SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Mapping Techniques

In this tutorial, we will look at how to render objects with textures, as well as normal mapping.

Texture Mapping

Previous tutorials covered using shaders to perform lighting computation. In this tutorial, we will look at how to get the fragment shader to look up a texture colour in determining the fragment colour.

The first thing that we need is to be able to read the contents of an image from a file. Open the Tutorial6a project. It contains two files: bmpfuncs.h and bmpfuncs.cpp.

These files contain the definition of a simple function to read the contents of a 24-bit colour (i.e. RGB) image in the bitmap (.bmp) file format. NOTE: it does not read 32-bit colour (i.e. RGBA) bitmap images, although it is trivial to modify it for RGBA bitmap images.

The following function reads the contents of a 24-bit colour bitmap image:

```
unsigned char* readBitmapRGBImage(const char *filename, int* widthOut, int* heightOut);
```

Bitmap images have a specific file format (BMP file format). It starts with a file header which is divided into two sections: the BitmapFileHeader (14 bytes) and BitmapInfoHeader (40 byte). The total header is 54 bytes in size. Among other things, the header contains the image width, height, size, etc. This is followed by the image data, i.e. the RGB pixels, each of which is 24-bits.

The contents of the readBitmapRGBImage() function are fairly straightforward. In summary, the function opens a file stream and reads the contents from the file header. From the header, it finds the starting location of pixel data within the file (i.e. the offset), along with the image width and height. Based on the image width and height, it allocates memory to store the image data and moves the file read position pointer by the offset and reads the pixel data into the allocated memory. At the end of the function, it returns a pointer to the image data and records the image size and width.

Have a look at the code in Tutorial6a.cpp. The following is an example of how to use the function:

```
unsigned char* g_texImage = NULL;
GLint imageWidth;
GLint imageHeight;
g_texImage = readBitmapRGBImage("images/check.bmp", &imageWidth, &imageHeight);
```

The next thing we will look at is texture coordinates. To be able to look up the correct colour from a texture when rendering a polygon, each vertex of the polygon must have a texture coordinate.

Textures are defined as *s* and *t* coordinates in texture space as shown in Figure 1, and mapped to *u* and *v* coordinates in object space.

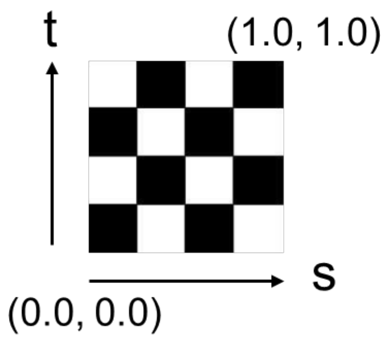

Figure 1

For each vertex of a textured object, we need a texture coordinate. This is added to the vertex struct, note that it has only 2 dimensions:

```
typedef struct Vertex
{
	GLfloat position[3];
	GLfloat normal[3];
	GLfloat texCoord[2];
} Vertex;
```

When defining the vertex information, texture coordinates for each vertex are defined:

```
Vertex g_vertices[] = {
	// Front: triangle 1
	// vertex 1
	-0.5f, 0.5f, 0.5f,	// position
	0.0f, 0.0f, 1.0f,	// normal
	0.0f, 1.0f,		// texture coordinate
	// vertex 2
	-0.5f, -0.5f, 0.5f,	// position
	0.0f, 0.0f, 1.0f,	// normal
	0.0f, 0.0f,		// texture coordinate
	// vertex 3
	0.5f, 0.5f, 0.5f,	// position
	0.0f, 0.0f, 1.0f,	// normal
	1.0f, 1.0f,		// texture coordinate

	...
	...
};
```

The shaders in this tutorial are based on the per pixel lighting shaders, with some modifications to handle texture mapping. Along with the other attribute variables, these texture coordinates must be passed to the shaders as attribute variables. Hence, in the vertex shader:

```
in vec2 aTexCoord;
```

This needs to be passed to the fragment shader, so in the vertex shader it is declared as an output variable:

```
out vec2 vTexCoord;
```

In the vertex shader's main() function, we simply pass the input to the output:

```
vTexCoord = aTexCoord;
```

In the fragment shader, the texture coordinate must be received as input from the vertex shader:

```
in vec2 vTexCoord;
```

For shaders, there is a special data type called sampler for textures. These are defined as uniform variables, since they will not change per vertex:

```
uniform sampler2D uTextureSampler;
```

As with other uniform variables, we will need to pass it the data before we render the polygons. To use the sampler, we can use the texture coordinate to map the correct colour from the texture:

```
texture(uTextureSampler, vTexCoord).rgb;
```

This is near the end of the fragment shader's main() function. It has been commented out for now.

To set up texturing, we need to add code in the application program. Look at the code in Tutorial6a.cpp. The following are declared to store the image data and to store the texture identifier:

```
unsigned char* g_texImage = NULL;
GLuint g_textureID;
```

In the init() function, we read the contents of a bitmap image (contained in the "images" folder) and store it in the unsigned char pointer that we previously declared:

```
GLint imageWidth;
GLint imageHeight;
g_texImage = readBitmapRGBImage("images/check.bmp", &imageWidth, &imageHeight);
```

For the shaders, we need to find and store the location of the texture coordinate attribute variable:

```
GLuint texCoordIndex = glGetAttribLocation(g_shaderProgramID, "aTexCoord");
```

And the location for the sampler uniform variable:

```
g_texSampler_Index = glGetUniformLocation(g_shaderProgramID, "uTextureSampler");
```

Next, we need to set up the texture. This is done by generating an identifier for a texture object and binding it to the appropriate texture target, i.e. GL_TEXTURE_2D:

```
glGenTextures(1, &g_textureID);
glBindTexture(GL_TEXTURE_2D, g_textureID);
```

Then, we create the texture using information about the image's width and height, specify the colour format and data type, and pass it the pointer to where the image data is stored:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageWidth, imageHeight, 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage);
```

The next command generates mipmaps (the purpose of mipmaps is explained in the lecture):

```
glGenerateMipmap(GL_TEXTURE_2D);
```

This is followed by setting up the texture filter and wrapping properties:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

When defining the vertex array object, we also need to set it up for textures:

```
glVertexAttribPointer(texCoordIndex, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
       reinterpret_cast<void*>(offsetof(Vertex, texCoord)));
```

And enable the attribute texture coordinate variable:

```
glEnableVertexAttribArray(texCoordIndex);
```

In the render_scene() function, before drawing the object, the texture unit must be set up. The following activates texture unit0 and binds the texture that was previously initialised to this texture unit:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, g_textureID);
```

In the previous line, we inform the shader that the sampler uniform variable will use texture unit0:

```
glUniform1i(g_texSampler_Index, 0);
```

To clean up, before exiting the program, memory for the image is deallocated and the texture is deleted:

```
if (g_texImage)
       delete[] g_texImage;


glDeleteTextures(1, &g_textureID);
```

Compile and run compile the program.

You will see that it is initially no different from the lighting and shading examples from the previous tutorial. This is because we haven't applied the texture to the colour of the fragment yet. Now in the fragment shader, uncomment the following line:

```
       //fColor *= texture(uTextureSampler, vTexCoord).rgb;
```

What this does is it looks up the texture colour based on the texture coordinate. The *= is so that the colour is modulated with the lighting colour. Notice that we do this after the lighting computation, since the colour based on lighting was already assigned as the fragment colour.

Compile and run compile the program. Now you should see the textured object with lighting.

Instead of combining it with the colour from the lighting calculations. We can simply replace the colour completely with the texture's colour. To do this, uncomment the following line:

```
	//fColor = texture(uTextureSampler, vTexCoord).rgb;
```

Compile and run compile the program. You will now see that the colour of the object completely uses the texture colour.

Multi-Texturing

In the previous example, we saw how to modulate the texture colour with the colour from the lighting calculations. We also saw how to activate texture unit0 and bind a texture to it. In the example in Tutorial6b, we will look at multi-texturing. In multi-texturing, we can texture an object with more than one texture. In this example, we simply have two (or more if you want more textures) of everything texture related.

In the vertex shader:

```
in vec2 aTexCoord1;
in vec2 aTexCoord2;
...
out vec2 vTexCoord1;
out vec2 vTexCoord2;
...
vTexCoord1 = aTexCoord1;
vTexCoord2 = aTexCoord2;
```

In the fragment shader:

```
in vec2 vTexCoord1;
in vec2 vTexCoord2;
...
uniform sampler2D uTextureSampler[2];
...
```

To use the texture colours, the following scales the texture colours and adds them together, before modulating the colour with the colour from the lighting computations:

```
fColor = (diffuse + specular + ambient).rgb
	* (0.8f * texture(uTextureSampler[0], vTexCoord1).rgb
	+ 0.2f * texture(uTextureSampler[1], vTexCoord2).rgb);
```

In Tutorial6b.cpp, we two texture coordinates. This is not necessary if your textures use the same texture coordinates. This example deliberately intends to show that we can use different texture coordinates for different textures:

```
struct Vertex
{
	GLfloat position[3];
	GLfloat normal[3];
	GLfloat texCoord1[2];
	GLfloat texCoord2[2];
};
```

Each vertex is defined with two texture coordinates:

```
Vertex g_vertices[] = {
	// Front: triangle 1
	// vertex 1
	-0.5f, 0.5f, 0.5f,	// position
	0.0f, 0.0f, 1.0f,	// normal
	0.0f, 1.0f,			// texture coordinate
	0.0f, 2.0f,			// texture coordinate
	// vertex 2
	-0.5f, -0.5f, 0.5f,	// position
	0.0f, 0.0f, 1.0f,	// normal
	0.0f, 0.0f,			// texture coordinate
	0.0f, 0.0f,			// texture coordinate

	...
	...
};
```

We read two images and set up two texture objects:

```
unsigned char* g_texImage[2];
GLuint g_textureID[2];
```

In the init() function, we read the contents from two images:

```
GLint imageWidth[2];
GLint imageHeight[2];
g_texImage[0] = readBitmapRGBImage("images/smile.bmp", &imageWidth[0], &imageHeight[0]);
g_texImage[1] = readBitmapRGBImage("images/check.bmp", &imageWidth[1], &imageHeight[1]);
```

Find the locations in the shaders:

```
GLuint texCoordIndex1 = glGetAttribLocation(g_shaderProgramID, "aTexCoord1");
GLuint texCoordIndex2 = glGetAttribLocation(g_shaderProgramID, "aTexCoord2");
...
g_texSampler_Index[0] = glGetUniformLocation(g_shaderProgramID, "uTextureSampler[0]");
g_texSampler_Index[1] = glGetUniformLocation(g_shaderProgramID, "uTextureSampler[1]");
```

Create and initialise two texture objects:

```
glGenTextures(2, g_textureID);
glBindTexture(GL_TEXTURE_2D, g_textureID[0]);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageWidth[0], imageHeight[0], 0, GL_BGR,
	GL_UNSIGNED_BYTE, g_texImage[0]);
...
glBindTexture(GL_TEXTURE_2D, g_textureID[1]);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageWidth[1], imageHeight[1], 0, GL_BGR,
        GL_UNSIGNED_BYTE, g_texImage[1]);
```

Initialise the vertex array object with the two texture coordinates:

```
glVertexAttribPointer(texCoordIndex1, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
        reinterpret_cast<void*>(offsetof(Vertex, texCoord1)));
glVertexAttribPointer(texCoordIndex2, 2, GL_FLOAT, GL_FALSE, sizeof(Vertex),
        reinterpret_cast<void*>(offsetof(Vertex, texCoord2)));


glEnableVertexAttribArray(texCoordIndex1);
glEnableVertexAttribArray(texCoordIndex2);
```

Before we render the object, we need to set up two texture units. The following activates texture unit0 and binds one of the textures to it:

```
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, g_textureID[0]);
```

The following activates texture unit1 and binds the other texture to it:

```
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, g_textureID[1]);
```

The preceding commands inform the shaders that sampler[0] will use texture unit0 and sampler[1] will use texture unit1:

```
        glUniform1i(g_texSampler_Index[0], 0);
        glUniform1i(g_texSampler_Index[1], 1);
```

Compile and run compile the program. You will now see that the colours of the two textures are combined and modulated to the colour obtain from lighting.

To demonstrate that you do not need different two different texture coordinates for multi-texturing if the vertices share the same texture coordinates, and to demonstrate the effect of using different weights, uncomment the following:

```
        fColor = (diffuse + specular + ambient).rgb
               * (0.5f * texture(uTextureSampler[0], vTexCoord1).rgb
               + 0.5f * texture(uTextureSampler[1], vTexCoord1).rgb);
```

Compile and run compile the program.

Now uncomment the following and observe how the difference in the rendered scene:

```
        fColor = (diffuse + specular + ambient).rgb
               * (0.3f * texture(uTextureSampler[0], vTexCoord2).rgb
               + 0.7f * texture(uTextureSampler[1], vTexCoord2).rgb);
```

Compile and run compile the program.

Normal Mapping

You should be familiar with texture mapping and multi-texturing from the previous examples. Instead of looking up a texture for colour, normal mapping is a technique of using a texture to obtain surface normals.

Figure 2 shows a couple of textures. We will use the texture on the left for its colour, whereas the texture on the right will be used to obtain surface normal. This is referred to as a normal map. Note that normal maps are typically blueish in colour, because surface normals are typically perpendicular to the surface. Since RGB represents XYZ, the surface normal has a larger Z component, which is represented by B.

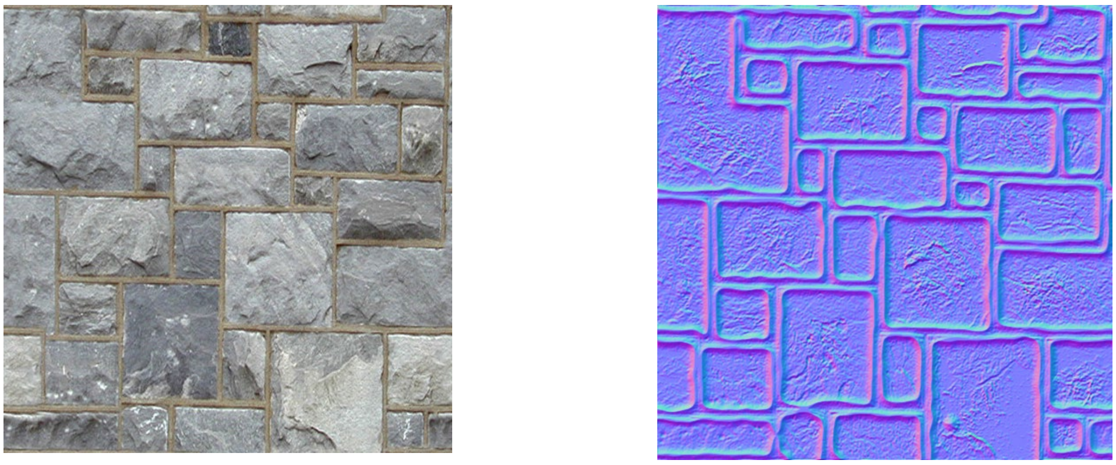

Figure 2: Textures

Open the Tutorial6c project.

Surface normals in normal maps are represented in tangent space, which is local to the surface of a polygon. Hence, to apply lighting correctly we need to transform the normal by a tangent, bitangent and normal matrix (commonly known as a TBN matrix). To construct this matrix, we need the tangent and bitangent vectors which are perpendicular to the normal vector. While we can calculate the tangent vector, for a flat plane like the one in Tutorial6c, it is easy to specify the tangent vector and calculate the bitangent vector as the cross product between the tangent and normal vectors.

To do this, the Vertex struct is modified to contain a tangent component, and each vertex is defined with an associated tangent vector:

```
// struct for vertex attributes
struct Vertex
{
	GLfloat position[3];
	GLfloat normal[3];
	GLfloat tangent[3];
	GLfloat texCoord[2];
};

Vertex g_vertices[] = {
	// Front: triangle 1
	// vertex 1
	-1.0f, 1.0f, 0.0f,	// position
	0.0f, 0.0f, 1.0f,	// normal
```

```
        1.0f, 0.0f, 0.0f,    // tangent
        0.0f, 1.0f,          // texture coordinate
        // vertex 2
        -1.0f, -1.0f, 0.0f,  // position
        0.0f, 0.0f, 1.0f,    // normal
        1.0f, 0.0f, 0.0f,    // tangent
        0.0f, 0.0f,          // texture coordinate

        ...
};
```

The vertex shader needs to receive the tangent vector:

```
in vec3 aTangent;
```

and outputs the vector in world space to be interpolated and passed as input to the fragment shader:

```
out vec3 vPosition;
out vec3 vNormal;
out vec3 vTangent;
out vec2 vTexCoord;
...
```

In the vertex shader's main() function:

```
vPosition = (uModelMatrix * vec4(aPosition, 1.0)).xyz;
vNormal = (uModelMatrix * vec4(aNormal, 0.0)).xyz;
vTangent = (uModelMatrix * vec4(aTangent, 0.0)).xyz;
vTexCoord = aTexCoord;
```

The fragment shader needs to receive these as input:

```
in vec3 vPosition;
in vec3 vNormal;
in vec3 vTangent;
in vec2 vTexCoord;
```

We will also need to pass the colour texture and normal map to the fragment shader as uniform variables:

```
uniform sampler2D uTextureSampler;
uniform sampler2D uNormalSampler;
```

We calculate the bitangent vector as the cross product between the tangent and normal vectors:

```
vec3 Normal = normalize(vNormal);
vec3 Tangent = normalize(vTangent);
vec3 BiTangent = normalize(cross(Tangent, Normal));
```

Note that the normal vector that is obtained from the normal map, is between the range of [0.0, 1.0]. We scale this to be within the range of [-1.0, 1.0]:

```
vec3 NormalMap = 2.0f * texture(uNormalSampler, vTexCoord).xyz - 1.0f;
```

We construct the TBN matrix and transform the surface normal into tangent space:

```
Normal = normalize(mat3(Tangent, BiTangent, Normal) * NormalMap);
```

We calculate lighting using this the newly calculated normal vector, and modulate the final lighting colour with the colour from the colour texture:

```
fColor = (diffuse + specular + ambient).rgb;
fColor *= texture(uTextureSampler, vTexCoord).rgb;
```

The code in Tutorial6c.cpp has to be modified appropriately. The following is a summary of changes from previous examples.

We need to store the location of the sampler for the normal map:

```
GLuint g_normalSamplerIndex;
```

In the init() function, we find the location of the tangent attribute variable and the uniform sampler for the normal map:

```
GLuint tangentIndex = glGetAttribLocation(g_shaderProgramID, "aTangent");
...
g_normalSamplerIndex = glGetUniformLocation(g_shaderProgramID, "uNormalSampler");
```

The tangents are specified in the vertex buffer objects and enabled:

```
glVertexAttribPointer(tangentIndex, 3, GL_FLOAT, GL_FALSE, sizeof(Vertex),
       reinterpret_cast<void*>(offsetof(Vertex, tangent)));
...
glEnableVertexAttribArray(tangentIndex);
```

We also need to load the contents of the colour texture and the normal map:

```
g_texImage[0] = readBitmapRGBImage("images/Fieldstone.bmp", &imageWidth[0],
       &imageHeight[0]);
g_texImage[1] = readBitmapRGBImage("images/FieldstoneBumpDOT3.bmp", &imageWidth[1],
       &imageHeight[1]);
```

These are used to create textures:

```
glGenTextures(2, g_textureID);
glBindTexture(GL_TEXTURE_2D, g_textureID[0]);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageWidth[0], imageHeight[0], 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage[0]);
glGenerateMipmap(GL_TEXTURE_2D);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glBindTexture(GL_TEXTURE_2D, g_textureID[1]);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, imageWidth[1], imageHeight[1], 0, GL_BGR,
       GL_UNSIGNED_BYTE, g_texImage[1]);
glGenerateMipmap(GL_TEXTURE_2D);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

In the render_scene() function, before the object is rendered we set the uniform variables for the samplers to refer to the respective texture units, and bind the respective textures to those texture units:

```
glUniform1i(g_texSamplerIndex, 0);
glUniform1i(g_normalSamplerIndex, 1);

glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, g_textureID[0]);

glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, g_textureID[1]);

glDrawArrays(GL_TRIANGLES, 0, 36);
```

Compile and run the program.

References

Among others, much of the material in this tutorial was sourced from:

- Angel & Shreiner, "Interactive Computer Graphics: A Top-Down Approach with OpenGL", Addison Wesley
- http://www.opengl-tutorial.org/
- http://ogldev.atspace.co.uk/index.html