

Laboratory work 6

1 Objectives

This laboratory presents you with the basic notions on the concept of texture and its usage within a 3D scene of objects. It will also provide you a basic framework for loading and including 3D objects into your OpenGL project and then briefly touch upon the subject of animation.

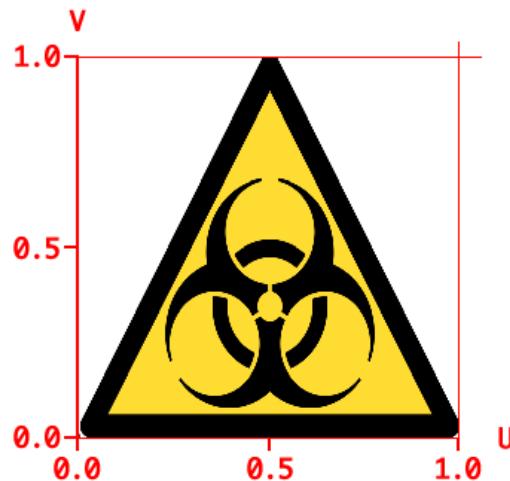
2 Theoretical background

You may think of a texture as a 2D image that can be applied on a 3D object. While it is also possible to have 1D and 3D textures, these are beyond the scope of this laboratory.

Generally, textures are applied on 3D objects in order to increase the photorealism, by adding detail onto their surfaces. One could conceivably add color detail to a model by applying different colors on a vertex-by-vertex basis, but this would be very wasteful, both in terms of the data volumes and processing power required.

Textures are basically 2D pixel maps loaded within the video memory, which can then be mapped to a given 3D object. The mapping is done by assigning each 3D vertex a point within the 2D image. The 2D texture space is usually expressed using the "u" and "v" parameter names, as opposed to "x" and "y" in order to avoid confusion with vertex position values.

As opposed to normal images, whose coordinates are integers expressing the value of a given pixel, a texture's UV coordinates are floating point values ranged within the interval [0, 1], where (0, 0) represents the bottom-left corner of the image and (1, 1) the top-right. These coordinates can be turned into pixels by multiplying them with the image's width and height.



In order to use a texture, one must first load it from an external file:

- Load the image data from an image file (preferably “.png” or “.tex” – always with width and height equal to a power of 2 – ex: 128, 256, 512 etc.)
- Generate a texture ID - using the function: “`glGenTextures`”
- Bind, or “activate” the texture – using the function “`glBindTexture`”
- Generate a texture from the image data – using the function “`glTexImage2D`”
- Set up texture parameters

The texture loading functionality is implemented within the “`ReadTextureFromFile`”. Take a look at it.

In order to apply the texture on an object, it suffices to bind it before drawing the object - using the function “`glBindTexture`”.

3 Tutorial

3.1 Texture application

For the purpose of demonstrating the use of textures, we will first try out a simple example.

Let's start by drawing two basic triangles. Define the vertex data – vertex position and texture UV coordinates:

```
//vertex position and UV coordinates
GLfloat vertexData[] = {
    // first triangle
    -5.0f, 0.0f, 0.0f,    0.0f, 0.0f,
    5.0f, 0.0f, 0.0f,    1.0f, 0.0f,
    0.0f, 8.0f, 0.0f,    0.5f, 1.0f,
    // second triangle
    0.1f, 8.0f, 0.0f,    0.0f, 0.0f,
    5.1f, 0.0f, 0.0f,    0.0f, 0.0f,
    10.1f, 8.0f, 0.0f,   0.0f, 0.0f
};
GLuint vertexIndices[] = {
    0,1,2,
    3,4,5
};
```

Load the triangle data by calling the predefined function “`loadTriangles()`” - before the start of the main loop:

```
int main(int argc, const char * argv[]) {

    .....
    loadTriangleData();

    while (!glfwWindowShouldClose(glWindow)) {
        renderScene();
```

```

        glfwPollEvents();
        glfwSwapBuffers(glWindow);
    }
    .....
}
```

Finally, draw the triangles:

```

void renderScene()
{
    .....

    glBindVertexArray(objectVAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
}
```

Build and run the application. You should see two black triangles.

In order to apply the texture, we need to send the texture coordinates to the Fragment Shader. This means that we have to pass them along from within the Vertex Shader:

```

#version 410 core

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexNormal;
layout(location = 2) in vec2 textcoord;

out vec3 colour;
out vec2 passTexture;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
    colour = vertexNormal;
    passTexture = textcoord;
    gl_Position = projection * view * model * vec4(vertexPosition, 1.0);
}
```

Inside the Fragment Shader, we need access to a texture unit, called “diffuseTexture”. The function “texture” will return a value from the currently active texture, based on the 2D UV coordinates it receives as argument.

Fragment Shader should look like this:

```

#version 410 core

in vec3 colour;
in vec2 passTexture;
```

```

out vec4 fragmentColour;

uniform sampler2D diffuseTexture;

void main() {
    fragmentColour = texture(diffuseTexture, passTexture);
}

```

Make sure you load the texture from the image file:

```

int main(int argc, const char * argv[]) {
    .....

    loadTriangleData();
    texture = ReadTextureFromFile("textures/hazard2.png");

    while (!glfwWindowShouldClose(glWindow)) {
        renderScene();

        glfwPollEvents();
        glfwSwapBuffers(glWindow);
    }
    .....
}

```

Now activate the texture before drawing the triangles:

```

void renderScene()
{
    .....

    glBindTexture(GL_TEXTURE_2D, texture);

    glBindVertexArray(objectVAO);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
}

```

Build and run the application.

Modify the texture UV coordinates for the second triangle, so that your application will look like this:



3.2 Loading 3D objects

For the purpose of loading and drawing complex 3D objects, one option you have is to use the Model3D class provided within this laboratory resource bundle. This class should be able to load anything, from simple objects to complex scenes of multiple, textured objects. As you can see inside “Model3D.hpp”, it holds a list of meshes, where each mesh represents a 3D object, as well as the collection of loaded textures necessary for those objects.

Let’s see how we can use the Model3D class in order to load a model from an “.obj” file. Modify the main function. Delete the previously added functionality and add:

```
int main(int argc, const char * argv[]) {  
    .....  
    //send matrix data to shader  
    GLint projLoc = glGetUniformLocation(myCustomShader.shaderProgram, "projection");  
    glUniformMatrix4fv(projLoc, 1, GL_FALSE, glm::value_ptr(projection));  
  
    myModel.LoadModel("objects/stall.obj");  
  
    while (!glfwWindowShouldClose(glWindow)) {  
        renderScene();  
  
        glfwPollEvents();  
        glfwSwapBuffers(glWindow);  
    }  
    .....  
}
```

To render the loaded model, modify and add the following lines within the “renderScene” function:

```
void renderScene()  
{  
    .....  
    //create rotation matrix  
    model = glm::rotate(model, glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));  
    model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
```

```

    //send matrix data to vertex shader
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
    myModel.Draw(myCustomShader);
}

```

Build and run the application. As you can see, **the model is not textured**. To apply a texture to it, you must load one from an external file:

```

int main(int argc, const char * argv[]) {
    .....
    myModel.LoadModel("objects/stall.obj");
    texture = ReadTextureFromFile("textures/stall_texture.png");
    .....
}

```

Activate the texture before drawing the model:

```

void renderScene()
{
    .....
    glBindTexture(GL_TEXTURE_2D, texture);

    myModel.Draw(myCustomShader);
}

```

Run the application. Use the “W”, “A”, “S” and “D” keys to navigate; “Q” and “E” to rotate the model.

Remember when we said that a Model3D object can hold both the 3D meshes and their textures? Most often, when you will be downloading 3D objects from online sources, you will be provided both with an “.obj” file and an “.mtl” one, together with a collection of applied texture images. The “.mtl” file specifies the texture association data for the object. **In these cases, the Model3D object will read the object and activate its associated textures as required – there will be no need for you to load and activate the textures yourselves.**

You may find “.obj” files at www.turbosquid.com, www.tf3dm.com and
<https://sketchfab.com/features/free-3d-models>.

Let's try to add an “.obj” file with included textures. Modify the main function to load the new object:

```

int main(int argc, const char * argv[]) {
    .....
    //argument 1 = obj file; argument 2 = textures folder
    myModel.LoadModel("objects/Farmhouse.obj", "textures/");
    //texture = ReadTextureFromFile("textures/stall_texture.png");

    while (!glfwWindowShouldClose(glWindow)) {
        renderScene();
    }
}

```

```

        glfwPollEvents();
        glfwSwapBuffers(glWindow);
    }
    .....
}

```

Comment the texture activation instruction from renderScene – this object has texture mapping information included, so the Model3D object will try to load the textures for you automatically:

```

void renderScene()
{
    .....
    //glActiveTexture(GL_TEXTURE0);
    //glUniform1i(glGetUniformLocation(myCustomShader.shaderProgram, "diffuseTexture"),0);
    // glBindTexture(GL_TEXTURE_2D, texture);

    myModel.Draw(myCustomShader);
}

```

General rules to follow when loading .obj files from the internet:

- Look for the .mtl file – it contains data about the object’s materials and textures. Inside the .obj you have a reference to the .mtl file. **Make sure that its path is correct** – it should be relative to your project’s location.
- Within the .mtl file, **look for texture files references** – make sure the texture paths are set up correctly – path relative to the project location. If they cannot be found, they will not be loaded.
- Make sure the textures are in “.png” or”.tga” format and have dimensions (width and height) equal to powers of 2 (ex: 128, 256, 512, etc.).
- Try to keep your project clean and ordered - keep the 3D objects and textures within their own separate folders.

3.3 Animation

Let us add a translation transformation to our object – translate on Ox axis by an amount equal to “delta”. Increase delta inside the “renderScene” function.

```

float delta = 0;
void renderScene()
{
    .....
    delta += 0.001;
    model = glm::translate(model, glm::vec3(delta, 0, 0));

    //create rotation matrix
    model = glm::rotate(model, glm::radians(angle), glm::vec3(0.0f, 1.0f, 0.0f));
    model = glm::rotate(model, glm::radians(180.0f), glm::vec3(0.0f, 1.0f, 0.0f));
    //send matrix data to vertex shader
    glUniformMatrix4fv(modelLoc, 1, GL_FALSE, glm::value_ptr(model));
}

```

```

    glBindTexture(GL_TEXTURE_2D, texture);
    myModel.Draw(myCustomShader);
}

```

Run the application. Then try drawing 500 instances of the same object. Notice the decrease in animation speed:

```

void renderScene()
{
    .....
    for (int i = 0; i < 500;i++)
        myModel.Draw(myCustomShader);
}

```

This animation is dependent upon the frequency with which the function is invoked – that is the refresh frequency of your application. As you might guess, the refresh frequency varies depending upon your hardware resources or scene complexity. Ideally, we would want the animation to run independently of the FPS capabilities of our application.

The solution is to increment delta based on a given speed variable – “units per second”. We measure the time between successive updates and compute the distance the object should move:

$$Distance = speed * time$$

```

float delta = 0;
float movementSpeed = 2; // units per second
void updateDelta(double elapsedSeconds) {
    delta = delta + movementSpeed * elapsedSeconds;
}
double lastTimeStamp = glfwGetTime();

void renderScene()
{
    .....
    //initialize the view matrix
    glm::mat4 view = myCamera.getViewMatrix();
    //send matrix data to shader
    GLint viewLoc = glGetUniformLocation(myCustomShader.shaderProgram, "view");
    glUniformMatrix4fv(viewLoc, 1, GL_FALSE, glm::value_ptr(view));

    // get current time
    double currentTimeStamp = glfwGetTime();
    updateDelta(currentTimeStamp - lastTimeStamp);
    lastTimeStamp = currentTimeStamp;
    model = glm::translate(model, glm::vec3(delta, 0, 0));
    .....
}

```

```
}
```

You can try the animation with increasing numbers of object instances, to see whether the movement speed holds.

4 Further reading

Function (and link)	Description
glGenTextures	Generate texture names
glBindTexture	Bind a named texture to a texturing target
glTexImage2D	Specify a two-dimensional texture image
glTexParameter	Set texture parameters

5 Assignment

- Load at least 3 models. Change the position of one object using keyboard controls and animate another object.