Laboratory work 2

1 Objectives

The objective of this laboratory is to present a brief introduction to the **OpenGL programmable pipeline**.

2 Theoretical background

OpenGL is considered an Application Programming Interface (API) and is used to develop graphical applications by accessing features available in the graphics hardware. However, OpenGL is a specification developed and maintained by the Khronos Group. The OpenGL specification describes what is the desired result or output of each function. The actual implementation can be different between various OpenGL libraries. These libraries are implemented mainly by the graphics card manufacturers.

The first versions of OpenGL used a so called **fixed function pipeline,** in which most of the functionality was predefined and the developers could only modify some parameters but not change the actual algorithms used to transform 3D objects to 2D images. This pipeline is very easy to understand, but on the other hand is very inefficient. Starting with OpenGL 3.2, this fixed function pipeline has become deprecated, developers preferring instead the larger degree of flexibility offered by the **programmable pipeline**.

There are some changes in writing OpenGL applications for the programmable pipeline compared to the fixed pipeline:

- *No immediate mode drawing* –glBegin() and glEnd() are no longer supported in order to specify graphical primitives
- *All data must be stored in buffer objects* – such as Vertex Buffer Objects (VBOs) or Vertex Array Objects (VAOs)
- *No transformations* – glTranslatef(), glRotatef(), glScalef(), gluLookAt(), gluPerspective() are no longer supported
- *No lighting* – glLight*() and glMaterial*() are no longer supported and you have to implement them in shaders

OpenGL operates as a large state machine, consisting of a collection of variables that define how it should operate. The OpenGL context represents the current state. Often, before rendering using the current context, we change the state by modifying some parameters and also manipulating some buffers. Being independent of the operating system or windowing system used, it offers flexibility for developing cross-platform applications. By writing small programs (called **shaders**) we can use the GPUs at their full capacity, compared to the fixed pipeline. Shaders are attached to an OpenGL applications and run on the GPU. They are written in OpenGL Shading Language (GLSL), which is very similar to the C language.

The rendering pipeline implemented in OpenGL is illustrated in the figure below. The programmable pipeline contains at least the **vertex shading** and the **fragment shading** stages, and replaces the fixed-

pipeline, which is no longer supported (lighting and transformations). The vertex shading stage processes each vertex independently of the other ones; vertex data is received from the application through vertex buffer objects. The rasterization phase generates a set of fragments which are processed in the fragment shading stage; the output is represented by the fragment's color and depth. Some stages (such as primitive setup, clipping and rasterization) are not programmable. Tesselation Shader and Geometry Shader are optional.

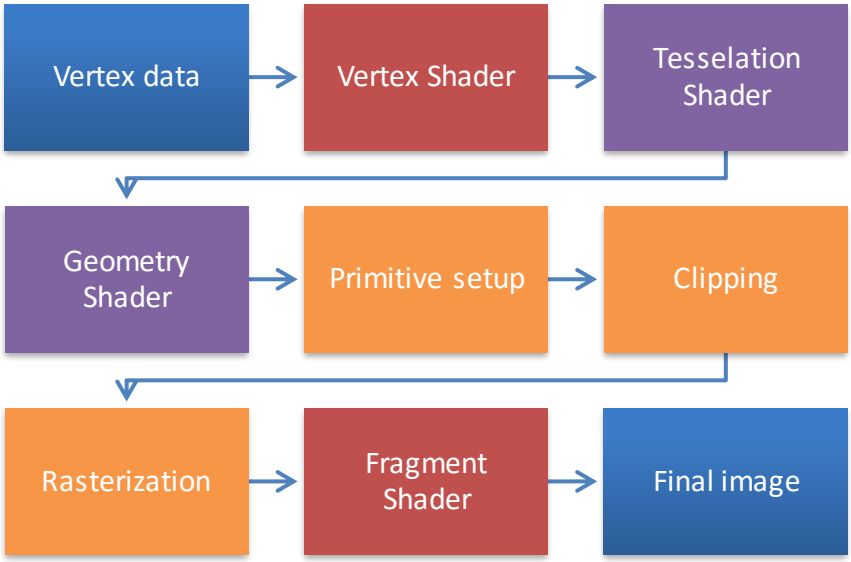

Figure 1 - OpenGL programmable pipeline

3 A basic OpenGL 4 application

3.1 Libraries

We are using the **GLFW** library instead of GLUT to create a window with OpenGL contexts. GLFW is written in C and has native support for Windows, OS X and many Unix-like systems. You can download pre-compiled binaries or the source code from http://www.glfw.org. A step by step tutorial on compiling the library on various platforms can be found here.

The OpenGL Extension Wrangler Library (**GLEW**) is a cross-platform open-source C/C++ extension loading library. You can download pre-compiled binaries or the source code from http://glew.sourceforge.net/. GLEW provides efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform.

3.2 Representation of 3D objects

A 3D object is represented using vertices and topological information. For each vertex we need attributes such as position, color, texture coordinate and other relevant information. Vertex data is stored in Vertex Buffer Objects (VBOs). A series of VBOs can be stored in a Vertex Array Object (VAO).

A **Vertex Buffer Object (VBO)** is a buffer used to hold an array of vertex attributes (such as position, color, normal vector, etc.). A **Vertex Array Object (VAO)** groups together multiple VBOs. In order to use VBOs we need to:

- generate VBO names (IDs): *glGenBuffers(…)*
- bind (select) a specific VBO for initialization: *glBindBuffer(GL_ARRAY_BUFFER, …)*
- load data into the VBO: *glBufferData(GL_ARRAY_BUFFER, …)*

A similar procedure is needed for Vertex Array Objects:

- generate VAO names (IDs): *glGenVertexArrays(…)*
- bind (select) a specific VAO for initialization: *glBindVertexArray(…)*
- update VBOs associated with this VAO: *glBindBuffer(…)* and *glVertexAttribPointer(…)*
- bind VAO for use in rendering: *glDrawArrays(…)*

3.3 Vertex and fragment shaders

In order to be able to render 3D objects, we need to write at least a vertex shader and a fragment shader. For this we need to:

- generate a shader object (referenced by its unique ID): *glCreateShader(…)*
- attach the shader source code to the shader object: glShaderSource(…)
- compile the shader: glCompileShader(…)

After we have compiled the vertex and fragment shaders we need to combine them in a so called shader program object. The steps for this are the following:

- create a shader program: *glCreateProgram(…)*
- attach the previous compiled shaders: *glAttachShader(…)*
- link them in the shader program: *glLinkProgram(…)*

4 Tutorial

This section presents a basic OpenGL 4 application. The starting source code is available on the laboratory web page.

4.1 Step 1

Open and run the OpenGL_4_Application project. It is a basic application that just creates an OpenGL window without rendering anything.

4.2 Step 2

We start by specifying a triangle, and for this, we need to define the coordinates of three vertices, a variable to store the Vertex Buffer Object ID and another one to store the Vertex Array Object ID.

```
//vertex coordinates in normalized device coordinates
GLfloat vertexCoordinates[] = {
   0.0f,  0.5f,      0.0f,
   0.5f, -0.5f,      0.0f,
```

```
    -0.5f, -0.5f,       0.0f
};

GLuint verticesVBO;
GLuint triangleVAO;
```

Add the following function, which will initialize the OpenGL objects.

```
void initObjects()
{
    //generate a unique ID corresponding to the verticesVBO
    glGenBuffers(1, &verticesVBO);
    //bind the verticesVBO buffer to the GL_ARRAY_BUFFER target,
    //any further buffer call made to GL_ARRAY_BUFFER will configure the
    //currently bound buffer, which is verticesVBO
    glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);
    //copy data into the currently bound buffer, the first argument specify
    //the type of the buffer, the second argument specify the size (in bytes) of data,
    //the third argument is the actual data we want to send,
    //the last argument specify how should the graphic card manage the data
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertexCoordinates), vertexCoordinates, GL_STATIC_DRAW);

    //generate a unique ID corresponding to the triangleVAO
    glGenVertexArrays(1, &triangleVAO);
    glBindVertexArray(triangleVAO);
    glBindBuffer(GL_ARRAY_BUFFER, verticesVBO);
    //set the vertex attributes pointers
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, NULL);
    glEnableVertexAttribArray(0);
    //unbind the triangleVAO
    glBindVertexArray(0);
}
```

4.3 Step 3

The vertex and fragment shader programs are already included in the project. The source code is laid out below. Do not worry about their content at this stage. In the next laboratory work, we will analyze the content and structure of these shader programs.

4.3.1 Vertex shader

```
#version 400

layout(location = 0) in vec3 vertex_position;

out vec3 colour;

void main() {
```

```
    //specify the vertex color
    colour = vec3(1.0, 0.0, 0.0);
    gl_Position = vec4(vertex_position, 1.0);
}
```

4.3.2 Fragment shader

```
#version 400

in vec3 colour;
out vec4 frag_colour;

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

4.4 Step 4

Next we need to read, compile and link the shader programs. The following function is already present in the source code; you don't need to add it.

```
GLuint initBasicShader(std::string vertexShaderFileName, std::string fragmentShaderFileName)
{
    //read, parse and compile the vertex shader
    std::string v = readShaderFile(vertexShaderFileName);
    const GLchar* vertexShaderString = v.c_str();
    GLuint vertexShader;
    vertexShader = glCreateShader(GL_VERTEX_SHADER);
    glShaderSource(vertexShader, 1, &vertexShaderString, NULL);
    glCompileShader(vertexShader);
    //check compilation status
    shaderCompileLog(vertexShader);

    //read, parse and compile the vertex shader
    std::string f = readShaderFile(fragmentShaderFileName);
    const GLchar* fragmentShaderString = f.c_str();
    GLuint fragmentShader;
    fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    glShaderSource(fragmentShader, 1, &fragmentShaderString, NULL);
    glCompileShader(fragmentShader);
    //check compilation status
    shaderCompileLog(fragmentShader);

    //attach and link the shader programs
    shaderProgram = glCreateProgram();
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    glLinkProgram(shaderProgram);
```

```
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    //check linking info
    shaderLinkLog(shaderProgram);

    return shaderProgram;
}
```

4.5 Step 5

Finally, we need to specify which primitive OpenGL should render. For this purpose, we use the glDrawArrays function, which employs the current active shader, VAO and VBOs in order to draw the specified primitive.

```
void renderScene()
{
    //clear the color and depth buffer before rendering the current frame
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //specify the background color
    glClearColor(0.8, 0.8, 0.8, 1.0);
    //specify the viewport location and dimension
    glViewport(0, 0, retina_width, retina_height);

    //process the keyboard inputs
    if (glfwGetKey(glWindow, GLFW_KEY_A)) {
        //TODO
    }

    if (glfwGetKey(glWindow, GLFW_KEY_D)) {
        //TODO
    }

    //bind the shader program, any further rendering call
    //will use this shader program
    glUseProgram(shaderProgram);

    //bind the VAO
    glBindVertexArray(triangleVAO);
    //specify the type of primitive, the starting index and
    //the number of indices to be rendered
    glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

5 Further reading

- OpenGL Programming Guide 8th edition – chapter 1

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 current rendering state
glDrawArrays	renders primitives from array data

6 Assignment

1. Define and display two triangles. Define different buffer objects for them (VBO and VAO) and arrange them in such a manner that they form a square.
2. Define a new shader program that uses a different fragment shader. Set the rendering color to green for this new fragment shader. Display one triangle using the old shader program and the other one using the newly defined shader program.
3. Change the shader program used to render each triangle by using different keyboard inputs.