08214 Lab 5: Textures

In this lab we will learn how to deal with textures. First you'll learn how to load a texture, and how to define how a texture is wrapped onto a mesh using texture coordinates. Next you'll learn how to use two textures and the same time. Finally you'll create a nice animated dissolving effect using two textures.

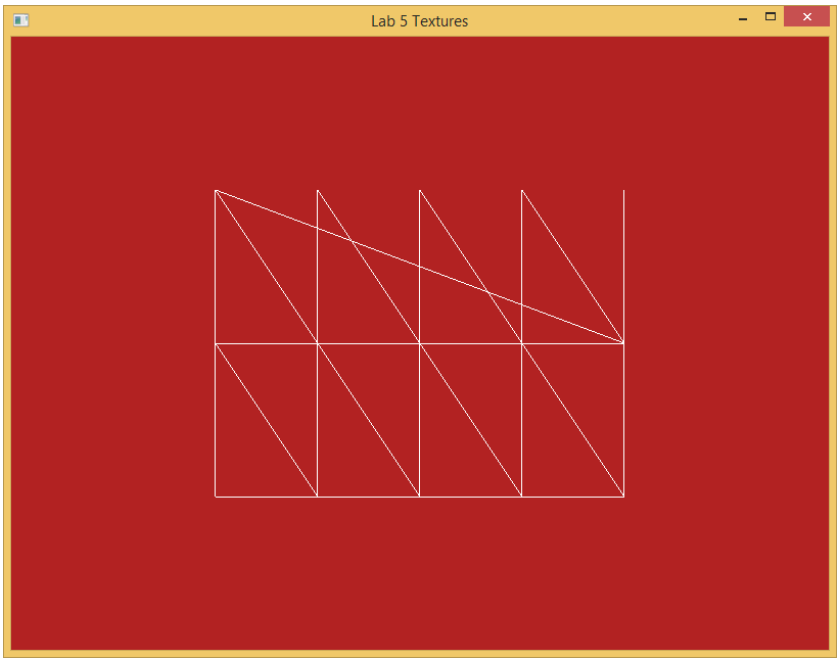
If you have any problems (or spot a mistake) one way you can get help is on the forum thread for lab 5 here at

http://forums.net.dcs.hull.ac.uk/t/1989.aspx

In the Program.cs file change the m_CurrentLab value to Lab.L5

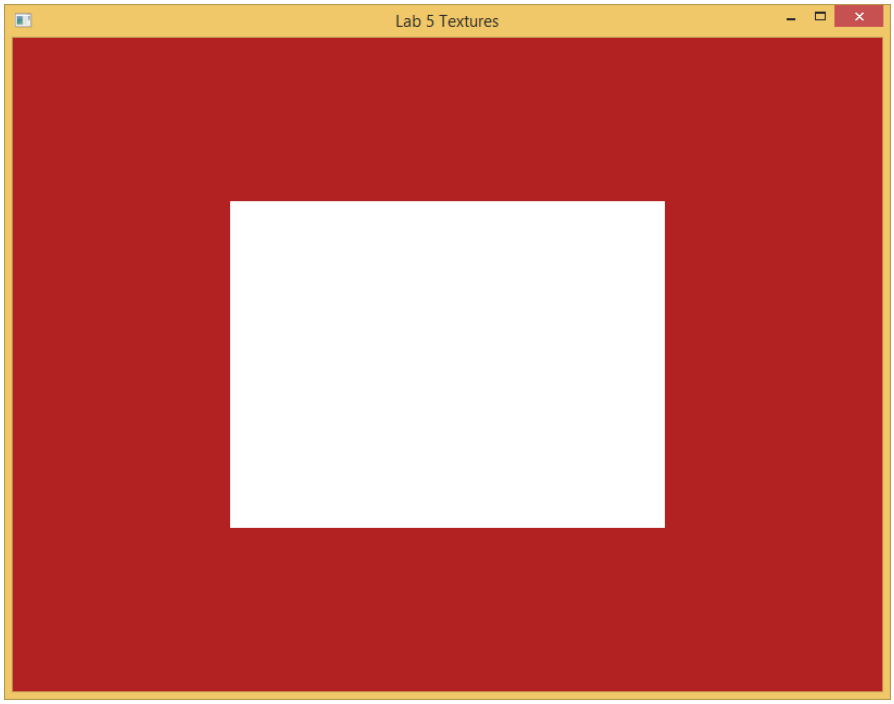
```
private static Lab m_CurrentLab = Lab.L5;
```

This will run the correct window for this lab. When you run the code you should see something like this:



This shows a series of triangle. They've been rendered using a line strip to demonstrate that there are several triangles and not just one. This will help you learn about texture coordinates and how they work.

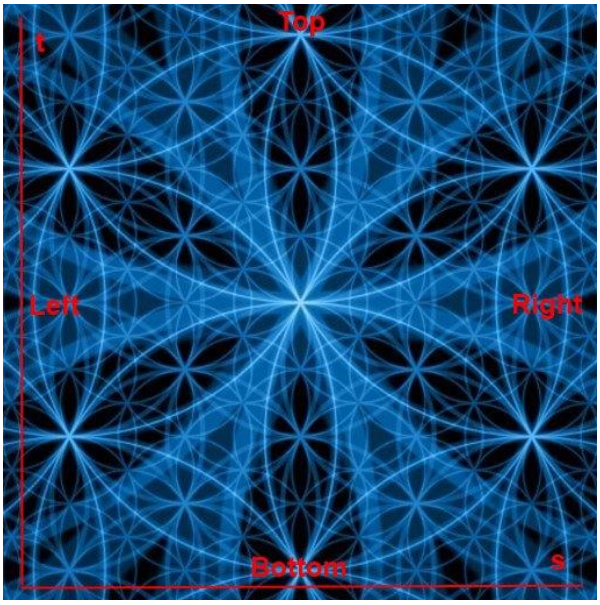
In the draw call change the primitive type to Triangles, and then run the code again.



Commit your code to SVN.

L5T1 Changes primitive type from lines to triangles

You will also need a texture file. You can use any image you like, however, on older hardware it's best to use images that have dimensions that are powers of two (e.g. 2, 4, 8, 16, 32, 64, 128, 256, 512). You can use any appropriate image you like. I'll be using this:



Note that I've annotated it to help later with texture coordinates. OpenGL uses texture coordinates that start at 0,0 at the bottom left and go to 1,1 at the top right. You should add this image to your project,

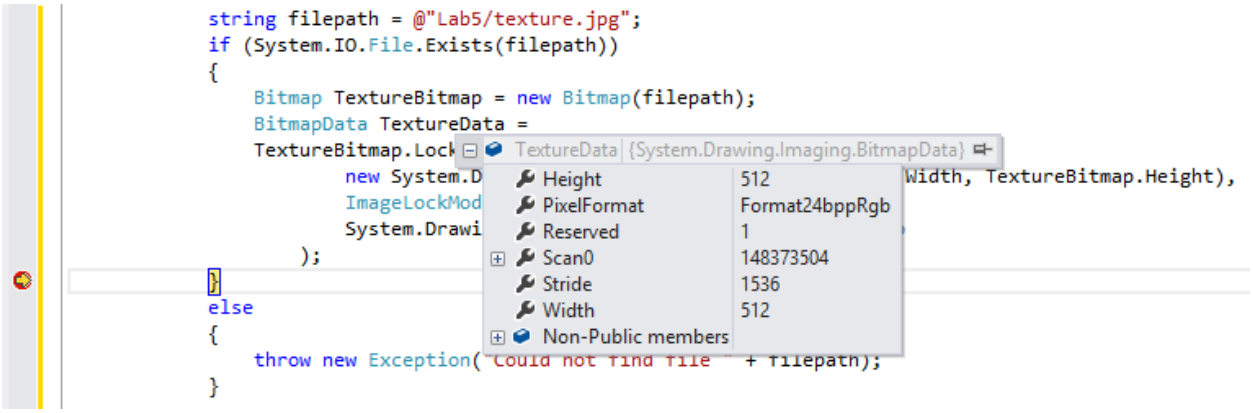
and set it to be copied to the output directory when the project is built. The first step is to load the texture into memory. This step has nothing to do with OpenGL. We're just trying to load the file from the hard disk into main memory.

```
string filepath = @"Lab5/texture.jpg";
if (System.IO.File.Exists(filepath))
{
        Bitmap TextureBitmap = new Bitmap(filepath);
        BitmapData TextureData = TextureBitmap.LockBits(
                new System.Drawing.Rectangle(0, 0, TextureBitmap.Width,
        TextureBitmap.Height), ImageLockMode.ReadOnly,
        System.Drawing.Imaging.PixelFormat.Format32bppRgb);
}
else
{
        throw new Exception("Could not find file " + filepath);
}
```

You will need to add some "using" statements for System.Drawing etc. The easiest way to do this is to highlight the lines that visual studio complains about and press ctrl and '.'. Visual studio should then figure out the correct using statements and offer to add them for you.

Note, be aware of adding System.Drawing.Imaging as some names in that namespace will clash with names in OpenTK. If you do this sometimes you will have to explicitly resolve between the two namespaces.

If you run your code nothing will happen yet, however, by adding a breakpoint we can get some idea that the texture was loaded correctly.



Commit your code to SVN.

L5T2 Loaded an image from the disk into memory

The next step is to get the texture from memory onto the graphics card. This process should be becoming pretty familiar to you. Just like with the Vertex Buffer Objects, Vertex Array Objects and Shader Programs in our main program we need to store a handle to the texture on the graphics card. First create an interger member variable to store the handle to the texture called mTexture_ID.

Next add the following code to the OnLoad method just after you load the texture.

```
GL.ActiveTexture(TextureUnit.Texture0);
GL.GenTextures(1, out mTexture_ID);
GL.BindTexture(TextureTarget.Texture2D, mTexture_ID);

GL.TexImage2D(TextureTarget.Texture2D,
        0, PixelInternalFormat.Rgba, TextureData.Width, TextureData.Height,
        0, OpenTK.Graphics.OpenGL.PixelFormat.Bgra,
        PixelType.UnsignedByte, TextureData.Scan0);

GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMinFilter,
(int)TextureMinFilter.Linear);
GL.TexParameter(TextureTarget.Texture2D, TextureParameterName.TextureMagFilter,
(int)TextureMagFilter.Linear);

TextureBitmap.UnlockBits(TextureData);
```

We set the active texture unit to be zero (the first texture unit). A texture unit is like a memory address for a texture on the graphics card. There are a limited number of texture units available (depending on the graphics card). However, texture handles allow us to load many more textures than we have texture units, and the swap them around during rendering with less of a performance hit.

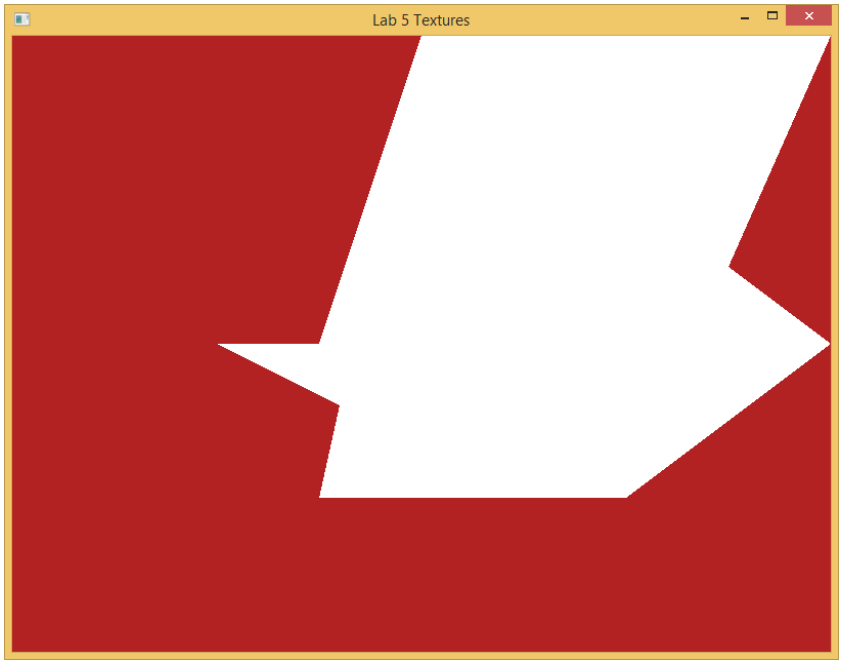
Next we generate a handle to a texture on the graphics card, as we have done many times before. Then we bind that hangle to the current texture unit. Next, the TexImage2D line loads the data from main memory to graphics memory. Finally, the TexParameter lines are instructions to do with mipmapping.

Check that your code compiles and commit to SVN.

L5T3 Loaded texture from memory onto the graphics card

So far so good, but we still need to modify our vertex buffer objects and our shaders. For every vertex we need to provide a coordinate in texture space. Add two floats (in the range 0.0f to 1.0f) to each vertex in the vertex buffer object. At this stage it doesn't matter what values you use, but use a variety.

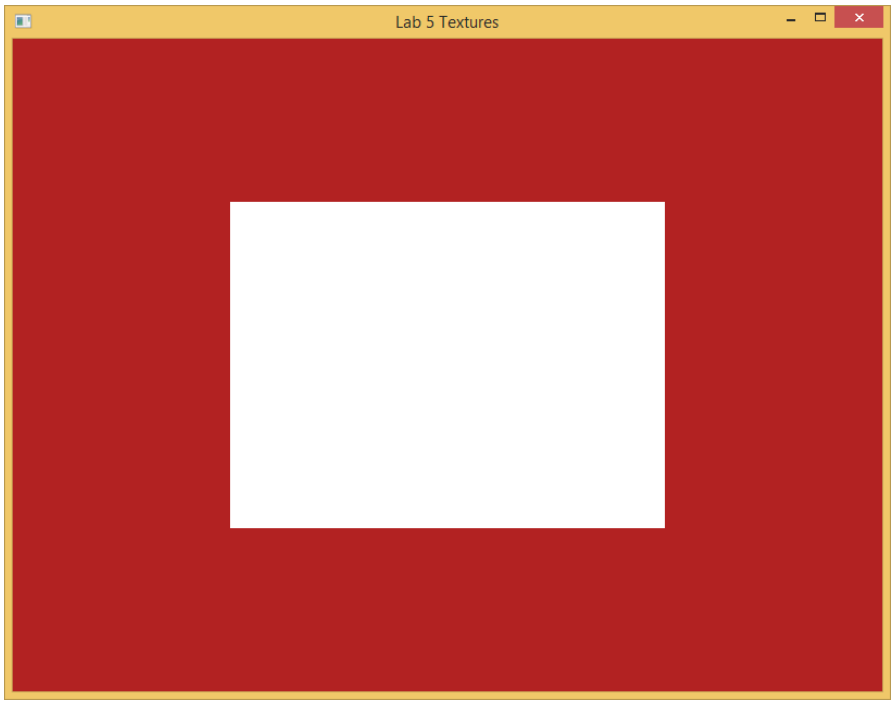
Run your code and you should see that your mesh has been messed up.



Commit your code to SVN with the following message:

L5T4 Added texture coordinates to vertices – mesh is broken

The mesh is broken because we didn't modify the commands that link the vertex position to the shader object. Fix the VertexAttribPointer line to deal with the new, larger stride and compile again.



Commit your code to SVN:

L5T5 Fixed mesh

Next, we need to add some code to the vertex and fragment shaders to read the texture coordinates from the vertex buffers. In the vertex shader we're just going to pass the texture coordinates to the fragment shader. Add a per vertex vec2 variable for the texture coordinates called vTexCoords and another called oTexCoords to pass out to the fragment shader. In the body of the vertex shaders just assign vTexCoords to oTexCoords.

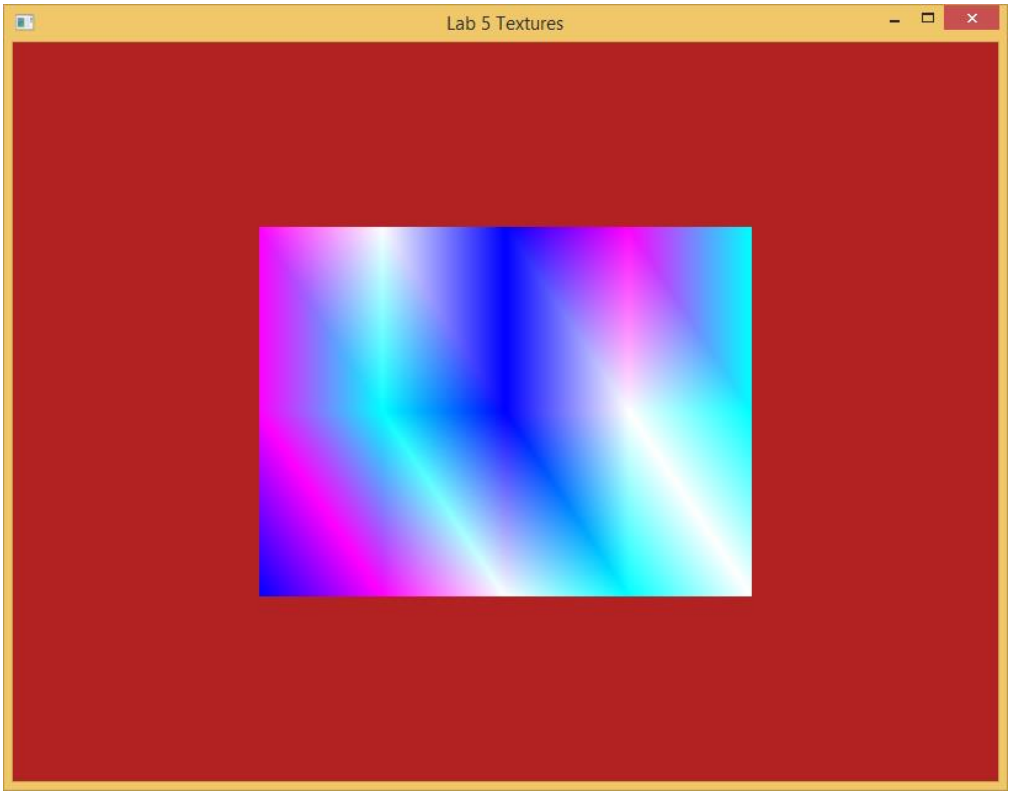
```
#version 330

in vec2 vPosition;
in vec2 vTexCoords;

out vec2 oTexCoords;

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

In the fragment shader take in the vec2 oTexCoords from the vertex shader and use them to colour the fragment.

```
#version 330

in vec2 oTexCoords;

out vec4 FragColour;

void main()
{
	FragColour = vec4(oTexCoords, 1, 1);
}
```

This won't do the texturing for us, but we will be able to visualize the texture coordinates in the buffer object, and be able to verify if they are being passed through the vertex shader into the fragment shader properly.

The final step is to link the vertex coordinates in the vertex buffer object to the vTexCoords variable in the shader. We've done this lots of times before so you should be used to that by now. You will also need to enable that variable.



Once you compile and run your code you should see something like this (depending on the values that you gave the texture coordinates). When you do that commit your code to SVN.

L5T6 Can visualize texture coordinates

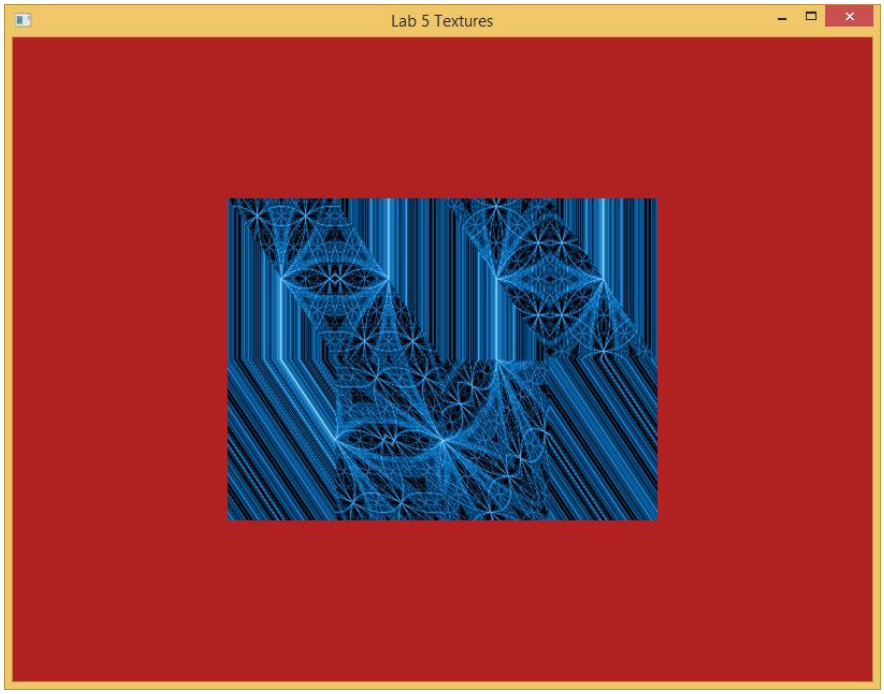
The final steps are to link the texture unit to a sampler2D in the fragment shader, and use that colour the fragments. Change the fragment shader by adding a uniform sampler2D variable, then use the texture function to look up the appropriate fragment colour from the texture.

```
#version 330

uniform sampler2D uTextureSampler;

in vec2 oTexCoords;

out vec4 FragColour;

void main()
{
        FragColour = texture(uTextureSampler, oTexCoords);
}
```

Finally, from the OnLoad method, just after the texture is loaded onto the graphics card set the sampler location to be zero (because we are using texture unit zero).

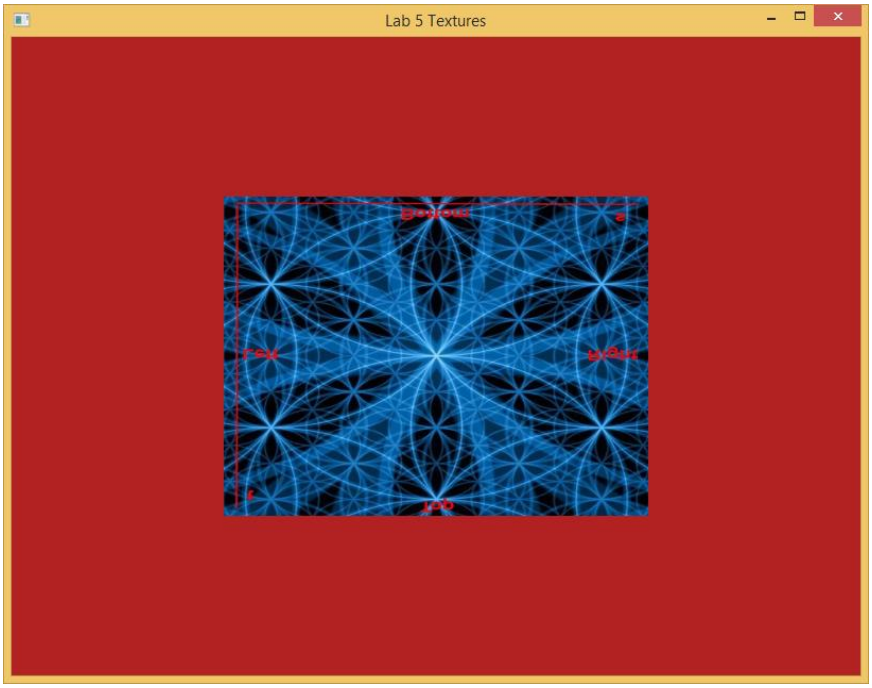
```
int uTextureSamplerLocation = GL.GetUniformLocation(mShader.ShaderProgramID,
"uTextureSampler");
GL.Uniform1(uTextureSamplerLocation, 0);
```

Run your code.



This isn't exactly what we wanted, but at least we have loaded a texture and used it to colour our triangles. Now the problem is that (unsurprisingly) our random allocation of texture coordinates has turned out a bit random. Work out what the texture coordinates should really be. Assuming you didn't use trial and error you'd probably notice something strange. The texture will appear to be upside down.

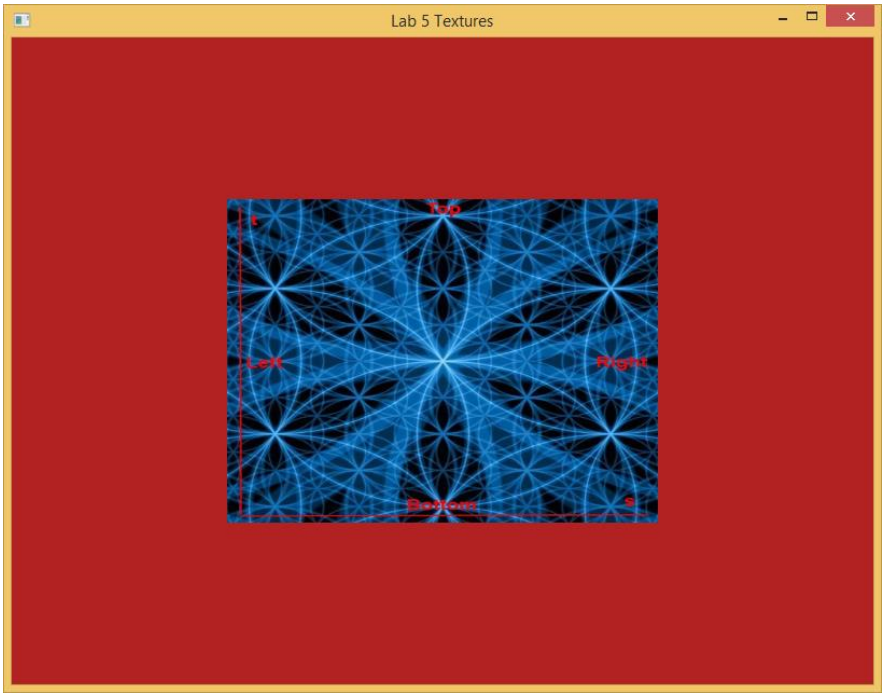
If this is the case there is nothing wrong with your texture coordinates. Instead, many image loading routines start from the top left, rather than the bottom left as openGL does.



We could compensate by changing our texture coordinates, but a better approach would be to flip the image before the load it onto the graphics card. Just after the texture is loaded, and before the data is extracted add the line to flip the image about the Y axis.

```
TextureBitmap.RotateFlip(RotateFlipType.RotateNoneFlipY);
```

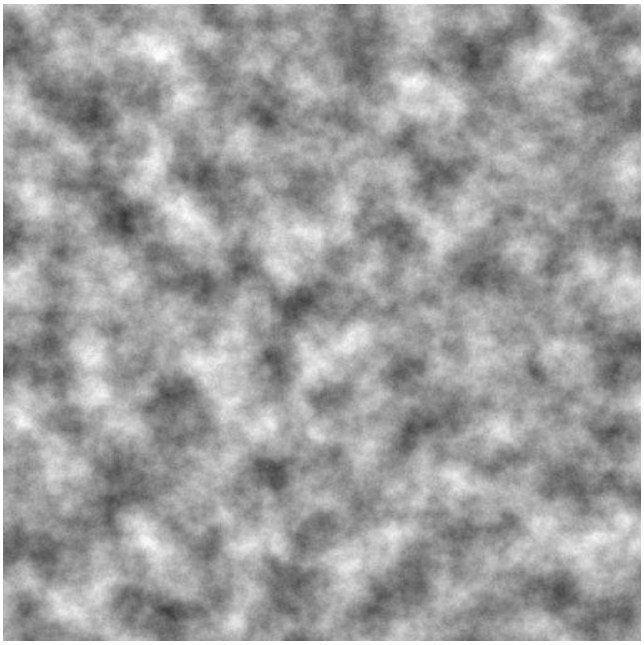
Check the fix worked and then check in to SVN.



L5T7 Flipped the image before loading onto the graphics card

The programmable pipeline offers huge flexibility, so you can use a texture for whatever you want. You could use it as a diffuse colour to add to your lighting model, or as a bump map to change the normal vectors, or as an alpha channel to make an object disappear.

The next section will demonstrate another way to use a texture, as well as using more than one texture at a time. I'm going to use this texture to build a dissolve effect on the mesh.

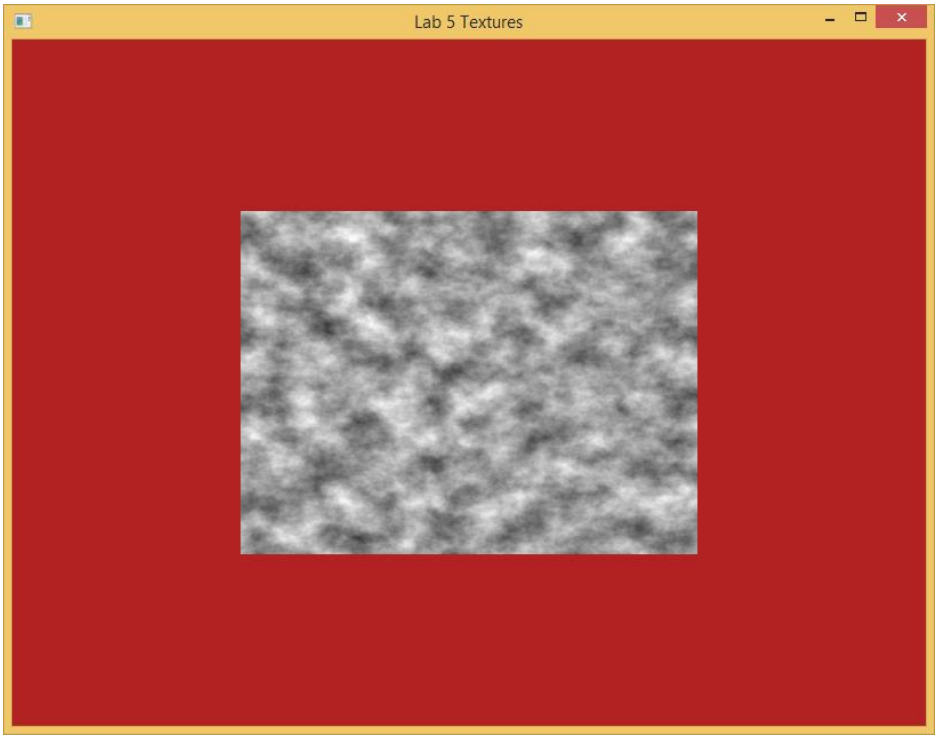


First we need to load the new texture in the same way that we did for the first texture. That means we need a new handle id. We also need to load the new texture into the next texture unit. The second texture should be loaded in the same way at the first, but this time we want to use the new texture handle, and a new texture unit. We also need a new sampler in the fragment shader to use. Create a second uniform sampler2D called uTextureSampler2 (Ideally you should come up with more meaningful names that uTextureSampler and uTextureSampler2).

Bind the new texture to texture unit 1 and link texture unit 1 to uTextureSampler2. Then, to verify that the new texture has loaded use uTextureSampler2 instead of uTextureSampler.

```
#version 330

uniform sampler2D uTextureSampler;
uniform sampler2D uTextureSampler2;

in vec2 oTexCoords;

out vec4 FragColour;

void main()
{
	FragColour = texture(uTextureSampler2, oTexCoords);
}
```

Run your code and you should see the new texture is used instead of the old one.



Commit your code to SVN

L5T8 Loaded a second texture

Now let's use both textures at the same time. This time we're going to use the shader keyword "discard" to discard a fragment in the value in the second texture is below some threshold. We could choose to hardcode that threshold in the shader, but instead let's create a uniform float to store the threshold and set it from our application.

```
uniform float uThreshold;
```

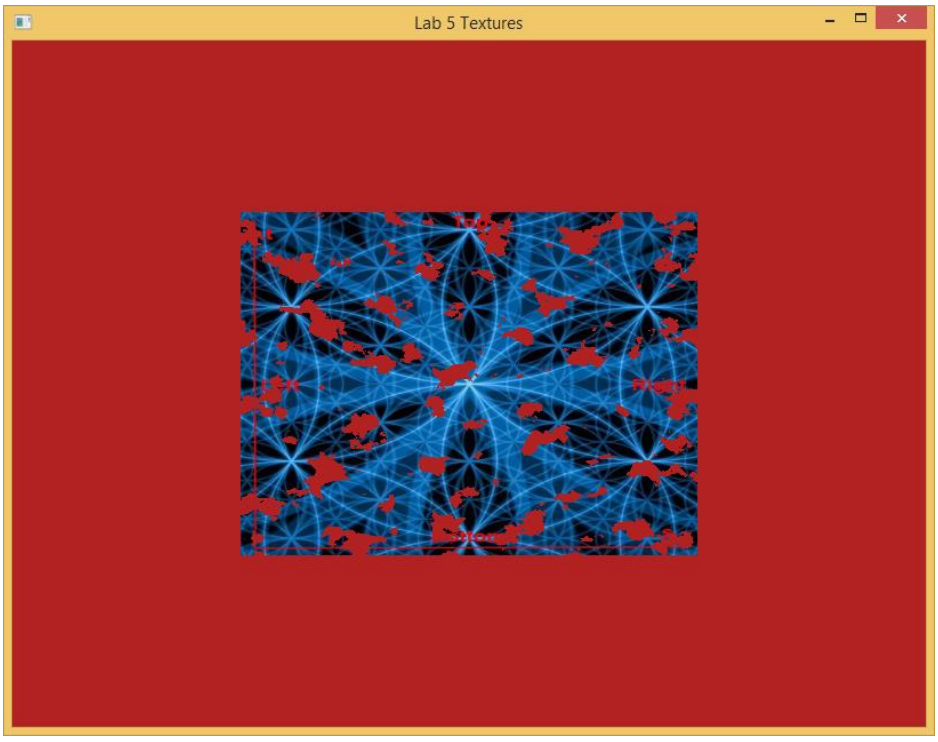
In the OnLoad method set uThreshold to some value between 0 and 1.

```
int uThresholdLocation = GL.GetUniformLocation(mShader.ShaderProgramID, "uThreshold");
GL.Uniform1(uThresholdLocation, 0.5f);
```

Finally, in the fragment shader write a condition to discard the fragment if the red channel of the second texture is less that the threshold value. Otherwise, select the fragment colour from the first texture.

```
if(texture(uTextureSampler2, oTexCoords).r < uThreshold)
{
        discard;
}
```

Run your code and you should see something similar to the image below.

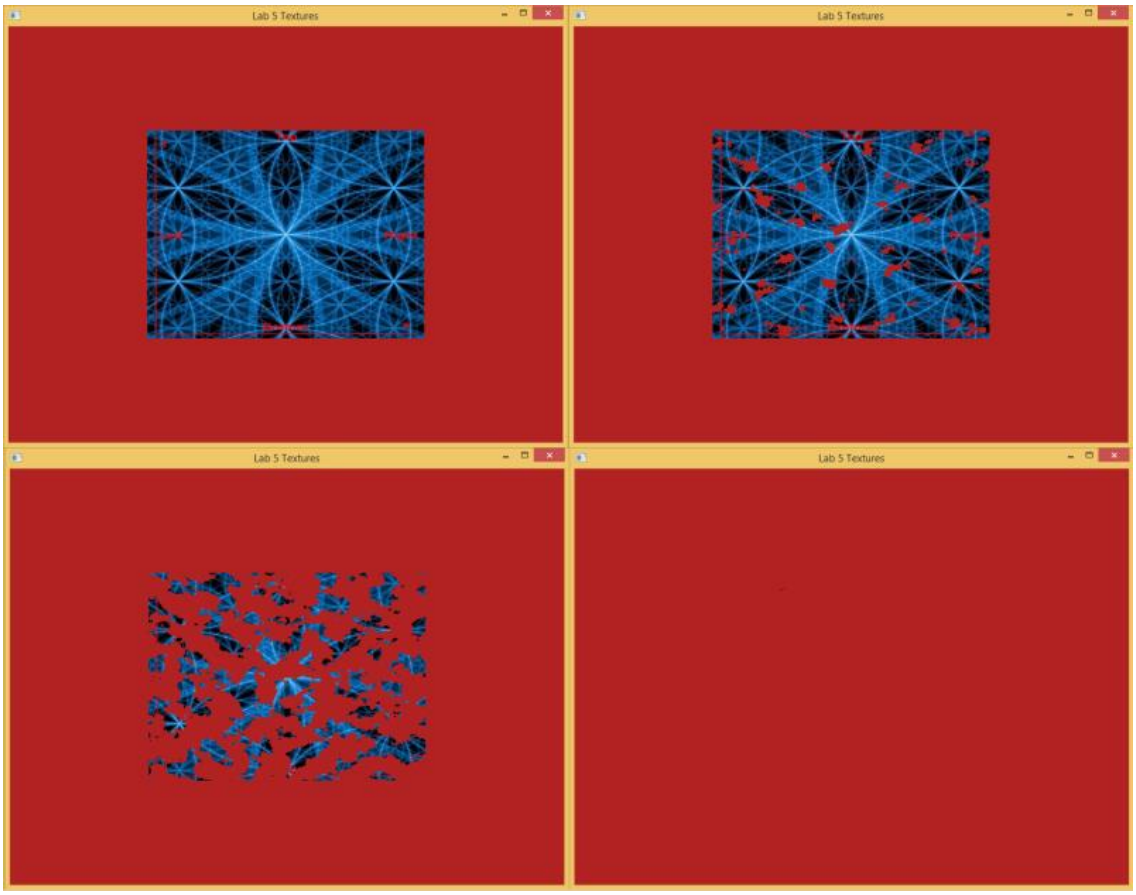


Commit your code to SVN with the message

L5T9 Used second texture to discard some fragments creating a dissolve effect

That effect is nice, but it would be better if it were a little more animated. To do this create an OnUpdateFrame method and alter the threshold value in there. You will need to store some variables and if you want your solution to be processor independent you should use the timestep to calculate the new threshold. The following code snippet should help you varying the threshold value from 0 to 1 and back down to 0.

```
float thresholdChange = mRateOfDissolve * timestep;
if (mThreshold + thresholdChange < 0 || mThreshold + thresholdChange > 1)
{
        mRateOfDissolve = -mRateOfDissolve;
}
mThreshold += mRateOfDissolve * timestep;
int uThresholdLocation = GL.GetUniformLocation(mShader.ShaderProgramID, "uThreshold");
GL.Uniform1(uThresholdLocation, mThreshold);
```



Commit your code to SVN

L5T10 Made a very cool animated dissolving effect