08214 Lab 1: Draw Calls and Vertex Buffer Objects

Welcome to the first practical lab for Simulation and 3D Graphics. An SVN repository has been created for this module. The URL is:

https://visualsvn.net.dcs.hull.ac.uk/svn/08214-1415/<xxxxxx>

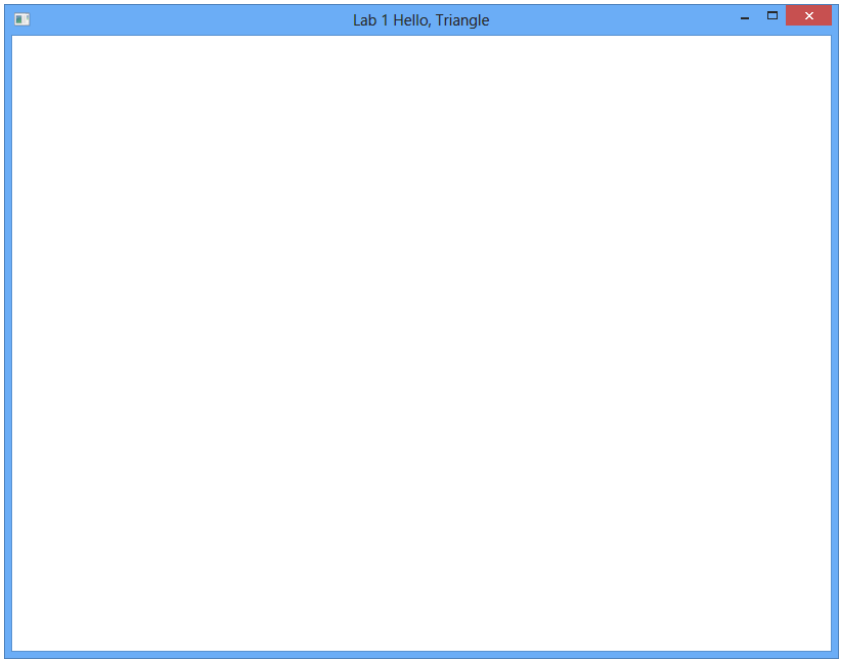
Where <xxxxxx> is your six digit login ID. If you do not have a repository contact Simon Grey as soon as possible. For more information on SVN, including a licence key for the visual studio SVN integration tool "Visual SVN", please see the help documents on the systems section of sharepoint. Starting points for each lab for this module have been already added to your SVN repository. The lab sheets themselves are on available on the module site on sharepoint. On your network drive create a directory that you wish to work in, right click and select SVN Checkout. Module staff can see the contents of these repositories, which makes providing support easier. Remember to check your work in regularly. For more supporting information about this module please read the 08214 Lab 0 Introduction document.

In this lab we will start drawing triangles! The goal for this lab is that you will learn how to load data onto the graphics card, and use a DrawCall to instruct the graphics card to send data into the graphics pipeline in different forms.

If you have any problems (or spot a mistake) one way you can get help is on the forum thread for lab 1 here at http://forums.net.dcs.hull.ac.uk/t/1937.aspx

Hello Triangle

Once you've checked the code out open the 08214.sln file and run the code. You should see the following screen:



This code was meant to draw a triangle! Let's take a look at the code to see what's wrong. The GameWindow class has three functions. OnLoad is called after the window has been created and a glContext exists. This means that we can make calls to the OpenGL API. OnRenderFrame is called when it is time to render the frame. This is where all our draw calls are done. OnUnload is called just before the glContext is destroyed. This gives us the chance to clean up after ourselves, and delete any data we've loaded on to the graphics card.

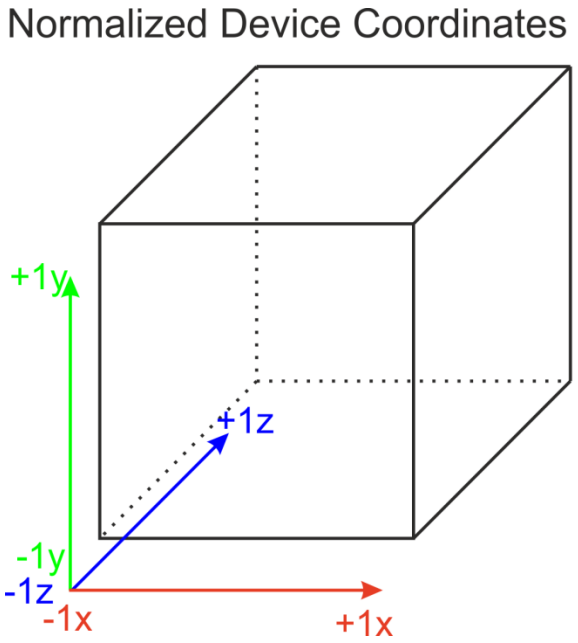
OnLoad is used to set various states on the graphics card, and to load some data onto the graphics card. It also uses a ShaderUtility function to load a shader. Shaders will be covered at greater depth next week.

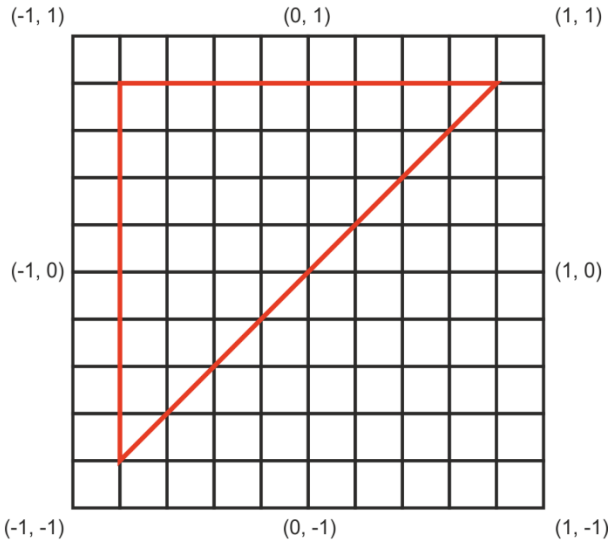
Let's take a look at the code that's already here:

```
GL.ClearColor(Color4.White);

float[] vertices = new float[] { -0.8f, -0.8f,
                                 -0.8f, 0.8f,
                                  0.8f, 0.8f };
```

First, the clear colour that OpenGL uses when it is instructed to clear the screen is set to White. Between frames it is usually required that the old frame be wiped before drawing the new frame. The clear colour dictates the default colour that each pixel should be reset to.

Next an array of six floats is created. Although it's not necessary from a syntactical point of view, to make the code easier to understand each line represents one vertex. These are the x and y values of a triangle. We are ignoring the third dimension for the time being. We're also drawing in Normalised Device Coordinates. Prior to rendering all objects in a scene are converted into Normalised Device Coordinates. In OpenGL Normalised device coordinates go from -1 to 1 in all three dimensions. Below is a diagram of normalized device coordinates, and the triangle that these three sets of x and y values draw.






```
GL.GenBuffers(1, out mVertexBufferObjectID);
GL.BindBuffer(BufferTarget.ArrayBuffer, mVertexBufferObjectID);
GL.BufferData(BufferTarget.ArrayBuffer, (IntPtr)(vertices.Length * sizeof(float)), vertices,
BufferUsageHint.StaticDraw);
```

Next we need to load the values in the vertices array onto the graphics card. To do that, we need to generate a Buffer ID, so that we can refer to the buffer later in our program. We'll store that value in a member variable called mVertexBufferObjectID. GenBuffers generates that buffer ID. Next we Bind the Buffer. That means we make that the active buffer, and any buffer related commands that follow will refer to that buffer. Finally we get to loading the data. The BufferData call both allocates memory on the graphics card, and copies the contents of vertices into that memory.

```
int size;
GL.GetBufferParameter(BufferTarget.ArrayBuffer, BufferParameterName.BufferSize, out size);

if (vertices.Length * sizeof(float) != size)
{
        throw new ApplicationException("Vertex data not loaded onto graphics card
correctly");
}
```

This code performs a check that checks that the correct amount of data has been copied onto the graphics card. If the size of the data on the card is not as expected an exception is thrown. There is a small region that loads a shader program. That will be covered next week. For the moment, we need to learn how to load model data on to the graphics card, and how to instruct the graphics card to draw that data.

Next, let's take a quick look at the OnUnload method. This is called when the program exits, but before the glContext has been disposed. This is our opportunity to delete the data we've allocated on the card.

```
GL.DeleteBuffers(1, ref mVertexBufferObjectID);
```

This line deletes the buffer with the same ID as is stored in mVertexBufferObjectID.

Finally, let's see how we instruct the graphics card to draw the triangle. There is a small region of code concerned with linking to the shader that had been omitted here. Linking to shaders will be covered next week.

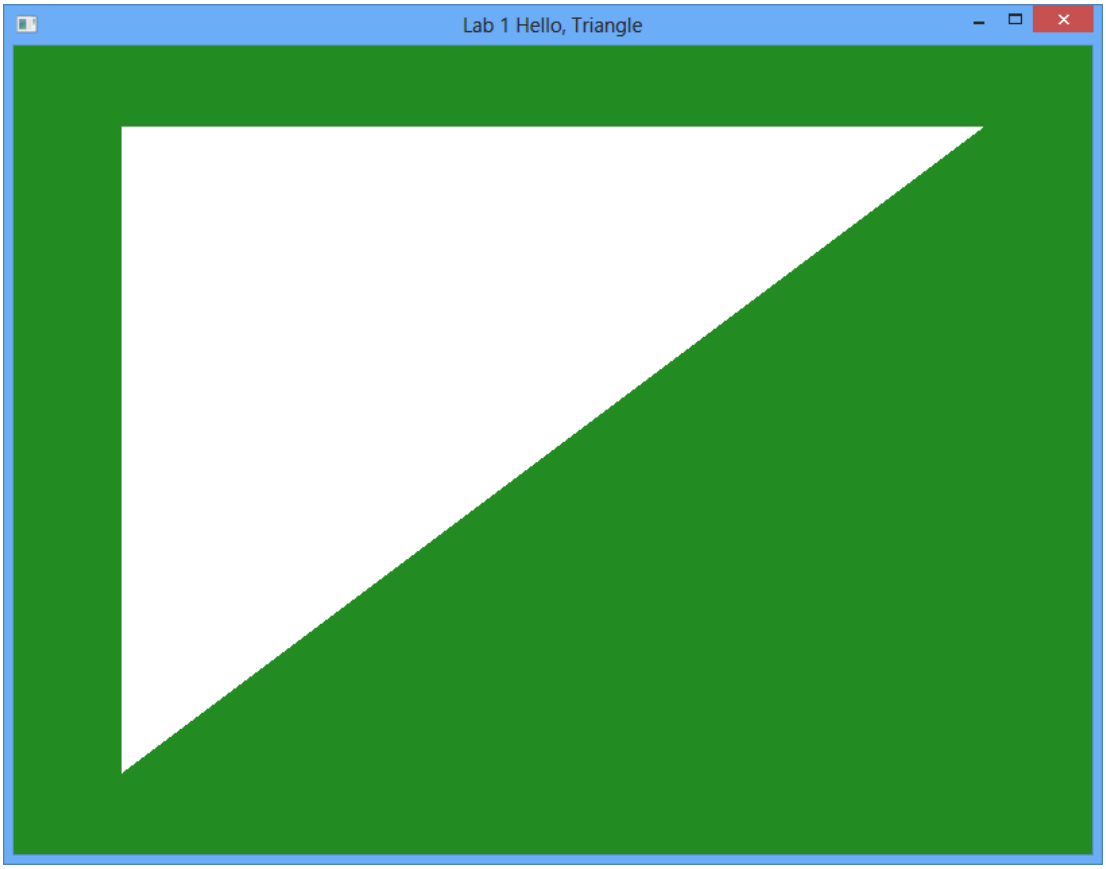
```
GL.Clear(ClearBufferMask.ColorBufferBit);

GL.BindBuffer(BufferTarget.ArrayBuffer, mVertexBufferObjectID);

GL.DrawArrays(PrimitiveType.Triangles, 0, 3);

this.SwapBuffers();
```

In most applications we use a technique called double buffering. That means that we draw to a backBuffer, that cannot be seen. Then once we've finished drawing we swap the back buffer and the front buffer. The first line clears the Colour data for the back buffer. This gives us a blank (in this case, white, as was set in the OnLoad method) slate to draw onto. Next we bind the buffer that we want to draw. This step is a bit of overkill here, because there is only one buffer loaded. If there were more you would need a line like this to switch between buffers. The DrawArrays command is the one that does all the drawing. It basically sends vertices in the array, in order, to the graphics pipeline, starting at the 0^{th} element, for 3 elements. The PrimitiveType specifies that these vertices should be used to draw Triangles. We'll cover other options for PrimitiveType later in this lab. Finally, we instruct OpenTK to swap the front and back buffers, presenting what we have rendered to the user.

So why can't we see a triangle as shown in the Normalised Device Coordinates diagram? Well, the answer is you can, but the current setup is drawing a white triangle on a white background.

Change the colour of the background to something other than white and recompile your code. I've used Forest Green.

Hint: *You need to change something in the OnLoad method.*



Once you've managed that right click on the solution in solution explorer and commit your work to SVN, with the following log message.

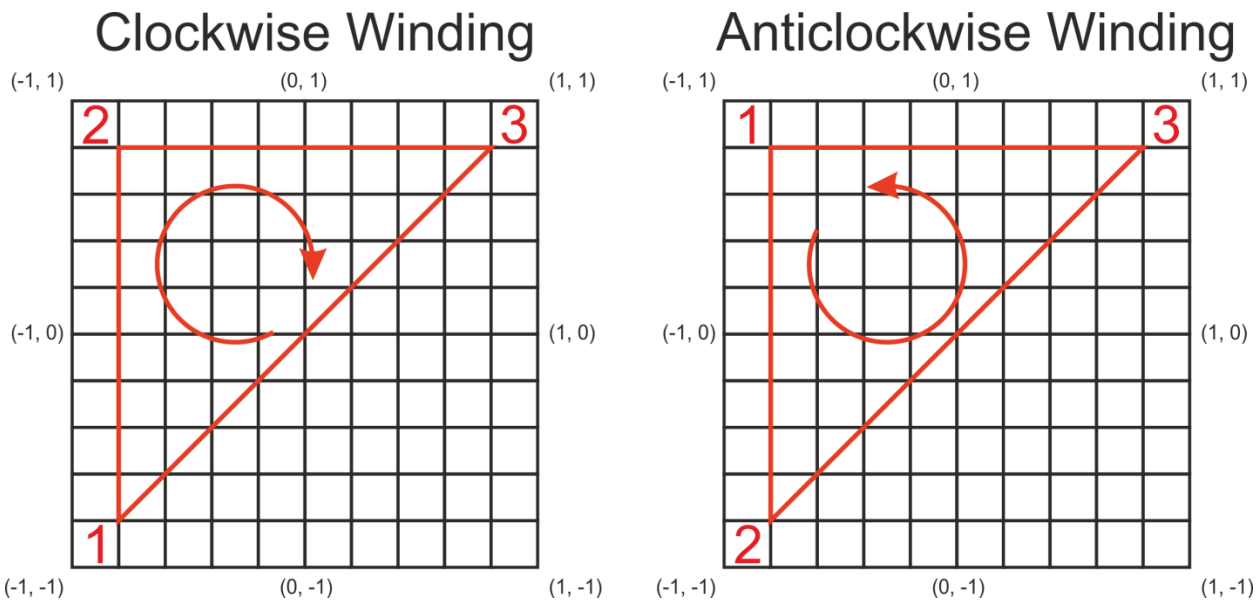
L1T1 Changed the clear colour and drew my first triangle!

Another state change

In the OnLoad method add the line

```
GL.Enable(EnableCap.CullFace);
```

Run the code again and you should see that your triangle has disappeared. This is because by enabling CullFace triangles that are facing the wrong way are no longer drawn. The direction a triangle is facing is determined by whether the order of the vertices is defined in a clockwise, or anticlockwise.



You can remedy this problem in a number of ways. One way it to swap two consecutive vertices in the vertex definition (i.e. swap 1 and 2, or 2 and 3, or 3 and 1). This reverses the "winding" of the triangle. Here I've swapped vertices 1 and 2:

```
float[] vertices = new float[] { -0.8f, 0.8f,
                                 -0.8f, -0.8f,
                                  0.8f, 0.8f };
```

Once you've done that, right click on the solution in solution explorer and commit your work to SVN, with the following log message.

L1T2 Enabled Back Face Culling and Fixed Triangle Winding.

Hello Square

Next, we're going to build on what we've learnt to draw a square by adding a second triangle in the gap on the bottom right.

First add three extra vertices to the vertices array:

```
float[] vertices = new float[] { -0.8f, 0.8f,
                                 -0.8f, -0.8f,
                                  0.8f, 0.8f,
                                 -0.8f, -0.8f,
                                  0.8f, 0.8f,
                                  0.8f, -0.8f };
```

Next, check the line that loads the vertices array into the graphics card.

```
GL.BufferData(BufferTarget.ArrayBuffer, (IntPtr)(vertices.Length * sizeof(float)), vertices,
BufferUsageHint.StaticDraw);
```

The second parameter is the number of bytes we want to copy on to the graphics card. Note that because we use the length of the array multiplied by the size of a float we don't need to adjust this, because the vertices.Length property of the array will be increased automatically as the size of the array is increased.

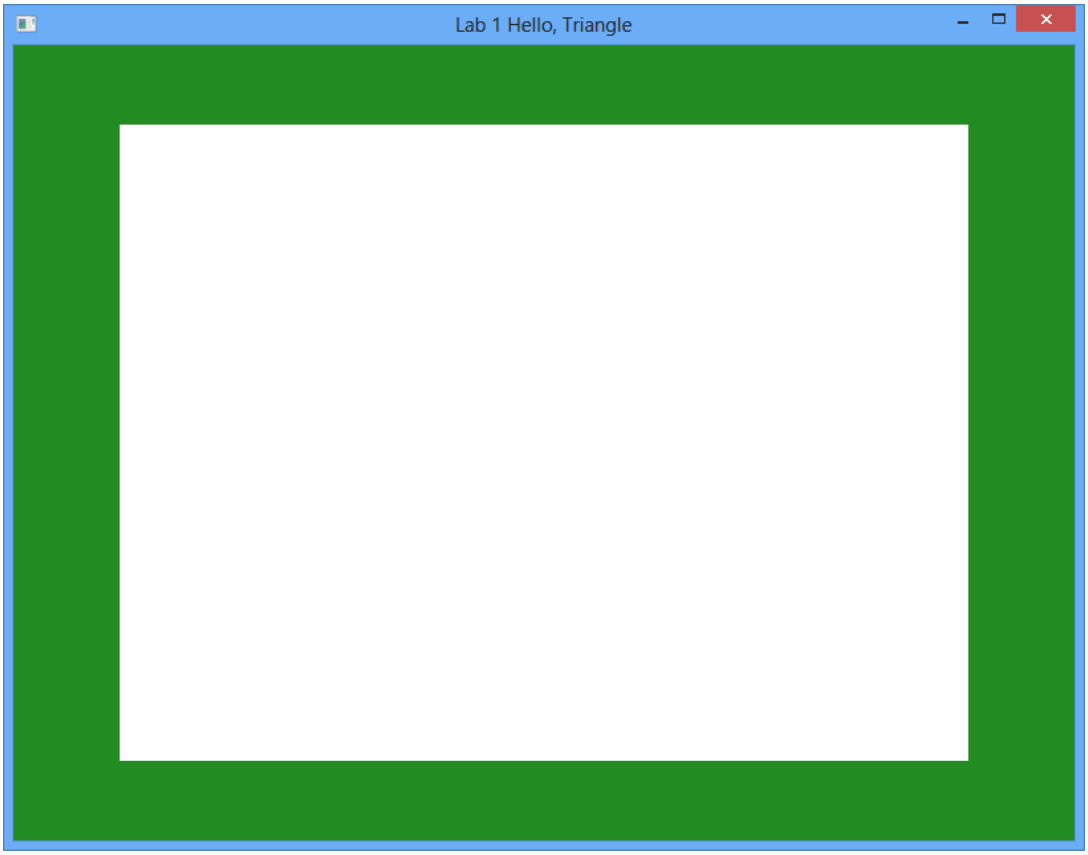
Finally, let's take a look at the draw call on OnRenderFrame.

```
GL.DrawArrays(PrimitiveType.Triangles, 0, 3);
```

This line sends the vertices from 0 to three to the graphics pipeline. We have 6 vertices to send, so change the last parameter to 6.

Run the code and you should notice that something's not quite right. See if you can fix the error.

Hint: *What order are the vertices in the second triangle?*



When you've drawn your square right click on the solution in solution explorer and commit your work to SVN, with the following log message.

L1T3 Drew a square by adding additional vertices to the vertices array, and adjusting the DrawArrays call.

In case you're wondering why your square isn't square it's because of the aspect ratio of the window. We'll fix that in a future lab.

Try experimenting with a different draw call. What do you think would happen if you change the draw call start at a later point in the vertices array?

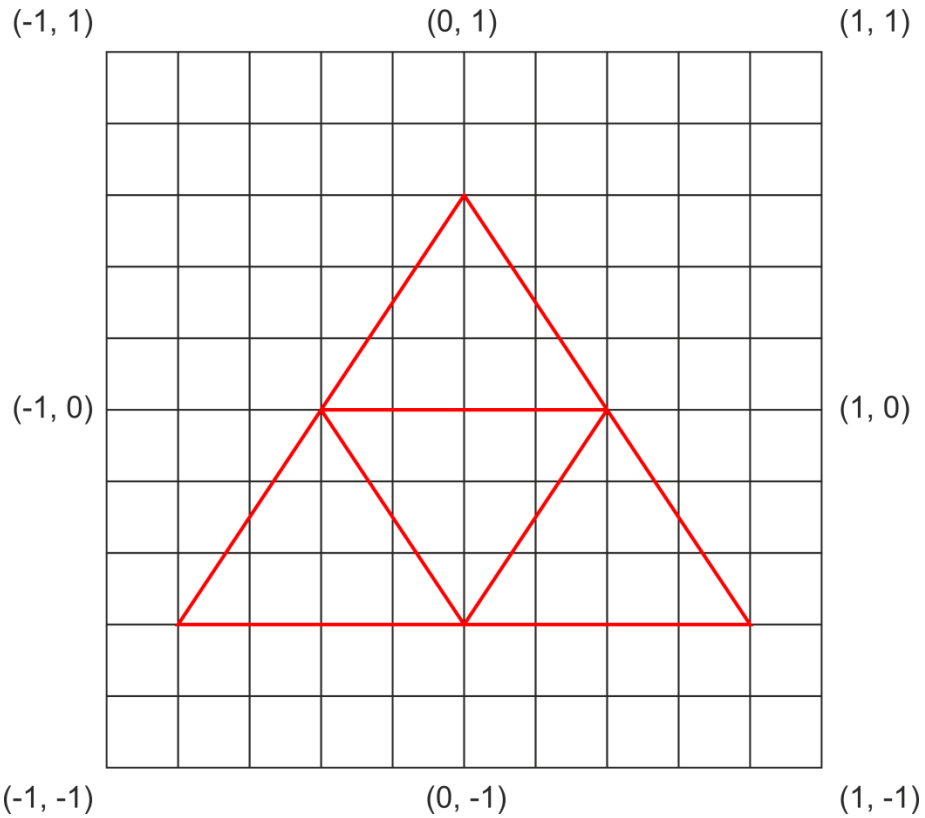
```
GL.DrawArrays(PrimitiveType.Triangles, 3, 6);
```

Commit your work to SVN, and add (and finish) the log message.

L1T4 This time only the second half of the square was drawn because…

Hello Triforce

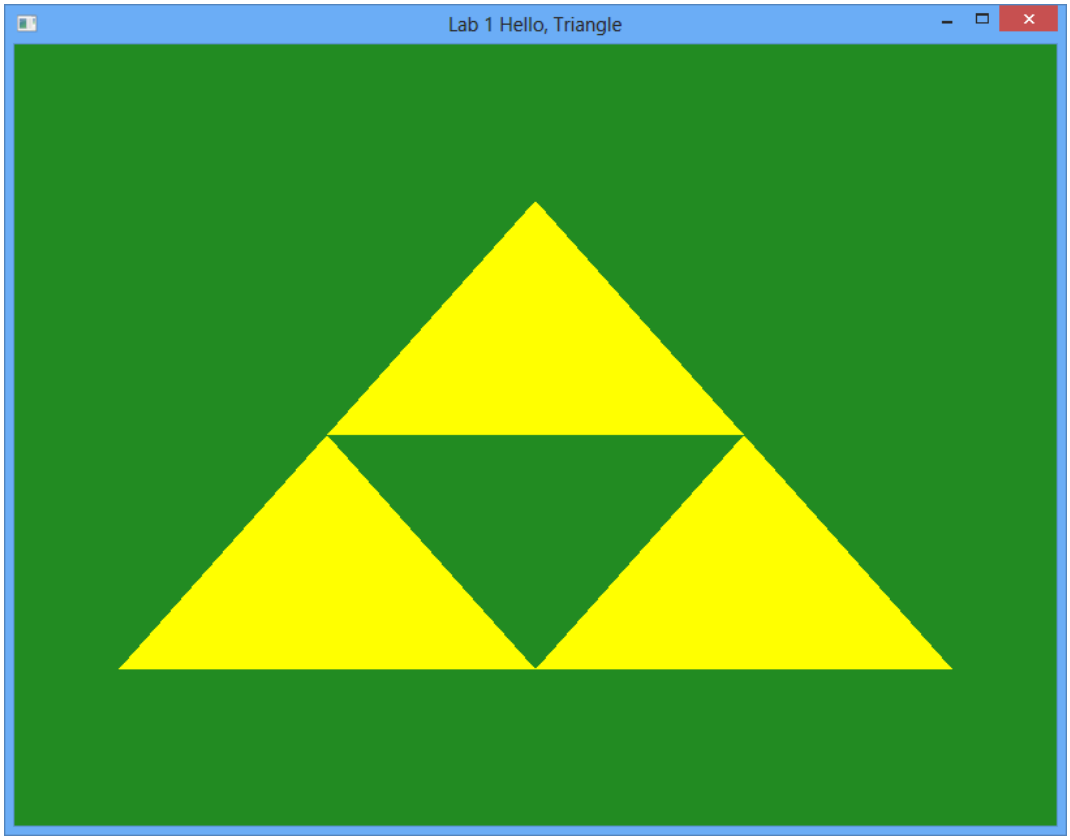
Next, we're going to draw a triforce symbol.



There are three triangles, so we need nine vertices. I've worked out the x and y values for the top triangle.

```
float[] vertices = new float[] { -0.4f, 0.0f,
                                  0.4f, 0.0f,
                                  0.0f, 0.6f };
```

See if you can figure out the x and y values for the remaining six vertices. Remember, you're also going to have to change the DrawArrays call in OnRenderFrame.



We will do more on shaders next week, but just for fun, to change the triangle colour to yellow open up the fragment shader fSimple.frag (in Lab1 -> Shaders) and change the line that says

```
FragColour = vec4(1.0, 1.0, 1.0, 1.0);
```

To

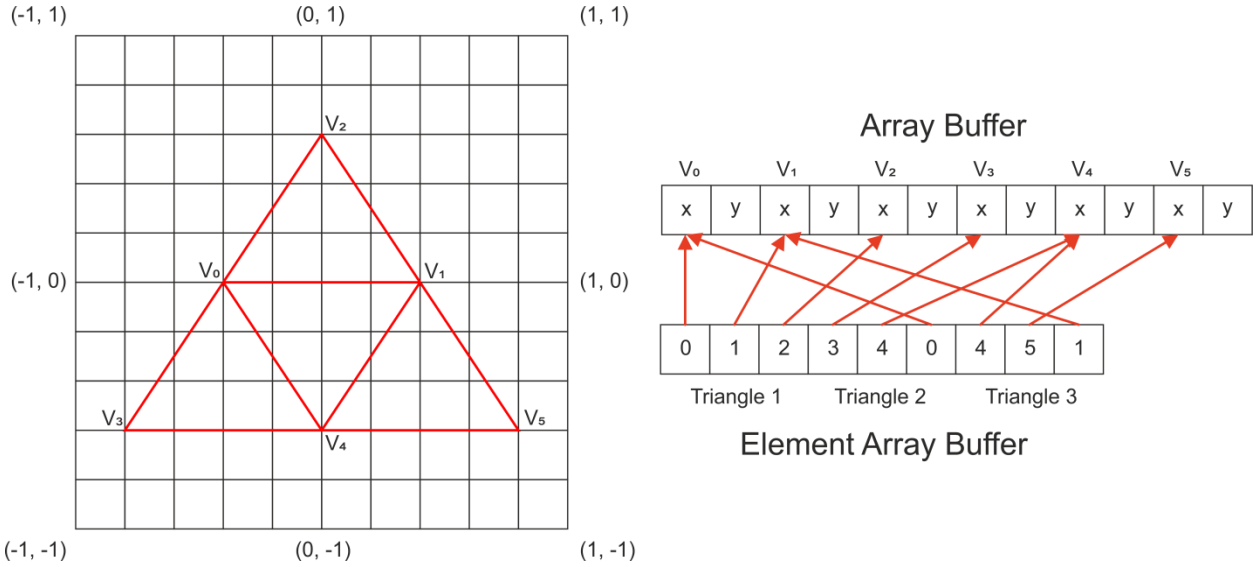
```
FragColour = vec4(1.0, 1.0, 0.0, 1.0);
```

Commit your work to SVN with the log message.

L1T5 Drew a TriForce Symbol by editing the vertices array and adjusting the DrawArrays call. Also modified the fragment shader to colour all fragments yellow.

You may have noticed that in our triforce three of the vertices are shared by two triangles each. This means that we're loading the same vertices more than once. This isn't a big problem right now, but if the model were made up of lots of vertices, and the vertices themselves contained more information then we might run out of memory on the graphics card.

Fortunately there is a way around this problem. We can use an element array buffer. An element array is an array of integer values that is used to reference an array of vertex data. The following diagram explains the concept. Instead of loading nine vertices, we load six and use the element array buffer to reuse three of them.



In order to do this, we need to generate another buffer object on the graphics card to store the indices. We could generate an entirely new member variable in the Lab1Window class, but instead we're going to change the existing integer member variable into an array. First change

```
private int mVertexBufferObjectID;
```

into an array that can hold a buffer ID for both the triangle indices and the vertex data.

```
private int[] mVertexBufferObjectIDArray = new int [2];
```

Then in the OnLoad method replace the code that creates the vertices array, generates a buffer ID and loads the data with the following code:

```
GL.GenBuffers(2, mVertexBufferObjectIDArray);

GL.BindBuffer(BufferTarget.ArrayBuffer, mVertexBufferObjectIDArray[0]);
GL.BufferData(BufferTarget.ArrayBuffer, (IntPtr)(vertices.Length * sizeof(float)), vertices,
BufferUsageHint.StaticDraw);

int size;
GL.GetBufferParameter(BufferTarget.ArrayBuffer, BufferParameterName.BufferSize, out size);

if (vertices.Length * sizeof(float) != size)
{
    throw new ApplicationException("Vertex data not loaded onto graphics card correctly");
}

GL.BindBuffer(BufferTarget.ElementArrayBuffer, mVertexBufferObjectIDArray[1]);
GL.BufferData(BufferTarget.ElementArrayBuffer, (IntPtr)(indices.Length * sizeof(uint)),
indices, BufferUsageHint.StaticDraw);

GL.GetBufferParameter(BufferTarget.ElementArrayBuffer, BufferParameterName.BufferSize, out
size);

if (indices.Length * sizeof(uint) != size)
{
    throw new ApplicationException("Index data not loaded onto graphics card correctly");
}
```

This code looks similar to the previous code, and you can probably guess what most of it does. Note the change to the GenBuffers call, to accommodate the array. The final two lines are the ones that bind and load our element array buffer. Note the different BufferTarget. You will also need to modify the vertices array according to the diagram above, and add an array of type uint called indices, again populated according to the diagram.

As we are loading more data to the graphics card, we must remember to delete it in the OnUnload method. Change the line that deletes the data buffer

```
GL.DeleteBuffers(1, ref mVertexBufferObjectID);
```

To a line that deletes both buffers at the same time.

```
GL.DeleteBuffers(2, mVertexBufferObjectIDArray);
```

Finally, we need to change the rendering call. Change

```
GL.BindBuffer(BufferTarget.ArrayBuffer, mVertexBufferObjectID);

// shader linking goes here

GL.DrawArrays(PrimitiveType.Triangles, 0, 9);
```

To

```
GL.BindBuffer(BufferTarget.ArrayBuffer, mVertexBufferObjectIDArray[0]);
GL.BindBuffer(BufferTarget.ElementArrayBuffer, mVertexBufferObjectIDArray[1]);

// shader linking goes here

GL.DrawElements(PrimitiveType.Triangles, 9, DrawElementsType.UnsignedInt, 0);
```

Note that we bind the element array separately, and that DrawElements using the currently bound element array buffer to refer to the currently bound array buffer.

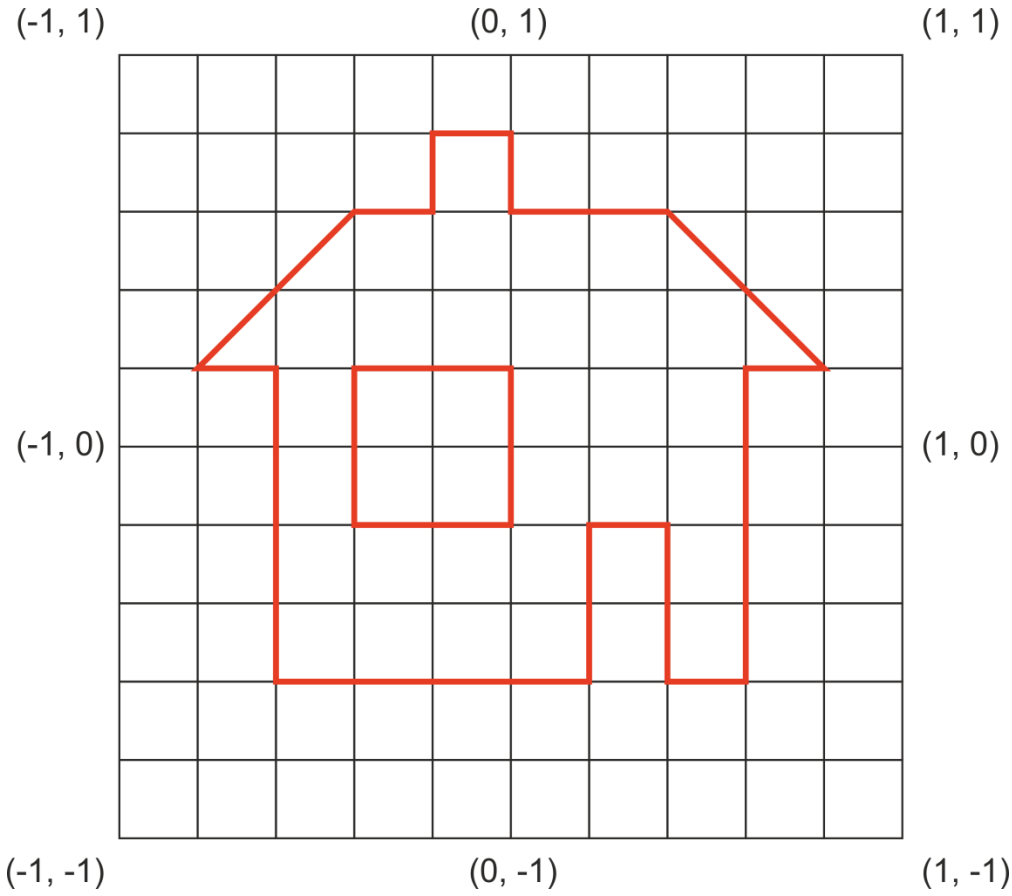
That was a lot of work to get to the same result, but hopefully the benefits in terms of memory efficiency on the graphics card, and load times loading data onto the card are apparent, especially when you consider larger models with more vertices, and vertices that store more data.

Commit your work to your SVN repository

L1T6 Converted TriForce Symbol to use element array buffers.

Hello, House!

Let's see how much you've understood so far. See if you can use what you've learnt to draw a house like the one plotted out on our Normalised Device Coordinates grid below.



I think this can be done using just 26 vertices and 48 indices to make 16 triangles. Can you do any better?

When you've drawn your house commit to SVN with an appropriate log message starting with…

L1T7 Drew a house by …

Using Different Drawing Primitives

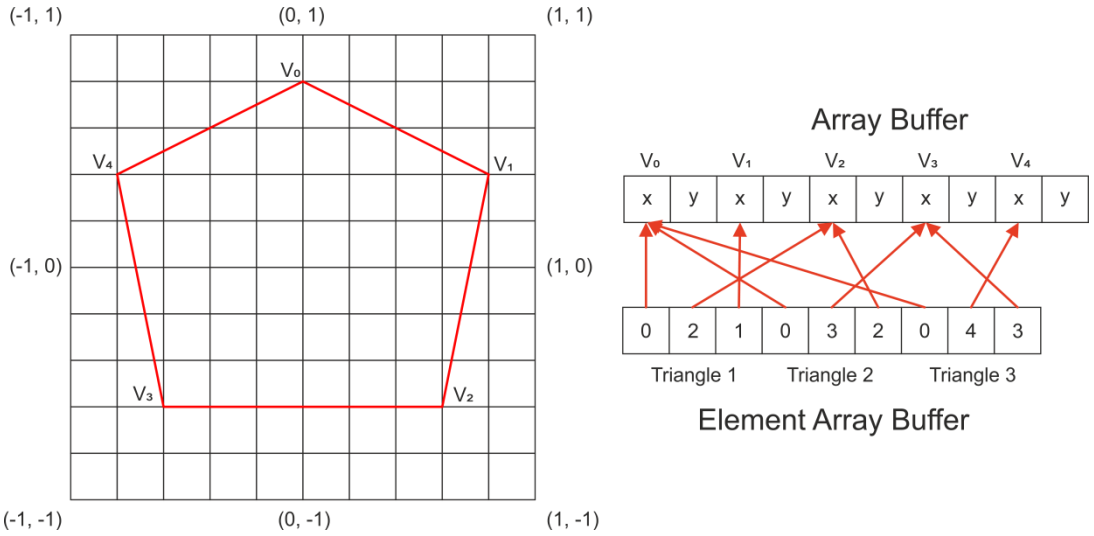
So we've managed to limit the amount of memory we're using, but there are better strategies for defining our triangles. If we look at the draw call we notice that the first parameter specifies the primitive type.

```
GL.DrawElements(PrimitiveType.Triangles, 9, DrawElementsType.UnsignedInt, 0);
```

This parameter controls how the vertices sent to the pipeline are converted into triangles. To illustrate the use of different primitives, and why this is important we need to modify our code to draw a pentagon.

Drawing a Pentagon

The diagram below shows what we are aiming for. We're going to draw a pentagon using five vertices, and index them using nine indices to draw three triangles.




```
float[] vertices = new float[] {  0.0f, 0.8f,
                                  0.8f, 0.4f,
                                  0.6f, -0.6f,
                                 -0.6f, -0.6f,
                                 -0.8f, 0.4f};

uint[] indices = new uint[] { 0,2,1,
                              0,3,2,
                              0,4,3};
```

Now you should see a pentagon shape. Commit your code to SVN with the log message:

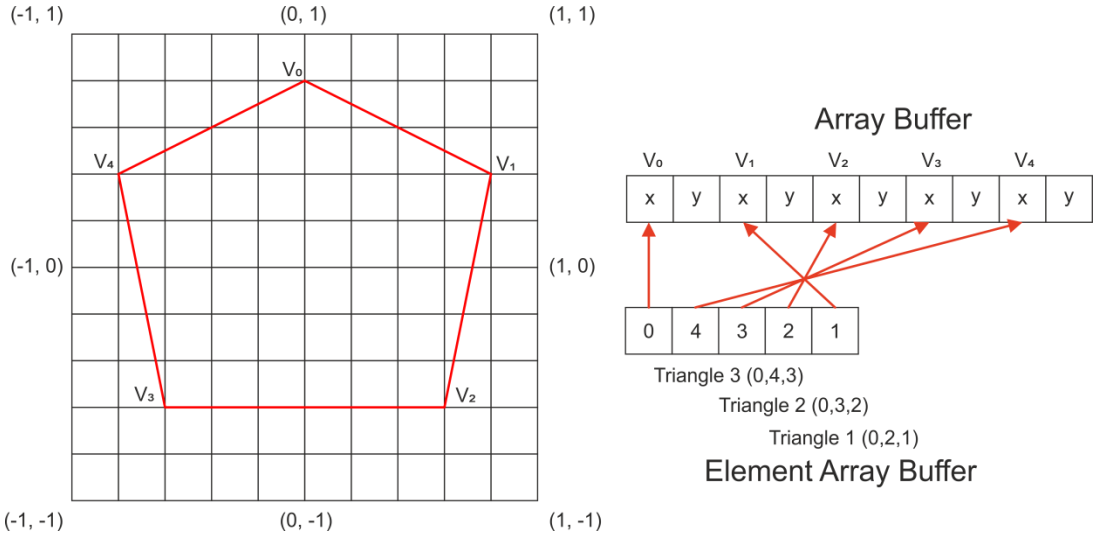
L1T8 Drew a pentagon using DrawPrimitive.Triangles

Hopefully it's clear that as vertices are copied to the graphics pipeline the triangles that should be drawn are defined in groups of three vertices. Also note that the first vertex (0) is sent to the pipeline three times. We can reduce that by using different DrawPrimitives. Change the draw primitive in the DrawElements call to TriangleFan. Note that the number of indices to vertices has been reduced from 9 to 5.

```
GL.DrawElements(PrimitiveType.TriangleFan, 5, DrawElementsType.UnsignedInt, 0);
```

We also need to change the indices array. A triangle fan builds triangles by using the first index that it was passed, and the last two. Change the indices array to be

```
uint[] indices = new uint[] { 0,4,3,
                              2,
                              1};
```



Note that we had to change the order the triangle were drawn in to preserve the correct winding. A triangle fan that is sent these indices will build triangles from vertex numbers [0, 4, 3].[0, 3, 2] and [0, 2, 1]. Compare that to what we were doing with the Triangles primitive and you should see that they are the same, but this time we've sent nearly 45% fewer vertices into the pipe.

Check that this is working, and commit your code to SVN with the message:

L1T9 Drew a pentagon using DrawPrimitives.TriangleFan using 45% fewer indices!

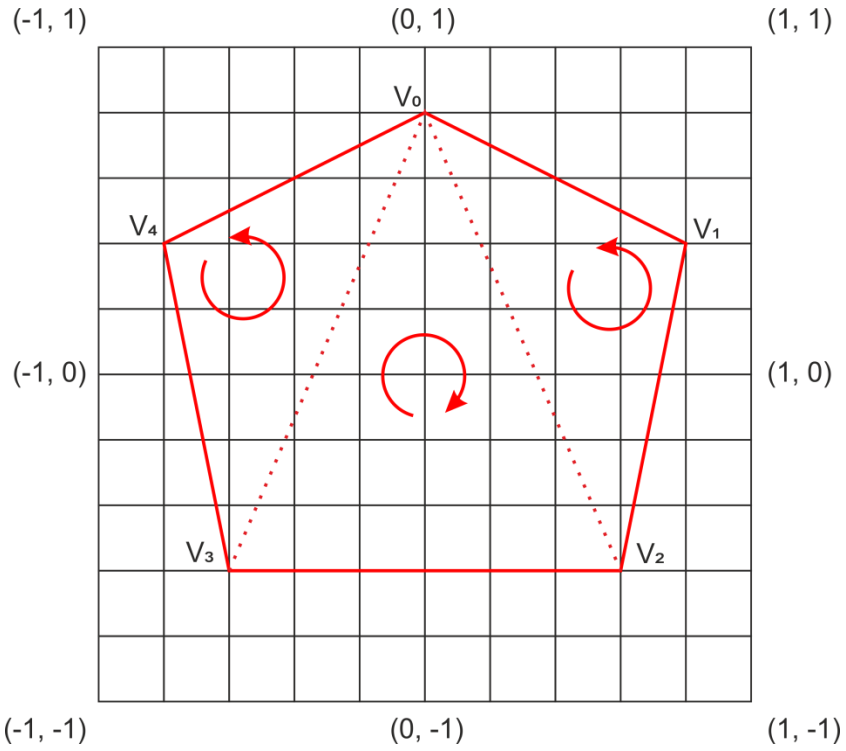
Another way of building triangles from vertices is to use a TriangleStrip. Remember that using the Triangles primitive takes indices in groups of three and reuses no vertices. Triangle fan uses the first index passed to it, and the last two to create triangles. Triangle Strip always uses the last three, so that each new vertex passed into the pipeline shares the previous two that were passed in. Change the draw call to be

```
GL.DrawElements(PrimitiveType.TriangleStrip, 5, DrawElementsType.UnsignedInt, 0);
```

See if you can figure out the order of the indices yourself, then commit to SVN with the message:

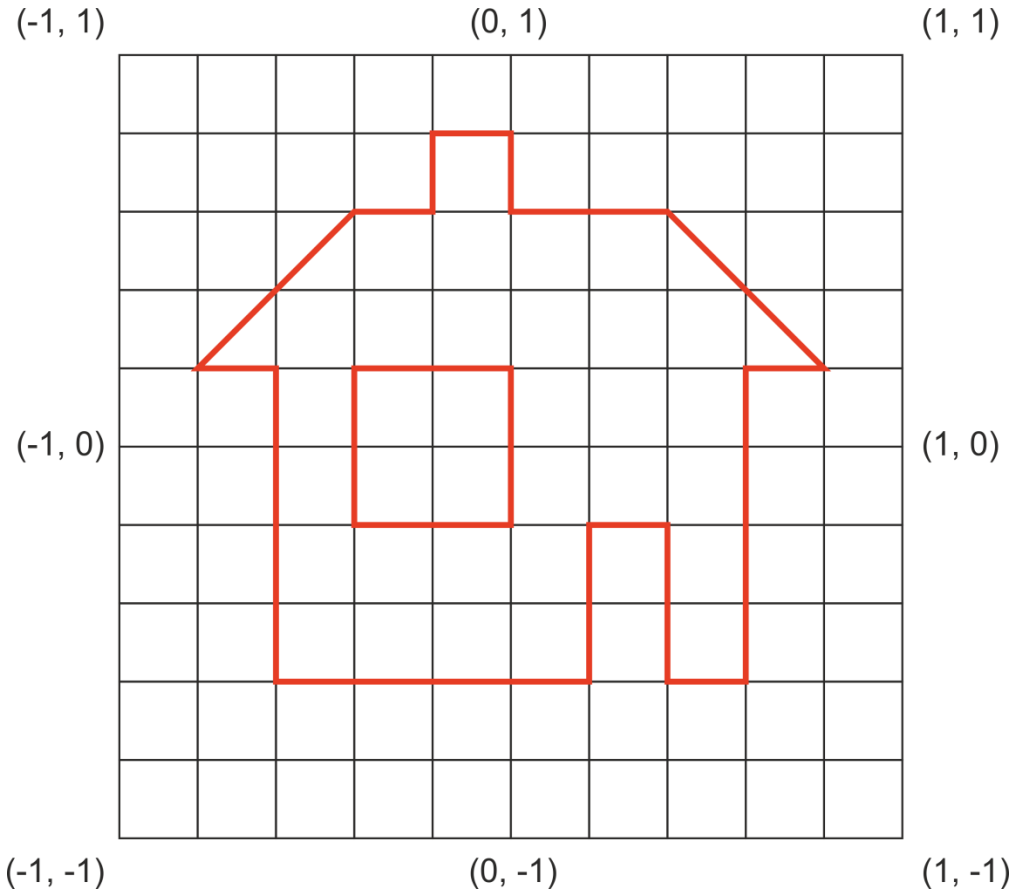
L1T10 Drew a pentagon using DrawPrimitive.TriangleStrip

You may have noticed that something odd happens when you use a triangle strip. The diagram below shows how by using a triangle strip the winding of the vertices in each triangle naturally flips from anticlockwise to clockwise and back.



Earlier we learnt how to ignore a triangle if the winding of the vertices is the wrong way round. That means that at least one of these triangles shouldn't be drawn. Fortunately for us, OpenGL understands this problem and with reverse the cull face for even numbers triangles, so we don't have to worry about it. Thanks OpenGL!

We're almost done, but before we move on let's take a look at our house again.



See how few indices you can send down the pipeline by using triangle fans and triangle strips instead of triangles. It might help to know that the last parameter in the DrawElements call is the start offset in bytes.

```
GL.DrawElements(PrimitiveType.TriangleFan, 4, DrawElementsType.UnsignedInt, 1 *
sizeof(uint));
```

A call like the one above will skip the first index and send the next 4 to the pipe.

Once you've done that, commit to SVN with the following log message – replace n and m with the appropriate numbers.

L1T11 Drew a house using <n> vertices and only <m> indices

Summary

During this lab we have looked at how to load data onto the graphics card, and to delete it when we are finished. We've learnt how to define triangles using data arrays, reduce the size of models in memory using element arrays, and reduce the number of vertices being sent to the pipeline using Triangle Strips and Triangle Fans.

In the next lab we'll look at the programmable parts of the pipeline, and how to link data being sent to the pipeline to the shaders that process vertices and fragments. We'll also look at Vertex Array Objects, which make state changes easier. Finally we'll learn about cameras and perspective to help us move around a virtual 3D world.