Laboratory work 3

1 Objectives

The objective of this laboratory is to present different methods in which vertex data (attributes) is transferred between the application (CPU) and GPU levels.

2 Theoretical background

2.1 OpenGL Shaders

The programmable pipeline in OpenGL relies on shaders, which are small programs written in the GLSL language (very similar with C, and oriented towards graphics operations). This language contains useful features related to vector and matrix operations. Every shader starts with a version declaration, followed by a list of input and output variables, uniforms and at least the main function (which is the entry point). The typical structure of a shader is the following:

```
//OpenGL version
#version versionNumber

//list of input variables
in type inputVariableName;

//list of output variables
out type outputVariableName;

//list of uniforms
uniform type uniformName;

void main()
{
   //implements the functionality and output the variables
   outputVariableName = processVariable();
}
```

2.2 Data types

In GLSL, we can use most of the default basic data types, such as int, float, double, uint and bool. Besides these basic types, GLSL also supports vectors and matrices. A vector represents a container for any of the basic types and may have different dimensions - of 1,2,3 or 4 elements. We can define vectors like this (usually we will be using **vecn**):

- **vecn**: vector of n floats
- **bvecn**: vector of n booleans

- **ivecn**: vector of n integers
- **uvecn**: vector of n unsigned integers
- **dvecn**: vector of n double components

In order to access different components from the vectors, we will be using **.x**, **.y**, **.z** and **.w** (first, second, third and fourth component). For code readability, we can use **.r**, **.g**, **.b** and **.a** for colors - or **.s**, **.t**, **.p** and **.q** for texture coordinates.

2.3 Inputs and outputs

The inputs and outputs are specified by the keywords **in** and **out**.

If the name of an input variable (from one shader) matches the name of an output variable (from another shader) then the data passes from one shader to the other.

The inputs of the vertex shader represent vertex attributes (could be position, color, normal vector, etc.). In order to match the vertex attributes specified at the CPU level with the ones defined at the GPU level, we use the following statement: **layout (location = locationID)**.

At the CPU level, we use the following functions:

```
glVertexAttribPointer(locationID, otherParameters);
glEnableVertexAttribArray(locationID);
```

At the GPU level, we specify the input variable:

```
layout(location = locationID) in vec3 vertex_position;
```

The output of the fragment shader should be a vec4 color output variable. Otherwise, the result (image) will be incorrect.

2.4 Uniforms

Another way of transferring data from our application (which runs on the CPU) to our shaders (which run on the GPU) is by **uniforms**. They are global (meaning unique for a shader program object), and can be accessed by any shader of the shader program. In addition, their value will remain the same until we modify it.

To declare a uniform at the shader level, we use:

```
uniform type uniformName;
```

In order to update the uniform, we need (at application level) to get the uniform location and then we can change its value.

```
glUseProgram(shaderProgram);
GLint uniformLocation = glGetUniformLocation(shaderProgram, "uniformName");
glUniform3f(uniformLocation, 0.15f, 0.0f, 0.84);
```

To update the uniform value, we use the function **glUniform*,** which has multiple definitions (different by the postfix):

- **f**: expecting a float
- **i**: expecting an int
- **ui**: expecting an unsigned int
- **3f**: expecting 3 floats
- **fv**: expecting a float vector/array

2.5 Vertex attributes

In the VBO we can store more attributes than just the vertices' position. For instance, we can store color, normal vector, texture coordinates, etc. Consider the following example, in which we have two attributes for each vertex (position and color).

Vertex 1						Vertex 2					
X	Y	Z	R	G	B	X	Y	Z	R	G	B
0…3	4…7	8…11	12…15	16…19	20…23	24…27	28…31	32…35	36…39	40…43	44…47
Vertex position stride: 24 Vertex offset: 0											
			Vertex color stride: 24 Vertex color offset: 12								

We need to modify the vertex shader to receive the color values, and for this, we need to set the corresponding location:

```
layout(location = locationID1) in vec3 vertexPosition;
layout(location = locationID2) in vec3 vertexColor;
```

We need to change also the way in which we specify the vertices attributes in our program:

```
//vertex position attribute
glVertexAttribPointer(locationID1, size, type, normalized, stride, pointer);
glEnableVertexAttribArray(locationID1);

//vertex colour attribute
glVertexAttribPointer(locationID2, size, type, normalized, stride, pointer);
glEnableVertexAttribArray(locationID2);
```

- *size* - number of components (1, 2, 3, 4)
- *type* - data type of each component (GL_BYTE, GL_UNSIGNED_BYTE, GL_SHORT, GL_UNSIGNED_SHORT, GL_INT, and GL_UNSIGNED_INT)
- *normalized* - specifies whether fixed-point data values should be normalized or not
- *stride* - the byte offset between consecutive generic vertex attributes
- *pointer* - the offset of the first component of the first generic vertex attribute

2.6 Element Buffer Objects

In order to avoid duplicating (re-defining) some vertices, we can use EBOs to store indices. Indices will be used by OpenGL to decide what vertices to choose from when forming polygons. The procedure of creating EBOs is very similar to the one used to define VBOs:

```
glGenBuffers(1, &verticesEBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, verticesEBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(vertexIndices), vertexIndices, GL_STATIC_DRAW);
```

When calling glDrawElements, we draw using the indices provided within the EBO currently bound. The first parameter specifies the type of the primitive, the second represents the number of elements we draw (vertices), the third parameter represents the type of indices (typically GL_UNSIGNED_INT) and the last one is an offset.

```
glDrawElements(GL_TRIANGLES, size, type, offset);
```

3 Tutorial

3.1 Transfering data from vertex to fragment shader

Start from the solution that you find on the website and modify the vertex and fragment shaders. Remember that if the name of an input variable (from one shader) matches the name of an output variable (from another shader), then the data passes from one shader to the other one.

The code from **vertex shader**:

```
#version 400

layout(location = 0) in vec3 vertexPosition;

//the color variable that we will send to the fragment shader
out vec3 colour;

void main() {
   colour = vec3(0.74, 0.16, 0.0);
   gl_Position = vec4(vertexPosition, 1.0);
}
```

The code from **fragment shader**:

```
#version 400

//the color variable that we received from the vertex shader
in vec3 colour;

out vec4 fragmentColour;
```

```
void main(){
   fragmentColour = vec4(colour, 1.0);

}
```

3.2 Sending data using uniforms

Start from the solution that you find on the website, and modify the vertex and fragment shaders.

The vertex shader:

```
#version 400

layout(location = 0) in vec3 vertexPosition;

void main(){
   gl_Position = vec4(vertexPosition, 1.0);
}
```

The fragment shader:

```
#version 400

out vec4 fragmentColour;

uniform vec3 uniformColour;

void main(){
   fragmentColour = vec4(uniformColour, 1.0);
}
```

Modify the main function to get the uniform location from the shader program and update the uniform value:

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

   .. .. ..

   myCustomShader.useShaderProgram();
   GLint uniformColourLocation = glGetUniformLocation(myCustomShader.shaderProgram, "uniformColour");
   glUniform3f(uniformColourLocation, 0.15f, 0.0f, 0.84);

   while (!glfwWindowShouldClose (glWindow)){
      .. .. ..

   return 0;
}
```

3.3 Element Buffer Object and attributes example

Start from the solution that you find on the website. In this example, we want to draw a rectangle composed of two triangles. Instead of duplicating some of the vertices' coordinates, we will use an Element Buffer Object in order to save storage space. Define, globally, the vertex data (representing position and color) and the buffers that will be used for it.

```
GLfloat vertexData[] = {
   //vertex position and vertex color
   -0.5f, 0.5f, 0.0f, 1.0f, 0.0f, 0.0f,
   -0.5f, -0.5f, 0.0f, 1.0f, 0.0f, 1.0f,
   0.5f, -0.5f, 0.0f, 0.0f, 1.0f, 0.0f,
   0.5f, 0.5f, 0.0f, 0.0f, 1.0f, 1.0f
};

GLuint vertexIndices[] = {
   0, 1, 2,
   0, 2, 3
};

GLuint verticesVBO;
GLuint verticesEBO;
GLuint objectVAO;
```

We need to update the initialization of our object. The difference is that, in this case, we have also indices and a new attribute for each vertex. For details about the parameters needed for each function, please read again the "theoretical background" section and also the "further reading" section.

```
void initObjects()
{
   glGenVertexArrays(1, &objectVAO);
   glBindVertexArray(objectVAO);

      glGenBuffers(1, &verticesVBO);
      glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);
      glBufferData(GL_ARRAY_BUFFER, sizeof(vertexData), vertexData, GL_STATIC_DRAW);

      glGenBuffers(1, &verticesEBO);
      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, verticesEBO);
      glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(vertexIndices), vertexIndices, GL_STATIC_DRAW);

      //vertex position attribute
      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)0);
      glEnableVertexAttribArray(0);

      //vertex colour attribute
      glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(GLfloat), (GLvoid*)(3 * sizeof(float)));
```

```
        glEnableVertexAttribArray(1);

    glBindVertexArray(0);
}
```

In the renderScene() function, we need to update the drawing call to glDrawElements.

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

Vertex shader:

```
#version 400

layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColour;

out vec3 colour;

void main() {
    colour = vertexColour;
    gl_Position = vec4(vertexPosition, 1.0);
}
```

Fragment shader:

```
#version 400

in vec3 colour;

out vec4 fragmentColour;

void main() {
    fragmentColour = vec4(colour, 1.0);
}
```

4 Further reading

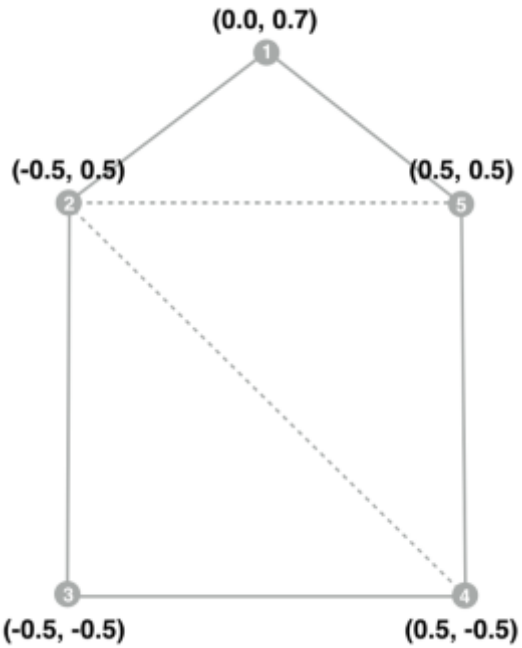
- OpenGL Programming Guide 8th edition – chapters 2 and 3

Function (and link)	Description
glGenBuffers	generates buffer object names
glBindBuffer	binds a buffer object to the specified buffer binding point
glBufferData	creates and initializes a buffer object's data store
glGenVertexArrays	generates vertex array object names
glBindVertexArray	binds a vertex array object
glVertexAttribPointer	defines an array of generic vertex attribute data

glEnableVertexAttribArray	enables a generic vertex attribute array
glCreateShader	creates a shader object
glShaderSource	replaces the source code in a shader object
glCompileShader	compiles a shader object
glCreateProgram	creates a program object
glAttachShader	attaches a shader object to a program object
glLinkProgram	links a program object
glDeleteShader	deletes a shader object
glUseProgram	installs a program object as part of the current rendering state
glDrawArrays	renders primitives from array data
glGetUniformLocation	returns the location of a uniform variable
glUniform	specifies the value of a uniform variable for the current program object
glDrawElements	renders primitives from array data

5 Assignment

1. Render the following figure using EBO and define different colors for each vertex. Do not replicate the vertices!!!



2. Modify the vertices coordinates (on x and y axis) using two uniforms that are sent to the vertex shader. Update the uniforms' values using different keyboard inputs.
3. Switch between flat and wireframe representation by using different keyboard inputs.
    - To change the rendering to a wireframe representation, you can use the following function:

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

- To change back the rendering to a flat representation, you the following function:

```
glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
```