four out of the six vertices are unique. The other two are redundant. One approach would be to define the four unique vertices in an array indexed 0 to 3, and to define the first triangle as vertices 1, 2 and 3 and the

second triangle as vertices 2, 3 and 4. This way, redundant vertex data is not duplicated. This is the idea behind indices. Suppose we would like to draw the two triangles below:



Normally we would have to define six vertices; now we only need to define five. Let's change our SetUpVertices method to use this approach, as follows:

```
private void SetUpVertices()
 {
     vertices = new VertexPositionColor[5];

     vertices[0].Position = new Vector3(0f, 0f, 0f);
     vertices[0].Color = Color.White;
     vertices[1].Position = new Vector3(5f, 0f, 0f);
     vertices[1].Color = Color.White;
     vertices[2].Position = new Vector3(10f, 0f, 0f);
     vertices[2].Color = Color.White;
     vertices[3].Position = new Vector3(5f, 0f, -5f);
     vertices[3].Color = Color.White;
     vertices[4].Position = new Vector3(10f, 0f, -5f);
     vertices[4].Color = Color.White;
 }
```

Vertices 0 to 2 are positioned on the positive X axis. Vertices 3 and 4 have a negative Z component, as XNA considers the negative Z axis to be 'Forward'. Since our vertices are defined along the Right and the Forward directions, the resulting triangles will appear to be lying flat on the ground.

Next, we will create a list of indices. As suggested above, the indices will refer to vertices in our array of vertices. The indices define the triangles, so for two triangles we will need to define six indices. We will begin by defining this array at the top of your (Game1.cs) class. Since indices are integers, we need to define an array capable of storing integers:

```
int[] indices;
```

Now create another method that fills the array of indices.

```
private void SetUpIndices()
 {
     indices = new int[6];

     indices[0] = 3;
     indices[1] = 1;
     indices[2] = 0;
     indices[3] = 4;
     indices[4] = 2;
     indices[5] = 1;
```

```
}
```

This method defines six indices, thus defining two triangles. Vertex Number 1 is used twice, as we can see it should be from the image above. In this example, the savings are rather small, but in larger applications (as we will see soon) the savings can be quite substantial. Also, note that the triangles have been defined in a clockwise order, so XNA will see them as facing the camera and will not cull them away.

Now make sure that we call `SetUpIndices()` in the LoadContent method :

```
SetUpIndices();
```

All that's left is to draw the triangles from our buffer. To accomplish this, change the following line in our Draw method:

```
device.DrawUserIndexedPrimitives(PrimitiveType.TriangleList, vertices, 0, vertices.Length, indices, 0,
            indices.Length / 3, VertexPositionColor.VertexDeclaration);
```

Instead of using the DrawUserPrimitives method, this time we call the DrawUserIndexedPrimitives method. This allows us to specify both an array of vertices and an array of indices. The second-to-last argument specifies how many triangles are defined by the indices. Since one triangle is defined by three indices, we specify the number of indices divided by 3.

Before you try this code, let's make our triangles stop rotating by resetting their World matrix to the Identity matrix, as follows:

```
Matrix worldMatrix = Matrix.Identity;
```

Run this code now. There will be not much to see. This is because both our triangles and our camera are positioned on the floor! We will have to reposition the camera so the triangles are in sight of the camera. So, in our SetUpCamera method, change the position of the camera so it is positioned above the (0,0,0) 3D origin, as follows:

```
viewMatrix = Matrix.CreateLookAt(new Vector3(0, 50, 0), new Vector3(0, 0, 0), new Vector3(0, 0, -1));
```

We have positioned the camera 50 units above the (0,0,0) 3D origin, as the Y axis is considered as the Up axis by XNA. However, because the camera is looking down, we can no longer specify the (0,1,0) Up vector as he Up vector for the camera. Therefore, we specify the (0,0,-1) Forward vector as Up vector for the camera.

Now when you run this code, you should see both triangles, but they are still solid. Try changing this property to our RenderState by adding this line in the Draw method beneath rs.CullMode:

```
rs.FillMode = FillMode.WireFrame;
```

This will only draw the edges of our triangles, instead of solid triangles.

Here is our code so far:

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
```

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

    Effect effect;
    VertexPositionColor[] vertices;
    int[] indices;
    Matrix viewMatrix;
    Matrix projectionMatrix;
    private float angle = 0f;

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

    private void SetUpVertices()
    {
        vertices = new VertexPositionColor[5];

        vertices[0].Position = new Vector3(0f, 0f, 0f);
        vertices[0].Color = Color.White;
        vertices[1].Position = new Vector3(5f, 0f, 0f);
        vertices[1].Color = Color.White;
        vertices[2].Position = new Vector3(10f, 0f, 0f);
        vertices[2].Color = Color.White;
        vertices[3].Position = new Vector3(5f, 0f, -5f);
        vertices[3].Color = Color.White;
        vertices[4].Position = new Vector3(10f, 0f, -5f);
        vertices[4].Color = Color.White;
    }

    private void SetUpIndices()
    {
        indices = new int[6];

        indices[0] = 3;
        indices[1] = 1;
        indices[2] = 0;
        indices[3] = 4;
        indices[4] = 2;
        indices[5] = 1;
    }

    private void SetUpCamera()
    {
        // Camera on positive Z axis
        // viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, 50), new Vector3(0, 0, 0), new
                Vector3(0, 1, 0));

        // Camera on negative Z axis
        // viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, -50), new Vector3(0, 0, 0), new
                Vector3(0, 1, 0));

        // Position camera 50 units above the (0,0,0) 3D origin,
        // as the Y axis is considered as Up axis by XNA. However,
        // because the camera is looking down, we can no longer
        // specify the (0,1,0) Up vector as Up vector for the camera!
        // Therefore, you specify the (0,0,-1) Forward vector as Up vector
```

```
    // for the camera.
    viewMatrix = Matrix.CreateLookAt(new Vector3(0, 50, 0), new Vector3(0, 0, 0), new
            Vector3(0, 0, -1));


    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
            device.Viewport.AspectRatio, 1.0f, 300.0f);
}

protected override void Initialize()
{
    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();
    Window.Title = "ATLS_4519_Lab7 - Terrain Tutorial";

    base.Initialize();
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    device = graphics.GraphicsDevice;

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

    SetUpCamera();
    SetUpVertices();
    SetUpIndices();
}

protected override void UnloadContent()
{
}

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

    angle += 0.005f;

    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    device.Clear(Color.DarkSlateBlue);

    // Uncomment the Code below to show triangles facing away from the camera
    RasterizerState rs = new RasterizerState();
    rs.CullMode = CullMode.None;
    rs.FillMode = FillMode.WireFrame;
    device.RasterizerState = rs;

    // Create a rotation
    // Matrix worldMatrix = Matrix.CreateRotationY(3 * angle);

    // Create a translation and rotation
```

```
            // Matrix worldMatrix = Matrix.CreateTranslation(-20.0f / 3.0f, -10.0f / 3.0f, 0) *
                    Matrix.CreateRotationZ(3 * angle);

            // Rotate about a changing arbitrary axis
            // Vector3 rotAxis = new Vector3(3 * angle, angle, 2 * angle);
            // rotAxis.Normalize();
            // Matrix worldMatrix = Matrix.CreateTranslation(-20.0f / 3.0f, -10.0f / 3.0f, 0) *
                    Matrix.CreateFromAxisAngle(rotAxis, angle);

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

            //effect.Parameters["xWorld"].SetValue(worldMatrix);
            effect.Parameters["xWorld"].SetValue(Matrix.Identity);

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();

                // device.DrawUserPrimitives(PrimitiveType.TriangleList,
                //     vertices, 0, 1, VertexPositionColor.VertexDeclaration);

                device.DrawUserIndexedPrimitives(PrimitiveType.TriangleList,
                            vertices, 0, vertices.Length, indices, 0, indices.Length / 3,
                           VertexPositionColor.VertexDeclaration);

            }

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

NOTE: If no triangles are visible, your graphics card may not be capable of rendering from many indices. To solve this, store your indices as shorts instead of ints, as follows:

```
short[] indices;
```

And in the SetUpIndices method, initialize it as follows:

```
indices = new short[6];
```

This should work, even on PCs with lower-end graphics cards.

Optional - Test your knowledge:

1. Try to render the triangle that connects vertices 1, 3 and 4. The only changes you need to make are in the SetUpIndices method.
2. Use another vector, such as (1,0,0) as Up vector for your camera's View matrix.

Rendering Terrain

We are finally ready to start creating something that might be considered terrain. Let's start small, by connecting a 4x3 grid of specified points. However, we will make our engine dynamic, so next chapter we can easily load a much larger number of points. To do this, we have to create two new variables in our Game1.cs class:

```
private int terrainWidth = 4;
```

```
private int terrainLength = 3;
```

We will assume the 4x3 points are equidistant. So the only thing we don't know about our points is the value of each point's Z coordinate. We will use an array to hold this information, so we also need to add this line to the top of our class:

```
private float[,] heightData;
```

For now, use the following method to fill this array:

```
private void LoadHeightData()
{
    heightData = new float[4, 3];
    heightData[0, 0] = 0;
    heightData[1, 0] = 0;
    heightData[2, 0] = 0;
    heightData[3, 0] = 0;

    heightData[0, 1] = 0.5f;
    heightData[1, 1] = 0;
    heightData[2, 1] = -1.0f;
    heightData[3, 1] = 0.2f;

    heightData[0, 2] = 1.0f;
    heightData[1, 2] = 1.2f;
    heightData[2, 2] = 0.8f;
    heightData[3, 2] = 0;
}
```

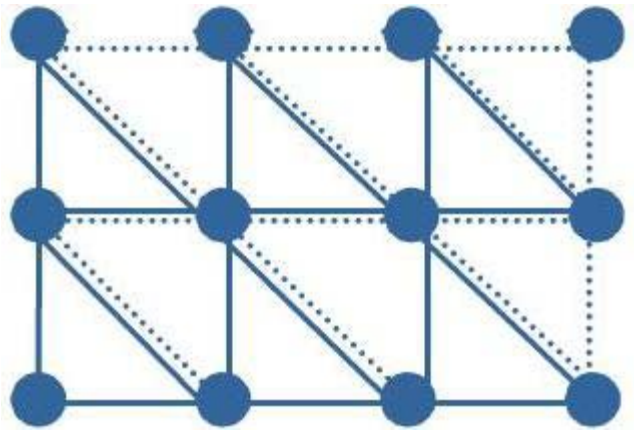
Now call this method from within our LoadContent method. Make sure to call it before the SetUpVertices method, since that method needs to use the contents of the heightData.

```
LoadHeightData();
```

With our height array filled, we can now create our vertices. Since we have a 4x3 terrain, twelve (terrainWidth*terrainLength) vertices are needed. The points are equidistant (the distance between them is the same), so we can easily change our SetUpVertices method. At first, we will not be using the Z coordinate.

```
private void SetUpVertices()
{
    vertices = new VertexPositionColor[terrainWidth * terrainLength];
    for (int x = 0; x < terrainWidth; x++)
    {
        for (int y = 0; y < terrainLength; y++)
        {
            vertices[x + y * terrainWidth].Position = new Vector3(x, 0, -y);
            vertices[x + y * terrainWidth].Color = Color.White;
        }
    }
}
```

There is nothing magical going on here, we simply define our 12 points and make them white. Note that the terrain will grow into the positive X direction (Right) and into the negative Z direction (Forward). Next comes a more difficult part: defining the indices that define the required triangles to connect the 12 vertices. The best way to do this is by creating two sets of vertices, as shown in the figure below. This will make more sense in a bit.



We'll start by drawing the set of triangles shown in solid lines. To do this, change our SetUpIndices as follows:

```
private void SetUpIndices()
 {
     indices = new int[(terrainWidth - 1) * (terrainLength - 1) * 3];
     int counter = 0;
     for (int y = 0; y < terrainLength - 1; y++)
     {
         for (int x = 0; x < terrainWidth - 1; x++)
         {
             int lowerLeft = x + y*terrainWidth;
             int lowerRight = (x + 1) + y*terrainWidth;
             int topLeft = x + (y + 1) * terrainWidth;
             int topRight = (x + 1) + (y + 1) * terrainWidth;

             indices[counter++] = topLeft;
             indices[counter++] = lowerRight;
             indices[counter++] = lowerLeft;
         }
     }
 }
```

Recall that terrainWidth and terrainLength are the horizontal and vertical number of vertices in our (small) terrain. We will need two rows of three triangles, giving six triangles. These will require 3*2 * 3 = 18 indices ((terrainWidth-1)*(terrainLength-1)*3), thus the first line creates an array capable of storing exactly this number of indices.

Then, we fill our array with indices. We scan the X and Y coordinates row by row, creating our triangles. During the first row pass, where y=0, we need to create six triangles from vertices 0 (bottom-left) to 7 (middle-right). Next, y becomes 1 and 1*terrainWidth=4 is added to each index: this time we create our six triangles from vertices 4 (middle-left) to 11 (top-right).

To make things easier, we first define four shortcuts for the four corner indices of a quad. For each quad we store three indices, defining one triangle. Remember culling? It requires us to define the vertices in clockwise order. So first we define the top-left vertex, then the bottom-down vertex and the bottom-left vertex.

The *counter* variable is an easy way to store vertices to an array, as we increment it each time an index has been added to the array. When the method finishes, the array will contain all indices required to render all bottom-left triangles.

We've coded our Draw method in such a way that XNA draws a number of triangles specified by the length of our indices array, so we can immediately run this code!

The triangles look tiny as rendered, so try positioning our camera at (0,10,0) and rerun the program. You should see six triangles in the right half of our window, every point of every triangle at the same Z coordinate. Now alter the vertices

position computation in SetUpVertices to change the height of our points according to our heightData array:

```
vertices[x + y * terrainWidth].Position = new Vector3(x, heightData[x,y], -y);
```

Run this code; you will notice the triangles are no longer positioned in the same plane.

Remember that we are still rendering only the bottom-left triangles. So if we render the triangles with their solid colors instead of only their wireframes, 50% of our grid would not be covered. To fix this, let's define some more indices to render the top-right triangles as well. We need the same vertices, so the only thing we have to change is the SetUpIndices method, as follows:

```
private void SetUpIndices()
{
    indices = new int[(terrainWidth - 1) * (terrainLength - 1) * 6];
    int counter = 0;
    for (int y = 0; y < terrainLength - 1; y++)
    {
        for (int x = 0; x < terrainWidth - 1; x++)
        {
            int lowerLeft = x + y*terrainWidth;
            int lowerRight = (x + 1) + y*terrainWidth;
            int topLeft = x + (y + 1) * terrainWidth;
            int topRight = (x + 1) + (y + 1) * terrainWidth;

            indices[counter++] = topLeft;
            indices[counter++] = lowerRight;
            indices[counter++] = lowerLeft;

            indices[counter++] = topLeft;
            indices[counter++] = topRight;
            indices[counter++] = lowerRight;
        }
    }
}
```

We will be drawing twice as many vertices now;, that is why the "*3" has been replaced by "*6". You see the second set of triangles also has been drawn clockwise relative to the camera: first the top-left corner, then the top-right and finally the bottom-right.

Running this code will give us a better 3 dimensional view. Since we have taken care to only use the variables terrainWidth and terrainLength, these are the only things we need change to increase the size of our terrain map, together with the contents of the heightData array. It would be nice to find a mechanism to fill this last one automatically, which we will do next.

Here is our code so far:

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
```

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

    Effect effect;
    VertexPositionColor[] vertices;
    int[] indices;
    Matrix viewMatrix;
    Matrix projectionMatrix;
    private float angle = 0f;

    private int terrainWidth = 4;
    private int terrainLength = 3;
    private float[,] heightData;

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

    private void SetUpVertices()
    {
        vertices = new VertexPositionColor[terrainWidth * terrainLength];
        for (int x = 0; x < terrainWidth; x++)
        {
            for (int y = 0; y < terrainLength; y++)
            {
                //vertices[x + y * terrainWidth].Position = new Vector3(x, 0, -y);
                vertices[x + y * terrainWidth].Position = new Vector3(x, heightData[x, y], -y);
                vertices[x + y * terrainWidth].Color = Color.White;
            }
        }
    }

    private void SetUpIndices()
    {
        indices = new int[(terrainWidth - 1) * (terrainLength - 1) * 6];
        int counter = 0;
        for (int y = 0; y < terrainLength - 1; y++)
        {
            for (int x = 0; x < terrainWidth - 1; x++)
            {
                int lowerLeft = x + y * terrainWidth;
                int lowerRight = (x + 1) + y * terrainWidth;
                int topLeft = x + (y + 1) * terrainWidth;
                int topRight = (x + 1) + (y + 1) * terrainWidth;

                indices[counter++] = topLeft;
                indices[counter++] = lowerRight;
                indices[counter++] = lowerLeft;

                indices[counter++] = topLeft;
                indices[counter++] = topRight;
                indices[counter++] = lowerRight;
            }
        }
    }
```

```
private void SetUpCamera()
{
    // Camera on positive Z axis
    // viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, 50), new Vector3(0, 0, 0), new
            Vector3(0, 1, 0));

    // Camera on negative Z axis
    // viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, -50), new Vector3(0, 0, 0), new
            Vector3(0, 1, 0));

    // Position camera 50 units above the (0,0,0) 3D origin,
    // as the Y axis is considered as Up axis by XNA. However,
    // because the camera is looking down, we can no longer
    // specify the (0,1,0) Up vector as Up vector for the camera!
    // Therefore, you specify the (0,0,-1) Forward vector as Up vector
    // for the camera.
    viewMatrix = Matrix.CreateLookAt(new Vector3(0, 10, 0), new Vector3(0, 0, 0), new
            Vector3(0, 0, -1));

    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
            device.Viewport.AspectRatio, 1.0f, 300.0f);
}

private void LoadHeightData()
{
    heightData = new float[4, 3];
    heightData[0, 0] = 0;
    heightData[1, 0] = 0;
    heightData[2, 0] = 0;
    heightData[3, 0] = 0;

    heightData[0, 1] = 0.5f;
    heightData[1, 1] = 0;
    heightData[2, 1] = -1.0f;
    heightData[3, 1] = 0.2f;

    heightData[0, 2] = 1.0f;
    heightData[1, 2] = 1.2f;
    heightData[2, 2] = 0.8f;
    heightData[3, 2] = 0;
}

protected override void Initialize()
{
    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();
    Window.Title = "ATLS_4519_Lab7 - Terrain Tutorial";

    base.Initialize();
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    device = graphics.GraphicsDevice;

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

    SetUpCamera();
```

```
    // Be sure to call LoadHeightData before SetUpVertices
    LoadHeightData();

    SetUpVertices();
    SetUpIndices();
}

protected override void UnloadContent()
{
}

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

    angle += 0.005f;

    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    device.Clear(Color.DarkSlateBlue);

    // Uncomment the Code below to show triangles facing away from the camera
    RasterizerState rs = new RasterizerState();
    rs.CullMode = CullMode.None;
    rs.FillMode = FillMode.WireFrame;
    device.RasterizerState = rs;

    // Create a rotation
    // Matrix worldMatrix = Matrix.CreateRotationY(3 * angle);

    // Create a translation and rotation
    // Matrix worldMatrix = Matrix.CreateTranslation(-20.0f / 3.0f, -10.0f / 3.0f, 0) *
            Matrix.CreateRotationZ(3 * angle);

    // Rotate about a changing arbitrary axis
    // Vector3 rotAxis = new Vector3(3 * angle, angle, 2 * angle);
    // rotAxis.Normalize();
    // Matrix worldMatrix = Matrix.CreateTranslation(-20.0f / 3.0f, -10.0f / 3.0f, 0) *
            Matrix.CreateFromAxisAngle(rotAxis, angle);

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

    //effect.Parameters["xWorld"].SetValue(worldMatrix);
    effect.Parameters["xWorld"].SetValue(Matrix.Identity);

    foreach (EffectPass pass in effect.CurrentTechnique.Passes)
    {
        pass.Apply();

        // device.DrawUserPrimitives(PrimitiveType.TriangleList,
        //     vertices, 0, 1, VertexPositionColor.VertexDeclaration);

        device.DrawUserIndexedPrimitives(PrimitiveType.TriangleList,
                    vertices, 0, vertices.Length, indices, 0, indices.Length / 3,
                   VertexPositionColor.VertexDeclaration);
```

```
            }

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

If you need to use shorts to access your indices, use the following code for SetUpIndices():

```
private void SetUpIndices()
{
    indices = new short[(terrainWidth - 1) * (terrainLength - 1) * 6];
    int counter = 0;
    for (int y = 0; y < terrainLength - 1; y++)
    {
        for (int x = 0; x < terrainWidth - 1; x++)
        {
            short lowerLeft = (short)(x + y * terrainWidth);
            short lowerRight = (short)((x + 1) + y * terrainWidth);
            short topLeft = (short)(x + (y + 1) * terrainWidth);
            short topRight = (short)((x + 1) + (y + 1) * terrainWidth);

            indices[counter++] = topLeft;
            indices[counter++] = lowerRight;
            indices[counter++] = lowerLeft;

            indices[counter++] = topLeft;
            indices[counter++] = topRight;
            indices[counter++] = lowerRight;
        }
    }
}
```

Optional - Test your knowledge:

1. Play around with the values of the heightData array.
2. Try to add an extra row to the grid.

Reading the HeightMap from a File

It's time to finally create a more realistic landscape. Instead of manually entering the data into our heightData array, we are going to obtain these data from a file. To do this, we are going to load a grayscale image of 128x128 pixels. We are going to use the 'white value' of every pixel as the height coordinate for the corresponding vertex. Begin by importing heightmap.bmp into your Content project, as we have done with other content.

Now we need to load image file in the LoadContent method. We will use a Texture2D variable for this purpose:

```
Texture2D heightMap = Content.Load<Texture2D> ("heightmap");
LoadHeightData(heightMap);
```

By using the default Content Pipeline, it doesn't matter whether you're using a .bmp, .jpg or .pgn file. The second line calls the LoadHeightData method (be sure to remove the extra call to LoadHeightData that used to appear in LoadContent), which we need to modify so that it processes our heightmap texture, as follows:

```
private void LoadHeightData(Texture2D heightMap)
```

```
{
    terrainWidth = heightMap.Width;
    terrainLength = heightMap.Height; // Texture2D uses the instance variable names Width and Height

    Color[] heightMapColors = new Color[terrainWidth * terrainLength];
    heightMap.GetData(heightMapColors);

    heightData = new float[terrainWidth, terrainLength];
    for (int x = 0; x < terrainWidth; x++)
        for (int y = 0; y < terrainLength; y++)
            heightData[x, y] = heightMapColors[x + y * terrainWidth].R / 5.0f; //5.0 is a scale factor
}
```

As you can see, this method receives the image as argument. Instead of using a predefined width and height for our terrain, we now use the resolution of our image. The first two lines read the width and length (unfortunately called Height because Texture2D uses the instance variable names Width and Height) of the image, and store them as width and length for the rest of our program. This will cause the rest of our code to automatically generate enough vertices and indices, and to render enough triangles. Because we want to access the color values of the pixels of the image, we need to create an array of Color objects, into which we will store the color of each pixel of the image, as shown.

Next, the code reshapes our 1D array of Colors into a 2D array of height data stored as floats. First we create a 2D array capable of storing these data. Next, we will cycle through all colors and select the Red value (notice the ".R"), which will be a value between 0 (no red) and 255 (fully red). Before we store this value inside the 2D array, we will scale it down by a factor of five (an arbitrarily chosen value), as otherwise our terrain would be too steep.

At this point, we have a 2D array containing the height for each point of our terrain. Our SetUpVertices method will generate a vertex for each point of the array. Our SetUpIndices method will generate three indices for each triangle that needs to be drawn to completely cover our terrain. Finally, our Draw method will render as many triangles as our indices array indicate. Woo hoo!

Run this code. Oops, we forgot to reposition our camera. Increase the height of our camera to 40; you should now see our terrain, but we will only see a corner. Let's move our terrain, so its center is shifted to the (0,0,0) 3D origin point. This can be done in the Draw method:

```
Matrix worldMatrix = Matrix.CreateTranslation(-terrainWidth / 2.0f, 0, terrainLength / 2.0f);
```

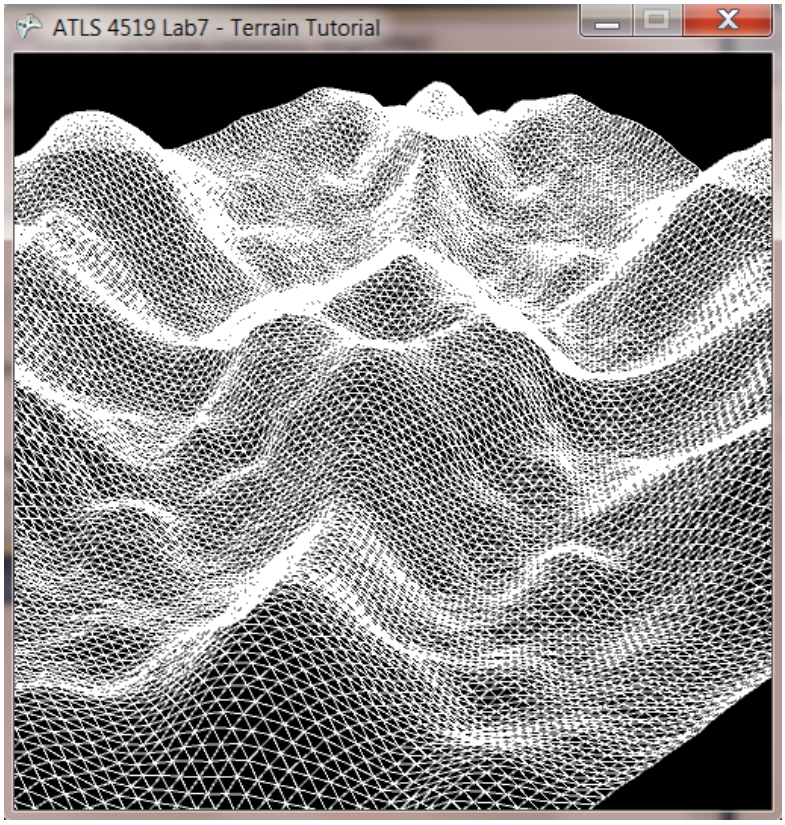
Now modify the code to use this world matrix:

```
effect.Parameters["xWorld"].SetValue(worldMatrix);
```

As the camera is still looking straight down on the terrain, we will get a better look by repositioning the camera (in the SetUpCamera method), as follows:

```
viewMatrix = Matrix.CreateLookAt(new Vector3(60, 80, -80), new Vector3(0, 0, 0), new Vector3(0, 1, 0));
```

When we set our clearing color to Color.Black, we should get the image shown below.



Optional - Test your knowledge:

1. This code scales the Red color value down by 5. Try other scaling values.
2. Open the heightmap.bmp image in Paint, and adjust some pixel values. Make some pixels pure red, others pure blue. Load this image into your program.
3. Find another image file and load it into your program. Look for images with a small resolution, as this approach will not allow you to render huge terrains.

Here is our code so far:

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        GraphicsDevice device;

        Effect effect;
        VertexPositionColor[] vertices;
        int[] indices;
        Matrix viewMatrix;
```

```
Matrix projectionMatrix;
private float angle = 0f;

private int terrainWidth = 4;
private int terrainLength = 3;
private float[,] heightData;

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

private void SetUpVertices()
{
    vertices = new VertexPositionColor[terrainWidth * terrainLength];
    for (int x = 0; x < terrainWidth; x++)
    {
        for (int y = 0; y < terrainLength; y++)
        {
            vertices[x + y * terrainWidth].Position = new Vector3(x, heightData[x, y], -y);
            vertices[x + y * terrainWidth].Color = Color.White;
        }
    }
}

private void SetUpIndices()
{
    indices = new int[(terrainWidth - 1) * (terrainLength - 1) * 6];
    int counter = 0;
    for (int y = 0; y < terrainLength - 1; y++)
    {
        for (int x = 0; x < terrainWidth - 1; x++)
        {
            int lowerLeft = x + y * terrainWidth;
            int lowerRight = (x + 1) + y * terrainWidth;
            int topLeft = x + (y + 1) * terrainWidth;
            int topRight = (x + 1) + (y + 1) * terrainWidth;

            indices[counter++] = topLeft;
            indices[counter++] = lowerRight;
            indices[counter++] = lowerLeft;

            indices[counter++] = topLeft;
            indices[counter++] = topRight;
            indices[counter++] = lowerRight;
        }
    }
}


private void SetUpCamera()
{
    viewMatrix = Matrix.CreateLookAt(new Vector3(60, 80, -80), new Vector3(0, 0, 0), new
            Vector3(0, 1, 0));
    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
            device.Viewport.AspectRatio, 1.0f, 300.0f);
}

private void LoadHeightData(Texture2D heightMap)
{
    terrainWidth = heightMap.Width;
```

```
    terrainLength = heightMap.Height;  // Texture2D uses instance var names Width and Height

    Color[] heightMapColors = new Color[terrainWidth * terrainLength];
    heightMap.GetData(heightMapColors);

    heightData = new float[terrainWidth, terrainLength];
    for (int x = 0; x < terrainWidth; x++)
        for (int y = 0; y < terrainLength; y++)
            heightData[x, y] = heightMapColors[x + y * terrainWidth].R / 5.0f;
}


protected override void Initialize()
{
    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();
    Window.Title = "ATLS 4519 Lab7 - Terrain Tutorial";

    base.Initialize();
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    device = graphics.GraphicsDevice;

    effect = Content.Load<Effect>("effects");
    Texture2D heightMap = Content.Load<Texture2D>("heightmap"); LoadHeightData(heightMap);

    SetUpCamera();

    // Be sure to call LoadHeightData before SetUpVertices
    // LoadHeightData();

    SetUpVertices();
    SetUpIndices();
}

protected override void UnloadContent()
{
}

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

    angle += 0.005f;

    base.Update(gameTime);
}

protected override void Draw(GameTime gameTime)
{
    device.Clear(Color.Black);

    // Uncomment the Code below to show triangles facing away from the camera
    RasterizerState rs = new RasterizerState();
    rs.CullMode = CullMode.None;
```

```
            rs.FillMode = FillMode.WireFrame;
            device.RasterizerState = rs;

            // Bring terrain to the center of our screen.
            Matrix worldMatrix = Matrix.CreateTranslation(-terrainWidth / 2.0f, 0, terrainLength /
                    2.0f);

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

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();

                // device.DrawUserPrimitives(PrimitiveType.TriangleList,
                //     vertices, 0, 1, VertexPositionColor.VertexDeclaration);

                device.DrawUserIndexedPrimitives(PrimitiveType.TriangleList,
                            vertices, 0, vertices.Length, indices, 0, indices.Length / 3,
                           VertexPositionColor.VertexDeclaration);

            }

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

Moving Terrain - Keyboard Input

Using XNA, it is very easy to read in the current state of the keyboard. The correct library, Microsoft.Xna.Framework.Input, was loaded by default when we started our XNA project, so we can immediately move on to the code that reads in the keyboard input, as follows.

Replace the line that updates our angle (`angle += 0.005f;`) with the following code in the Update method:

```
KeyboardState keyState = Keyboard.GetState();
 if (keyState.IsKeyDown(Keys.PageUp))
     angle += 0.05f;
 if (keyState.IsKeyDown(Keys.PageDown))
     angle -= 0.05f;
```

When the user presses the PageUp or PageDown key, the value of the 'angle' variable will be adjusted.

In our Draw method, we need to make sure that this rotation is incorporated into our World matrix:

```
Matrix worldMatrix = Matrix.CreateTranslation(-terrainWidth / 2.0f, 0, terrainLength / 2.0f) *
       Matrix.CreateRotationY(angle);
```

The terrain is thus rotated around the Y Up axis before it is moved to the (0,0,0) 3D origin. When you run the code, you can rotate the terrain simply by pressing the PageUp and PageDown keys.

On your own, add code that reads in the Arrow keys, and adjusts the World matrix so that the terrain is moved in the direction of the Arrow key being pressed. Let's also clean things up a bit to compute translation and rotation separately. Here is the code after we have done this:

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        GraphicsDevice device;

        Effect effect;
        VertexPositionColor[] vertices;
        int[] indices;
        Matrix viewMatrix;
        Matrix projectionMatrix;

        private int terrainWidth = 4;
        private int terrainLength = 3;
        private float[,] heightData;

        Matrix worldMatrix = Matrix.Identity; // World matrix identity
        Matrix worldTranslation = Matrix.Identity; // Translation
        Matrix worldRotation = Matrix.Identity; // Rotation

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

        private void SetUpVertices()
        {
            vertices = new VertexPositionColor[terrainWidth * terrainLength];
            for (int x = 0; x < terrainWidth; x++)
            {
                for (int y = 0; y < terrainLength; y++)
                {
                    vertices[x + y * terrainWidth].Position = new Vector3(x, heightData[x, y], -y);
                    vertices[x + y * terrainWidth].Color = Color.White;
                }
            }
        }

        private void SetUpIndices()
        {
            indices = new int[(terrainWidth - 1) * (terrainLength - 1) * 6];
            int counter = 0;
            for (int y = 0; y < terrainLength - 1; y++)
            {
                for (int x = 0; x < terrainWidth - 1; x++)
                {
                    int lowerLeft = x + y * terrainWidth;
                    int lowerRight = (x + 1) + y * terrainWidth;
                    int topLeft = x + (y + 1) * terrainWidth;
```

```
            int topRight = (x + 1) + (y + 1) * terrainWidth;

            indices[counter++] = topLeft;
            indices[counter++] = lowerRight;
            indices[counter++] = lowerLeft;

            indices[counter++] = topLeft;
            indices[counter++] = topRight;
            indices[counter++] = lowerRight;
        }
    }
}


private void SetUpCamera()
{
    viewMatrix = Matrix.CreateLookAt(new Vector3(60, 80, -80), new Vector3(0, 0, 0), new
            Vector3(0, 1, 0));
    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
            device.Viewport.AspectRatio, 1.0f, 300.0f);
}

private void LoadHeightData(Texture2D heightMap)
{
    terrainWidth = heightMap.Width;
    terrainLength = heightMap.Height;

    Color[] heightMapColors = new Color[terrainWidth * terrainLength];
    heightMap.GetData(heightMapColors);

    heightData = new float[terrainWidth, terrainLength];
    for (int x = 0; x < terrainWidth; x++)
        for (int y = 0; y < terrainLength; y++)
            heightData[x, y] = heightMapColors[x + y * terrainWidth].R / 5.0f;
}


protected override void Initialize()
{
    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();
    Window.Title = "ATLS 4519 Lab7  - Terrain Tutorial";

    base.Initialize();
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    device = graphics.GraphicsDevice;

    effect = Content.Load<Effect>("effects");
    Texture2D heightMap = Content.Load<Texture2D>("heightmap"); LoadHeightData(heightMap);

    SetUpCamera();

    SetUpVertices();
    SetUpIndices();
```

```
    worldMatrix *= Matrix.CreateTranslation(-terrainWidth / 2.0f, 0, terrainLength / 2.0f);
}

protected override void UnloadContent()
{
}

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

    KeyboardState keyState = Keyboard.GetState();

    //Rotation
    if (keyState.IsKeyDown(Keys.PageUp))
    {
        worldRotation = Matrix.CreateRotationY(0.1f);
    }
    else if (keyState.IsKeyDown(Keys.PageDown))
    {
        worldRotation = Matrix.CreateRotationY(-0.1f);
    }
    else
    {
        worldRotation = Matrix.CreateRotationY(0);
    }

    //Translation
    if (keyState.IsKeyDown(Keys.Left))
    {
        worldTranslation = Matrix.CreateTranslation(-.1f, 0, 0);
    }
    else if (keyState.IsKeyDown(Keys.Right))
    {
        worldTranslation = Matrix.CreateTranslation(.1f, 0, 0);
    }
    else if (keyState.IsKeyDown(Keys.Up))
    {
        worldTranslation = Matrix.CreateTranslation(0, .1f, 0);
    }
    else if (keyState.IsKeyDown(Keys.Down))
    {
        worldTranslation = Matrix.CreateTranslation(0, -.1f, 0);
    }
    else if (keyState.IsKeyDown(Keys.Q))
    {
        worldTranslation = Matrix.CreateTranslation(0, 0, .1f);
    }
    else if (keyState.IsKeyDown(Keys.Z))
    {
        worldTranslation = Matrix.CreateTranslation(0, 0, -.1f);
    }
    else
    {
        worldTranslation = Matrix.CreateTranslation(0, 0, 0);
    }

    worldMatrix *= worldTranslation * worldRotation;

    base.Update(gameTime);
}
```

```
        protected override void Draw(GameTime gameTime)
        {
            device.Clear(Color.Black);

            // Uncomment the Code below to show triangles facing away from the camera
            RasterizerState rs = new RasterizerState();
            rs.CullMode = CullMode.None;
            rs.FillMode = FillMode.WireFrame;
            device.RasterizerState = rs;

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

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();

                // device.DrawUserPrimitives(PrimitiveType.TriangleList,
                //     vertices, 0, 1, VertexPositionColor.VertexDeclaration);

                device.DrawUserIndexedPrimitives(PrimitiveType.TriangleList,
                            vertices, 0, vertices.Length, indices, 0, indices.Length / 3,
                           VertexPositionColor.VertexDeclaration);

            }

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

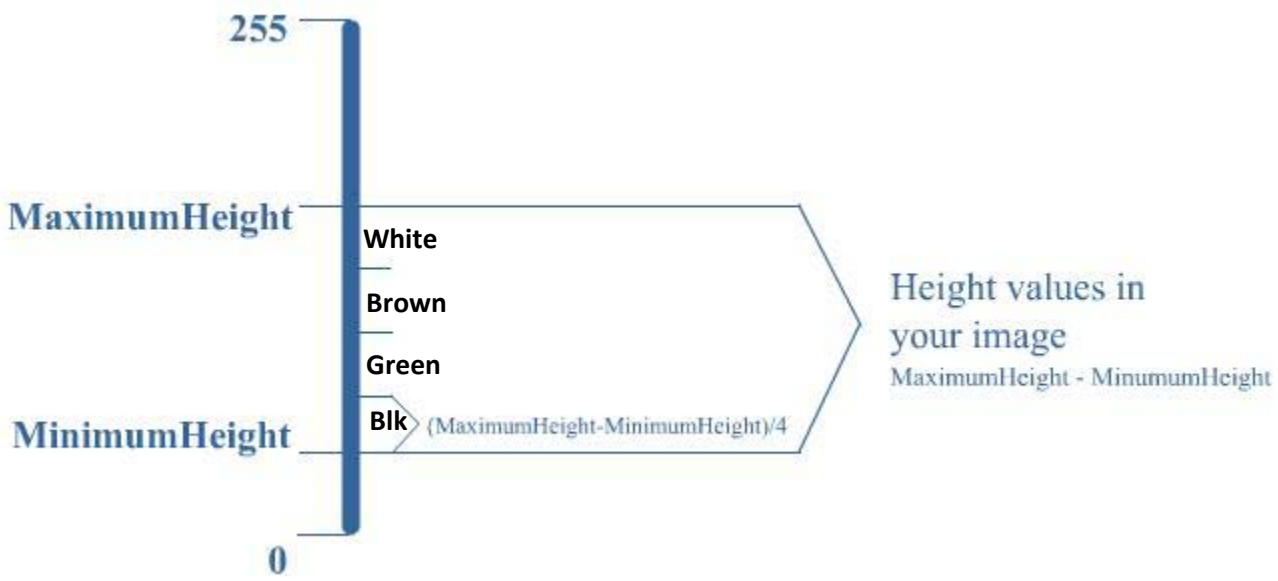
Adding Color to Terrain

Now we will color our terrain based upon height. For example, at the lowest level we might have blue lakes, then the green trees, the brown mountain and finally snow topped peaks. We will extend our SetUpVertices method to store the correct colors in each vertex. However, we cannot expect every image to have a lake at height 0, and a mountain peak at height 255 (the maximum value for a .bmp pixel). Imagine an image with height values only between 50 and 200, this image would then probably produce a terrain without any lakes or snow topped peaks. How do we fix this?

To respond to this situation, we will first detect the minimum and maximum heights in our image. We will then store these in the minHeight and maxHeight variables. Add the following code at the top of the SetUpVertices method:

```
float minHeight = float.MaxValue;
float maxHeight = float.MinValue;
for (int x = 0; x < terrainWidth; x++)
{
     for (int y = 0; y < terrainLength; y++)
     {
         if (heightData[x, y] < minHeight)
             minHeight = heightData[x, y];
         if (heightData[x, y] > maxHeight)
             maxHeight = heightData[x, y];
     }
}
```

Now we will check each point of our grid to see if that point's height is below the current minHeight or above the current maxHeight. If it is, store the current height in the corresponding variable. Now that we know minHeight and maxHeight, we can specify the 4 color regions, as shown in the figure below:



The ideas is to compute the total range (maxHeight - minHeight), and divide that range by 4. This gives us the size of each color range. Now when we declare our vertices and their colors, we are going to assign the desired color to the correct height regions as follows: (in the "for (int y = 0; y < terrainLength; y++)" loop)

```
vertices[x + y * terrainWidth].Position = new Vector3(x, heightData[x, y], -y);
if (heightData[x, y] < minHeight + (maxHeight - minHeight) / 4)
       vertices[x + y * terrainWidth].Color = Color.Blue;
else if (heightData[x, y] < minHeight + (maxHeight - minHeight) * 2 / 4)
       vertices[x + y * terrainWidth].Color = Color.Green;
else if (heightData[x, y] < minHeight + (maxHeight - minHeight) * 3 / 4)
       vertices[x + y * terrainWidth].Color = Color.Brown;
else
       vertices[x + y * terrainWidth].Color = Color.White;
```

Run the code at this point. You should see a nicely colored network of lines. If we want to see the whole colored terrain, we just have to remove this line (or set it to FillMode.Solid) in the Draw method:

```
rs.FillMode = FillMode.WireFrame;
```

Run the code, and rotate the terrain a couple of times. On *some* computers, you will see that sometimes the middle peaks get overdrawn by the 'invisible' lake behind it. This is because we have not yet defined a 'Z-buffer'. This Z buffer is an array where our video card keeps track of the depth coordinate of every pixel that should be drawn on your screen (in our case, a 500x500 array). Every time our card receives a triangle to draw, it checks whether the triangle's pixels are closer to the screen than the pixels already present in the Z-buffer. If they are closer, the Z-buffer's contents is updated with these pixels for that region.

This whole process if fully automated. All we have to do is to initialize our Z buffer with the largest possible distance to start with. So we first fill our buffer with ones. To do this automatically every update of our screen, change the following line in the Draw method:

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

(The | is a bitwise OR operator, in this case it means both the Target (the colors) as well as the DepthBuffer have to be cleared). Now everyone should see the terrain rotating as expected.

Here is code so far:

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        GraphicsDevice device;

        Effect effect;
        VertexPositionColor[] vertices;
        int[] indices;
        Matrix viewMatrix;
        Matrix projectionMatrix;

        private int terrainWidth = 4;
        private int terrainLength = 3;
        private float[,] heightData;

        Matrix worldMatrix = Matrix.Identity;
        Matrix worldTranslation = Matrix.Identity;
        Matrix worldRotation = Matrix.Identity;

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

        private void SetUpVertices()
        {
            float minHeight = float.MaxValue;
            float maxHeight = float.MinValue;
            for (int x = 0; x < terrainWidth; x++)
            {
                for (int y = 0; y < terrainLength; y++)
                {
                    if (heightData[x, y] < minHeight)
                        minHeight = heightData[x, y];
                    if (heightData[x, y] > maxHeight)
                        maxHeight = heightData[x, y];
                }
            }

            vertices = new VertexPositionColor[terrainWidth * terrainLength];
            for (int x = 0; x < terrainWidth; x++)
```

```
    {
        for (int y = 0; y < terrainLength; y++)
        {
            vertices[x + y * terrainWidth].Position = new Vector3(x, heightData[x, y], -y);
            if (heightData[x, y] < minHeight + (maxHeight - minHeight) / 4)
                vertices[x + y * terrainWidth].Color = Color.Blue;
            else if (heightData[x, y] < minHeight + (maxHeight - minHeight) * 2 / 4)
                vertices[x + y * terrainWidth].Color = Color.Green;
            else if (heightData[x, y] < minHeight + (maxHeight - minHeight) * 3 / 4)
                vertices[x + y * terrainWidth].Color = Color.Brown;
            else
                vertices[x + y * terrainWidth].Color = Color.White;

        }
    }
}

private void SetUpIndices()
{
    indices = new int[(terrainWidth - 1) * (terrainLength - 1) * 6];
    int counter = 0;
    for (int y = 0; y < terrainLength - 1; y++)
    {
        for (int x = 0; x < terrainWidth - 1; x++)
        {
            int lowerLeft = x + y * terrainWidth;
            int lowerRight = (x + 1) + y * terrainWidth;
            int topLeft = x + (y + 1) * terrainWidth;
            int topRight = (x + 1) + (y + 1) * terrainWidth;

            indices[counter++] = topLeft;
            indices[counter++] = lowerRight;
            indices[counter++] = lowerLeft;

            indices[counter++] = topLeft;
            indices[counter++] = topRight;
            indices[counter++] = lowerRight;
        }
    }
}


private void SetUpCamera()
{
    viewMatrix = Matrix.CreateLookAt(new Vector3(60, 80, -80), new Vector3(0, 0, 0), new
             Vector3(0, 1, 0));
    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
             device.Viewport.AspectRatio, 1.0f, 300.0f);
}

private void LoadHeightData(Texture2D heightMap)
{
    terrainWidth = heightMap.Width;
    terrainLength = heightMap.Height;

    Color[] heightMapColors = new Color[terrainWidth * terrainLength];
    heightMap.GetData(heightMapColors);

    heightData = new float[terrainWidth, terrainLength];
    for (int x = 0; x < terrainWidth; x++)
        for (int y = 0; y < terrainLength; y++)
            heightData[x, y] = heightMapColors[x + y * terrainWidth].R / 5.0f;
```

```
}


protected override void Initialize()
{
    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();
    Window.Title = "ATLS 4519 Lab7  - Terrain Tutorial";

    base.Initialize();
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    device = graphics.GraphicsDevice;

    effect = Content.Load<Effect>("effects");
    Texture2D heightMap = Content.Load<Texture2D>("heightmap"); LoadHeightData(heightMap);

    SetUpCamera();

    SetUpVertices();
    SetUpIndices();

    worldMatrix *= Matrix.CreateTranslation(-terrainWidth / 2.0f, 0, terrainLength / 2.0f);
}

protected override void UnloadContent()
{
}

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

    KeyboardState keyState = Keyboard.GetState();

    //Rotation
    if (keyState.IsKeyDown(Keys.PageUp))
    {
        worldRotation = Matrix.CreateRotationY(0.1f);
    }

    else if (keyState.IsKeyDown(Keys.PageDown))
    {
        worldRotation = Matrix.CreateRotationY(-0.1f);
    }
    else
    {
        worldRotation = Matrix.CreateRotationY(0);
    }

    //Translation
    if (keyState.IsKeyDown(Keys.Left))
    {
        worldTranslation = Matrix.CreateTranslation(-.1f, 0, 0);
    }
```

```
            else if (keyState.IsKeyDown(Keys.Right))
            {
                worldTranslation = Matrix.CreateTranslation(.1f, 0, 0);
            }
            else if (keyState.IsKeyDown(Keys.Up))
            {
                worldTranslation = Matrix.CreateTranslation(0, .1f, 0);
            }
            else if (keyState.IsKeyDown(Keys.Down))
            {
                worldTranslation = Matrix.CreateTranslation(0, -.1f, 0);
            }
            else if (keyState.IsKeyDown(Keys.Q))
            {
                worldTranslation = Matrix.CreateTranslation(0, 0, .1f);
            }
            else if (keyState.IsKeyDown(Keys.Z))
            {
                worldTranslation = Matrix.CreateTranslation(0, 0, -.1f);
            }
            else
            {
                worldTranslation = Matrix.CreateTranslation(0, 0, 0);
            }

            worldMatrix *= worldTranslation * worldRotation;

            base.Update(gameTime);
        }

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

            // Uncomment the Code below to show triangles facing away from the camera
            RasterizerState rs = new RasterizerState();
            rs.CullMode = CullMode.None;
            //rs.FillMode = FillMode.WireFrame;
            rs.FillMode = FillMode.Solid;
            device.RasterizerState = rs;

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

            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();

                device.DrawUserIndexedPrimitives(PrimitiveType.TriangleList,
                            vertices, 0, vertices.Length, indices, 0, indices.Length / 3,
                            VertexPositionColor.VertexDeclaration);

            }

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

Save a copy of your code at this point to a text file. We will need it again in a bit.

Basic Lighting

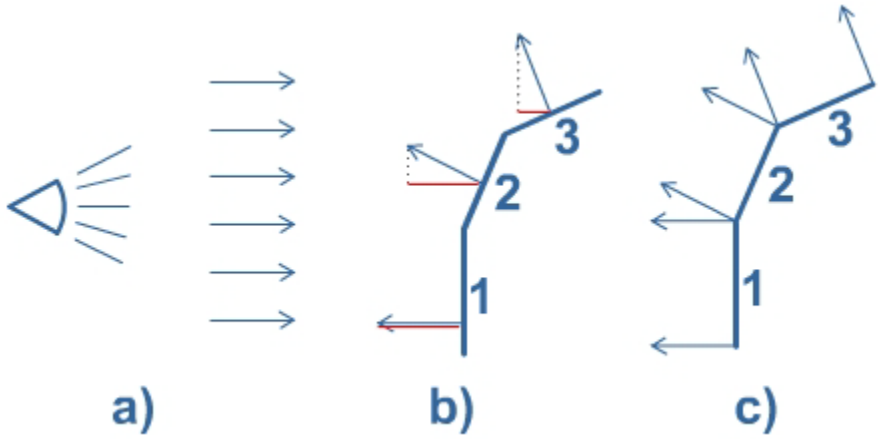
Even when using colors and a Z buffer, our terrain seems to miss some depth detail when we turn on the Solid FillMode. By adding some lighting, it will look much better. Before we do that however, we will see the impact of lighting and shading on two simple triangles, so we can have a better understanding of how lights work in XNA. We will be using the code from the 'World space' section, so reload that code now (you can find a copy in the Lab7_Content.zip file previously downloaded (filename: Game1.cs-Page38.txt); just replace the current Game1.cs with the contents of Game1.cs-Page38.txt), or use the code below.

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        GraphicsDevice device;

        Effect effect;
        VertexPositionColor[] vertices;
        Matrix viewMatrix;
        Matrix projectionMatrix;

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

        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 500;
            graphics.PreferredBackBufferHeight = 500;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 7 - Terrain Tutorial";

            base.Initialize();
        }

        protected override void LoadContent()
        {
            spriteBatch = new SpriteBatch(GraphicsDevice);

            device = graphics.GraphicsDevice;

            effect = Content.Load<Effect>("effects");
            SetUpCamera();

            SetUpVertices();
```

```
        }

        protected override void UnloadContent()
        {
        }

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

            vertices[0].Position = new Vector3(0f, 0f, 0f);
            vertices[0].Color = Color.Red;
            vertices[1].Position = new Vector3(10f, 10f, 0f);
            vertices[1].Color = Color.Yellow;
            vertices[2].Position = new Vector3(10f, 0f, -5f);
            vertices[2].Color = Color.Green;
        }

        private void SetUpCamera()
        {
            viewMatrix = Matrix.CreateLookAt(new Vector3(0, 0, 50), new Vector3(0, 0, 0), new
                    Vector3(0, 1, 0));
            projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
                    device.Viewport.AspectRatio, 1.0f, 300.0f);
        }

        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(Color.DarkSlateBlue);

            RasterizerState rs = new RasterizerState();
            rs.CullMode = CullMode.None;
            device.RasterizerState = rs;

            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.DrawUserPrimitives(PrimitiveType.TriangleList, vertices, 0, 1,
                    VertexPositionColor.VertexDeclaration);
            }

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

In this example, we will be using a single directional light, i.e., the light will travel from its source in one particular direction. The effect of the light hitting a triangle will depend upon the direction from which the light emanates relative to

the orientation of the triangle. Therefore, to calculate the effect of light hitting a triangle, XNA needs another input: the 'normal' in each of the triangles' vertices. Consider the figure below:



If the light source (a) shines it on the three surfaces shown (1, 2 and 3), how does XNA know that surface 1 should be lit more intensely than surface 3? If you look at the thin red lines in figure b), you will notice that their length appears to be a good indication of how much light we would want to be reflected (and thus seen) on every surface. So how can we calculate the length of these lines? Actually, graphics cards are very good at this kind of computation. All we have to do is give the blue arrow perpendicular (with an angle of 90 degrees, the thin blue lines) to every surface and XNA does the rest (a simple cosine computation) for us!

This is why we need to add normals (the perpendicular blue lines) to our vertex data. The VertexPositionColor is no longer adequate, as it does not allow us to store a normal for each vertex. Surprisingly, XNA does not offer a convenient structure that can contain a position, a color and a normal. However, we can easily create one of our own. Let's define this structure at the top of the Game1 class, immediately above our variable declarations:

```
public struct VertexPositionColorNormal
        {
            public Vector3 Position;
            public Color Color;
            public Vector3 Normal;

            public readonly static VertexDeclaration vertexDeclaration = new VertexDeclaration
            (
                new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
                new VertexElement(sizeof(float) * 3, VertexElementFormat.Color,
                            VertexElementUsage.Color, 0),
                new VertexElement(sizeof(float) * 3 + 4, VertexElementFormat.Vector3,
                            VertexElementUsage.Normal, 0)
            );
        }
```

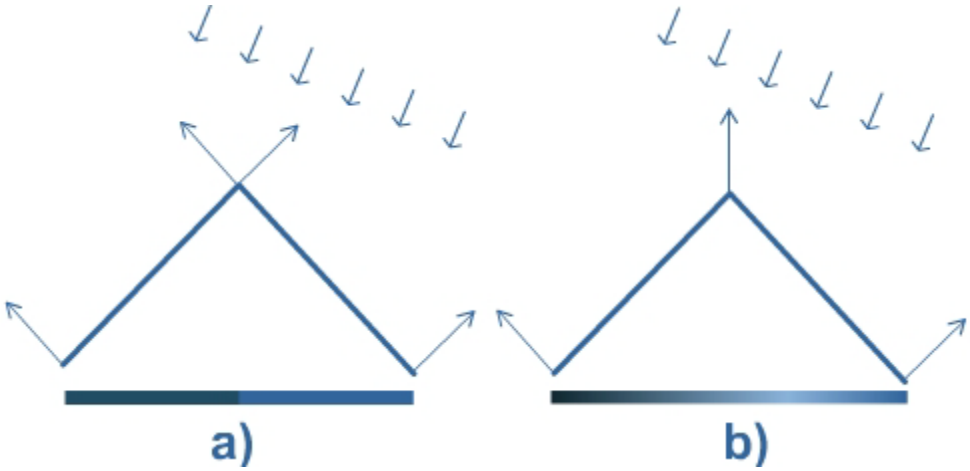
Now change our vertex variable declaration (below "Effect effect;") to use this structure:

```
VertexPositionColorNormal[] vertices;
```

Don't worry if XNA gives a "Cannot implicitly convert type […]" message at this point.

Now we can start defining triangles with normals. But first, study the figure below. The arrows at the top represent the direction of the light. The normals of each vertex are as shown, and the color bar below the drawing represents the color of every pixel along our surface. Notice that the top vertex of the triangle actually has two normal vectors, one for each

face.



If we simply use the perpendicular vectors as the normals, there will be an 'edge' in the lighting (see the bar directly above the ‘a’). This is because the right surface is lit more than the left surface. As a result, it will be easy to see the surface is made of separate triangles. However, if we replace the two normal vectors of the shared top vertex with a shared 'normal' as shown in figure b), XNA can automatically interpolate the lighting for every point of our surface. This will give a much smoother effect, as you can see in the bar above the b). This shared normal vector is simply the average of the two original normal vectors of a). The average of two vectors can be found by summing them and by dividing them by two.

To demonstrate this example, we will first reset the camera position:

```
viewMatrix = Matrix.CreateLookAt(new Vector3(0, -40, 100), new Vector3(0, 50, 0),
                    new Vector3(0, 1, 0));
```

Next, we will update our SetUpVertices method with 6 vertices that define the 2 triangles of the example above:

```
private void SetUpVertices()
        {
            vertices = new VertexPositionColorNormal[6];

            vertices[0].Position = new Vector3(0f, 0f, 40f);
            vertices[0].Color = Color.Blue;
            vertices[0].Normal = new Vector3(1, 0, 1);
            vertices[1].Position = new Vector3(40f, 0f, 0f);
            vertices[1].Color = Color.Blue;
            vertices[1].Normal = new Vector3(1, 0, 1);
            vertices[2].Position = new Vector3(0f, 40f, 40f);
            vertices[2].Color = Color.Blue;
            vertices[2].Normal = new Vector3(1, 0, 1);

            vertices[3].Position = new Vector3(-40f, 0f, 0f);
            vertices[3].Color = Color.Blue;
            vertices[3].Normal = new Vector3(-1, 0, 1);
            vertices[4].Position = new Vector3(0f, 0f, 40f);
            vertices[4].Color = Color.Blue;
            vertices[4].Normal = new Vector3(-1, 0, 1);
            vertices[5].Position = new Vector3(0f, 40f, 40f);
            vertices[5].Color = Color.Blue;
            vertices[5].Normal = new Vector3(-1, 0, 1);

            for (int i = 0; i < vertices.Length; i++)
                vertices[i].Normal.Normalize();
```

```
        }
```

This code defines two triangles. By adding a Z coordinate (other than 0), the triangles are now 3D. Notice that we have defined the normal vectors perpendicular to the triangles, as in example (a) of the figure above.

At the end, we "normalize" our "normals." These two words have absolutely nothing to do with each other. It means we scale our normal vectors so their lengths become exactly one. This is required to compute correct lighting, as the length of the normals has an impact on the amount of lighting.

Now change the Draw method to use our vertices with normal information:

```
device.DrawUserPrimitives(PrimitiveType.TriangleList, vertices, 0, 2,
       VertexPositionColorNormal.vertexDeclaration);
```

We also need to let the graphics card know that from now on it has to use the Colored technique (which uses the normal information to compute lighting and shading). This technique works exactly the same as the ColoredNoShading technique, but also adds correct lighting (in the cases where we provide normal information):

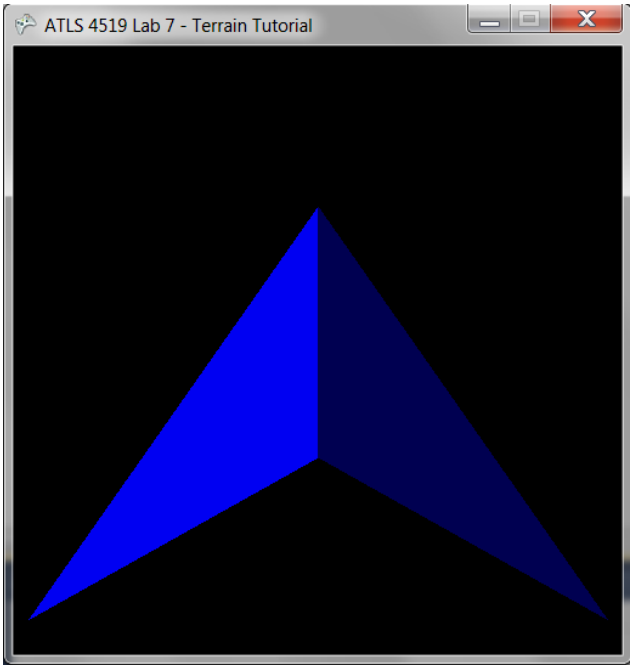
```
effect.CurrentTechnique = effect.Techniques["Colored"];
```

When you run this code, you should see an arrow (our 2 triangles), but you won't see any shading because we have not yet defined the light. We do this by setting the following additional parameters of our effect directly below where we just specified which technique to use:

```
effect.Parameters["xEnableLighting"].SetValue(true);
Vector3 lightDirection = new Vector3(0.5f, 0, -1.0f);
lightDirection.Normalize();
effect.Parameters["xLightDirection"].SetValue(lightDirection);
```

This instructs our technique to enable lighting calculations (now the technique needs the normals), and sets the direction of our light. Note again that we need to normalize this vector, so its lengths becomes one (otherwise the length of this vector influences the strength of the shading, while you want the shading to depend solely on the direction of the incoming light). We might also want to change the background color to black, so we get a better view.

Now run this code and we see the "edged lighting" described above: the light shines brightly on the left panel and the right panel is darker. We can clearly see the difference between the two triangles in the figure below.



Now we will combine the two normal vectors on the shared top vertex. First, note that the two triangles are formed by vertices (0,1,2) and (3,4,5), respectively. Thus vertices 2 and 5 are shared, and vertices 0 and 4 are shared. If we average the normal vectors for these pairs of vertices ((-1,0,1) and (1,0,1), respectively), the computation in each case is:

(-1+1,0+0,1+1)/2 = (0,0,1)

Now modify the code to use these new computed normals:

```
vertices[0].Position = new Vector3(0f, 0f, 40f);
vertices[0].Color = Color.Blue;
vertices[0].Normal = new Vector3(0, 0, 1);
vertices[1].Position = new Vector3(40f, 0f, 0f);
vertices[1].Color = Color.Blue;
vertices[1].Normal = new Vector3(1, 0, 1);
vertices[2].Position = new Vector3(0f, 40f, 40f);
vertices[2].Color = Color.Blue;
vertices[2].Normal = new Vector3(0, 0, 1);

vertices[3].Position = new Vector3(-40f, 0f, 0f);
vertices[3].Color = Color.Blue;
vertices[3].Normal = new Vector3(-1, 0, 1);
vertices[4].Position = new Vector3(0f, 0f, 40f);
vertices[4].Color = Color.Blue;
vertices[4].Normal = new Vector3(0, 0, 1);
vertices[5].Position = new Vector3(0f, 40f, 40f);
vertices[5].Color = Color.Blue;
vertices[5].Normal = new Vector3(0, 0, 1);
```

When you run this code, you will see that the reflection is nicely distributed from the dark right tip to the brighter left panel. You can imagine that this effect will give a much nicer and smoother image for a large number of triangles, as we will see in a moment.

Also, since the middle vertices use the same normal, we could again combine the 2x2 shared vertices to 2x1 vertices using an index buffer. However, in the case were we really want to create a sharp lighting edge, we need to specify the two separate normal vectors.

Optional - Test your knowledge:

1. Try moving the normals a bit to the left and right in both the separate and shared case.

Here is the final code for this section:

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
    public struct VertexPositionColorNormal
    {
        public Vector3 Position;
        public Color Color;
        public Vector3 Normal;

        public readonly static VertexDeclaration vertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Color,
                    VertexElementUsage.Color, 0),
            new VertexElement(sizeof(float) * 3 + 4, VertexElementFormat.Vector3,
                    VertexElementUsage.Normal, 0)
        );
    }

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

        Effect effect;
        VertexPositionColorNormal[] vertices;
        Matrix viewMatrix;
        Matrix projectionMatrix;

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

        protected override void Initialize()
        {
            graphics.PreferredBackBufferWidth = 500;
            graphics.PreferredBackBufferHeight = 500;
            graphics.IsFullScreen = false;
            graphics.ApplyChanges();
            Window.Title = "ATLS 4519 Lab 7 - Terrain Tutorial";

            base.Initialize();
```

```
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    device = graphics.GraphicsDevice;

    effect = Content.Load<Effect>("effects");
    SetUpCamera();

    SetUpVertices();
}

protected override void UnloadContent()
{
}

private void SetUpVertices()
{
    vertices = new VertexPositionColorNormal[6];

    vertices[0].Position = new Vector3(0f, 0f, 40f);
    vertices[0].Color = Color.Blue;
    vertices[0].Normal = new Vector3(0, 0, 1);
    vertices[1].Position = new Vector3(40f, 0f, 0f);
    vertices[1].Color = Color.Blue;
    vertices[1].Normal = new Vector3(1, 0, 1);
    vertices[2].Position = new Vector3(0f, 40f, 40f);
    vertices[2].Color = Color.Blue;
    vertices[2].Normal = new Vector3(0, 0, 1);

    vertices[3].Position = new Vector3(-40f, 0f, 0f);
    vertices[3].Color = Color.Blue;
    vertices[3].Normal = new Vector3(-1, 0, 1);
    vertices[4].Position = new Vector3(0f, 0f, 40f);
    vertices[4].Color = Color.Blue;
    vertices[4].Normal = new Vector3(0, 0, 1);
    vertices[5].Position = new Vector3(0f, 40f, 40f);
    vertices[5].Color = Color.Blue;
    vertices[5].Normal = new Vector3(0, 0, 1);

    for (int i = 0; i < vertices.Length; i++)
        vertices[i].Normal.Normalize();
}


private void SetUpCamera()
{
    viewMatrix = Matrix.CreateLookAt(new Vector3(0, -40, 100), new Vector3(0, 50, 0),
            new Vector3(0, 1, 0));

    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
             device.Viewport.AspectRatio, 1.0f, 300.0f);
}

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(Color.Black);

            RasterizerState rs = new RasterizerState();
            rs.CullMode = CullMode.None;
            device.RasterizerState = rs;

            effect.CurrentTechnique = effect.Techniques["Colored"];

            // turn on the light
            effect.Parameters["xEnableLighting"].SetValue(true);
            Vector3 lightDirection = new Vector3(0.5f, 0, -1.0f);
            lightDirection.Normalize();
            effect.Parameters["xLightDirection"].SetValue(lightDirection);


            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.DrawUserPrimitives(PrimitiveType.TriangleList, vertices, 0, 2,
                    VertexPositionColorNormal.vertexDeclaration);
            }

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

Creating Lighted Terrain

We are finally ready to create lighted terrain. We will do this by adding normal data to all vertices of our terrain, so that our graphics card can perform the required lighting calculations. The normal will be perpendicular to the triangle associated with each vertex. In cases where the vertex is shared among multiple triangles (as will usually be the case with terrain) we will find the normal of all triangles that use the vertex and then store the average of those normals for the vertex.

Begin by replacing our current Game1.cs code with the code you saved earlier. You can also find a copy in the Lab7_Content.zip file previously downloaded (filename: Game1.cs-Page40.txt). Next, add the struct (above our Game1 class definition) that will allow normal data to be added to our vertices:

```
public struct VertexPositionColorNormal
    {
        public Vector3 Position;
        public Color Color;
        public Vector3 Normal;

        public readonly static VertexDeclaration vertexDeclaration = new VertexDeclaration
        (
            new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
            new VertexElement(sizeof(float) * 3, VertexElementFormat.Color,
                    VertexElementUsage.Color, 0),
            new VertexElement(sizeof(float) * 3 + 4, VertexElementFormat.Vector3,
                    VertexElementUsage.Normal, 0)
```

```
        );
    }
```

As before, update our vertices declaration:

```
VertexPositionColorNormal[] vertices;
```

During the instantiation of this array (in SetUpVertices, above "`for (int x = 0; x < terrainWidth; x++)`"), add:

```
vertices = new VertexPositionColorNormal[terrainWidth * terrainLength];
```

As well as the line that actually renders our triangles in our Draw method:

```
device.DrawUserIndexedPrimitives(PrimitiveType.TriangleList, vertices, 0, vertices.Length, indices, 0,
            indices.Length / 3, VertexPositionColorNormal.vertexDeclaration);
```

Now we are ready to calculate the normals. Let's create a CalculateNormals method to do this. We will begin by clearing the normals in each of our vertices. Place the following method after the end of the SetUpIndices method.

```
private void CalculateNormals()
        {
            for (int i = 0; i < vertices.Length; i++)
                vertices[i].Normal = new Vector3(0, 0, 0);
        }
```

Next, we are going to iterate through each of our triangles in the same method. For each triangle, we will calculate its normal. Finally, we will add this normal to the normal each of the triangle's three vertices, as follows:

```
            for (int i = 0; i < indices.Length / 3; i++)
            {
                int index1 = indices[i * 3];
                int index2 = indices[i * 3 + 1];
                int index3 = indices[i * 3 + 2];

                Vector3 side1 = vertices[index1].Position - vertices[index3].Position;
                Vector3 side2 = vertices[index1].Position - vertices[index2].Position;
                Vector3 normal = Vector3.Cross(side1, side2);

                vertices[index1].Normal += normal;
                vertices[index2].Normal += normal;
                vertices[index3].Normal += normal;
            }
```

This code works as follows: we first look up the indices for the three vertices of each triangle. If we know two sides of a triangle, we can find its normal by taking the cross product of these two sides (this is because, given any two vectors, their cross product gives us the vector that is perpendicular to both of the original vectors). Thus, we first find two sides of the triangle by subtracting the position of one corner from the position of another. Then we find the normal by taking the cross product of these two vectors. Finally, we add this cross product to the normal for each of the three vertices.

At this point, all vertices contain large normal vectors, but they need to be of length one. So we finish our CalculateNormals method by normalizing all of the normal vectors:

```
for (int i = 0; i < vertices.Length; i++)
              vertices[i].Normal.Normalize();
```

That's it for the normals, except we need to call this method at the end of our LoadContent method:

```
CalculateNormals();
```

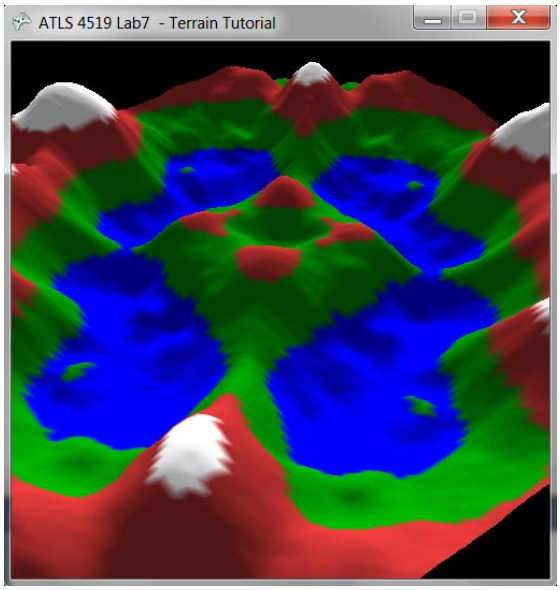
In our Draw method, tell the graphics card to use the correct effect:

```
effect.CurrentTechnique = effect.Techniques["Colored"];
```

And while we're in Draw, turn on the light:

```
Vector3 lightDirection = new Vector3(1.0f, -1.0f, -1.0f);
lightDirection.Normalize();
effect.Parameters["xLightDirection"].SetValue(lightDirection);
effect.Parameters["xAmbient"].SetValue(0.5f);
effect.Parameters["xEnableLighting"].SetValue(true);
```

The last line turns on ambient lighting, so even the parts of the terrain that are not lit at all by the directional light still receive some lighting. When you run this code, you should see the image below:



Here is our code so far:

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        public struct VertexPositionColorNormal
        {
            public Vector3 Position;
            public Color Color;
            public Vector3 Normal;

            public readonly static VertexDeclaration vertexDeclaration = new VertexDeclaration
```

```
    (
        new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
        new VertexElement(sizeof(float) * 3, VertexElementFormat.Color,
                  VertexElementUsage.Color, 0),
        new VertexElement(sizeof(float) * 3 + 4, VertexElementFormat.Vector3,
                  VertexElementUsage.Normal, 0)
    );
}

GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;
GraphicsDevice device;

Effect effect;
VertexPositionColorNormal[] vertices;
int[] indices;
Matrix viewMatrix;
Matrix projectionMatrix;

private int terrainWidth = 4;
private int terrainLength = 3;
private float[,] heightData;

Matrix worldMatrix = Matrix.Identity;
Matrix worldTranslation = Matrix.Identity;
Matrix worldRotation = Matrix.Identity;

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

private void SetUpVertices()
{
    float minHeight = float.MaxValue;
    float maxHeight = float.MinValue;
    for (int x = 0; x < terrainWidth; x++)
    {
        for (int y = 0; y < terrainLength; y++)
        {
            if (heightData[x, y] < minHeight)
                minHeight = heightData[x, y];
            if (heightData[x, y] > maxHeight)
                maxHeight = heightData[x, y];
        }
    }

    vertices = new VertexPositionColorNormal[terrainWidth * terrainLength];
    for (int x = 0; x < terrainWidth; x++)
    {
        for (int y = 0; y < terrainLength; y++)
        {
            vertices[x + y * terrainWidth].Position = new Vector3(x, heightData[x, y], -y);
            if (heightData[x, y] < minHeight + (maxHeight - minHeight) / 4)
                vertices[x + y * terrainWidth].Color = Color.Blue;
            else if (heightData[x, y] < minHeight + (maxHeight - minHeight) * 2 / 4)
                vertices[x + y * terrainWidth].Color = Color.Green;
            else if (heightData[x, y] < minHeight + (maxHeight - minHeight) * 3 / 4)
                vertices[x + y * terrainWidth].Color = Color.Brown;
            else
                vertices[x + y * terrainWidth].Color = Color.White;
```

```
        }
    }
}

private void SetUpIndices()
{
    indices = new int[(terrainWidth - 1) * (terrainLength - 1) * 6];
    int counter = 0;
    for (int y = 0; y < terrainLength - 1; y++)
    {
        for (int x = 0; x < terrainWidth - 1; x++)
        {
            int lowerLeft = x + y * terrainWidth;
            int lowerRight = (x + 1) + y * terrainWidth;
            int topLeft = x + (y + 1) * terrainWidth;
            int topRight = (x + 1) + (y + 1) * terrainWidth;

            indices[counter++] = topLeft;
            indices[counter++] = lowerRight;
            indices[counter++] = lowerLeft;

            indices[counter++] = topLeft;
            indices[counter++] = topRight;
            indices[counter++] = lowerRight;
        }
    }
}

private void CalculateNormals()
{
    for (int i = 0; i < vertices.Length; i++)
        vertices[i].Normal = new Vector3(0, 0, 0);

    for (int i = 0; i < indices.Length / 3; i++)
    {
        int index1 = indices[i * 3];
        int index2 = indices[i * 3 + 1];
        int index3 = indices[i * 3 + 2];

        Vector3 side1 = vertices[index1].Position - vertices[index3].Position;
        Vector3 side2 = vertices[index1].Position - vertices[index2].Position;
        Vector3 normal = Vector3.Cross(side1, side2);

        vertices[index1].Normal += normal;
        vertices[index2].Normal += normal;
        vertices[index3].Normal += normal;
    }

    for (int i = 0; i < vertices.Length; i++)
        vertices[i].Normal.Normalize();

}

private void SetUpCamera()
{
    viewMatrix = Matrix.CreateLookAt(new Vector3(60, 80, -80), new Vector3(0, 0, 0), new
            Vector3(0, 1, 0));
    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
            device.Viewport.AspectRatio, 1.0f, 300.0f);
}
```

```
private void LoadHeightData(Texture2D heightMap)
{
    terrainWidth = heightMap.Width;
    terrainLength = heightMap.Height;

    Color[] heightMapColors = new Color[terrainWidth * terrainLength];
    heightMap.GetData(heightMapColors);

    heightData = new float[terrainWidth, terrainLength];
    for (int x = 0; x < terrainWidth; x++)
        for (int y = 0; y < terrainLength; y++)
            heightData[x, y] = heightMapColors[x + y * terrainWidth].R / 5.0f;
}


protected override void Initialize()
{
    graphics.PreferredBackBufferWidth = 500;
    graphics.PreferredBackBufferHeight = 500;
    graphics.IsFullScreen = false;
    graphics.ApplyChanges();
    Window.Title = "ATLS 4519 Lab7  - Terrain Tutorial";

    base.Initialize();
}

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    device = graphics.GraphicsDevice;

    effect = Content.Load<Effect>("effects");
    Texture2D heightMap = Content.Load<Texture2D>("heightmap"); LoadHeightData(heightMap);

    SetUpCamera();

    SetUpVertices();
    SetUpIndices();

    CalculateNormals();

    worldMatrix *= Matrix.CreateTranslation(-terrainWidth / 2.0f, 0, terrainLength / 2.0f);
}

protected override void UnloadContent()
{
}

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

    KeyboardState keyState = Keyboard.GetState();

    //Rotation
    if (keyState.IsKeyDown(Keys.PageUp))
    {
        worldRotation = Matrix.CreateRotationY(0.01f);
    }
```

```
        else if (keyState.IsKeyDown(Keys.PageDown))
        {
            worldRotation = Matrix.CreateRotationY(-0.01f);
        }
        else
        {
            worldRotation = Matrix.CreateRotationY(0);
        }

        //Translation
        if (keyState.IsKeyDown(Keys.Left))
        {
            worldTranslation = Matrix.CreateTranslation(-.1f, 0, 0);
        }

        else if (keyState.IsKeyDown(Keys.Right))
        {
            worldTranslation = Matrix.CreateTranslation(.1f, 0, 0);
        }

        else if (keyState.IsKeyDown(Keys.Up))
        {
            worldTranslation = Matrix.CreateTranslation(0, .1f, 0);
        }

        else if (keyState.IsKeyDown(Keys.Down))
        {
            worldTranslation = Matrix.CreateTranslation(0, -.1f, 0);
        }
        else
        {
            worldTranslation = Matrix.CreateTranslation(0, 0, 0);
        }

        worldMatrix *= worldTranslation * worldRotation;

        base.Update(gameTime);
    }

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

        // Uncomment the Code below to show triangles facing away from the camera
        RasterizerState rs = new RasterizerState();
        rs.CullMode = CullMode.None;
        //rs.FillMode = FillMode.WireFrame;
        rs.FillMode = FillMode.Solid;
        device.RasterizerState = rs;

        Vector3 lightDirection = new Vector3(1.0f, -1.0f, -1.0f);
        lightDirection.Normalize();
        effect.Parameters["xLightDirection"].SetValue(lightDirection);
        effect.Parameters["xAmbient"].SetValue(0.5f);
        effect.Parameters["xEnableLighting"].SetValue(true);


        effect.CurrentTechnique = effect.Techniques["Colored"];
        effect.Parameters["xView"].SetValue(viewMatrix);
        effect.Parameters["xProjection"].SetValue(projectionMatrix);
        effect.Parameters["xWorld"].SetValue(worldMatrix);
```

```
            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Apply();

                device.DrawUserIndexedPrimitives(PrimitiveType.TriangleList, vertices, 0,
                    vertices.Length, indices, 0, indices.Length / 3,
                    VertexPositionColorNormal.vertexDeclaration);
            }

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

Using Graphics Card Memory to Buffer Vertices and Indices

Our terrain is fully working, but in each frame, all vertices and indices are being sent from our computer memory to our graphics card memory. This is a relatively slow process, and we are in fact sending over exactly the same data each time. We can optimize this process by remembering that the graphics card has a lot of memory.

Thus, we want to send the data over to the graphics card only once, after which the graphics card should store it in its own (fast) memory. This can be done by storing our vertices in what is called a VertexBuffer, and our indices in an IndexBuffer.

Start by declaring these two variables at the top of your code:

```
VertexBuffer myVertexBuffer;
IndexBuffer myIndexBuffer;
```

We will initialize and fill the VertexBuffer and IndexBuffer in a new method, CopyToBuffers. Put this code after the end of SetUpCamera.

```
private void CopyToBuffers()
        {
            myVertexBuffer = new VertexBuffer(device, VertexPositionColorNormal.vertexDeclaration,
                    vertices.Length, BufferUsage.WriteOnly);
            myVertexBuffer.SetData(vertices);

            myIndexBuffer = new IndexBuffer(device, typeof(int), indices.Length,
                    BufferUsage.WriteOnly);
            myIndexBuffer.SetData(indices);
        }
```

The first line creates the VertexBuffer, which allocates a piece of memory on the graphics card that is large enough to store all our vertices. Therefore, we need to tell the graphics card how many bytes of memory we need. This is done by specifying the number of vertices in our array, as well the VertexDeclaration (which contains the size in bytes for one vertex). The second line actually copies the data from our local vertices array into the memory on our graphics card. Then we need do the same for our indices.

Now call this method at the end of our LoadContent method:

```
CopyToBuffers();
```

All that is left is to instruct our graphics card to fetch the vertex and index data from its own memory using the DrawIndexedPrimitives method instead of the DrawUserIndexedPrimitives method. Before we call this method, we need

to let your graphics card know it should read from the buffers stored in its own memory. Put this code after pass.Apply() in the Draw method:

```
device.Indices = myIndexBuffer;
device.SetVertexBuffer(myVertexBuffer);
device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0, vertices.Length, 0, indices.Length / 3);
```

Running this code will give us the same result as before, but this time however, all vertex and index data is transferred only once to your graphics card. How cool is that! Save your code at this point.

Our final code for Lab 7:

```
using System;
using System.Collections.Generic;
using System.Linq;
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.Media;

namespace ATLS_4519_Lab7
{
    public class Game1 : Microsoft.Xna.Framework.Game
    {
        public struct VertexPositionColorNormal
        {
            public Vector3 Position;
            public Color Color;
            public Vector3 Normal;

            public readonly static VertexDeclaration vertexDeclaration = new VertexDeclaration
            (
                new VertexElement(0, VertexElementFormat.Vector3, VertexElementUsage.Position, 0),
                new VertexElement(sizeof(float) * 3, VertexElementFormat.Color,
                            VertexElementUsage.Color, 0),
                new VertexElement(sizeof(float) * 3 + 4, VertexElementFormat.Vector3,
                            VertexElementUsage.Normal, 0)
            );
        }

        GraphicsDeviceManager graphics;
        SpriteBatch spriteBatch;
        GraphicsDevice device;

        Effect effect;
        VertexPositionColorNormal[] vertices;
        int[] indices;
        Matrix viewMatrix;
        Matrix projectionMatrix;

        VertexBuffer myVertexBuffer;
        IndexBuffer myIndexBuffer;


        private int terrainWidth = 4;
        private int terrainLength = 3;
        private float[,] heightData;
```

```
Matrix worldMatrix = Matrix.Identity;
Matrix worldTranslation = Matrix.Identity;
Matrix worldRotation = Matrix.Identity;

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

private void SetUpVertices()
{
    float minHeight = float.MaxValue;
    float maxHeight = float.MinValue;
    for (int x = 0; x < terrainWidth; x++)
    {
        for (int y = 0; y < terrainLength; y++)
        {
            if (heightData[x, y] < minHeight)
                minHeight = heightData[x, y];
            if (heightData[x, y] > maxHeight)
                maxHeight = heightData[x, y];
        }
    }

    vertices = new VertexPositionColorNormal[terrainWidth * terrainLength];
    for (int x = 0; x < terrainWidth; x++)
    {
        for (int y = 0; y < terrainLength; y++)
        {
            vertices[x + y * terrainWidth].Position = new Vector3(x, heightData[x, y], -y);
            if (heightData[x, y] < minHeight + (maxHeight - minHeight) / 4)
                vertices[x + y * terrainWidth].Color = Color.Blue;
            else if (heightData[x, y] < minHeight + (maxHeight - minHeight) * 2 / 4)
                vertices[x + y * terrainWidth].Color = Color.Green;
            else if (heightData[x, y] < minHeight + (maxHeight - minHeight) * 3 / 4)
                vertices[x + y * terrainWidth].Color = Color.Brown;
            else
                vertices[x + y * terrainWidth].Color = Color.White;

        }
    }
}

private void SetUpIndices()
{
    indices = new int[(terrainWidth - 1) * (terrainLength - 1) * 6];
    int counter = 0;
    for (int y = 0; y < terrainLength - 1; y++)
    {
        for (int x = 0; x < terrainWidth - 1; x++)
        {
            int lowerLeft = x + y * terrainWidth;
            int lowerRight = (x + 1) + y * terrainWidth;
            int topLeft = x + (y + 1) * terrainWidth;
            int topRight = (x + 1) + (y + 1) * terrainWidth;

            indices[counter++] = topLeft;
            indices[counter++] = lowerRight;
            indices[counter++] = lowerLeft;

            indices[counter++] = topLeft;
```

```
            indices[counter++] = topRight;
            indices[counter++] = lowerRight;
        }
    }
}

private void CalculateNormals()
{
    for (int i = 0; i < vertices.Length; i++)
        vertices[i].Normal = new Vector3(0, 0, 0);

    for (int i = 0; i < indices.Length / 3; i++)
    {
        int index1 = indices[i * 3];
        int index2 = indices[i * 3 + 1];
        int index3 = indices[i * 3 + 2];

        Vector3 side1 = vertices[index1].Position - vertices[index3].Position;
        Vector3 side2 = vertices[index1].Position - vertices[index2].Position;
        Vector3 normal = Vector3.Cross(side1, side2);

        vertices[index1].Normal += normal;
        vertices[index2].Normal += normal;
        vertices[index3].Normal += normal;
    }

    for (int i = 0; i < vertices.Length; i++)
        vertices[i].Normal.Normalize();


}

private void SetUpCamera()
{
    viewMatrix = Matrix.CreateLookAt(new Vector3(60, 80, -80), new Vector3(0, 0, 0), new
            Vector3(0, 1, 0));
    projectionMatrix = Matrix.CreatePerspectiveFieldOfView(MathHelper.PiOver4,
            device.Viewport.AspectRatio, 1.0f, 300.0f);
}

private void CopyToBuffers()
{
    myVertexBuffer = new VertexBuffer(device, VertexPositionColorNormal.vertexDeclaration,
            vertices.Length, BufferUsage.WriteOnly);
    myVertexBuffer.SetData(vertices);

    myIndexBuffer = new IndexBuffer(device, typeof(int), indices.Length,
            BufferUsage.WriteOnly);
    myIndexBuffer.SetData(indices);
}


private void LoadHeightData(Texture2D heightMap)
{
    terrainWidth = heightMap.Width;
    terrainLength = heightMap.Height;

    Color[] heightMapColors = new Color[terrainWidth * terrainLength];
    heightMap.GetData(heightMapColors);

    heightData = new float[terrainWidth, terrainLength];
    for (int x = 0; x < terrainWidth; x++)
```

```
            for (int y = 0; y < terrainLength; y++)
                heightData[x, y] = heightMapColors[x + y * terrainWidth].R / 5.0f;
    }


    protected override void Initialize()
    {
        graphics.PreferredBackBufferWidth = 500;
        graphics.PreferredBackBufferHeight = 500;
        graphics.IsFullScreen = false;
        graphics.ApplyChanges();
        Window.Title = "ATLS 4519 Lab7  - Terrain Tutorial";

        base.Initialize();
    }

    protected override void LoadContent()
    {
        spriteBatch = new SpriteBatch(GraphicsDevice);

        device = graphics.GraphicsDevice;

        effect = Content.Load<Effect>("effects");
        Texture2D heightMap = Content.Load<Texture2D>("heightmap"); LoadHeightData(heightMap);

        SetUpCamera();

        SetUpVertices();
        SetUpIndices();

        CalculateNormals();
        CopyToBuffers();

        worldMatrix *= Matrix.CreateTranslation(-terrainWidth / 2.0f, 0, terrainLength / 2.0f);
    }

    protected override void UnloadContent()
    {
    }

    protected override void Update(GameTime gameTime)
    {
        if (GamePad.GetState(PlayerIndex.One).Buttons.Back == ButtonState.Pressed)
            this.Exit();
        //this is directly copied
        KeyboardState keyState = Keyboard.GetState();

        //Rotation
        if (keyState.IsKeyDown(Keys.PageUp))
        {
            worldRotation = Matrix.CreateRotationY(0.01f);
        }

        else if (keyState.IsKeyDown(Keys.PageDown))
        {
            worldRotation = Matrix.CreateRotationY(-0.01f);
        }
        else
        {
            worldRotation = Matrix.CreateRotationY(0);
        }
```

```
        //Translation
        if (keyState.IsKeyDown(Keys.Left))
        {
            worldTranslation = Matrix.CreateTranslation(-.1f, 0, 0);
        }

        else if (keyState.IsKeyDown(Keys.Right))
        {
            worldTranslation = Matrix.CreateTranslation(.1f, 0, 0);
        }

        else if (keyState.IsKeyDown(Keys.Up))
        {
            worldTranslation = Matrix.CreateTranslation(0, .1f, 0);
        }

        else if (keyState.IsKeyDown(Keys.Down))
        {
            worldTranslation = Matrix.CreateTranslation(0, -.1f, 0);
        }
        else
        {
            worldTranslation = Matrix.CreateTranslation(0, 0, 0);
        }

        worldMatrix *= worldTranslation * worldRotation;

        base.Update(gameTime);
    }

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

        // Uncomment the Code below to show triangles facing away from the camera
        RasterizerState rs = new RasterizerState();
        rs.CullMode = CullMode.None;
        //rs.FillMode = FillMode.WireFrame;
        rs.FillMode = FillMode.Solid;
        device.RasterizerState = rs;

        Vector3 lightDirection = new Vector3(1.0f, -1.0f, -1.0f);
        lightDirection.Normalize();
        effect.Parameters["xLightDirection"].SetValue(lightDirection);
        effect.Parameters["xAmbient"].SetValue(0.5f);
        effect.Parameters["xEnableLighting"].SetValue(true);


        effect.CurrentTechnique = effect.Techniques["Colored"];
        effect.Parameters["xView"].SetValue(viewMatrix);
        effect.Parameters["xProjection"].SetValue(projectionMatrix);
        effect.Parameters["xWorld"].SetValue(worldMatrix);

        foreach (EffectPass pass in effect.CurrentTechnique.Passes)
        {
            pass.Apply();
            device.Indices = myIndexBuffer;
            device.SetVertexBuffer(myVertexBuffer);
            device.DrawIndexedPrimitives(PrimitiveType.TriangleList, 0, 0, vertices.Length, 0,
                indices.Length / 3);
        }
```

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

That completes our introductory terrain tutorial. We have interesting, but non-realistic terrain. That is the subject of the next terrain lab.