08214 Lab 2_1: Linking to Shaders and using Vertex Array Objects

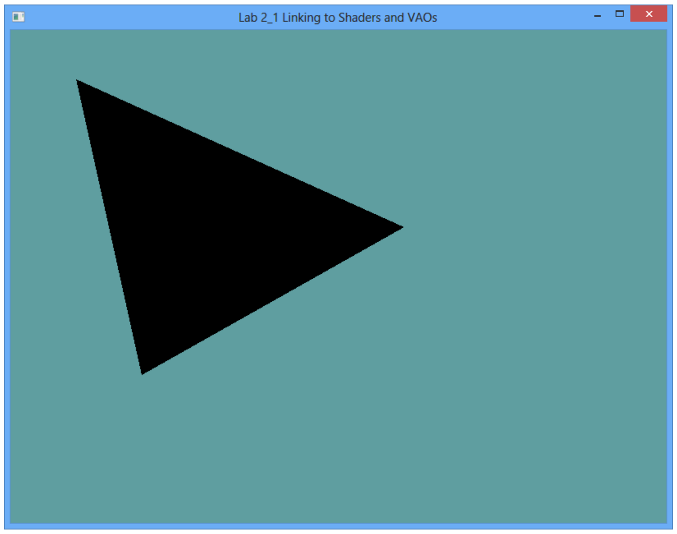
This is the second lab for Simulation and 3D Graphics. In the last lab we learnt how vertex buffer objects are loaded onto the graphics card, and how to use draw calls to build triangles by sending vertices through the pipeline. In this lab we will take a closer look at two of the programmable parts of the pipeline – the vertex shader and the fragment shader. Before we begin, update your repository to pull any changes that might have been made to the labs from SVN.

If you have any problems (or spot a mistake) one way you can get help is on the forum thread for lab 2.1 here at http://forums.net.dcs.hull.ac.uk/t/1938.aspx

In the Program.cs file change the m_CurrentLab value to Lab.L2_1

```
private static Lab m_CurrentLab = Lab.L2_1;
```

This will run the correct window for this lab. In the solution explorer window in the Lab2 folder open Lab2_1Window.cs. Hit F5 and the solution should build and run without a problem. You should see the following window.



Commit your code to SVN, add the log message

L21T1 Rendered a big black triangle!

It is important to understand how the graphics hardware works. If you have read the introduction document you should have a good idea, but just to reiterate. Shaders are small, C-like programs designed to be run on graphics hardware. The graphics hardware includes a parallel processor capable of running many threads at the same time. When a shader is run the same shader program is run for multiple sets of data at the same time (a “Single Program Multiple Data” parallelism model). In our simplified view of the graphics pipeline after the vertices are processed triangles are created, and the rasterizer turns these triangles into fragments. A fragment is similar to a pixel (in fact, fragment shaders are sometimes called pixel shaders). The difference is that a fragment can be thought of as the part of a triangle that inhabits a pixel in screen space. This means that because triangles can overlap you will likely have far more fragments than you do pixels.

Before we start programming, take a look in the Utility folder at the ShaderUtility class. This contains some helpers that should assist in shader loading. The ShaderUtility constructor takes two strings, which should be the file path for a vertex shader and a fragment shader pair.

```
StreamReader reader;
VertexShaderID = GL.CreateShader(ShaderType.VertexShader);
reader = new StreamReader(pVertexShaderFile);
GL.ShaderSource(VertexShaderID, reader.ReadToEnd());
reader.Close();
GL.CompileShader(VertexShaderID);

int result;
GL.GetShader(VertexShaderID, ShaderParameter.CompileStatus, out result);
if (result == 0)
{
    throw new Exception("Failed to compile vertex shader!" +
GL.GetShaderInfoLog(VertexShaderID));
}
```

This code creates a vertex shader and stores the ID. This is similar to the way OpenGL handles vertex buffers. Next the file is read and the contents is sent to the shader ID as the source code. The reader is closed, and then the shader is compiled. Next we check to see that the shader compiled successfully, and if not we throw and exception.

```
FragmentShaderID = GL.CreateShader(ShaderType.FragmentShader);
reader = new StreamReader(pFragmentShaderFile);
GL.ShaderSource(FragmentShaderID, reader.ReadToEnd());
reader.Close();
GL.CompileShader(FragmentShaderID);

GL.GetShader(FragmentShaderID, ShaderParameter.CompileStatus, out result);
if (result == 0)
{
    throw new Exception("Failed to compile fragment shader!" +
GL.GetShaderInfoLog(FragmentShaderID));
}
```

The fragment shader is processed in the same way.

```
ShaderProgramID = GL.CreateProgram();
GL.AttachShader(ShaderProgramID, VertexShaderID);
GL.AttachShader(ShaderProgramID, FragmentShaderID);
GL.LinkProgram(ShaderProgramID);
```

Finally, a shader program is created and the vertex and fragment shaders are attached to the shader program. The shader program is linked, meaning that the output of the vertex shader is linked to the input of the fragment shader, which must match one another. Now our shader is ready to use.

Before we finish looking in ShaderUtility take a quick look at the Delete method.

```
public void Delete()
{
    GL.DetachShader(ShaderProgramID, VertexShaderID);
    GL.DetachShader(ShaderProgramID, FragmentShaderID);
    GL.DeleteShader(VertexShaderID);
    GL.DeleteShader(FragmentShaderID);
    GL.DeleteProgram(ShaderProgramID);
}
```

This detachs the shader programs from the shader and deletes them all.

Next let’s look at the shader files themselves. For this lab, the shaders that are being loaded are in the Shaders folder inside the Lab2 folder.

```
#version 330

in vec2 vPosition;

void main()
{
        gl_Position = vec4(vPosition, 0, 1);
}
```

The “in” keyword signifies that this variable will be provided as a per vertex attribute, and should come from the previous stage in the pipeline. In this case the variable is a vec2 type (a 2d floating point vector). The shader’s entry point is the main function. This sets gl_Position to a vec4 with x and y values from the vPosition input, a z value of 0 and a w value of 1 (w comes from the fact we are working in a homogenous 4 dimensional space. Don’t worry about it for now). gl_Position is a built in output variable of type vec4 (4d floating point vector). This gets passed in to the fragment shader.

```
#version 330

uniform vec4 uColour;

out vec4 FragColour;

void main()
{
        FragColour = uColour;
}
```

The fragment shader shown above includes a uniform vec4 variable uColour. The “uniform” keyword signifies that this variable is linked from the application, and is not a per fragment value. Here the output is simple set to the uniform colour. We’ll play around with this soon, so you should be able to better understand what is happening.

Next we should take a look at is in the Lab2_1Window.cs file in the OnLoad method. There is a region that deals with linking to the shader. First we user the ShaderUtility to load, compile and link the two shaders.

```
mShader = new ShaderUtility(@"Lab2/Shaders/vLab21.vert", @"Lab2/Shaders/fSimple.frag");
```

Finally let’s take a look at what’s going on in the OnRenderFrame method.

```
GL.UseProgram(mShader.ShaderProgramID);

int vPositionLocation = GL.GetAttribLocation(mShader.ShaderProgramID, "vPosition");
GL.EnableVertexAttribArray(vPositionLocation);
GL.VertexAttribPointer(vPositionLocation, 2, VertexAttribPointerType.Float, false, 2 *
sizeof(float), 0);
```

Then we set OpenGL to actually use the shader we just loaded. Then we find the index for the location of vPosition in the shader and enable it. Next comes the really important stuff! When a buffer is loaded on to the graphics card it can contain all sorts of information. It could contain position information, and colour information, and texture information. This region tells the shader program how the data in the ArrayBuffer should be interpreted. In this case, we’re passing in 2 float values which should correspond to the vPosition vec2 attribute in the vertex shader. First we get the location of the “vPosition” attribute from the shader program. This should be a non-negative value. A vPositionLocation value of -1 indicates that something has gone wrong. Next, we enable that parameter. Finally, we provide a pointer to the data for vPosition. The parameters are the attribute location, the number of variables in the attribute, the type of the variables, whether the variable should be normalized or not, the stride of the entire vertex format in bytes, and the offset from the start of the vertex definition. Again, this method should become clear as we use more complex vertex definition and shaders.

It should be noted that this is not the only way to link buffer data to attributes in the shader, but we will be using this method during these labs.

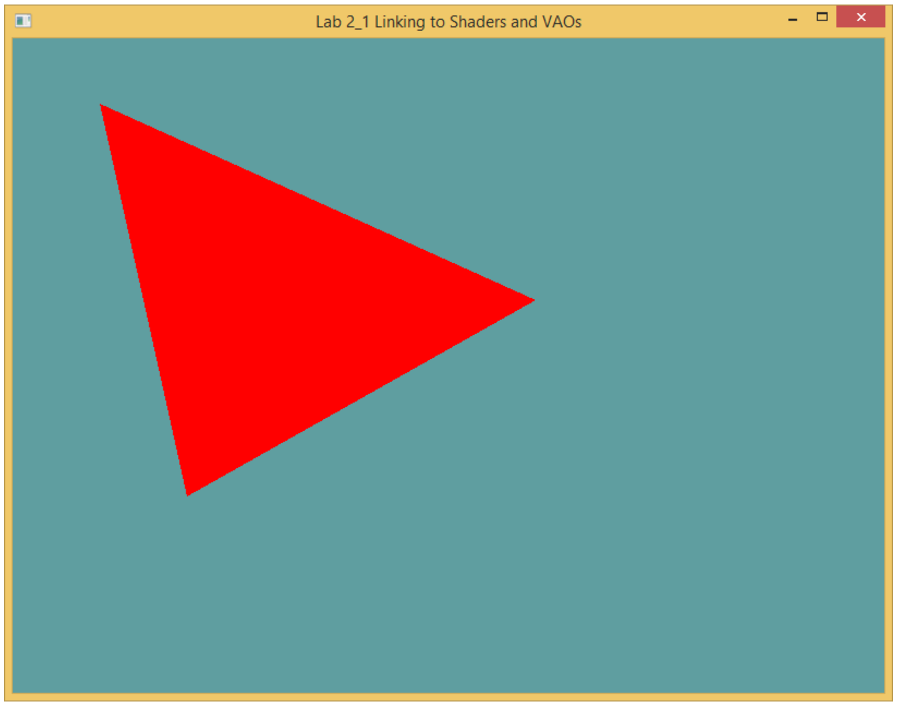
Setting the Fragment Colour

Remember that the fragment shader contained a uniform colour variable. As we have not set that value the fragments are all coloured black. Let's change that.

In the OnRenderFrame method add the following lines before the draw call.

```
int uColourLocation = GL.GetUniformLocation(mShader.ShaderProgramID, "uColour");
GL.Uniform4(uColourLocation, Color4.Red);
```

This sets the uColour variable in the fragment shader to red. Run the code and you should see that your black triangle has turned to red.

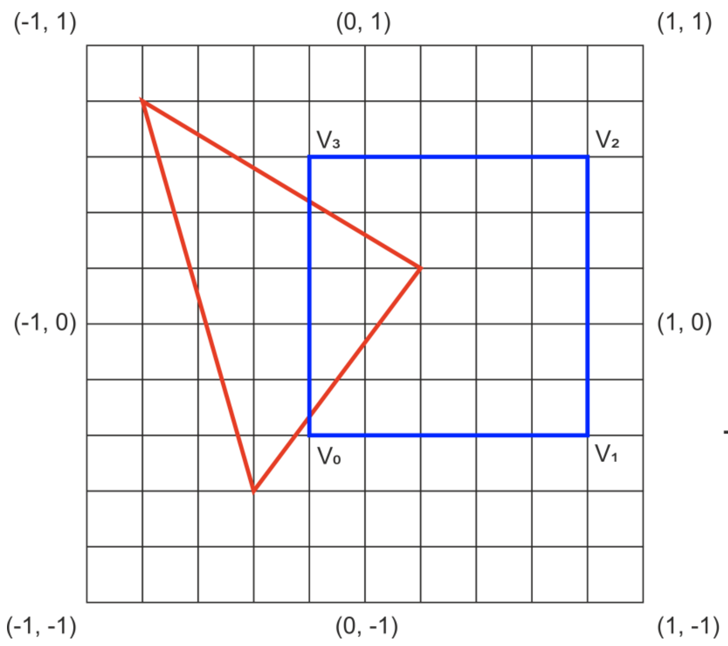


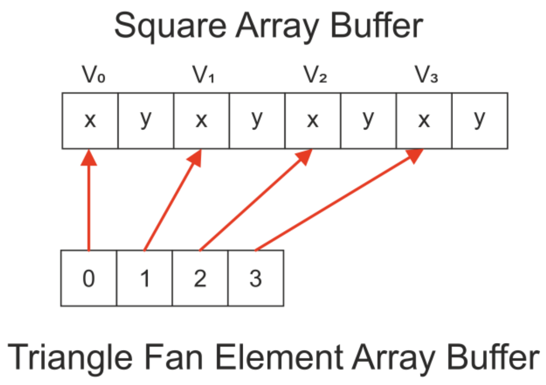
Commit to SVN with the message

L21T2 Set uniform variable in the fragment shader to colour all fragments red

Next let's draw a square with the triangle. Theoretically we could do this using the same buffers we already have, but that wouldn't really make a lot of sense, so use what you learnt in the last lab to create a new array buffer and element buffer for the square. To make the code clearer I've changed the existing array of buffer IDs from being called mVertexBufferObjectIDArray to mTriangleVertexBufferObjectIDArray. To make sure that this change is propogated throughout your code double click the current variable name and press the F2 key to bring up visual studio's renaming tool.

For the square create a second mSquareVertexBufferObjectIDArray. You can work out the vertex data values from the diagram below.





In the OnLoad method you will need to:

- Add an member variable array of two integers called mSquareVertexBufferArray
- Get OpenGL to generate Buffer IDs for these two new arrays
- Create temporary data arrays for square vertex data and indices (see diagram)
- Bind the data buffer
- Load the data buffer
- Test the load was successful
- Bind the element buffer
- Load the element buffer
- Test the load was successful

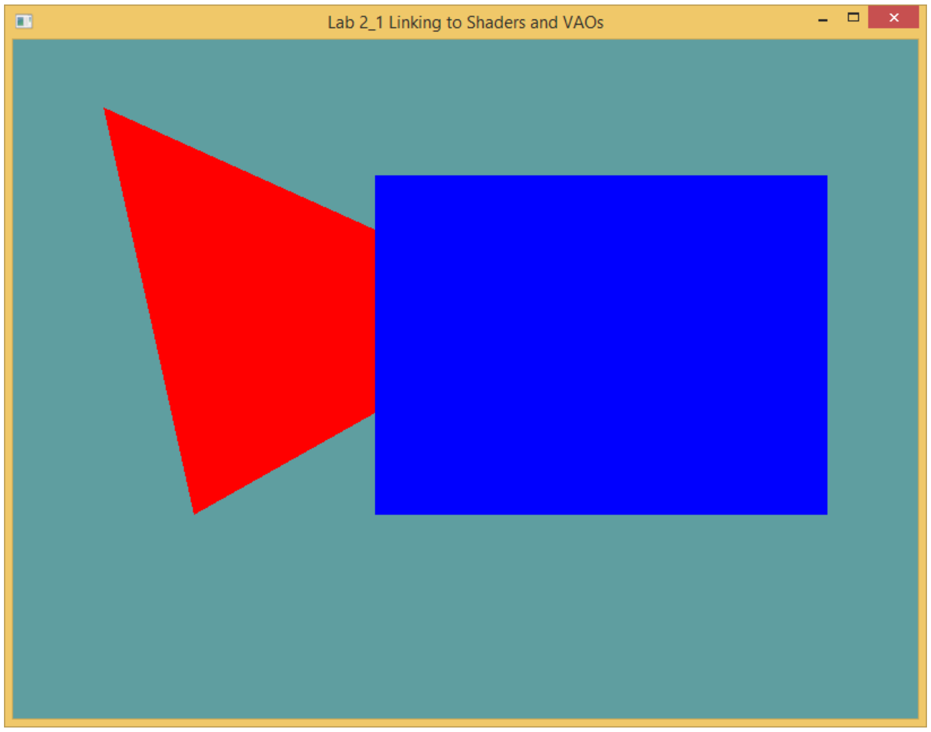
You will also need to remember to delete the new buffer IDs in the OnUnload method.

If you've done all that, let's take a closer look at what we need in the OnRenderFrame method. The following code is added to draw the square using the new buffers:

```
GL.Uniform4(uColourLocation, Color4.Blue);

GL.BindBuffer(BufferTarget.ArrayBuffer, mSquareVertexBufferObjectIDArray[0]);
GL.BindBuffer(BufferTarget.ElementArrayBuffer, mSquareVertexBufferObjectIDArray[1]);
GL.VertexAttribPointer(vPositionLocation, 2, VertexAttribPointerType.Float, false, 2 *
sizeof(float), 0);
GL.DrawElements(PrimitiveType.TriangleFan, 4, DrawElementsType.UnsignedInt, 0);
```

The first line changes the uniform variable in the fragment shader to blue. All pixels sent to the fragment shader will now be blue! Next we bind the data buffer, and bind the element array buffer, then we link these new buffers to the shader and finally we make the draw call.



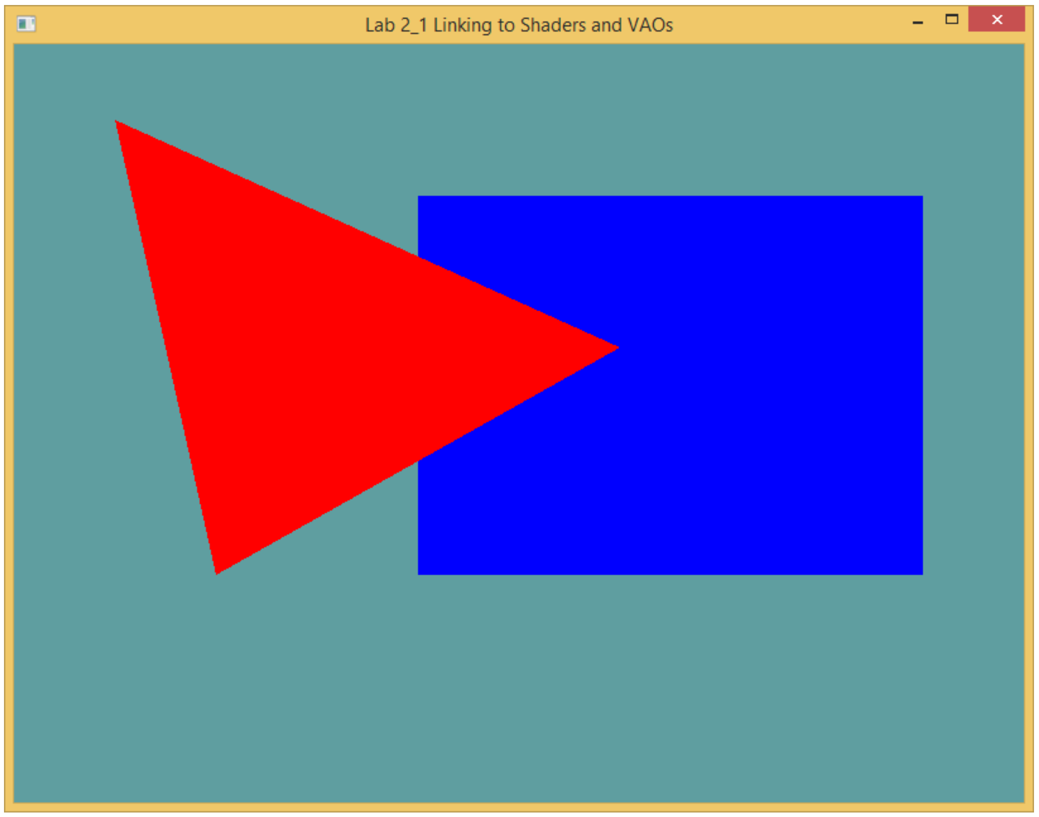
Once you’re drawing the triangle and the square commit to SVN with the message

L21T3 Added a blue square by creating new data and element arrays, linking the appropriate variables to the shader and changing the fragment shaders uniform colour variable

Note that the red triangle appears to be behind the blue square. That’s not really the case. In fact, the rasterizer generated a bunch of fragments for the triangle that were sent through the fragment shader, but we can’t see. That’s because the draw call for the blue square was made after the red triangle, so the fragments from the red triangle were overwritten by the fragments from the blue square.

See if you can swap the order so that the red triangle is on top of the blue square.

Hint: It’s not just as simple as swapping the draw calls, you also have to consider how the buffers are bound and how the shader is linked etc. The order that things happen in is important!



Once you've done that commit to SVN with the message

L21T4 Moved the red triangle so it is drawn on top of the blue square

Next let's try and move the red triangle so it is actually deeper than the blue square. First, we need to add a depth value (z) to each of the vertices in for both the triangle and the square.

We want the square to be behind the triangle. That means that we need to give all the vertices in both the triangle and the square a z value. First we need to give every pixel a depth value.

```
float[] squareVertices = new float[] { -0.2f, -0.4f, 0.2f,
                                        0.8f, -0.4f, 0.2f,
                                        0.8f, 0.6f, 0.2f,
                                       -0.2f, 0.6f, 0.2f};

float[] triangleVertices = new float[] { -0.8f, 0.8f, 0.4f,
                                         -0.6f, -0.4f, 0.4f,
                                          0.2f, 0.2f, 0.4f};
```

The size of the data we are loading will be updated automatically, because we are using the length of the array in the parameter calculation. We do need to change the attribute declarations for both the triangle and the square.

Instead of two floats, each position now contains three floats. Change the two lines that say

```
GL.VertexAttribPointer(vPositionLocation, 2, VertexAttribPointerType.Float, false, 2 *
sizeof(float), 0);
```

To

```
GL.VertexAttribPointer(vPositionLocation, 3, VertexAttribPointerType.Float, false, 3 *
sizeof(float), 0);
```

We also need to tell the vertex shader that vPosition now has three floats instead of two.

```
#version 330

in vec3 vPosition;

void main()
{
        gl_Position = vec4(vPosition, 1);
}
```

Run your code. You should see that the red triangle is still on top of the blue square. This is because like the face culling we need to enable depth testing. This means that for every fragment that is processed OpenGL will check the depth of any fragment that has already been processed for a particular pixel. If the fragment that has already been processed is further away the colour in the colour buffer is replaced by the current fragment colour. Otherwise the current fragment is thrown away.

In the OnLoad method enable depth testing.

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

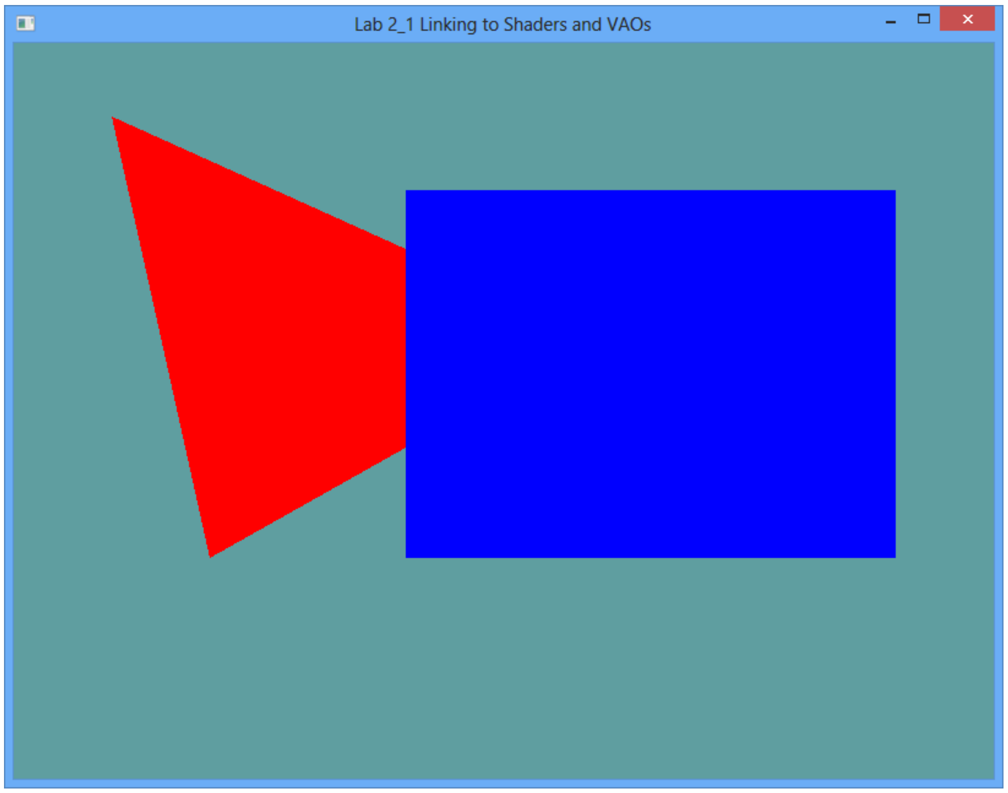
Like the colour buffer, inbetween frames we need to clear the depth buffer. Otherwise when we compare the depth of fragments for the next frame we will be using depth values from the last frame, which may have changed. To clear both the colour and the depth buffer in the OnRenderFrame method change the line that says

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

To

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

Now if you run your code you should see that the red triangle is behind the blue square



Commit your code to SVN with the message

L21T5 Added depth by adding an extra vertex parameter, changing the shader variable and linking and enabling depth testing

When we learnt about saving data by reusing vertices it was mentioned that the per vertex data might also store other values, such as colour. Let's see how that would work by assigning a colour per vertex. To do this we are going to use a technique called interleaved arrays. This means that we are going to store all the different types of data in the same array. Each vertex will be made up of six floats corresponding to the x, y and z position values and the red, green and blue colour values.

```
float[] squareVertices = new float[] { -0.2f, -0.4f, 0.2f, 0.0f, 1.0f, 1.0f,
                                        0.8f, -0.4f, 0.2f, 0.0f, 1.0f, 1.0f,
                                        0.8f, 0.6f, 0.2f, 0.0f, 1.0f, 1.0f,
                                       -0.2f, 0.6f, 0.2f, 0.0f, 1.0f, 1.0f};

float[] triangleVertices = new float[] { -0.8f, 0.8f, 0.4f, 1.0f, 0.0f, 1.0f,
                                         -0.6f, -0.4f, 0.4f, 1.0f, 0.0f, 1.0f,
                                          0.2f, 0.2f, 0.4f, 1.0f, 0.0f, 1.0f};
```

Once again, because we used the array length in the loading code we don't need to change that. We do need to change the shaders though.

```
#version 330

in vec3 vColour;
in vec3 vPosition;

out vec4 oColour;

void main()
{
        gl_Position = vec4(vPosition, 1);
        oColour = vec4(vColour, 1);
}
```

Now we've added a new in parameter. Remember in means that this is a per vertex attribute. We've also added an out parameter of type vec4 called oColour. This value is being passes out to the fragment shader. In the main function we simply add a fourth value of 1 to the colour (for the alpha channel) and put the new vec4 value into oColour to be sent along the pipe.

```
#version 330

in vec4 oColour;

out vec4 FragColour;

void main()
{
        FragColour = oColour;
}
```

Remember that the output of the vertex shader and the input of the fragment shader must match up, otherwise they won't link together properly. Here we've added an in vec4 parameter with the same name as the out vec4 parameter in the pixel shader. We've also removed the uniform colour variable as it is no longer needed. Be aware that if you don't use a variable the compiler will usually optimize that variable out. This can lead to some confusion if you are trying to link to a variable that isn't being used. The main body of the fragment shader is changed so that the FragColour output is given the value of oColour. Next, we should ensure that the colour attribute is enabled. As we are only using one shader and won't be changing this inbetween draw calls we only need do this once we can do this in the OnLoad method.

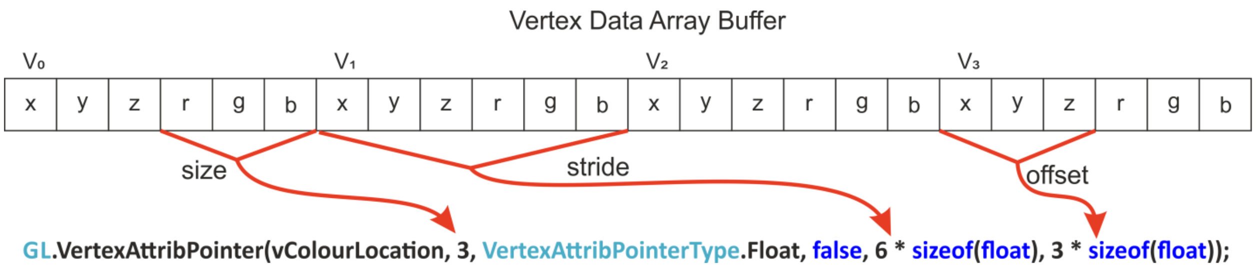
```
int vColourLocation = GL.GetAttribLocation(mShader.ShaderProgramID, "vColour");
GL.EnableVertexAttribArray(vColourLocation);
```

Finally, we need to link our data buffers to our shaders. This is done in the OnRenderFrame method. Change the linking code to look like this:

```
int vColourLocation = GL.GetAttribLocation(mShader.ShaderProgramID, "vColour");
int vPositionLocation = GL.GetAttribLocation(mShader.ShaderProgramID, "vPosition");

GL.VertexAttribPointer(vPositionLocation, 3, VertexAttribPointerType.Float, false, 6 *
sizeof(float), 0);
GL.VertexAttribPointer(vColourLocation, 3, VertexAttribPointerType.Float, false, 6 *
sizeof(float), 3 * sizeof(float));
```

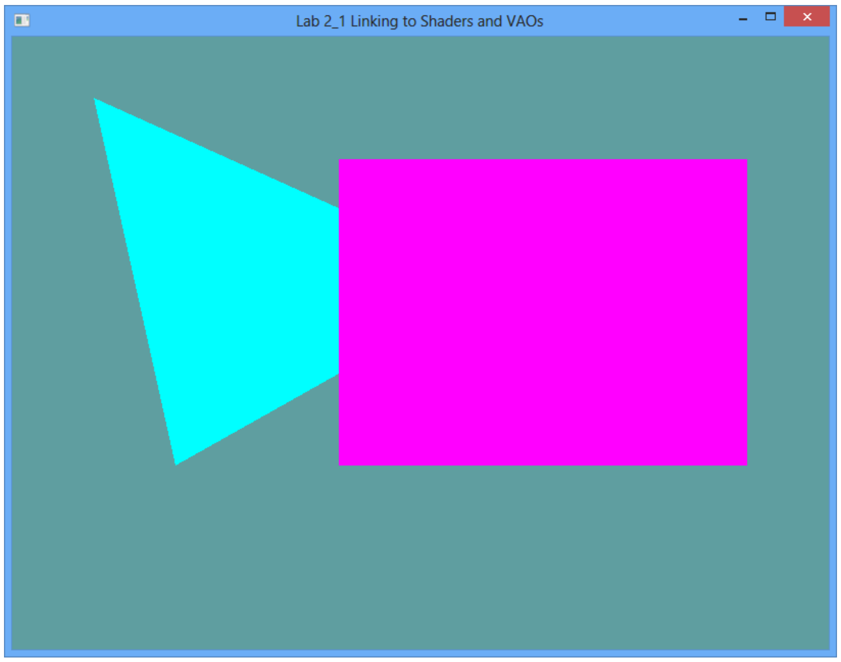
This needs to be done for both shapes that you're drawing. Let's take some time to consider what the changes to the parameters mean. Below is an example of this function as used for the per vertex colour attribute.



Vertex Data Array Buffer
V₀
V₁
V₂
V₃
x y z r g b x y z r g b x y z r g b x y z r g b
size
stride
offset
GL.VertexAttribPointer(vColourLocation, 3, VertexAttribPointerType.Float, false, 6 * sizeof(float), 3 * sizeof(float));

The first parameter, vColourLocation is the index that refers to the colour attribute's location in the shader. The second is the number of elements this attribute uses. In this case there is a float for the red, green and blue components of the colour, so there are 3 floats. The VertexAttribPointerType specifies what type the pointer points to. The next parameter specifies if this value should be normalized as it is passed through the pipeline. Next is the stride. The stride is the size in bytes of the whole vertex definition. In this case, the vertex definition consists of a float for the x, y, z, r, g and b values, so the stride is 6 * sizeof(float). Finally, the offset specifies the offset in bytes from the start of the vertex definition to the start of the colour definition. When picking out colours we need the pipeline to skip over the x, y and z values and start at the r value. The offset is 3 * sizeof(float).

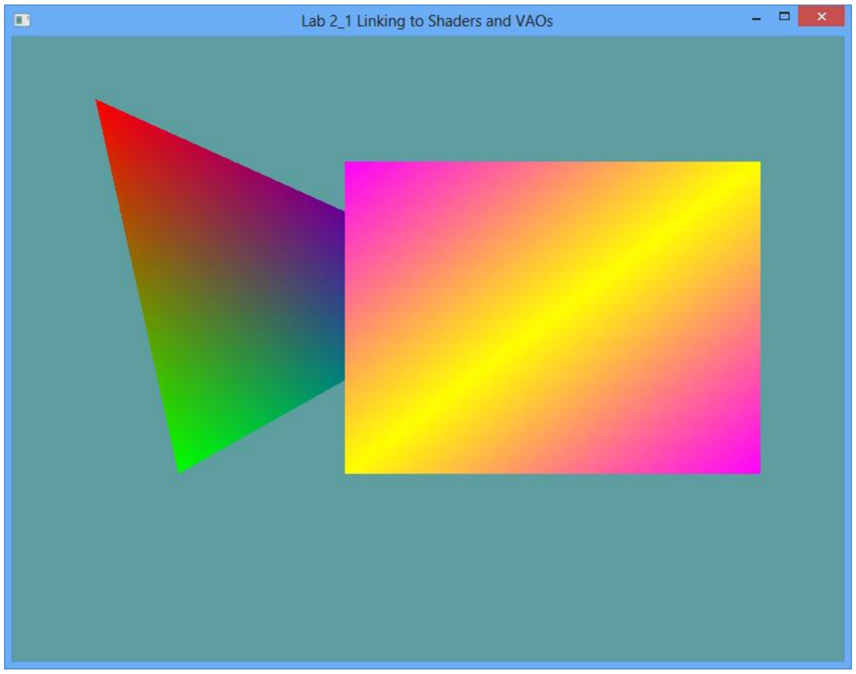
Run your code and hopefully you should see something like this:



If you do then commit to SVN with the log message

L21T6 Added per vertex colour to the vertex shader, passed it to the fragment shader and linked the data buffer to the shader program

Right now we're providing a colour value per vertex. We're using the same colours for every vertex in a shape and we're getting a solid block of colour. What would happen if vertices that are being used in a triangle had different colour values? Let's try it and find out!



When per vertex values are passed to the fragment shader, for the fragments in between vertices the per vertex values are interpolated between the three values from the three vertices that make up the triangle!

Commit your work to SVN. Give it a log message of

L21T7 Changed vertex colours to see fragments colour values being blended together

Right now your OnRenderFrame method probably looks a bit like this one – a bit of a mess.

```
protected override void OnRenderFrame(FrameEventArgs e)
{
  base.OnRenderFrame(e);
  GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);

  GL.UseProgram(mShader.ShaderProgramID);

  int vColourLocation = GL.GetAttribLocation(mShader.ShaderProgramID, "vColour");
  int vPositionLocation = GL.GetAttribLocation(mShader.ShaderProgramID, "vPosition");

  GL.BindBuffer(BufferTarget.ArrayBuffer, mSquareVertexBufferObjectIDArray[0]);
  GL.BindBuffer(BufferTarget.ElementArrayBuffer, mSquareVertexBufferObjectIDArray[1]);
  GL.VertexAttribPointer(vPositionLocation, 3, VertexAttribPointerType.Float, false, 6 *
sizeof(float), 0);
  GL.VertexAttribPointer(vColourLocation, 3, VertexAttribPointerType.Float, false, 6 *
sizeof(float), 3 * sizeof(float));
  GL.DrawElements(PrimitiveType.TriangleFan, 4, DrawElementsType.UnsignedInt, 0);

  GL.BindBuffer(BufferTarget.ArrayBuffer, mTriangleVertexBufferObjectIDArray[0]);
  GL.BindBuffer(BufferTarget.ElementArrayBuffer, mTriangleVertexBufferObjectIDArray[1]);
  GL.VertexAttribPointer(vPositionLocation, 3, VertexAttribPointerType.Float, false, 6 *
sizeof(float), 0);
  GL.VertexAttribPointer(vColourLocation, 3, VertexAttribPointerType.Float, false, 6 *
sizeof(float), 3 * sizeof(float));
  GL.DrawElements(PrimitiveType.Triangles, 3, DrawElementsType.UnsignedInt, 0);

  this.SwapBuffers();
}
```

There’s a lot of state switching going on every frame, and pretty much every call is a call to the graphics card, which is slow. The states that we’re switching to aren’t changing either. Surely we could do this is a smarter way.

That’s where Vertex Array Objects come in. Vertex Array Objects store a set of state on the graphics card, so that we can switch between the states with a single call. We need to create two vertex array objects; one for the triangle and one for the square. Like the buffer objects we use an ID number refer to vertex array objects on the graphics card.

The goal is to end up with a nice, clean OnRenderFrame method like this:

```
protected override void OnRenderFrame(FrameEventArgs e)
{
        base.OnRenderFrame(e);
        GL.Clear(ClearBufferMask.ColorBufferBit | ClearBufferMask.DepthBufferBit);

        GL.BindVertexArray(mVertexArrayObjectIDs[1]);
        GL.DrawElements(PrimitiveType.TriangleFan, 4, DrawElementsType.UnsignedInt, 0);

        GL.BindVertexArray(mVertexArrayObjectIDs[0]);
        GL.DrawElements(PrimitiveType.Triangles, 3, DrawElementsType.UnsignedInt, 0);

        GL.BindVertexArray(0);

        this.SwapBuffers();
}
```

Note how all the state change we did before is reduced to a single call that binds a new Vertex Array. This encompasses binding the correct data and element buffer, linking the shader, and enabling vertex attributes in just one call. All the hard work is now done setting up the vertex array objects in the OnLoad method. Note that at the end we bind an array index of 0. This effectively unbinds the current Vertex Array Object so that we don't accidently change its state elsewhere in our code.

Before we look at the OnLoad function we need to create an array for our Vertex Array Object ID numbers.

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

Next we need to make some changes in our OnLoad function. I'll give you the code to deal with the vertex array object for the triangle. You need to set up the other vertex array object for the square.

```
GL.GenVertexArrays(2, mVertexArrayObjectIDs);

GL.BindVertexArray(mVertexArrayObjectIDs[0]);
GL.BindBuffer(BufferTarget.ArrayBuffer, mTriangleVertexBufferObjectIDArray[0]);
GL.BindBuffer(BufferTarget.ElementArrayBuffer, mTriangleVertexBufferObjectIDArray[1]);
GL.VertexAttribPointer(vPositionLocation, 3, VertexAttribPointerType.Float, false, 6 *
sizeof(float), 0);
GL.VertexAttribPointer(vColourLocation, 3, VertexAttribPointerType.Float, false, 6 *
sizeof(float), 3 * sizeof(float));
GL.EnableVertexAttribArray(vColourLocation);
GL.EnableVertexAttribArray(vPositionLocation);
```

This sort of code should be getting pretty familiar by now. We get some IDs from the graphics card, then we bind one of them to make it current, then we set up the state for that thing. You need to take care to do things in a sensible order. For example, set up the buffers and vertex attributes before we load the buffers and create the shader. A good order of working would be to load the shader, then load the buffers, then set up the vertex array object.

Set up the Vertex Array Object for the square in a similar way. When you are done you should get the same result that you did before. The last thing to do is to delete the vertex array objects in the OnUnload method.

```
GL.BindVertexArray(0);
GL.DeleteVertexArrays(2, mVertexArrayObjectIDs);
```

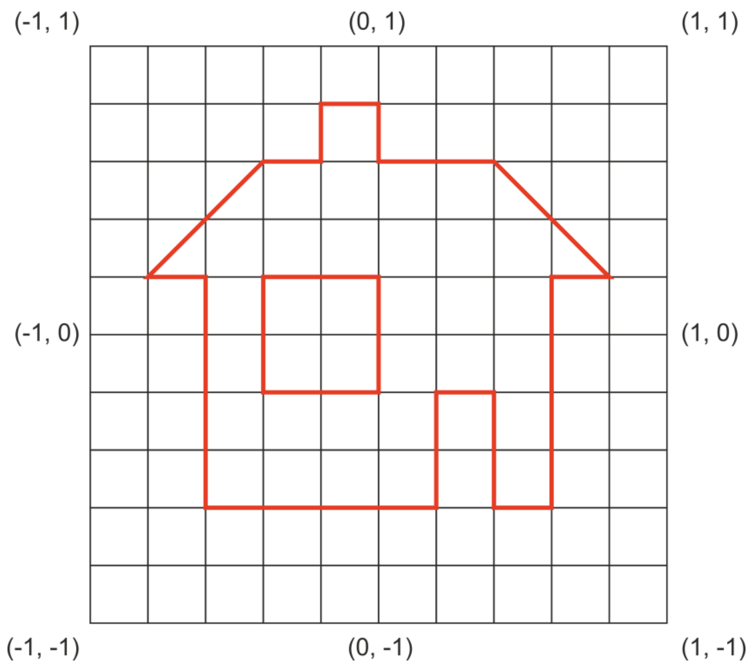
When you've done that commit your code with the message:

L21T8 Made state changes easier using Vertex Array Objects

There is one more thing we can do to make our easier to follow.

Challenge: Build a better house!

Use what you've learnt to build a house like the one you made last week!



When you've finished making your house submit it to SVN with the (completed) log message:

L21T9 Made a better house by using ...

Summary

In this lab we've learnt a little about programming shaders. We've looked at uniform variables that don't change very often, and vertex attributes that change for every vertex. We've learnt how to link data in a buffer array to the appropriate vertex attributes. Finally we've learnt how to group sets of state together on the graphics card in Vertex Array Objects, to make changing between different sets of state a far easier process. We also covered how the graphics pipeline deals with depth.