Computer Graphics Labs

Abel J. P. Gomes

LAB. 1

Department of Computer Science and Engineering
University of Beira Interior
Portugal
2019

LAB. 1

INTRODUCTION TO OpenGL

1. Getting Started
2. Installing and setting up graphics libraries: OpenGL, GLEW, and GLFW
4. A First OpenGL/GLFW Program
5. Programming Exercises

Lab. 1

INTRODUCTION TO OpenGL

1. Getting Started

A starting point for learning computer graphics via OpenGL API (Application Programming Interface) is the following wiki:

http://www.opengl.org/wiki/Getting_started

Before starting to program it is good idea to know details about system, say renderer/graphics card and CPU, as well as the OpenGL version installed in your computer. For Unix-like operating systems this task can performed using the command **glxinfo**.

The idea of producing pictures or images using a computer requires two processes: modeling and rendering. **Modeling** has to do with the creation, manipulation, and storage of geometric objects on computer, while rendering just means converting a scene to an image. **Rendering** involves a set of transformations, namely: rasterization, shading, illumination, and animation of the image.

Computer graphics has been widely used in many areas of interest from graphics presentation, paint systems, computer-aided design (CAD), and scientific visualization to simulation of natural phenomena, virtual reality, computer games and, in general, entertainment.

2. Installing and setting up graphics libraries: OpenGL, GLEW, and GLFW

To install and set up the graphics-related libraries, please have a look at the course webpage: http://www.di.ubi.pt/~agomes/cg/, where you find supporting videos and complementary information about such procedures.

3. A First OpenGL/GLUT Program

The following program draws nothing; it just opens a window in blue.

```
// Include standard headers
#include <stdio.h>
#include <stdlib.h>

// Include GLEW
#include <GL/glew.h>

// Include GLFW
#include <GLFW/glfw3.h>
GLFWwindow* window;
```

```
// Include GLM
#include <glm/glm.hpp>
using namespace glm;

int main( void )
{
    // Initialise GLFW
    if( !glfwInit() )
    {
        fprintf( stderr, "Failed to initialize GLFW\n" );
        getchar();
        return -1;
    }
    glfwWindowHint(GLFW_SAMPLES, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // make Mac happy
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
   // Open a window and create its OpenGL context
   window = glfwCreateWindow( 1024, 768, "Tutorial 01", NULL, NULL);
   if( window == NULL ){
       fprintf( stderr, "Failed to open GLFW window." );
       getchar();
       glfwTerminate();
       return -1;
   }
   glfwMakeContextCurrent(window);
   // Initialize GLEW
   if (glewInit() != GLEW_OK) {
       fprintf(stderr, "Failed to initialize GLEW\n");
       getchar();
       glfwTerminate();
       return -1;
   }
   // Ensure we can capture the escape key being pressed below
   glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);
   // Dark blue background
   glClearColor(0.0f, 0.0f, 0.4f, 0.0f);
   do{
         // Clear the screen.
         glClear( GL_COLOR_BUFFER_BIT );
         // Draw nothing, see you in tutorial 2 !

        // Swap buffers
        glfwSwapBuffers(window);
        glfwPollEvents();
  } // Check if the ESC key was pressed or the window was closed
  while( glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS &&
```

```
        glfwWindowShouldClose(window) == 0 );
    // Close OpenGL window and terminate GLFW
    glfwTerminate();

    return 0;
}
```

Comments:

- Before going through the following exercises, make sure you have succeeded in drawing a triangle as shown in Tutorial 2 at http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/, which draws a triangle from its vertices.
- Every single geometry must be encoded as a set of points, which are moved to graphics card using a vertex buffer object (VBO); for example, a triangle is encoded as a set of three points stored in a VBO.

4. Programming Exercises

1. Write a program to draw two triangles with different colors (red and green).
2. Write a program to draw the following points: (0.0,0.0), (20.0,0.0), (20.0,20.0), (0.0,20.0) and (10.0,25.0). For this purpose, use the **GL_POINTS** primitive. Then change the point size from 1 to 10.
3. Re-write the previous program to draw a 2D house. The 2D house consists of two components: a square (front wall composed by 2 triangles) and a triangle (roof). The first four points given above define the square, while the last three points define the triangle. For this purpose, use the **GL_TRIANGLES** primitive. Then, let us construct two windows and the door of the house.
4. Write a program to draw the sinc function. This function is given by

$$\text{sinc}(x) = \frac{\sin(x)}{x}$$

   with $x \in [-10.0,10.0]$. For this purpose, use the **GL_LINE_STRIP** primitive. The sampling step is 0.25 when *x* varies in the given interval. Recall that sinc(0.0)=1.0.
5. Write a program to draw a circle centered at (cc_x, c_y) with a given radius r, whose points are given by the following equations

$$\begin{cases} x = c_x + r\cos(\theta) \\ y = c_y + r\sin(\theta) \end{cases}$$

   REMARK: The corresponding graphics applications that solve the previous programming exercises follow below.

5. Programming Solutions

Every single modern OpenGL program involves at least three programs:

a) a C++/OpenGL program: main.cpp;
b) a vertex shader program in GLSL (OpenGL Shading Language): vertexshader.vs;

c) a fragment shader program in GLSL: fragmentshader.fs.

The vertex shader allows to re-program part of the graphics pipeline, while the fragment shader allows us to re-program another part of such pipeline. Thus, the graphics pipeline is programmable in modern OpenGL. Let us then see how these three programs work together in the graphics application that solves the first programming exercise above.

Graphics Application #1

To draw two triangles with different colors on screen, we first need to transfer two arrays from the CPU to GPU. The first array contains 2x3 triangle vertices, and the second 2x3 colors. The code in red shows the differences relative to the one in Tutorial 2 (http://www.opengl-tutorial.org/beginners-tutorials/tutorial-2-the-first-triangle/). These differences come from the fact that now we have an array for colors.

main.cpp

```
// Include standard headers
#include <stdio.h>
#include <stdlib.h>

// Include GLEW
#include <GL/glew.h>

// Include GLFW
#include <GLFW/glfw3.h>
GLFWwindow* window;

// Include GLM
#include <glm/glm.hpp>
using namespace glm;

//Include vertex and fragment shaders
#include <common/shader.hpp>

int main( void )
{
  // Initialise GLFW
  if( !glfwInit() )
  {
    fprintf( stderr, "Failed to initialize GLFW\n" );
    getchar();
    return -1;
  }

  glfwWindowHint(GLFW_SAMPLES, 4);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // To make MacOS happy
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

  // Open a window and create its OpenGL context
  window = glfwCreateWindow( 1024, 768, "Tutorial 02 - Red triangle", NULL, NULL);
  if( window == NULL )
  {
    fprintf( stderr, "Failed to open GLFW window. Check your GPU compatibility.\n" );
    getchar();
    glfwTerminate();
```

```
    return -1;
}
glfwMakeContextCurrent(window);

// Initialize GLEW
glewExperimental = true; // Needed for core profile
if (glewInit() != GLEW_OK)
{
    fprintf(stderr, "Failed to initialize GLEW\n");
    getchar();
    glfwTerminate();
    return -1;
}

// Ensure we can capture the escape key being pressed (keyboard event)
glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);

// Dark blue background of window
glClearColor(0.0f, 0.0f, 0.4f, 0.0f);

GLuint VertexArrayID;
glGenVertexArrays(1, &VertexArrayID);
glBindVertexArray(VertexArrayID);

// Create and compile our GLSL program from the shaders
GLuint programID = LoadShaders( "SimpleVertexShader.vertexshader", "SimpleFragmentShader.fragmentshader" );

// Geometry: vertices for 2 triangles in the default domain [-1,1]x[-1,1]x[-1,1]
static const GLfloat g_vertex_buffer_data[] = {
        -1.0f, -1.0f, 0.0f, // vertices of the 1st triangle
        1.0f, -1.0f, 0.0f,
        0.0f, 1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,  // vertices of the 2nd triangle
        1.0f, 1.0f, 0.0f,
        0.0f, -1.0f, 0.0f,
};

// Colors: 1 color per vertex. The first triangle in red and the second in green
static const GLfloat g_color_buffer_data[] = {
        1.0f,  0.0f,  0.0f,
        1.0f,  0.0f,  0.0f,
        1.0f,  0.0f,  0.0f,
        0.0f,  1.0f,  0.0f,
        0.0f,  1.0f,  0.0f,
        0.0f,  1.0f,  0.0f,
};

// Transfer vertex geometry to GPU memory
GLuint vertexbuffer;
glGenBuffers(1, &vertexbuffer);
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL_STATIC_DRAW);

// Transfer vertex colors to GPU memory
GLuint colorbuffer;
glGenBuffers(1, &colorbuffer);
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(g_color_buffer_data), g_color_buffer_data, GL_STATIC_DRAW);
```

```
  do{// for each frame
      // Clear the screen
      glClear( GL_COLOR_BUFFER_BIT );

      // Use our shader
      glUseProgram(programID);

      // 1rst attribute buffer : vertices
      glEnableVertexAttribArray(0);
      glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
      glVertexAttribPointer(
            0,                  // attribute 0 must match the layout in the shader
            3,                  // size
            GL_FLOAT,           // type
            GL_FALSE,           // normalized?
            0,                  // stride
            (void*)0            // array buffer offset
      );

      // 2nd attribute buffer : colors
      glEnableVertexAttribArray(1);
      glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
      glVertexAttribPointer(
            1,                  // attribute 1 must match the layout in the shader
            3,                  // size
            GL_FLOAT,           // type
            GL_FALSE,           // normalized?
            0,                  // stride
            (void*)0            // array buffer offset
      );

      // Draw 2 triangles
      glDrawArrays(GL_TRIANGLES, 0, 6); // 6 indices starting at 0 -> 1 triangle

      glDisableVertexAttribArray(0);
      glDisableVertexAttribArray(1);

      // Swap buffers
      glfwSwapBuffers(window);
      glfwPollEvents();

  } // Check if the ESC key was pressed or the window was closed
  while ( glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS && glfwWindowShouldClose(window) == 0 );

  // Cleanup VBO
  glDeleteBuffers(1, &vertexbuffer);
  glDeleteBuffers(1, &colorbuffer);
  glDeleteVertexArrays(1, &VertexArrayID);
  glDeleteProgram(programID);

  // Close OpenGL window and terminate GLFW
  glfwTerminate();

  return 0;
}
```

The **vertex shader** is as follows:

vertexshader.vs

```
#version 330 core

// Input vertex data and color data
layout(location = 0) in vec3 vertexPosition;
layout(location = 1) in vec3 vertexColor;

// Output fragment data
out vec3 fragmentColor;

void main()
{
	// position of each vertex in homogeneous coordinates
	gl_Position.xyz = vertexPosition;
	gl_Position.w = 1.0;

	// the vertex shader just passes the color to fragment shader
	fragmentColor = vertexColor;
}
```

The **connection** between the main.cpp and vertexshader.vs is done through the first argument of glVertexAttribPointer, which must match the layout location in the shader. In the case of the vertex buffer, the first of argument of glVertexAttribPointer is 0, so it corresponds to the layout 0 in the shader. In the case of the color buffer, the first of argument of glVertexAttribPointer is 1, so it corresponds to the layout 1 in the shader.

In turn, the code of the fragment shader is as follows:

fragmentshader.fs

```
#version 330 core

// Interpolated values from the vertex shaders
in vec3 fragmentColor;

// Ouput data
out vec3 color;

void main()
{
	// Output color = color specified in the vertex shader,
	// interpolated between all 3 surrounding vertices
	color = fragmentColor;
}
```

Recall that the fragment shader handles the color of each pixel spanned by each triangle.

The shading (coloring) of the triangle pixels is performed in parallel on GPU; that is, the two shaders run for each triangle pixel, but this task is parallelized on GPU for all pixels. The execution of the function glDrawArrays, with GL_TRIANGLES mode on, triggers the parallel shading process on GPU.

To better understand the structure of a graphics applications, let us rewrite **main.cpp** in a more organized manner:

```
// Include standard headers
#include <stdio.h>
#include <stdlib.h>

// Include GLEW
#include <GL/glew.h>

// Include GLFW
#include <GLFW/glfw3.h>
GLFWwindow* window;

// GLM header file
#include <glm/glm.hpp>
using namespace glm;

// shaders header file
#include <common/shader.hpp>

// Vertex array object (VAO)
GLuint VertexArrayID;

// Vertex buffer object (VBO)
GLuint vertexbuffer;

// color buffer object (CBO)
GLuint colorbuffer;

// GLSL program from the shaders
GLuint programID;

//----------------------------------------------------------------------------
void transferDataToGPUMemory(void)
{
	// VAO
	glGenVertexArrays(1, &VertexArrayID);
	glBindVertexArray(VertexArrayID);

	// Create and compile our GLSL program from the shaders
	programID = LoadShaders( "SimpleVertexShader.vertexshader", "SimpleFragmentShader.fragmentshader" );
	// vertices for 2 triangles
	static const GLfloat g_vertex_buffer_data[] = {
		-1.0f, -1.0f, 0.0f,
		 1.0f, -1.0f, 0.0f,
		 0.0f,  1.0f, 0.0f,
		-1.0f, 1.0f, 0.0f,
		 1.0f, 1.0f, 0.0f,
		 0.0f,  -1.0f, 0.0f,
	};

	// One color for each vertex. They were generated randomly.
	static const GLfloat g_color_buffer_data[] = {
		1.0f,  0.0f,  0.0f,
		1.0f,  0.0f,  0.0f,
		1.0f,  0.0f,  0.0f,
		0.0f,  1.0f,  0.0f,
		0.0f,  1.0f,  0.0f,
```

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

        // Move vertex data to video memory; specifically to VBO called vertexbuffer
        glGenBuffers(1, &vertexbuffer);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
        glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data), g_vertex_buffer_data, GL_STATIC_DRAW);

        // Move color data to video memory; specifically to CBO called colorbuffer
        glGenBuffers(1, &colorbuffer);
        glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
        glBufferData(GL_ARRAY_BUFFER, sizeof(g_color_buffer_data), g_color_buffer_data, GL_STATIC_DRAW);

}
//---------------------------------------------------------------------------
void cleanupDataFromGPU()
{
        glDeleteBuffers(1, &vertexbuffer);
        glDeleteBuffers(1, &colorbuffer);
        glDeleteVertexArrays(1, &VertexArrayID);
        glDeleteProgram(programID);
}
//---------------------------------------------------------------------------
void draw (void)
{
        // Clear the screen
        glClear( GL_COLOR_BUFFER_BIT );

        // Use our shader
        glUseProgram(programID);

        // 1rst attribute buffer : vertices
        glEnableVertexAttribArray(0);
        glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
        glVertexAttribPointer(
                0,              // attribute 0. No particular reason for 0, but must match the layout in the shader.
                3,              // size
                GL_FLOAT,  // type
                GL_FALSE,   // normalized?
                0,              // stride
                (void*)0      // array buffer offset
        );

        // 2nd attribute buffer : colors
        glEnableVertexAttribArray(1);
        glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);
        glVertexAttribPointer(
                1,               // attribute. No particular reason for 1, but must match the layout in the shader.
                3,               // size
                GL_FLOAT,   // type
                GL_FALSE,    // normalized?
                0,               // stride
                (void*)0       // array buffer offset
        );

        glEnable(GL_PROGRAM_POINT_SIZE);
        //glPointSize(10);
        // Draw the 2 triangles !
```

```
	glDrawArrays(GL_TRIANGLES, 0, 6); // 6 indices starting at 0 -> 1 triangle
	//glDrawArrays(GL_POINTS, 0, 6); // 6 indices starting at 0 -> 1 triangle

	glDisableVertexAttribArray(0);
	glDisableVertexAttribArray(1);
}
//----------------------------------------------------------------------------
int main( void )
{
	// Initialise GLFW
	glfwInit();

	glfwWindowHint(GLFW_SAMPLES, 4);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
	glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
	glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE); // To make MacOS happy; should not be needed
	glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

	// Open a window
	window = glfwCreateWindow( 1024, 768, "Two Triangles in Red and Green", NULL, NULL);

	// Create window context
	glfwMakeContextCurrent(window);

	// Initialize GLEW
	glewExperimental = true; // Needed for core profile
	glewInit();

	// Ensure we can capture the escape key being pressed below
	glfwSetInputMode(window, GLFW_STICKY_KEYS, GL_TRUE);

	// Dark blue background
	glClearColor(0.0f, 0.0f, 0.4f, 0.0f);

	// transfer my data (vertices, colors, and shaders) to GPU side
	transferDataToGPUMemory();

	// render scene for each frame
	do{
		// drawing callback
		draw();
		// Swap buffers
		glfwSwapBuffers(window);
		// looking for input events
		glfwPollEvents();

	} // Check if the ESC key was pressed or the window was closed
	while( glfwGetKey(window, GLFW_KEY_ESCAPE ) != GLFW_PRESS &&
		   glfwWindowShouldClose(window) == 0 );

	// Cleanup VAO, VBOs, and shaders from GPU
	cleanupDataFromGPU();

	// Close OpenGL window and terminate GLFW
	glfwTerminate();

	return 0;
}
```