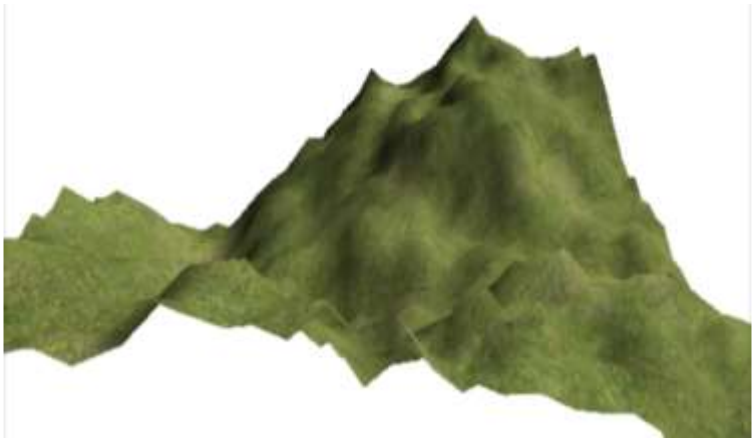
Advanced Graphics Programming

Workshop Seven – Lighting and Texturing

This workshop assumes that you have already procedurally generated a terrain which was done in workshop 6. This workshop has two main sections, lighting and texturing and the combination should look like the following:



Step 1. Enable the depth buffer. In the setup enable depth testing so that parts of the terrain that are obscured by others are not rendered. `Add it in the main function.`

```
glEnable(GL_DEPTH_TEST);
```

Then at the start of your draw function you should clear the depth buffer as well as the colour buffer. `Add it in the drawScene function.`

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Step 2: Turn off the wireframe mode by commenting out the following line:

```
//glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

The default colour is black so if you run the code with a small 5x5 grid you should see something similar to:

5 x 5 terrain

Lighting

The lighting in the latest version of OpenGL is do it yourself, so you will need to create structures to hold the light and material properties as well as add shader code to do the lighting calculations. Once you have done this once you will be able to re-use the code in future applications.

Materials

Step 3. Set the material properties of the terrain in the application. First, create a Material struct to hold the material properties, where the first three variables hold the ambient, diffuse and specular properties of the material, the fourth variable contains the emitted light by the material and the fifth variable is the shininess of the material.

```
struct Material
{
   vec4 ambRefl;
   vec4 difRefl;
   vec4 specRefl;
   vec4 emitCols;
   float shininess;
};
```

Next set the desired properties of your terrain in your application:

```
// Front and back material properties.
static const Material terrainFandB =
{
       vec4(1.0, 1.0, 1.0, 1.0),
       vec4(1.0, 1.0, 1.0, 1.0),
       vec4(1.0, 1.0, 1.0, 1.0),
       vec4(0.0, 0.0, 0.0, 1.0),
       50.0f

};
```

Finally, set the material uniform locations in your application so that they can be accessed by the shader:

```
       glUniform4fv(glGetUniformLocation(programId, "terrainFandB.ambRefl"), 1,
&terrainFandB.ambRefl[0]);
       glUniform4fv(glGetUniformLocation(programId, "terrainFandB.difRefl"), 1,
&terrainFandB.difRefl[0]);
       glUniform4fv(glGetUniformLocation(programId, "terrainFandB.specRefl"), 1,
&terrainFandB.specRefl[0]);
       glUniform4fv(glGetUniformLocation(programId, "terrainFandB.emitCols"), 1,
&terrainFandB.emitCols[0]);
       glUniform1f(glGetUniformLocation(programId, "terrainFandB.shininess"),
terrainFandB.shininess);
```

Ambient light

Step 4. Set up the global ambient light in the application and set the uniform location in your application so that they can be accessed by the shader.

```
static const vec4 globAmb = vec4(0.2, 0.2, 0.2, 1.0);
```

…

```
glUniform4fv(glGetUniformLocation(programId, "globAmb"), 1, &globAmb[0]);
```

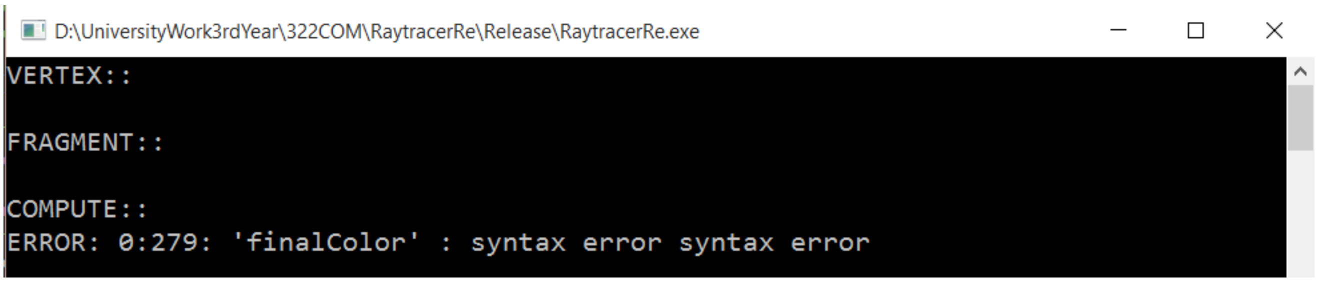
Step 5. Before making any changes to the shaders you need to add some error checking code to your application. This is a very important step as it is easy to make errors in the shaders and by default compiler errors will not be reported you will just see a white screen when you run the code which will be very difficult to debug.

```
void shaderCompileTest(GLuint shader)
{
	GLint result = GL_FALSE;
	int logLength;
	glGetShaderiv(shader, GL_COMPILE_STATUS, &result);
	glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &logLength);
	std::vector<GLchar> vertShaderError((logLength > 1) ? logLength : 1);
	glGetShaderInfoLog(shader, logLength, NULL, &vertShaderError[0]);
	std::cout << &vertShaderError[0] << std::endl;
}
```

An example of how to use it:

```
//Compiles the vertex shader

glShaderSource(vertex, 1, &vertSource, NULL);
glCompileShader(vertex);
//Test for vertex shader compilation errors
std::cout << "VERTEX::" << std::endl;
shaderCompileTest(vertex);
```

To test it works delete a ';' on line 279 in the compute shader the error should be output in the console (as shown below).


D:\UniversityWork3rdYear\322COM\RaytracerRe\Release\RaytracerRe.exe

```
VERTEX::

FRAGMENT::

COMPUTE::
ERROR: 0:279: 'finalColor' : syntax error syntax error
```

You can also use `glGetError()` after every OpenGL call in the C++ code to check if any errors occurred.

Step 6. Write the ambient lighting equation in the vertex shader.

```
uniform vec4 globAmb;

struct Material
{
   vec4 ambRefl;
   vec4 difRefl;
   vec4 specRefl;
   vec4 emitCols;
   float shininess;
};
uniform Material terrainFandB;
…

colorsExport = globAmb * terrainFandB.ambRefl;
```

Completed vertex shader codes can be found in Github

Light source

Step 7. Set up a directional light in the application. First, create a Light struct to hold the light source properties, where the first three variables hold the ambient, diffuse and specular properties of the light source and the fourth variable contains the direction of the light.

```
struct Light
{
   vec4 ambCols;
   vec4 difCols;
   vec4 specCols;
   vec4 coords;
};
```

Next set the desired properties of your directional light in your application:

```
static const Light light0 =
{
       vec4(0.0, 0.0, 0.0, 1.0),
       vec4(1.0, 1.0, 1.0, 1.0),
       vec4(1.0, 1.0, 1.0, 1.0),
       vec4(1.0, 1.0, 0.0, 0.0)
};
```

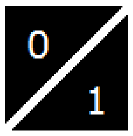
Finally, set the directional light uniform locations in your application so that they can be accessed by the shader:

```
   glUniform4fv(glGetUniformLocation(programId, "light0.ambCols"), 1,
&light0.ambCols[0]);
   glUniform4fv(glGetUniformLocation(programId, "light0.difCols"), 1,
&light0.difCols[0]);
   glUniform4fv(glGetUniformLocation(programId, "light0.specCols"), 1,
&light0.specCols[0]);
   glUniform4fv(glGetUniformLocation(programId, "light0.coords"), 1,
&light0.coords[0]);
```

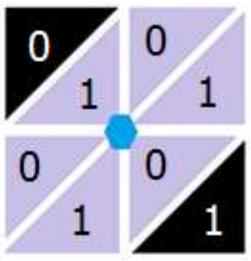
Normals

Generating the normals is a two step process. The first step is calculate the normals for each triangle of the heightmap. The second step is to calculate the average of the triangle normals for each vertex.

Step 8. Calculate the normals for each quad of the heightmap, where each quad consists of two triangles. You need to calculate the normal with a counter clockwise ordering of the triangles. The normal can be calculated by taking the cross product between two edges of your triangle. These can be stored in a temporary data structure as this is an intermediate step to enable you to calculate the vertex normal in the next step.



Step 9. Calculate the normals for each vertex in the heightmap by summing the surrounding triangle normals and then normalising the final vector. The triangle normals that need to be added for a particular vertex are illustrated and described below:



- On upper-left side of our vertex, there is one adjacent triangle (if you aren't in top row of heightmap or leftmost column)
- On the upper-right of our vertex there are two adjacent triangles (if you aren't in top row of heightmap or in rightmost column)
- On bottom-right side of our vertex, there is one adjacent triangle (if you aren't in bottom row of heightmap or rightmost column)
- On the bottom-left of our vertex there are two adjacent triangles (if you aren't in bottom row of heightmap or in leftmost column)

You will need to access the final normals in the shaders so you should store them with the vertex co-ordinate data, as you have setup a material for the terrain the vertex colours are no longer required.

```
struct Vertex
{
	vec4 coords;
	vec3 normals;
};
```

Note that the normals are a `vec3` rather than the colours which were a `vec4` so you will need to update the vertex attributes so that they can be accessed correctly from the shader.

```
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(terrainVertices[0]), 0);
	glEnableVertexAttribArray(0);
	glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(terrainVertices[0]), 
(GLvoid*)sizeof(terrainVertices[0].coords));
	glEnableVertexAttribArray(1);
```

Step 10. Make the lighting robust to changes in the model view. Setup a normal matrix in the application and make it available to the shader.

```
static mat3 normalMat = mat3(1.0);

…

normalMatLoc = glGetUniformLocation(programId,"normalMat");

…

// Calculate and update normal matrix, after any changes to the view matrix
normalMat = transpose(inverse(mat3(modelViewMat)));
glUniformMatrix3fv(normalMatLoc, 1, GL_FALSE, value_ptr(normalMat));
```

Diffuse lighting

Step 11. Diffuse lighting equations. Now you have set up your own material, light source and the normal of the terrain, you are finally ready to update the vertex shader with the diffuse lighting equations.

```
layout(location=0) in vec4 terrainCoords;
layout(location=1) in vec3 terrainNormals;

uniform mat3 normalMat;

struct Light
{
   vec4 ambCols;
   vec4 difCols;
   vec4 specCols;
   vec4 coords;
};
uniform Light light0;

….

normal = normalize(normalMat * terrainNormals);
lightDirection = normalize(vec3(light0.coords));
colorsExport = max(dot(normal, lightDirection), 0.0f) * (light0.difCols *
terrainFandB.difRefl);
```

You can also put codes into fragment shader see completed shader codes in Moodle

This is similar to the example code in BumpMappingShaderized.sln for per-vertex lighting (same as legacy OpenGL). There is also example code in BumpMappingPerPixelLight.sln for per-pixel lighting where the lighting equations are performed in the fragment shader rather than the vertex shader.

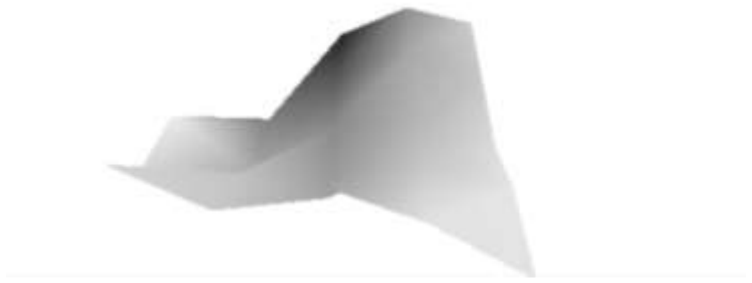
You are now ready to run you code and check the terrain is correctly displayed. If you have any problems firstly check if you shaders compile correctly using the example code in worksheet 7. If they compile correctly the next is to check for errors after each OpenGL call you can do this by using the following command after each OpenGL call and checking the error:

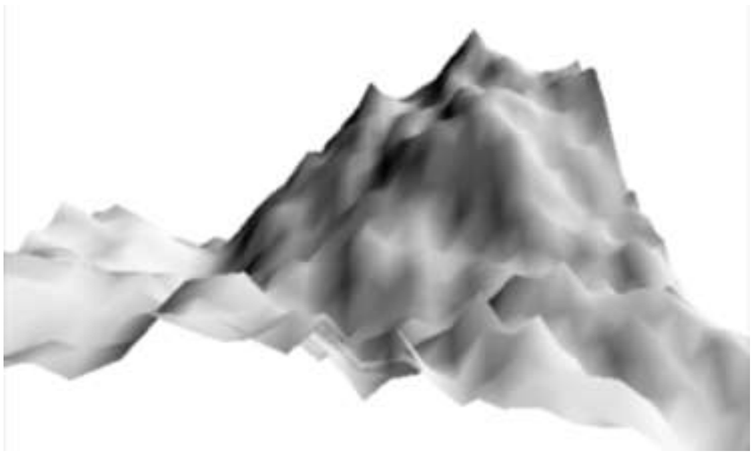
```
GLenum error = glGetError();
```

The error codes are shown below:

```
1280 GL_INVALID_ENUM
1281 GL_INVALID_VALUE
1282 GL_INVALID_OPERATION
1283 GL_STACK_OVERFLOW
1284 GL_STACK_UNDERFLOW
1285 GL_OUT_OF_MEMORY
```

If you run your code now you should see something similar to the images below.


5x5 terrain


33x33 terrain

Step 12: Cull back faces. To increase efficiency you can turn on culling so that the back faces of the terrain are not rendered. You will need to add the following commands to your setup function:

```
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);
```

Textures

Step 13: Find and download a texture that you will apply to your terrain e.g. grass or rock. The format you will need is .bmp if you want to use the bmp reader that will be provided for you so if they are not already in that format you will need to convert them.

You can also find the grass textures used in this example in C:\OpenGL\ExperimenterSource\Textures

The following website provides free low res versions of the textures direct from the website but for high res versions you need to download and install their free viewer: http://www.spiralgraphics.biz/packs/index.htm

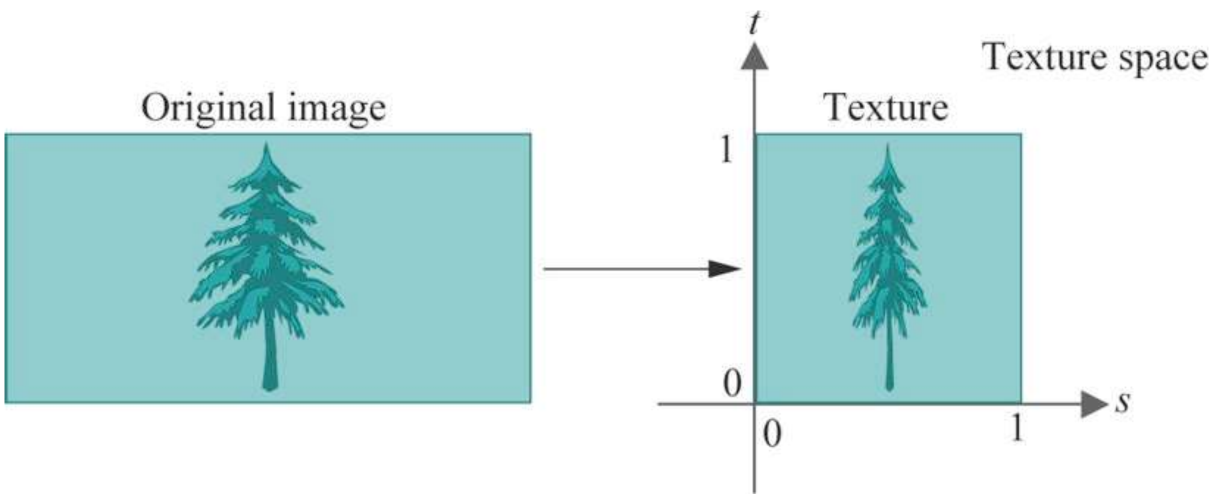
Step 14: Declare the texture variables in the application. You will start by loading one texture but you will create a global array to hold the texture ids and storage for the bmp image data so that it is easy to add more textures later.

```
static BitMapFile *image[1]; // Local storage for bmp image data.

static unsigned int
texture[1], // Array of texture ids.
grassTexLoc;
```

Step 15: Load the texture into your application, remember it must saved in a bmp file format. Download getbmp.zip from Moodle which contains getbmp.cpp and getbmp.h which is a routine to read an uncompressed 24-bit color RGB bmp file into a 32-bit color RGBA bitmap file (A value being set to 1). To load a texture for a file you will need the following set of OpenGL commands. They only need to be performed once so add them to your setup() function. Make sure you set the correct file path for the texture that you want to use.

```
   // Load the image.
   image[0] = getbmp("../../Textures/grass.bmp");

   // Create texture id.
   glGenTextures(1, texture);

   // Bind grass image.
   glActiveTexture(GL_TEXTURE0);
   glBindTexture(GL_TEXTURE_2D, texture[0]);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, image[0]->sizeX, image[0]->sizeY, 0,
                GL_RGBA, GL_UNSIGNED_BYTE, image[0]->data);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);

   grassTexLoc = glGetUniformLocation(programId, "grassTex");
   glUniform1i(grassTexLoc, 0);
```

Step 16: Map the textures to quads on your terrain. All textures are mapped to a square texture space, shown below with axis `s` and axis `t`. Note that (0,0) is the bottom left point of the texture which is important to know when setting the texture coordinates for each vertex of your polygon.



The texture co-ordinates should be added to the vertex structure that contains the vertex coordinates and normals. Note that the texture co-ordinates are a `vec2`.

```
struct Vertex
{
	vec4 coords;
	vec3 normals;
	vec2 texcoords;
};
```

You also need to add an attribute data pointer so that you can access the texture co-ordinates from the shader.

```
	glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, sizeof(terrainVertices[0]), 0);
	glEnableVertexAttribArray(0);
	glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, sizeof(terrainVertices[0]),
(GLvoid*)sizeof(terrainVertices[0].coords));
	glEnableVertexAttribArray(1);
	glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, sizeof(terrainVertices[0]),
(GLvoid*)(sizeof(terrainVertices[0].coords) + sizeof(terrainVertices[0].normals)));
	glEnableVertexAttribArray(2);
```

Now you can generate the texture coordinates in the same for loop where the terrain vertices co-ordinates were already calculated. You need to decide how many times you should map the texture across the heightmap. It depends on the size of the texture. That's why there are two variables `fTextureS` and `fTextureT` which represent how many times the texture is mapped across each dimension.

```
// Generate texture co-ordinates
float fTextureS = float(MAP_SIZE)*0.1f;
float fTextureT = float(MAP_SIZE)*0.1f;

for (int y = 0; y < MAP_SIZE; y++)
	{
		for (int x = 0; x < MAP_SIZE; x++)
		{

			terrainVertices[i].coords = { (float)x, terrain[x][y], (float)y,
			1.0 };

			float fScaleC = float(x) / float(MAP_SIZE - 1);
			float fScaleR = float(y) / float(MAP_SIZE - 1);
```

```
                        terrainVertices[i].texcoords = vec2(fTextureS*fScaleC,
                        fTextureT*fScaleR);

                        i++;
                }
        }
```

Texturing is done with the following simple commands in the fragment shader.

```
uniform sampler2D grassTex;
…
fieldTexColor = texture(grassTex, texCoordsExport);
colorsOut = fieldTexColor;
```

The texture coordinates must also be passed through the vertex shader.

```
layout(location=0) in vec4 terrainCoords;
layout(location=1) in vec3 terrainNormals;
layout(location=2) in vec2 terrainTexCoords;

uniform mat4 projMat;
uniform mat4 modelViewMat;
uniform mat3 normalMat;

out vec2 texCoordsExport;
```

See FieldAndSkyFilteredShaderized.sln for a similar example.

If you run your code you should now see your textured terrain. You will notice that the lighting is no longer being displayed we will come back this later.

33 x 33 terrain

Step 17: Reducing aliasing. If you move your camera around your textured scene the grass may seem to shimmer or flash. This is caused by aliasing, which can be reduced with linear filtering and mipmaps. Mipmaps are an efficient filtering option based on pre-assigning a set of textures to be used at different levels of minification. Generating mipmaps automatically is simple. The command `glGenerateMipmap(GL_TEXTURE_2D)` generates a full set of mipmaps for the texture associated with target. You can call this function in your setup function just after setting the new linear filtering options. You also need to comment out the previous filtering options.

```
            //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
            //glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MAPMAP_LINEAR);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
            glGenerateMipmap(GL_TEXTURE_2D);
```

Run your code again and you should now see that the aliasing problems should be dramatically reduced, as shown below.

Texturing and Lighting

Step 18. Combining texture and light. If the lighting was done in the vertex shader, followed by the texturing in the fragment shader they can easily be combined in the fragment shader to give the final result:

```
        colorsOut = frontAmbDiffExport * texColor;
```

See LitTexturedCylinderShaderized.sln for similar example code.

Extra steps: An improvement is to use different textures for different areas of the terrain. Using multiple textures will require blending for the terrain to appear realistic. A good article demonstrating how to use OpenGL for multi-textured terrain can be found here: http://www.mbsoftworks.sk/index.php?page=tutorials&series=1&tutorial=24