IM420 Advanced Computer Graphics – WS 2020
Assignment 2- 2D Scene Behind The Window with Textures

Sahabaj Barbhuiya, Faezeh Asgharkermani

10th November, 2020

1.1 Task

Based on the assignment 1 and project template from the lectures, the task was to draw a window in front (on top) of the 2D scene and switch between transparent and opaque window for the 2D scene. In the first assignment we used only Vertex Buffer Objects (VBOs) and Vertex Array Objects (VAOs) to draw the 2D scene. But for assignment 2, we have also used the concept of Index Buffer Objects (IBOs) and Indices to draw the 2D scene. This second assignment also demonstrates the complex practices of blending function of OpenGL.

1.2 Approach to the solution

For this assignment, we used vertices for the triangles as well as indices to draw the 2D scene. We have to calculate the indices before defining and using it to render our 2D scene in OpenGL. After calculating the indices we can define the total number indices to be used together with the vertices and also the Index Buffer. We also have to take into consideration that we need to draw a window on top of the 2D scene.

We will use 6 vertices to draw our window, but we will not use any indices for the window. Hence, we have 54 vertices for the 2D scene and 6 vertices for the window, and a total of 60 vertices. After calculating the vertices we define the total number indices(only for the 2D) to be used together with the vertices (2D scene and the window) and also the Index Buffer which is as follows:

```
1 #define USE\_INDEX_BUFFER 1
2 #define NUM\_VERTICES 60 // 54 for the scene and 6 for the window
3 #define NUM\_INDICES 72 // indices are only for the scene not for the window
```

Now, to use the textures, we also have to define ***texCoordID*** and ***textureID***. Moreover, we are going to use more than one shader (greyscale fragment shader and window shader) and more than one texture(wood and tile textures), we define *currentShader* and *currentTexture* as follows:

```
1 int texCoordID, textureID;
2 int currentShader, currentTexture;
```

We also define a boolean value ***transparent*** and set to *true* which will later be used later to change and switch the alpha value of the window to make the window appear transparent or opaque.

Now, we define the vertices for the 2D scene for all the triangles and also add the vertices for the coordinates of the window to be drawn on the top of the 2D scene. The vertices can be defined as:

```
GLfloat vertices[] = {
  //2D SCENE
  //TREE 1
    -0.6f, -0.6f, 0.0f,//0
    -0.6f, -0.4f, 0.0f,//1
    -0.4f, -0.6f, 0.0f,//2
    -0.4f, -0.4f, 0.0f,//3
    ....

  //WINDOW
    -0.5f, -0.5f, 0.0f,//54
    -0.5f, 0.5f, 0.0f,//55
     0.5f, 0.5f, 0.0f,//56

    0.5f, 0.5f, 0.0f,//57
    0.5f, -0.5f, 0.0f,//58
    -0.5f, -0.5f, 0.0f//59
    };
```

Next, we add the colors for the 2D scene with the triangles and also add color for the window, which is shown in the following code snippet:

```
GLfloat colors[] = {
  //COLOR FOR THE SCENE
  //TREE 1
  //Brown
  0.5f, 0.35f, 0.05f, 1.0f,
  0.5f, 0.35f, 0.05f, 1.0f,
  0.5f, 0.35f, 0.05f, 1.0f,
  0.5f, 0.35f, 0.05f, 1.0f,
  ....

  // COLOR FOR THE WINDOW
  1.0f, 1.0f, 0.0f, 1.0f, //window color with alpha value
  1.0f, 1.0f, 0.0f, 1.0f,
  1.0f, 1.0f, 0.0f, 1.0f,

  1.0f, 1.0f, 0.0f, 1.0f,
  1.0f, 1.0f, 0.0f, 1.0f,
  1.0f, 1.0f, 0.0f, 1.0f,
};
```

Now, we need to define the texture coordinates as *texCoords* for the 2D scene. For the 2D we use 23 triangles, but some pair of triangles uses indices and hence share vertices. Such triangles are drawn as a square.

Moreover, in order to map a texture to the triangles in our 2D scene, we need to tell each vertex of the triangles which part of the texture it corresponds to. As texture coordinates range from 0 to 1, the texture is basically shaped as an square coordinate map, ranging from 0 to 1.

Hence, after calculating the texture coordinates, we only need 15 set of texture coordinates, which can be defined as following:

```
GLfloat texcoords[] = {
  0.0f, 0.0f,//1
  1.0f, 0.0f,
  1.0f, 1.0f,
  0.0f, 1.0f,

  ....

  0.0f, 0.0f,//15
  1.0f, 0.0f,
  1.0f, 1.0f,
  0.0f, 1.0f
}
```

Finally, we define the indices only to be used to draw the 2D scene consisting of all the triangles, which can be defined as the following:

```
GLuint indices[] = {  //INDICES ONLY FOR THE 2D SCENE
  0, 1, 2, 1, 3, 2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 14, 15, 16,
  17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 27, 28, 29, 30, 31, 32,
  31, 32, 33, 34, 35, 36, 35, 36, 37, 38, 39, 40, 38, 40, 41, 42, 43, 44,
  43, 44, 45, 46, 47, 48, 46, 48, 49, 50, 51, 52, 50, 52, 53
    };
```

1.3 Shaders and Textures

1.3.1 Three different Shaders

All the three shaders has the same variables, since all of them are using a texture. The texture gets transmitted into the shader and the variable *texCoord* gets forwarded from the vertex shader.

In the *colorShader* fragment shader program, a texture gets transmitted as a sampler2D variable, with the texture() function and the *texCoord* a texture with a vec4 gets created. This vector is then used when calculating the color of the pixel by multiplying it with the color of the 2D scene.

```
#version 330 core

in vec3 color;
in vec2 texCoord;

uniform sampler2D tex;

out vec4 fragColor;

void main() {
  vec4 texture = texture(tex, texCoord);
  fragColor = vec4(texture * vec4(color, 1.0));
}
```

The *greyscaleShader* fragment shader changes the color of the texture to grey scale. This is achieved by defining luminance and a *vec3 GSW* which is the greyscale weight,

to calculate the luminance, which is the dot product of texCoord with rgb and GSW value, and outputs the final *fragColor* as grey color.

```
float lum; //luminance

vec3 GSW = vec3(0.2125,0.7154, 0.0721); //greyscale weight

out vec4 fragColor;

void main() {
   lum = dot(texture(tex,texCoord).rgb, GSW);
   fragColor = vec4(lum, lum, lum, 1.0);
}
```

The last shader *windowShader* fragment shader, is used for the window and outputs nothing but the color, and uses the texture as the color of the 2D scene with the window.

```
void main() {
  vec4 texture = texture(tex, texCoord);
  fragColor = texture;
}
```

1.3.2 Delegating the Textures

Textures are defined by the variable *textureID* which then changes the imported image. This value of the variable changes by key input provided by the user or before the window is drawn. A texture gets generated with the image. If the image is the window which also includes alpha values, the function glTexImage2D() has to use RGBA values instead of RGB values. The following code snippet shows the *loadTexture()* function:

```
void loadTexture() {

  glGenTextures(1, &texture);
  glBindTexture(GL_TEXTURE_2D, texture);

  int width, height;
  unsigned char* image;

  // Switch statement determining which texture to load, depending on currentTexture value

  switch (currentTexture)
  {
  case 0:
    image = SOIL_load_image("images/tile.png", &width, &height, 0, SOIL_LOAD_RGB);
    break;
  case 1:
    image = SOIL_load_image("images/wood.jpg", &width, &height, 0, SOIL_LOAD_RGB);
    break;
  default:
    image = SOIL_load_image(windowFilename, &width, &height, 0, SOIL_LOAD_RGBA);
    break;
  }

  // Nested else if statement, loading the texture as RGB or RGBA, depending on if it has
     an alpha channel

```

```
26   if (image && currentTexture == 2)
27   {
28     glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
       GL_UNSIGNED_BYTE, image);
29     glGenerateMipmap(GL_TEXTURE_2D);
30   } else if (image)
31     {
32       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
       GL_UNSIGNED_BYTE, image);
33       glGenerateMipmap(GL_TEXTURE_2D);
34     } else
35     {
36       std::cout << "Failed to load texture: " << SOIL_last_result() << std::endl;
37     }
38 }
```

The *loadTexture()* gets called in the *initBuffers()* method. We also have to access the location of *texCoord* variable within the shader, assign and forward the relevant *texCoord* value for each vertex, access the texture variable within the shader program, and enable the *texCoordID* attribute inside the *initBuffers()* method usinf the following code snippet:

```
1     void initBuffers() {
2     ....
3     texCoordID = glGetAttribLocation(shaderProgram, "inTexCoord");
4     ....
5     glVertexAttribPointer(texCoordID, 2, GL_FLOAT, GL_FALSE, 0,
6             BUFFER_OFFSET(3 * NUM_VERTICES * sizeof(GLfloat)
7               + NUM_VERTICES * 4 * sizeof(GLfloat)));
8   ....
9   textureID = glGetUniformLocation(shaderProgram, "tex");
10  ....
11  glEnableVertexAttribArray(texCoordID);
12  }
```

1.3.3 Toggling the Textures and Transparency

We extend the function *processInput()* to add the key event functionality so that a user can switch between the colored scene, greyscale scene, transparent scene with the window and the opaque 2D scene with the window on top. The textures and shaders should be switched by pressing the key ***C***.

When key ***C*** is pressed, the program uses the wood texture in combination with the greyscale shader for the 2D scene shown in the Figure.1.1 else it uses the tile texture in combination with the normal *colorShader* for the 2D scene with the window on the top shown in the Figure. 1.2.

The transparency of the window should switch by pressing the key ***B***, the variable for transparency also changes in the *processInput()* function.

```
1     void processInput(GLFWwindow *window) {
2     ....
3
4   if (glfwGetKey(window, GLFW_KEY_C) == GLFW_PRESS) {
5     Shader = 1,
6     Texture = 1;
```

```
7
8       loadShaders();
9       initBuffers();
10    }
11    else {
12      Shader = 0,
13      Texture = 0;
14
15      loadShaders();
16      initBuffers();
17
18    }
19    if (glfwGetKey(window, GLFW_KEY_B) == GLFW_PRESS) {
20      transparent = false;
21    }
22    else {
23      transparent = true;
24      windowFilename = "images/window.png";
25    }
26  }
```

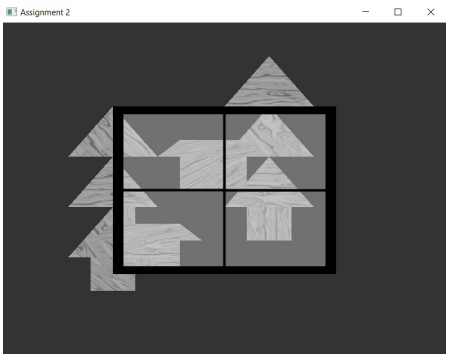

Figure 1.1: The 2D scene with a wood texture and a gray scale shader.

Figure 1.2: The 2D scene with a tile texture and a normal color mixing shader.

To render a window in front of the 2D scene the function *renderScene()* which has to be changed. First the shader and the texture have to be changed for the quad (as some pairs of triangles are drawn quad sharing the same vertices) and it has to be rendered.

To render a transparent window , we use *glEnable(GL_BLEND)* function. Since it is a loop, we start the with with disabling the blending using *glDisable(GL_BLEND)* function else the 2D scene will get blended as well. Figure.1.3. shows the 2D with opaque window which is not being blended with the 2D scene

```
1     void renderScene() {
2   glDisable(GL_BLEND);
3   glBindTexture(GL_TEXTURE_2D, texture);
4
5 #define USE_INDEX_BUFFER
6   glDrawElements(GL_TRIANGLES, NUM_INDICES, GL_UNSIGNED_INT, NULL); //draw all the
      tringles of the 2D scene
7   if (transparent)
```

```
8   {
9     glEnable(GL_BLEND);
10    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
11  }
12
13  Shader = 2;
14  Texture = 2;
15
16  loadShaders();
17  initBuffers();
18
19  glDrawArrays(GL_TRIANGLES, 54, 6); // 54 is the starting point from where the
20                      //function starts drawing the 6 specified  vertices  for  the  window.
21 }
```

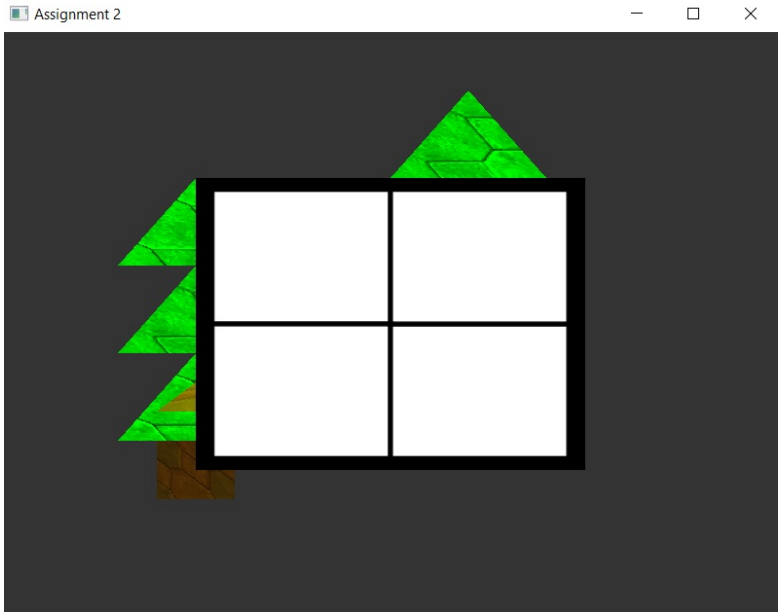

Figure 1.3: Opaque window with no transparency

1.4 Summary

After completing the assignment 2, we know how to use the concepts of IBOs(indices), how to work with textures and blending function in OpenGL. In this assignment, user is able to switch between three different textures using the key ***C*** and ***B***.

The most challenging part was to calculate the texture coordinates and the number of texture coordinates to be used to render the texture in the 2D scene. Moreover, drawing and blending the window in combination with the 2D scene was also very challenging.