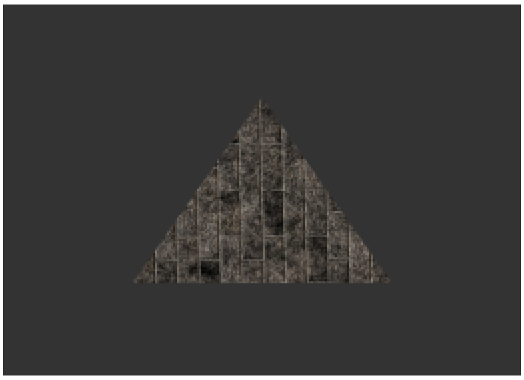
Tutorial 3: Texture Mapping



Summary

In this tutorial, you are going to learn about texture mapping, by performing some texture mapping operations on the triangle you've been using in the previous tutorials.

New Concepts

Texture maps, mip-mapping, texture filtering, texture coordinates, texture matrix, texture units, multitexturing, shader samplers, swizzling

Introduction

Texture mapping is the process of applying an image to the polygons that make up your scene, in order to improve visual realism. OpenGL does a lot of the hard work for us, but we must still manually configure the texture units of our graphics hardware in order to achieve the graphical results we want.

Texture Data

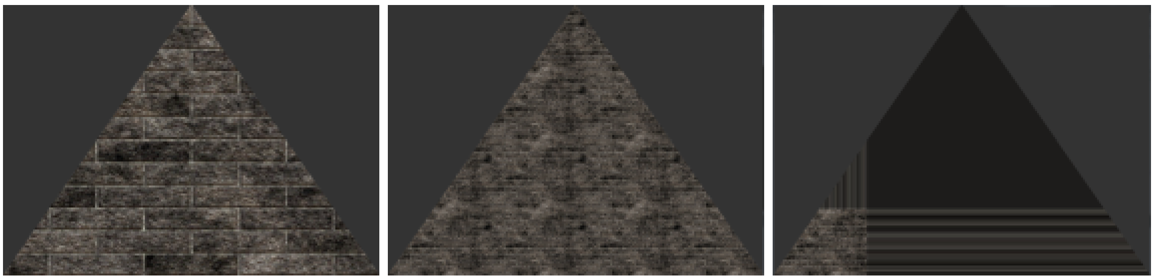
Generally, textures are stored as *bitmaps* - a big linear chunk of memory containing the red, green, blue, and optionally *alpha* information for each texel that makes up an individual texture - textures are measured in *tex*ture *el*ements. This means that compressed file formats like JPEG or PNG must be *decompressed* before being loaded into graphics memory. However, modern graphics hardware quite often supports *native* texture compression - generally a variation on DXT compression, developed by S3 Graphics. This allows textures to take up far less memory, for a minor speed hit on compressing and decompressing the texture data.
It was once a requirement, or at least beneficial from a speed perspective, for textures to be limited to having dimensions that were powers of 2. That is, a texture of 512 by 256 was fine, but one of 512 by 320 would be much slower to work with. Again, modern graphics hardware can generally support textures of any dimension size.

Texture Coordinates

Once a texture is loaded into graphics memory, it can be *sampled*, to obtain a colour from it. We do this via *texture coordinates.* These are vertex attributes like position and colour, stored as vectors, that determine how far along each axis of the texture the sample is to be taken from. The size of the vector depends on the number of dimensions the texture has - a 2D texture uses a 2-component vector for its texture sampling coordinates, and so on. These coordinates are *normalised* - no matter how large the texture we use is, its texture coordinates range from 0.0 to 1.0. This means that OpenGL applications don't have to worry about the size of a texture once it has been loaded. For example, in normalised coordinates, 0.5 is always half way along a texture's axis, whereas '256' would be a meaningless value; it might be half way along a 512-texel texture, or only quarter way along a 1024-texel texture. Like other vertex attributes, texture coordinates are automatically interpolated between vertices, creating smooth even samples across the texture.

Texture Wrapping

Although a texture is defined as having normalised coordinates, vertices can use coordinates outside this range. What happens then? That depends on the current *texture wrapping* mode of the texture - either *clamped* or *repeating*. If a texture uses clamping, then texture sampling coordinates outside of the normalised range will be locked to the 0.0 to 1.0 range. However, if repeating is allowed, then the sampling coordinate 'loops round' the texture, so a coordinate of 1.5 2.5 or even 100.5 would sample the middle of the texture along that axis. The pictures below will demonstrate this more clearly - the left is a picture of a triangle with all of its texture coordinates in the 0.0 to 1.0 range, while the other two have texture coordinates in the range of 0.0 to 4.0 and texture wrapping set to repeating and clamped, respectively. You should be able to see how wrapping allows a texture to be repeated across a single polygon multiple times - useful for naturally repetitive scenes such as brick walls! The large grey area of the right triangle is caused by the clamping setting both texture coordinate axis to 1.0 - so the last texel in the very top right corner is sampled repeatedly for the rest of the triangle.

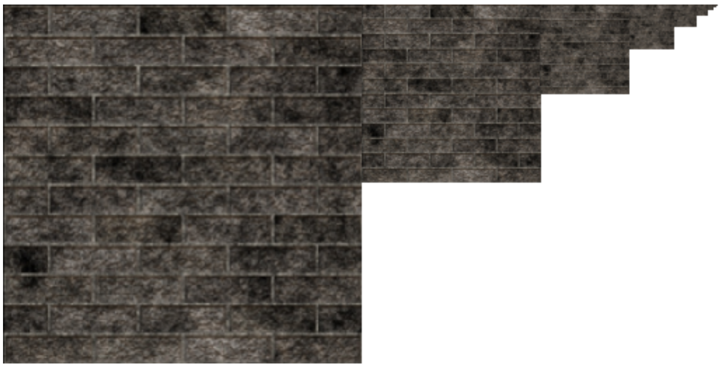


Texture Units

Although textures area really just arrays of data in memory, in order to turn them into a properly transformed and filtered pixel on screen, graphics cards use special *Texture Units* - hardware dedicated to the task of manipulating texture data. Different graphics cards have different numbers of texture units, although you can generally assume to have *at least* 16 on modern hardware. To sample a texture, it must be *bound* to a texture unit - much the same as how a Vertex Buffer must be bound to manipulate it.

Mip-Mapping

In large scenes, it can often be the case that geometry with large textures applied to it is in the distance - for example a racetrack, which has a large advertisement billboard at the end of a long straight, with a 2048 by 2048 texel texture applied to it. Even though it is a long way off, the large texture must still be sampled. Every texel that might contribute to the final pixel must be processed - even if our billboard only took up a single pixel on screen, every texel of the 2048 by 2048 texture would still have to be sampled! This is bandwidth and processing intensive, but fortunately there is a solution - *mipmaps*. A mipmap is a series of smaller lower-detail copies of a texture - so our 2048*2048 texture would have a 1024*1024 copy, a 512*512 copy, and so on all the way down to 1*1. Then, when sampling a mipmapped texture, the appropriate mipmap is used instead, based on the distance of the fragment being textured. Now, our theoretical 1*1 pixel billboard in the distance has to only sample the 1*1 mipmap - that's a huge texture sampling saving! This sampling performance does come at the expense of more texture memory being used for each texture, though, to store the mipmaps. If the texture is not both square and power-of-two, the lowest mipmap will be the whole value each axis can reach - so a 512*320 texture will generate mipmaps of sizes 256*160, 128*80, 64*40, 32*20, 16*10, and 8*5. For this reason, it is common to limit textures to be at least a power-of-two dimension, and preferably square, too.



Mipmaps from original image to 1x1

Texture Filtering

When texturing 3D geometry, it's unlikely that there's an exact 1:1 ratio between the texels of the texture, and the pixels on screen. So how does the geometry get fully textured? When a texture is sampled, either the *nearest* texel can be picked, or *interpolated* between several nearby pixels - just like how the colours were interpolated across vertices in the first tutorial! The most basic form of texture filtering is *Bilinear Interpolation* - the closest texels on both the x and y axis are taken into consideration when interpolating the final pixel colour. Below are examples of our brick texture being used with the nearest texel and Bilinear Interpolation methods of texture filtering. Note how the nearest method results in blocky 'pixelation', while the Bilinear Interpolation looks slightly blurry.

The drawback to Bilinear Interpolation is when using mip-maps - the filtering of nearby texels does not cross mip-map boundaries, so there will be a noticeable 'jump' in blurriness at the boundaries. The solution to this is the more computationally complex *Trilinear Interpolation*, which samples multiple mip-maps Bilinearly, and then interpolates between *those*, resulting in smoother mip-map transitions. Both bilinear and trilinear tend to fail when trying to texture polygons that are quite **oblique** to the camera - as they perform their interpolation in the same way across each axis, the obliqueness of the polygon causes lower and lower mip-maps to be sampled, resulting in a blurry final image. To get around this, an even more computationally expensive filtering option is available - *Anisotropic* filtering. This filtering method samples a texture by a different amount on each axis (*an-iso-tropic* means *not-same-direction*), with a ratio determining filtering amount on each axis. The image on the next page shows a classic case where anisotropic filtering is beneficial - the picture on the left shows the road markings becoming blurry in the distance due to a loss in filtering accuracy even when trilinear filtered, while the picture on the right shows how anisotropic filtering keeps the road markings sharp, even in the distance.

Multitexturing

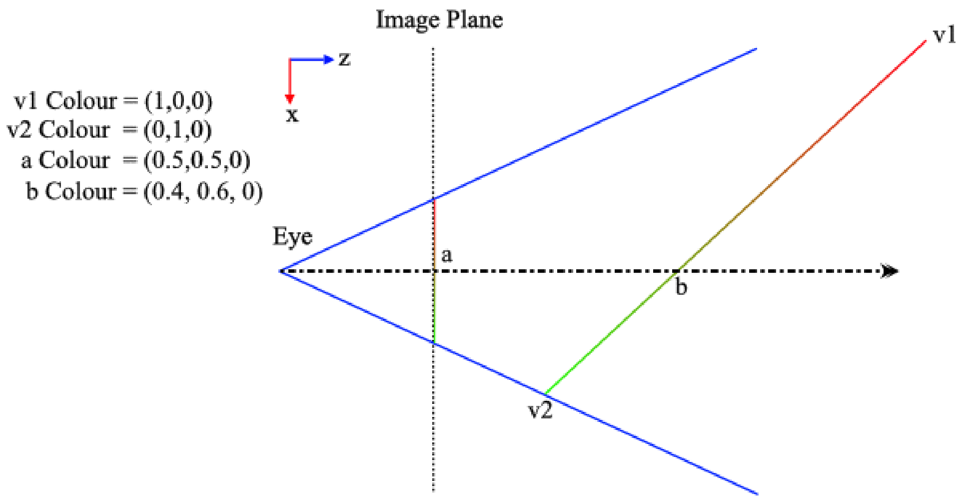
You aren't just limited to a single texture when texturing a triangle in your scene. A fragment shader can sample as many textures as you have texture units available. As fragment shaders are completely programmable, what you do with the data you sample from a texture is entirely up to you - you might blend them all together for example, or take channel components from a number of individual texture samplers. They can each have an individual texture coordinate too - just pass in more unique vertex attribute data to your vertex shader!

Texture Matrices

There *is* a case where a normalised coordinate of 0.5 wouldn't necessarily sample a texel half-way along a texture's axis. As with vertex position coordinates, texture coordinates can be *transformed* by a matrix. These are 4*4 matrices just like the ones you were introduced to in tutorial 2 - they can contain scaling, translation, and rotation components. Texture matrices aren't used very often, but it's useful to know they exist, and the code we'll be writing later shows an example of a texture matrix that will rotate the texture map on our triangle.

Interpolating texture coordinates

In the Introduction tutorial, you learnt that vertex values are typically interpolated from one to the other according to the currently processed fragment's position. Then, in tutorial 2, you were introduced to perspective matrices, the perspective divide, and how the projection matrix is used to map 3D space onto our 2D screen. However, the forshortening effect we want for triangles can also have undesirable side effects for interpolated vertex values. Take this example of a line in 3D space being flattened onto our image plane:



Once we flatten the triangle using our matrix pipeline, we find that the value we see when we appear to be looking at the centre of the triangle (position a) is not the part of the line we're actually looking at in 3D space (position b) - giving us slightly different colours. If instead of colours, these were interpolated texture coordinates, we'd be sampling the wrong part of the texture, and the 3D for-shortening effect would be broken! In order to calculate perspective correct texture coordinates (or any other *varying* value in a shader), the texture coordinates must be divided by the homogenous w value of our vertex coordinate, which can then be interpolated as described in the introduction tutorial. Finally, these interpolated values are divided by 1/w, turning them back into 'world space' interpolated values, only at the correct position. Nearly all shader languages will automatically perform this perspective correction between the vertex and fragment stages, along with the rest of interpolation, so you don't really need to think about this too much, it's just useful to know exactly what the hardware is doing!

Texturing example program

Now to implement some of the texture mapping theory you've just learnt. We're going to make a simple program, that instead of drawing a triangle with interpolated vertex colours as in Tutorial 1, draws a triangle with a 2D texture map applied to it! Very simple, but will get you well on the way to understanding how to use textures in OpenGL. We'll also need to update the *Mesh* and *Shader* classes we created in the last tutorial, to support our new texturing functionality. For this example program, you'll need to create 3 files in the *Tutorial3* Visual Studio project - a short cpp file containing our **main** function, the header and class file for our texturing OpenGL *Renderer* class, and a file each in */Shaders/* for the new vertex and fragment shaders. We'll also be using a texture file, *brick.tga*, provided for you, in the */Textures/* folder.

Mesh header file

We want our meshes to have a texture map applied to them, so we need a new member variable to store the OpenGL name of the texture we want to use - just like with VBOs and shader programs, textures are referenced by numerical *names*. Create a **protected** member variable called *texture* to your Mesh class, a pointer to some **Vector2s** called *textureCoords*, and the **public** accessor functions *SetTexture* and *GetTexture*.

```
protected:
   GLuint       texture;
   Vector2*     textureCoords;
public:
   void      SetTexture(GLuint tex)   {texture = tex;}
   GLuint    GetTexture()             {return texture;}
```

Mesh.h

We must also update the *MeshBuffer* **enum**, as we need a VBO to buffer texture coordinates onto graphics memory.So, we'll add *TEXTURE_BUFFER* before *MAX_BUFFER*, ensuring we'll end up with a *bufferObject* large enough to store enough VBO names.

```
enum MeshBuffer {
   VERTEX_BUFFER, COLOUR_BUFFER, TEXTURE_BUFFER,MAX_BUFFER
};
```

Mesh.h

Mesh class file

In the **constructor** of the *Mesh* class, we must set our new *texture* and *textureCoords* variables to 0, and in the **destructor**, we should **delete** the textureCoords, and also the texture, using the

glDeleteTextures function.This takes in the number of textures you wish to delete, and a reference to the OpenGL names to delete as parameters.

```
Mesh::Mesh(void)  {
..
   texture       = 0;
   textureCoords = NULL;
..
```

Mesh.cpp

```
Mesh::~Mesh(void) {
...
   glDeleteTextures(1,&texture);
   delete[]textureCoords;
...
```

Mesh.cpp

In order to use the correct texture map when drawing a Mesh, we have to change the *Draw* function so that it binds the texture to a texture unit - by default this will be unit 0. This is done using the **glBindTexture** function, with the OpenGL name of the texture as a parameter. So, change your Mesh *Draw* function to be as follows:

```
void Mesh::Draw() {
   glBindTexture(GL_TEXTURE_2D, texture);
   glBindVertexArray(arrayObject);
   glDrawArrays(type, 0, numVertices);
   glBindVertexArray(0);
   glBindTexture(GL_TEXTURE_2D, 0);
}
```

Mesh.cpp

OK, our *Mesh* class can now store a texture map, and apply it when drawing. But that's not much use if your mesh doesn't have texture coordinates! We're also going to have to modify the *GenerateTriangle* and *BufferData* functions.

The modified *GenerateTriangle* function will now create 3 texture coordinates, in much the same way as it generates 3 vertices and 3 colours. Note the texture coordinates used! It is otherwise much the same as in tutorial 1.

```
Mesh* Mesh::GenerateTriangle()   {
   Mesh*m = new Mesh();
   m->numVertices = 3;

   m->vertices    = new Vector3[m->numVertices];
   m->vertices[0] = Vector3(0.0f,   0.5f, 0.0f);
   m->vertices[1] = Vector3(0.5f,  -0.5f, 0.0f);
   m->vertices[2] = Vector3(-0.5f, -0.5f, 0.0f);

   m->textureCoords     = new Vector2[m->numVertices];
   m->textureCoords[0]  = Vector2(0.5f,   0.0f);
   m->textureCoords[1]  = Vector2(1.0f,   1.0f);
   m->textureCoords[2]  = Vector2(0.0f,   1.0f);

   m->colours     = new Vector4[m->numVertices];
   m->colours[0]  = Vector4(1.0f, 0.0f, 0.0f,1.0f);
   m->colours[1]  = Vector4(0.0f, 1.0f, 0.0f,1.0f);
   m->colours[2]  = Vector4(0.0f, 0.0f, 1.0f,1.0f);
```

```
    m->BufferData();

    return m;
}
```

Mesh.cpp

BufferData is also fairly similar to tutorial 1, only now it also uploads the texture coordinates to the *TEXTURE_BUFFER* VBO, in exactly the same way as vertices and colours. Make sure you use the correct **sizeof** value and **glVertexAttribPointer** size!

```
void Mesh::BufferData() {
    glBindVertexArray(arrayObject);
    glGenBuffers(1, &bufferObject[VERTEX_BUFFER]);
    glBindBuffer(GL_ARRAY_BUFFER, bufferObject[VERTEX_BUFFER]);
    glBufferData(GL_ARRAY_BUFFER, numVertices*sizeof(Vector3),
                 vertices, GL_STATIC_DRAW);
    glVertexAttribPointer(VERTEX_BUFFER, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(VERTEX_BUFFER);
    if(textureCoords) { //This bit is new!
        glGenBuffers(1, &bufferObject[TEXTURE_BUFFER]);
        glBindBuffer(GL_ARRAY_BUFFER, bufferObject[TEXTURE_BUFFER]);
        glBufferData(GL_ARRAY_BUFFER, numVertices*sizeof(Vector2),
                     textureCoords, GL_STATIC_DRAW);
        glVertexAttribPointer(TEXTURE_BUFFER, 2, GL_FLOAT, GL_FALSE,0,0);
        glEnableVertexAttribArray(TEXTURE_BUFFER);
    }
    if (colours) {
        glGenBuffers(1, &bufferObject[COLOUR_BUFFER]);
        glBindBuffer(GL_ARRAY_BUFFER, bufferObject[COLOUR_BUFFER]);
        glBufferData(GL_ARRAY_BUFFER, numVertices*sizeof(Vector4),
                colours, GL_STATIC_DRAW);
        glVertexAttribPointer(COLOUR_BUFFER, 4, GL_FLOAT, GL_FALSE,0,0);
        glEnableVertexAttribArray(COLOUR_BUFFER);
    }
    glBindVertexArray(0);
}
```

Mesh.cpp

Shader class file

We only have to change one thing in our *Shader* class file. In the function *SetDefaultAttributes*, we want to bind a VBO to the **uniform** value *texCoord*, as well as the colour and vertex VBO attributes we used in the first tutorial. So, update the function to be as follows:

```
void Shader::SetDefaultAttributes() {
    glBindAttribLocation(program, VERTEX_BUFFER, "position");
    glBindAttribLocation(program, COLOUR_BUFFER, "colour");
    glBindAttribLocation(program, TEXTURE_BUFFER, "texCoord");
}
```

Shader.cpp

Remember, for shaders that don't use texture coordinates, it is still safe to attempt to bind the *texCoord* vertex attribute. It will of course fail, but will otherwise not impede the program in any way.

Renderer header file

With our *Mesh* and *Shader* classes updated to allow the use of textures, we can create the *Renderer* class for this tutorial. The header file is much the same as last tutorial, but now we two **bools** - *filtering* and *repeating*. There's also three new **public** functions - *UpdateTextureMatrix*, *ToggleRepeating*, and *ToggleFiltering*.

```
#pragma once

#include "./nclgl/OGLRenderer.h"

class Renderer : public OGLRenderer {
public:
   Renderer(Window &parent);
   virtual ~Renderer(void);

   virtual void RenderScene();

   void  UpdateTextureMatrix(float rotation);
   void  ToggleRepeating();
   void  ToggleFiltering();

protected:
   Mesh* triangle;

   bool  filtering;
   bool  repeating;
};
```

renderer.h

Renderer Class file

As with previous tutorials, we begin by creating a triangle using our *Mesh* class. We load in our texture using the Simple OpenGL Image Library. This handles all the hard work of generating texture data, and returns the OpenGL name of the texture - just like how our shaders, and vertex buffers and arrays have name, so do textures. If we get a texture index of 0, that means OpenGL has failed to generate the texture (wrong filename, perhaps?), so we quit our application.

```
#include "Renderer.h"

Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
   triangle = Mesh::GenerateTriangle();

   triangle->SetTexture(SOIL_load_OGL_texture(TEXTUREDIR"brick.tga",
                SOIL_LOAD_AUTO,SOIL_CREATE_NEW_ID, 0));

   if(!triangle->GetTexture()) {
      return;
   }
```

renderer.cpp

Our shader loading is the same as last tutorial - the code we added to the *Shader* class *SetDefaultAttributes* function will handle the setting of the new vertex attribute *texCoord*. We also set useful default values for an othographic projection matrix, and our **bools**.

```
    currentShader = new Shader(SHADERDIR"TexturedVertex.glsl",
                               SHADERDIR"texturedfragment.glsl");

    if(!currentShader->LinkProgram()) {
        return;
    }

    init = true;

    projMatrix = Matrix4::Orthographic(-1,1,1,-1,1,-1);

    filtering = true;
    repeating = false;
}
```

renderer.cpp

Our *Renderer* class **destructor** stays the same as last tutorial, as our *Mesh* class will **delete** the texture we create in its own **destructor**.

```
Renderer::~Renderer(void)  {
    delete triangle;
}
```

renderer.cpp

Our Renderer class **RenderScene** function is also quite similar to that of last tutorial. We bind our shader, and update our matrices, then draw our triangle. This time, our fragment shader will have a sampler unifrom variable called *diffuseTex*, and we want it to sample from texture unit *0*. The changes we made to the *Mesh* class *Draw* function earlier will bind the correct texture for us, so there's nothing else we need to do in *RenderScene*, except swap our buffers, just like last tutorial.

```
void Renderer::RenderScene()  {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glUseProgram(currentShader->GetProgram());

    UpdateShaderMatrices();

    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
     "diffuseTex"), 0);

    triangle->Draw();

    glUseProgram(0);
    SwapBuffers();
}
```

renderer.cpp

In the example program, when the left and right keys are pressed, we want the triangle's texture to rotate - this is handled by the *UpdateTextureMatrix* function. the rotation matrix variable probably makes sense, it's just like the rotation matrices we were applying to the model matrix in the last tutorial, but what about the push and pop matrices? As our rotation matrix has no translation data, it will rotate our texture coordinates about the *origin* - 0,0,0. In texture-coordinates, that's our bottom left point of our triangle, so our texture would rotate about this point. That might be the effect we want in some instances, but we're going to have our texture rotating about the centre of our triangle. The centre of our triangle is at 0.5,0.5,0 in texture-coordinate space, so we multiply our rotation matrix by a translation matrix to translate to the centre of our texture, *then* rotate.

Then we translate back to the origin to complete our rotation transformation. This method of 'pushing' and 'popping' matrices is useful for rotating around a point other than the origin - so bear it in mind!

```
44 void Renderer::UpdateTextureMatrix( float value )  {
45    Matrix4 pushPos    = Matrix4::Translation(Vector3(0.5f,0.5f,0));
46    Matrix4 popPos     = Matrix4::Translation(Vector3(-0.5f,-0.5f,0));
47    Matrix4 rotation   = Matrix4::Rotation(value,Vector3(0,0,1));
48    textureMatrix      = pushPos * rotation * popPos;
49 }
```

renderer.cpp

Our little texture mapping test application is going to support the toggling of texture repeating - this is controlled by the **ToggleRepeating** function. On line 51, we toggle our *repeating* class member variable using the **NOT** *boolean operator*. Texure parameters such as the wrapping state are set per texture, so to apply a state to our triangles' brick texture, we must *bind* it, just like we do in *RenderScene*. We then set the texture wrap state for our texture maps two axis' using the **glTexParameteri** OpenGL function, and a *ternary operator* - if our filtering **bool** is set to **true**, it sets **GL_LINEAR** as the filtering value, otherwise it sets **GL_NEAREST**. Finally, we *unbind* the texture, as we're done modifying it.

```
50 void  Renderer::ToggleRepeating()    {
51    repeating = !repeating;
52    glBindTexture(GL_TEXTURE_2D, triangle->GetTexture());
53    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, //x axis
54                    repeating ? GL_REPEAT : GL_CLAMP);
55    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, //y axis
56                    repeating ? GL_REPEAT : GL_CLAMP);
57    glBindTexture(GL_TEXTURE_2D, 0);
58 }
```

renderer.cpp

Our final *Renderer* class function is *ToggleFiltering*. Much like with *ToggleRepeating*, it flips our member variable, and sets a texture parameter; this time it switches between bilinear filtering, and nearest-neighbour filtering on our triangles' brick texture.

```
59 void  Renderer::ToggleFiltering()    {
60    filtering = !filtering;
61    glBindTexture(GL_TEXTURE_2D, triangle->GetTexture());
62    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
63                    filtering ? GL_LINEAR : GL_NEAREST);
64    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
65                    filtering ? GL_LINEAR : GL_NEAREST);
66    glBindTexture(GL_TEXTURE_2D, 0);
67 }
```

renderer.cpp

Vertex Shader

Our vertex shader is going to be pretty simple. We have four uniform matrices this time, as we're using a texture matrix in addition to the model, view, and projection matrices that were introduced in the previous tutorial. As with the previous tutorial, we transform our vertex using the combined model-view-projection matrix.

Just like the vertex position, we must also expand the vertex texture coordinates to a 4-component vector in order to multiply it by our texture matrix - but note how we then convert it back to a vec2 using **.xy**. This is an example of *vector swizzling*, which will be explained in more detail later.

```
#version 150 core

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projMatrix;
uniform mat4 textureMatrix;

in  vec3 position;
in  vec2 texCoord;

out Vertex  {
   vec2 texCoord;
} OUT;

void main(void)   {
   mat4 mvp         = projMatrix * viewMatrix * modelMatrix;
   gl_Position      = mvp * vec4(position, 1.0);
   OUT.texCoord     = (textureMatrix * vec4(texCoord, 0.0, 1.0)).xy;
}
```

TexturedVertex.glsl

Fragment Shader

Another simple fragment shader, similar to the one you've used in previous tutorials. On line 3 we have a new **uniform** variable, a **texture sampler**. This is a special type of GLSL variable that corresponds to a texture unit on your graphics hardware. Note how it is explicitly a *2D* sampler - there's 1D and 3D sampler types as well. We use this texture sampler on line 12, with the **texture** GLSL function. This function takes a texture sampler and a texture coordinate as parameters, and returns a **vec4** containing the colour of our texture at this point - this may be an *interpolated* colour if bilinear filtering has been set.

```
1  #version 150 core
2
3  uniform sampler2D diffuseTex;
4
5  in Vertex   {
6     vec2 texCoord;
7  } IN;
8
9  out vec4 fragColour;
10
11 void main(void)   {
12    fragColour   = texture(diffuseTex, IN.texCoord);
13 }
```

TexturedFragment.glsl

Vector Swizzling

As mentioned, the texture GLSL function returns a **vec4** - corresponding to the red, green, blue, and alpha components of the texture map. What if we wanted to use the red channel component, how would we access it? Well, we could access it using **.x** if wanted to, as red is the first component of the **vec4**. Thankfully however, GLSL allows us to use the more intuitive **.r** / **.g** / **.b** / **.a** to access a **vec4** if we want when dealing with colours. Note that we can't mix and match the 'geometric' and 'colour' component labels in a single operation. What we *can* do, however, is **swizzle** the components. If we want to, we can extract the values in a different order - we could swap the red and green values of a colour by accessing it with **.grbw**, or even expand the first channel of a source **vec4** to all of the colour components using **.rrrr** (or **.xxxx**!). Swizzle operations are cheap to perform, so feel free to use them wherever appropriate in your code. Swizzle operations aren't just limited to **vec4s** you are using for colours, any vector type being used for any purpose in a shader program can be swizzled!

```
   //Allowed
   fragColour  = texture(diffuseTex, IN.texCoord).rgba;
   //Allowed
   fragColour  = texture(diffuseTex, IN.texCoord).xyzw;
   //DISallowed!
   fragColour  = texture(diffuseTex, IN.texCoord).rgzw;
   //Swizzling
   fragColour  = texture(diffuseTex, IN.texCoord).bgra;
   //More swizzling!
   fragColour  = texture(diffuseTex, IN.texCoord).xxxw;
```

Vector Swizzling

Main file

The main file for this tutorial should look fairly familiar by now! Note how we have a few more key checks this time, to handle our texture rotation, filtering, and repetition settings. Other than that, it's all the same.

```
#pragma comment(lib, "nclgl.lib")
#include "./nclgl/window.h"
#include "Renderer.h"

int main() {
   Window w("Texture Mapping!", 800,600,false);
   if(!w.HasInitialised()) {
      return -1;
   }

   Renderer renderer(w);
   if(!renderer.HasInitialised()) {
      return -1;
   }

   float rotate = 0.0f;
   while(w.UpdateWindow() &&
    !Window::GetKeyboard()->KeyDown(KEYBOARD_ESCAPE)){
      if(Window::GetKeyboard()->->KeyDown(KEYBOARD_LEFT) ) {
         --rotate;
         renderer.UpdateTextureMatrix(rotate);
      }
      if(Window::GetKeyboard()->->KeyDown(KEYBOARD_RIGHT) ) {
         ++rotate;
```

```
            renderer.UpdateTextureMatrix(rotate);
        }
        if(Window::GetKeyboard()->->KeyTriggered(KEYBOARD_1) ) {
            renderer.ToggleFiltering();
        }
        if(Window::GetKeyboard()->->KeyTriggered(KEYBOARD_2) ) {
            renderer.ToggleRepeating();
        }

        renderer.RenderScene();
    }
    return 0;
}
```

Tutorial3.cpp

Tutorial Summary

If compilation is successful, you should see a textured triangle on screen when you run the program. You'll be able to rotate the texture using the arrow keys, and toggle bilinear filtering and texture repetition using the 1 and 2 keys. Congratulations! You now know the basics of texturing, and should be well prepared for the texture operations you'll be performing throughout the rest of this tutorial series. Next lesson we'll be taking texture mapping a bit further, by looking at *transparency*, as well as how the *depth buffer* is used.

Further Work

1) Try playing with the texture coordinates of the triangle we define in the *Renderer* **constructor**. Get a hang of how they are influenced by the texture wrapping commands. Remember - texture wrapping can be defined per-axis!

2) What happens to our rotation matrix if we increase our triangles texture coordinate range from 0.0 - 1.0 to 0.0 - 10.0? Does it still rotate our triangle's matrix about it's centre? How else, other than directly by our vertex buffer texture coordinates, could we scale how many times a texture is repeated across our triangle?

3) In the first tutorial, we coloured our triangle using colour Vertex Buffers. Try adding the colour code from Tutorial 1 to the *Renderer* for this tutorial, and blend the colour and texture sample together in the fragment shader. You could even vary how much to blend the texture sampled colour using a **uniform float** sent to the fragment shader from your *Renderer* class...

4) Try adding another texture to your *Renderer*, and blend both textures together in the fragment shader. You'll need two texture2D samplers in your fragment shader...

5) The Simple OpenGL Image Library used by NCLGL can automatically generate mipmaps for a texture if **SOIL_FLAG_MIPMAPS** enum is used when loading an image. Try adding a *Camera* to your *Renderer*, and see how the texture filtering settings you have learnt about in this tutorial affect how the triangle looks when viewed from different angles and distances. Trilinear filtering can be enabled with a *min* filter setting of **GL_LINEAR_MIPMAP_NEAREST**. Why shouldn't it be enabled for a *mag* filter?

6) Investigate how to determine the maximum amount of anisotropic filtering your graphics hardware can perform, and how to enable it on a per-texture basis.

7) Investigate the noperspective interpolation qualifier for vertex output values. What do you suppose this does, then?

Appendix A: Loading textures the *hard way*

In these tutorials, we're using an external library to load in texture data. But what if you already have texture data in main memory (for example if you use your own texture loading functions, or generate them procedurally)? The following code demonstrates how to generate a 2D texture, and load your texture into graphics memory, with mip-mapping enabled:

```
1   GLuint texture;
2   glGenTextures(1, &texture);
3
4   glActiveTexture(GL_TEXTURE0);
5   glBindTexture(GL_TEXTURE_2D, texture);
6
7   glTexImage2D(GL_TEXTURE_2D, mipmapLevel, internalFormat,
8                width, height,  border, format, datatype, data);
9
10  glGenerateMipmap(GL_TEXTURE_2D);
11
12  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
13                  GL_LINEAR_MIPMAP_LINEAR);
14  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
15
16  glBindTexture(GL_TEXTURE_2D, 0);
```

OpenGL Texture Loading

Just like with Vertex Array Objects and Vertex Buffer Objects, we must *generate* a new name for a texture (Line 2), and then *bind* it to a chosen texture unit(lines 4 and 5).

Line 7 actually uploads our texture data, using the OpenGL function **glTexImage** - which is more fully documented on the OpenGL specification website. *mipmapLevel* is the mipmap level this texture will represent - usually this will be 0. The *internalFormat* parameter tells OpenGL what type of colour components your texture will have - generally either *GL_RGB* or *GL_RGBA*, although other formats exist.*width* and *height* are the dimensions of the texture being loaded, while *border* is deprecated functionality, and so should be 0. *format* describes what colour components are actually being *loaded* - normally this will match the *internalFormat*, but you could, for example, have an internal format of *GL_RGBA*, but only load in the red colour component, using a format of *GL_RED*. *datatype* tells OpenGL what type of data each pixel in main memory is - such as a *GL_UNSIGNED_BYTE* or a *GL_FLOAT*. Finally, data is a pointer to the start of the memory containing your texture data - OpenGL will be able to work out the size in bytes of your texture using the *width*, *height* and *datatype* parameters.

Once our texture data is in graphics memory, we can tell OpenGL to generate the mipmaps for the texture (line 10), and finally, set the min and mag filters (lines 12 and 14). Finally, we can optionally clean up and *unbind* our texture from the texture unit.