UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Texture Mapping and Normal Mapping

In this lab, we will look at how to render objects with textures.

Textures

To use textures with OpenGL, we need to load texture data and generate texture objects. We will use the stb_image library for this. Note that the stb_image.h file must be in the project folder. In addition, a Texture class has been created to make it easier to use textures.

Open the lab7a project and have a look at the Texture class defined in the Texture.h and Texture.cpp files. The class has:

- A bind() member function that binds a texture for use;
- Two member functions for setting the texture filtering parameters and wrapping parameters, respectively;
- Overloaded member functions for generating texture objects.
- Member variables for texture ID, target and parameters.

The code for the class is quite straightforward. We'll have a closer look at one of the member functions. Look at the code in void Texture::generate(const std::string filename). It starts by loading image data using the stb_image library.

```
int width, height, channels;
unsigned char* imageData = stbi_load(filename.c_str(), &width, &height, &channels, 0);
```

If the image data loaded successfully, the code generates a texture object and binds it to GL_TEXTURE_2D.

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

Next, the texture and mipmaps of it are created.

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, imageData);
glGenerateMipmap(GL_TEXTURE_2D);
```

The texture parameters are set, based on the respective member variables.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, mMagFilter);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, mMinFilter);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, mWrapS);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, mWrapT);
```

Finally, the image data is freed (no longer need this as the texture object has been created) and the target is set.

Now look at the code in lab7a.cpp. To use the Texture class, include the Texture.h header and declare a texture object:

```
#include "Texture.h"
...
Texture gTexture;
```

The texture is loaded in the init() function.

```
gTexture.generate("./images/check.bmp");
```

For textures, texture coordinates must be defined as vertex attributes:

```
std::vector<GLfloat> vertices =
{
	-1.0f, 0.0f, 1.0f,	// vertex 0: position
	0.0f, 1.0f, 0.0f,	// vertex 0: normal
	0.0f, 0.0f,		// vertex 0: texture coordinate
	1.0f, 0.0f, 1.0f,	// vertex 1: position
	0.0f, 1.0f, 0.0f,	// vertex 1: normal
	1.0f, 0.0f,		// vertex 1: texture coordinate
...
};
```

The shaders must be set up to accept the texture coordinates as vertex attributes. In the init() function:

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

The shaders in this lab are based on the point lighting shader, with some modifications to handle texture mapping. Along with other vertex attributes, texture coordinates for each vertex will be accepted as input to the vertex shader.

```
layout(location = 2) in vec2 aTexCoord;
```

This needs to be passed to the fragment shader, hence:

```
out vec2 vTexCoord;

void main() {
	...
	vTexCoord = aTexCoord;
}
```

The output of the vertex shader will be interpolated, and will be the input to the fragment shader. In the fragment shader:

```
in vec2 vTexCoord;
```

For shaders, there is a special data type called sampler for textures, which declared as uniform variables as they do not change per vertex.

```
uniform sampler2D uTextureSampler;
```

The following looks-up the colour from a texture using texture coordinates, and modulates it with the intensity of reflected light.

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

In the render_scene() function, before the object is rendered the sampler must be set. The following sets the sampler to texture unit 0:

```
gShader.setUniform("uTextureSampler", 0);
```

Texture unit 0 is then activated and we bind the texture object to the active texture unit, before rendering the object.

```
glActiveTexture(GL_TEXTURE0);
gTexture.bind();
```

Compile and run the program.

Multi-Texturing (Add the code in this section yourself)

In multi-texturing, we can texture an object with more than one texture, as graphics devices have more than one texture unit. In this example, we'll use two textures and allow the user to blend between them.

Create two textures and a blend factor:

```
std::map<std::string, Texture> gTextures;
...
float gFactor = 0.5f;
```

In the init() function:

```
gTextures["Check"].generate("./images/check.bmp");
gTextures["Smile"].generate("./images/smile.bmp");
```

In the fragment shader, add another sampler and the blend factor:

```
uniform sampler2D uTextureSampler2;
uniform float uFactor;
```

Then, in the render_scene() function, set the two samplers to the respective texture units and the blend factor:

```
gShader.setUniform("uTextureSampler", 0);
gShader.setUniform("uTextureSampler2", 1);
gShader.setUniform("uFactor", gFactor);
```

Activate the texture units and bind them to the respective textures:

```
glActiveTexture(GL_TEXTURE0);
gTextures["Check"].bind();

glActiveTexture(GL_TEXTURE1);
gTextures["Smile"].bind();
```

In the create_UI() function, add an entry to the user interface to allow the user change the blend factor:

```
TwAddVarRW(twBar, "Blend", TW_TYPE_FLOAT, &gFactor,
       " group='Texture' min=0.0 max=1.0 step=0.01 ");
```

Finally, in the fragment shader, lookup the colour from the textures and interpolate between them using the blend factor:

```
fColor *= mix(texture(uTextureSampler, vTexCoord).rgb,
       texture(uTextureSampler2, vTexCoord).rgb, uFactor);
```

Compile and run the program. Change the blend factor to observe the results.

Multi-Texturing using Different Texture Coordinates (Add the code in this section yourself)

Both textures do not have to share the same texture coordinates. We will add code to allow for different texture coordinates.

First, add another set of texture coordinates to the vertices:

```
std::vector<GLfloat> vertices =
{
	-1.0f, 0.0f, 1.0f,   // vertex 0: position
	0.0f, 1.0f, 0.0f,    // vertex 0: normal
	0.0f, 0.0f,          // vertex 0: texture coordinate 1
	0.0f, 0.0f,          // vertex 0: texture coordinate 2
	1.0f, 0.0f, 1.0f,    // vertex 1: position
	0.0f, 1.0f, 0.0f,    // vertex 1: normal
	1.0f, 0.0f,          // vertex 1: texture coordinate 1
	2.0f, 0.0f,          // vertex 1: texture coordinate 2
	-1.0f, 0.0f, -1.0f,  // vertex 2: position
	0.0f, 1.0f, 0.0f,    // vertex 2: normal
	0.0f, 1.0f,          // vertex 2: texture coordinate 1
	0.0f, 2.0f,          // vertex 2: texture coordinate 2
	1.0f, 0.0f, -1.0f,   // vertex 3: position
	0.0f, 1.0f, 0.0f,    // vertex 3: normal
	1.0f, 1.0f,          // vertex 3: texture coordinate 1
	2.0f, 2.0f,          // vertex 3: texture coordinate 2
};
```

Initialise the vertex attribute pointers to accommodate the new attribute variable:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexNormTex2),
	reinterpret_cast<void*>(offsetof(VertexNormTex2, position)));
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(VertexNormTex2),
	reinterpret_cast<void*>(offsetof(VertexNormTex2, normal)));
glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(VertexNormTex2),
	reinterpret_cast<void*>(offsetof(VertexNormTex2, texCoord1)));
glVertexAttribPointer(3, 2, GL_FLOAT, GL_FALSE, sizeof(VertexNormTex2),
	reinterpret_cast<void*>(offsetof(VertexNormTex2, texCoord2)));
```

Enable the attribute:

```
glEnableVertexAttribArray(3);
```

In the vertex shader, add:

```
layout(location = 3) in vec2 aTexCoord2;
...
out vec2 vTexCoord2;

void main(){
	...
	vTexCoord2 = aTexCoord2;
}
```

In the fragment shader, add:

```
in vec2 vTexCoord2;
```

Then, use different texture coordinates when looking up texture colour:

```
fColor *= mix(texture(uTextureSampler, vTexCoord).rgb,
       texture(uTextureSampler2, vTexCoord2).rgb, uFactor);
```

Compile and run the program.

Also try:

```
fColor *= mix(texture(uTextureSampler, vTexCoord2).rgb,
       texture(uTextureSampler2, vTexCoord).rgb, uFactor);
```

Compile and run the program.

Texturing Models

We can load models with texture coordinates, and texture those models. The example in lab7b demonstrates this using the SimpleModel class, described in a previous lab. In the lab7b.cpp file, a SimpleModel object is declared:

```
SimpleModel gModel;
```

Using the SimpleModel class, when loading the model we load its texture coordinates, by setting the 2nd argument to true. In the init() function:

```
gModel.loadModel("./models/cube.obj", true);
```

After setting the shader uniforms, including the two texture units and binding the respective textures, the model is rendered using:

```
gModel.drawModel();
```

The rest of the code should be familiar.

Compile and run the program.

Normal Mapping

Instead of looking up a texture for colour, normal mapping is a technique of using a texture to obtain surface normals.

Figure 1 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 normals. 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 1: Textures

Open the Lab7c project. The textures are loaded using the Texture class:

```
gTexture["Stone"].generate("./images/Fieldstone.bmp");
gTexture["StoneNormalMap"].generate("./images/FieldstoneBumpDOT3.bmp");
```

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 that are perpendicular to the normal vector. While we can calculate the tangent vector, for a flat plane like the one in Lab7c 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 (defined in the utilities.h file) is modified to contain a tangent component, and each vertex is defined with an associated tangent vector:

```
struct VertexNormTanTex
{
	GLfloat position[3];
	GLfloat normal[3];
	GLfloat tangent[3];
	GLfloat texCoord[2];
};
```

In the init() function where the vertex data is defined, each vertex is specified with these four properties:

```
std::vector<GLfloat> vertices = {
	-1.0f, -1.0f, 0.0f,  // vertex 0: position
	0.0f, 0.0f, 1.0f,    // vertex 0: normal
	1.0f, 0.0f, 0.0f,    // vertex 0: tangent
	0.0f, 0.0f,          // vertex 0: texture coordinate
	1.0f, -1.0f, 0.0f,   // vertex 1: position
	0.0f, 0.0f, 1.0f,    // vertex 1: normal
	1.0f, 0.0f, 0.0f,    // vertex 1: tangent
	1.0f, 0.0f,          // vertex 1: texture coordinate
	...
	...
};
```

Vertex attribute pointers specify this format:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexNormTanTex),
	reinterpret_cast<void*>(offsetof(VertexNormTanTex, position)));
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(VertexNormTanTex),
	reinterpret_cast<void*>(offsetof(VertexNormTanTex, normal)));
glVertexAttribPointer(2, 3, GL_FLOAT, GL_FALSE, sizeof(VertexNormTanTex),
	reinterpret_cast<void*>(offsetof(VertexNormTanTex, tangent)));
glVertexAttribPointer(3, 2, GL_FLOAT, GL_FALSE, sizeof(VertexNormTanTex),
	reinterpret_cast<void*>(offsetof(VertexNormTanTex, texCoord)));
```

Then, each attribute is enabled:

```
glEnableVertexAttribArray(0);
glEnableVertexAttribArray(1);
glEnableVertexAttribArray(2);
glEnableVertexAttribArray(3);
```

These attributes are input to the vertex shader:

```
layout(location = 0) in vec3 aPosition;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec3 aTangent;
layout(location = 3) in vec2 aTexCoord;
```

Modified in the vertex shader's main() function and assigned as the vertex shader's output:

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

void main() {
	...
	vPosition = (uModelMatrix * vec4(aPosition, 1.0f)).xyz;
	vNormal = uNormalMatrix * aNormal;
	vTangent = uNormalMatrix * aTangent;
	vTexCoord = aTexCoord;
}
```

The fragment shader receives the interpolated values as input:

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

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

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

In the fragment shader's main() function, the normal and tangent vectors are normalised, then used to calculate the bitangent vector, which is the cross product between the tangent and normal vectors:

```
vec3 n = normalize(vNormal);
vec3 tangent = normalize(vTangent);
vec3 biTangent = normalize(cross(tangent, n));
```

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

```
n = normalize(mat3(tangent, biTangent, n) * normalMap);
```

Lighting is calculated using this normal vector.

Now look at the code in the render_scene() function located in lab7c.cpp. Along with other uniform shader variables, the texture and normal map must be passed to the shader:

```
gShader.setUniform("uTextureSampler", 0);
gShader.setUniform("uNormalSampler", 1);

glActiveTexture(GL_TEXTURE0);
gTexture["Stone"].bind();
glActiveTexture(GL_TEXTURE1);
gTexture["StoneNormalMap"].bind();
```

Compile and run the program.

You will see that even though the polygon is flat, the surface is shaded as it there are bumps on the surface. Try moving the point light source and changing the material properties.

Exercises

You should now understand texture mapping. Try modifying the programs to do the following:

- Try changing the texture coordinates and observing the results.
- Load different models and render these using textures (use to the check.bmp texture to see how the texture coordinates are defined)
- Try using the different textures and normal maps in Lab7c's image folder.
- Create a room with walls that use normal mapping.

  Hint: you only need the vertices of a single normal mapped plane. Use transformation matrices to move and rotate the plane to appropriate positions.

References

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

- Angel & Shreiner, "Interactive Computer Graphics: A Top-Down Approach with OpenGL", Addison Wesley
- https://learnopengl.com/