●Real-Time Graphics Programming Course OpenGL Project Report

●Emir Erdogdu – Matriculation number: V08870 – emir.erdogdu@studenti.unimi.it

●1. Introduction

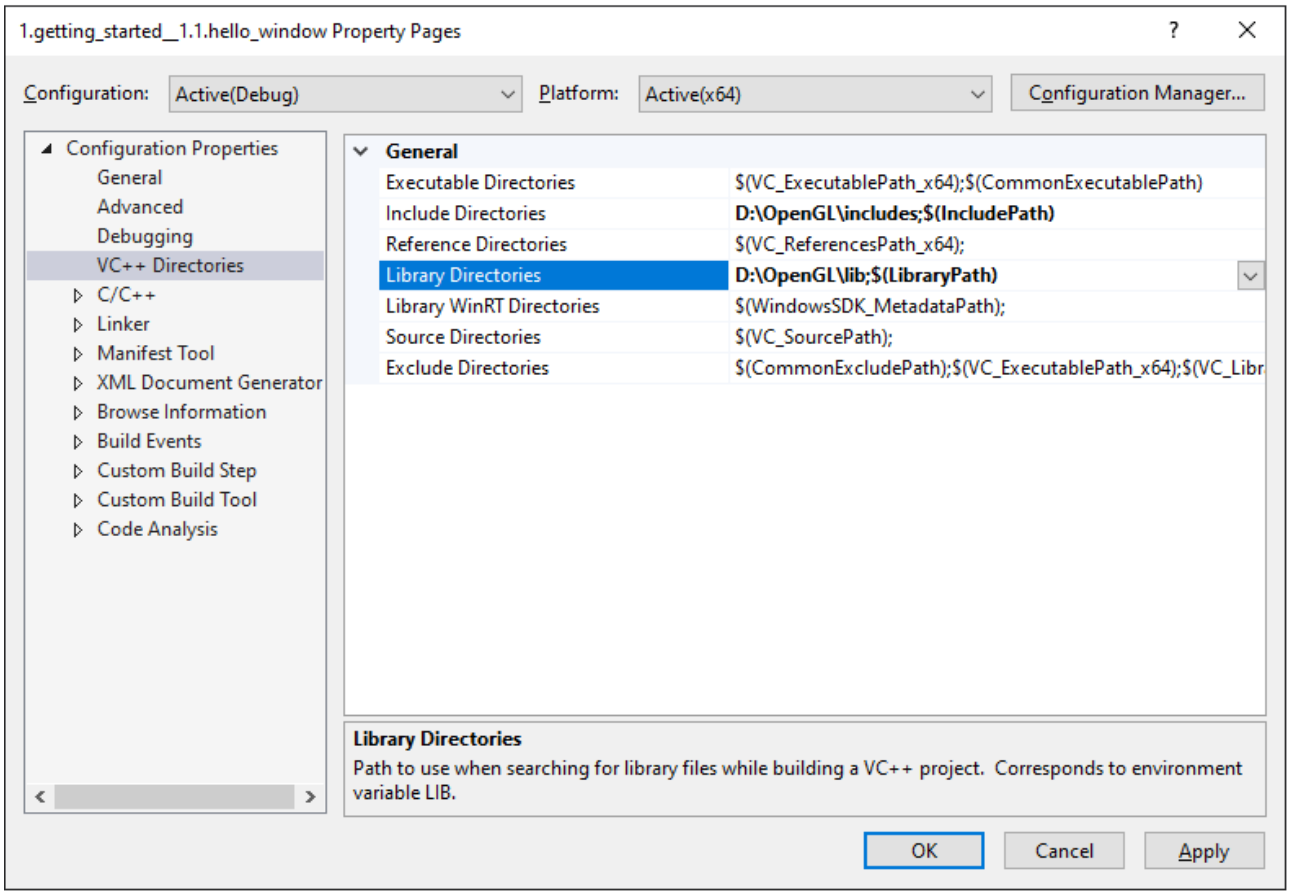
Open Graphics Library (OpenGL) is an API used to draw 2D or 3D graphics, in other words, it is a library. This library gives software developers a chance to manage their graphics hardware. OpenGL is essentially considered an API (Application Programming Interface) that provides a large number of functions that can be used to render graphics and images. However, OpenGL is not an API per se, but a specification developed and maintained solely by the Khronos Group. This library is independent of the operating system and the platform on which the operating system is running. Just as writing to the screen and receiving data from the user are standardized in ANSI C with functions such as printf() and scanf(), and these two functions do the same job no matter what operating system you go, the OpenGL library has standardized drawing graphics to the screen. Thanks to OpenGL, programming is done independent of hardware factors such as the model of the graphics card or the architecture of the processor. In addition, programming independent of the operating system is also done. OpenGL has become a popular tool because of its ease of use and these "portability" features.

The people who develop the actual OpenGL libraries are usually the graphics card manufacturers. Every graphics card supports certain versions of OpenGL, which are versions of OpenGL developed specifically for that card (series). When using an Apple system, the OpenGL library is maintained by Apple itself, and under Linux there is a combination of versions of graphics providers and hobbyists' adaptations of these libraries. This also means that when OpenGL exhibits strange behavior that it shouldn't, it's most likely the fault of the graphics card manufacturers (or whoever developed/maintained the library).

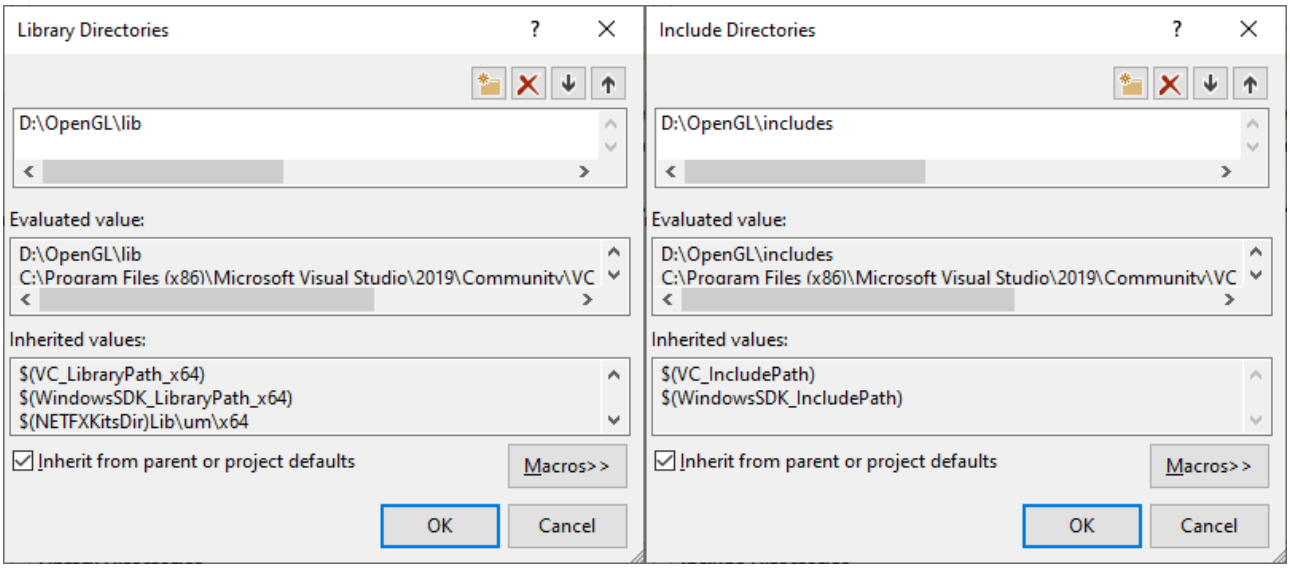
●2. OpenGL Installation

Visual Studio opens and a new project is created. C++ is selected and the blank project is created. Once this is done, we are now ready to create our first OpenGL application.

Linking, need to link the library to our project so that the project can use GLFW. This is done by specifying in the linker settings that we want to use glfw3.lib. It is necessary to add the directory to the project.

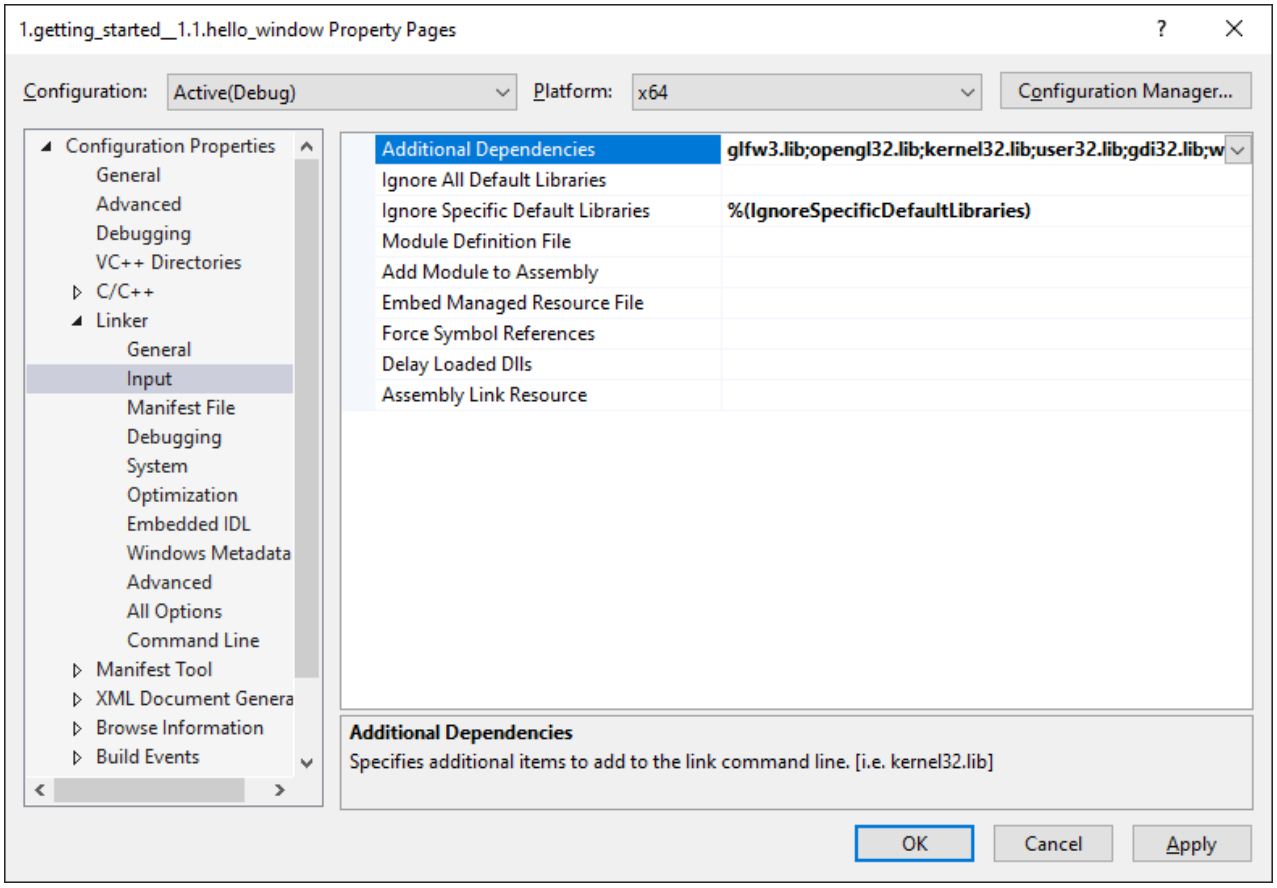


Relevant directories are added to tell the project where to search. This is done by manually placing in the text or by clicking the appropriate location string and selecting the edit option.



When searching for library and header files, the IDE will also search for these directories. As soon as Include folder is added from GLFW, you will be able to find all header files for GLFW by adding

<GLFW/..>. The same is true for library directories. Since VS can now find all the required files,



finally GLFW can be linked to the project by going to the Linker tab and Input.

Header files for GLFW are added as follows.

```
#include <GLFW/glfw3.h>
```

Because there are many different versions of OpenGL drivers, the location of many of their functions is unknown at compile time and must be queried at runtime. This process can be complex and tedious to redefine every function you may need. The GLAD library can be used to solve this situation.

●Installing GLAD

GLAD is an open-source library that manages many cumbersome and cumbersome operations. GLAD uses a web service (http://glad.dav1d.de/) where we can tell GLAD for which version of OpenGL we want to define and load all relevant OpenGL functions based on that version. C++ language is selected from GLAD web service and API is determined at least 3.3. Build is selected to generate the resulting library files. Two include folders and a glad.c file are obtained. Copy both include folders (GLAD and KHR) to the include(s) directory and add the glad.c file to the project. To include the Glad library in the project:

```
#include <glad/glad.h>
```

•3. Project Description

The libraries used in the project are listed below.

The libraries used in the project are listed below.

```
#include <iostream>
#include <glad/glad.h>
#include <GLFW/glfw3.h>
#include <string>
```

The following libraries are used for matrices used to create and manage polygons.

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
```

The following libraries are used for matrices used to create and manage polygons.

```
#include "graphics/shader.h"
#include "graphics/texture.h"
#include "graphics/model.h"
#include "graphics/light.h"
#include "graphics/models/cube.hpp"
#include "graphics/models/lamp.hpp"
```

The following libraries are used for matrices used to create and manage polygons.

```
#include "io/keyboard.h"
#include "io/mouse.h"
#include "io/joystick.h"
#include "io/screen.h"
#include "io/camera.h"
```

Its main function is created, with which we will instantiate the GLFW window.

```
glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
```

Shaders: Graphics card (GPU) widgets that are used to define the properties of a pixel or vertex.
Vertex: Vertex represents a graphical point. There are 2 types of Shaders.
Fragment Shader: It is a 2D Shader type that is used to process and define attributes such as color, alpha channel, z-depth of the given pixel.
Vertex Shader: Graphics Pipeline is used to process and define the attributes related to the Vertexes it uses, such as the positions of the Vertexes, their text coordinates, their colors.

The following code snippet defines shader files.

```
Shader shader("assets/object.vs", "assets/object.fs");
Shader lampShader("assets/object.vs", "assets/lamp.fs");
```

The code block used to create the models. Models are defined in the cube.hpp and lamp.hpp header files in the src/graphics/models folder.

```
glm::vec3 cubePositions[] = {
                        glm::vec3(0.0f,  0.0f,  0.0f),
};
Cube cubes(cubePositions[0], glm::vec3(1.0f));
cubes.init();
```

The code block used to create lights.

```
DirLight dirLight = { glm::vec3(-0.2f, -1.0f, -0.3f), glm::vec3(0.1f), glm::vec3(0.4f), glm::vec3(0.5f) };

	glm::vec3 pointLightPositions = glm::vec3(0.7f, 0.2f, 2.0f);
	Lamp lamps;
		lamps = Lamp(glm::vec3(1.0f),
			glm::vec3(0.05f), glm::vec3(0.8f), glm::vec3(1.0f),
			1.0f, 0.07f, 0.032f,
			pointLightPositions, glm::vec3(0.25f));
		lamps.init();


	SpotLight s = {
		Camera::defaultCamera.cameraPos, Camera::defaultCamera.cameraFront,
		glm::cos(glm::radians(12.5f)), glm::cos(glm::radians(20.0f)),
		1.0f, 0.07f, 0.032f,
		glm::vec3(0.0f), glm::vec3(1.0f), glm::vec3(1.0f)
	};
```

With the following code block, the graphic elements are refreshed until the screen is turned off.

```
while (!screen.shouldClose()) {
		// calculate dt
		double currentTime = glfwGetTime();
		deltaTime = currentTime - lastFrame;
		lastFrame = currentTime;

		// process input
		processInput(deltaTime);

		// render
		screen.update();

		// draw shapes
		shader.activate();
```

The code blocks used to change texture.

```
if (texture1) {
			cubes.cleanup();
			char text[100] = "assets/texture1.png";
			cubes.setTexturename(text);
			cubes.init();
			texture1 = !texture1;
		}
```

The code blocks used to move lamp.

```
		if (lampUp) {
			lamps.pointLight.position.y += 0.1f;
			lamps.pos.y = lamps.pointLight.position.y;
			lampUp = !lampUp;
		}
```

With the code block below, the switching on and off of previously defined lamps is managed.

```
if (lightOn) {
	dirLight.render(shader);
	lamps.pointLight.render(shader, 0);
	shader.setInt("noPointLights", 1);
}
else
{
	shader.setInt("noPointLights", 0);
}


if (flashlightOn) {
	s.position = Camera::defaultCamera.cameraPos;
	s.direction = Camera::defaultCamera.cameraFront;
	s.render(shader, 0);
	shader.setInt("noSpotLights", 1);
}
else {
	shader.setInt("noSpotLights", 0);
}
```

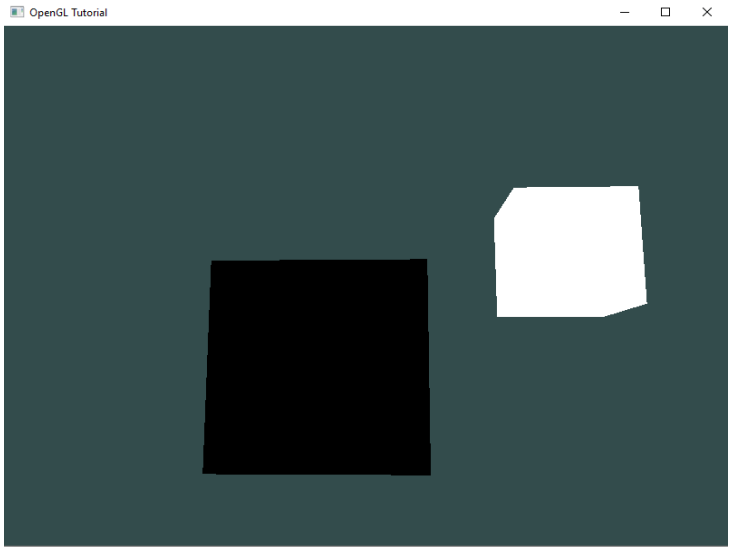
OpenGL doesn't have any matrix or vector information built in, so we need to define our own math classes and functions. In this project, it is preferred to abstract from all small mathematical details and manage this process with the OpenGL math library called GLM (OpenGL Mathematic). GLM objects are created with the following code block. Matrices such as view, and projection are used to manage the scene.

```
glm::mat4 view = glm::mat4(1.0f);
glm::mat4 projection = glm::mat4(1.0f);
view = Camera::defaultCamera.getViewMatrix();
projection = glm::perspective(
	glm::radians(Camera::defaultCamera.zoom),
	(float)Screen::SCR_WIDTH / (float)Screen::SCR_HEIGHT, 0.1f, 100.0f);

shader.setMat4("view", view);
shader.setMat4("projection", projection);

cubes.render(shader);

lampShader.activate();
lampShader.setMat4("view", view);
lampShader.setMat4("projection",projection);
lamps.render(lampShader);

// send new frame to window
screen.newFrame();
glfwPollEvents();
cubes.cleanup();
lamps.cleanup();

glfwTerminate();
return 0;
```

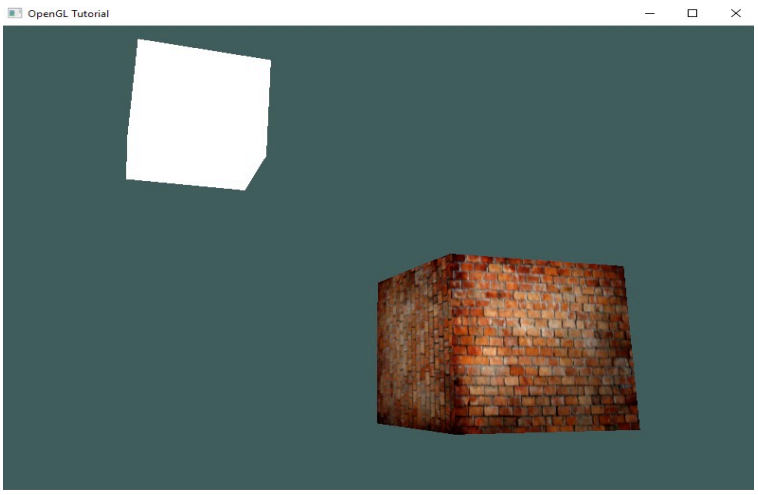
Finally, the application is managed by capturing the keyboard movements with the following code block within the main function.

```
void processInput(double deltaTime) {
	if (Keyboard::key(GLFW_KEY_ESCAPE)) {
		screen.setShouldClose(true);
	}

	if (Keyboard::keyWentDown(GLFW_KEY_L)) {
		lightOn = !lightOn;
	}
```

When the application is run, a screen like the one below greets us. A light source and an object appear on the screen.

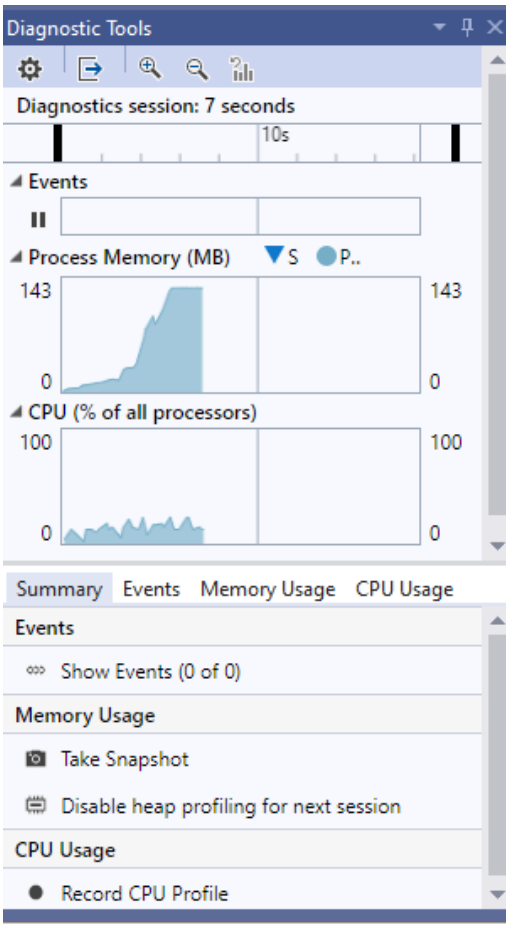


Camera control is provided with W - A - S and D keys. This is how the scene moves. The light source can be moved within the scene with the arrow keys. The texture of the object can be changed with the 1 - 2 and 3 keys. In addition, 2 different light sources can be turned on and off with the L and O keys.



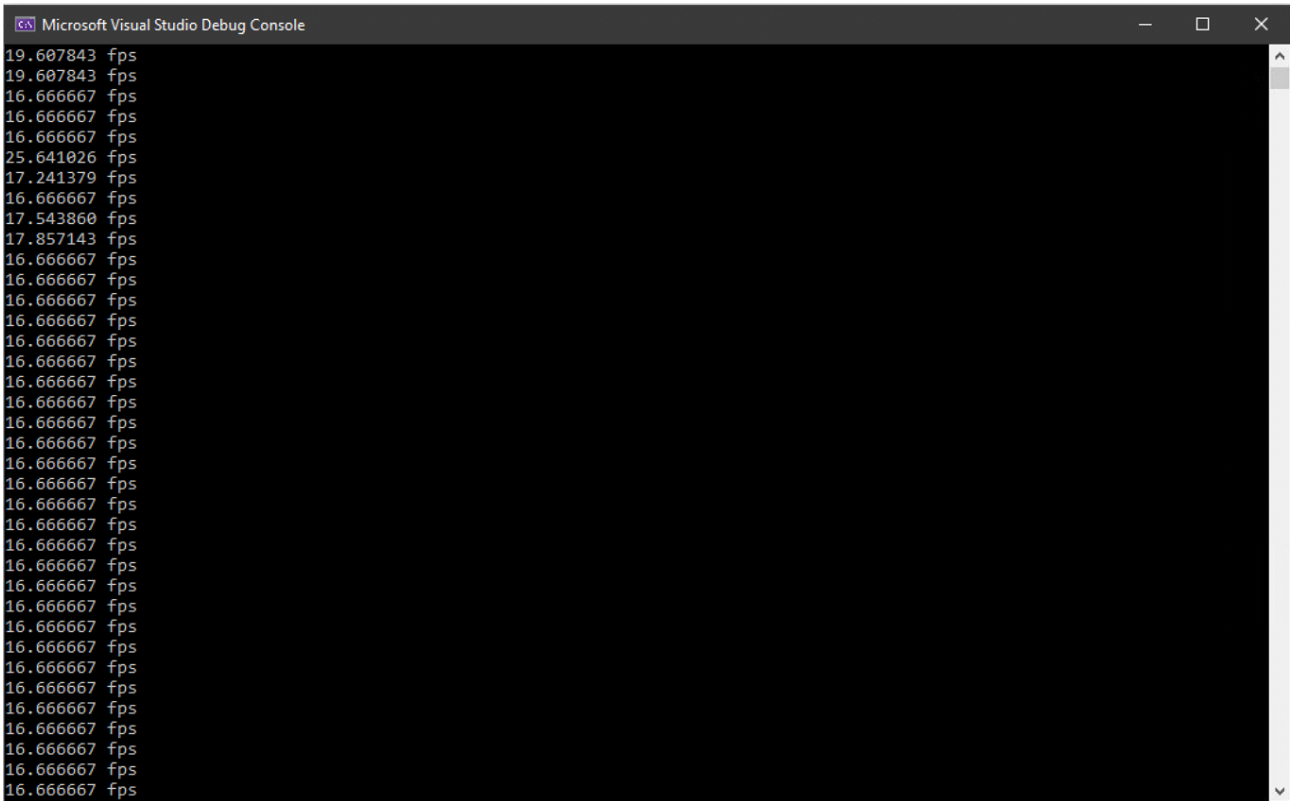
●4. Performance Evaluation

When the performance of the application is examined, the following result is encountered.



When the application is run, the fps value is measured with the code block seen below.

```
while (!screen.shouldClose()) {

        // Measure speed
        double currentTime = glfwGetTime();
        nbFrames++;
        if (currentTime - lastTime >= 1.0) { // If last prinf() was more than 1 sec ago
                // printf and reset timer
                printf("%f fps\n", 1000.0 / double(nbFrames));
                nbFrames = 0;
                lastTime += 1.0;
        }
```



The performance test was performed on computer equipped with an Intel Core i7 6700k CPU running at 4 GHz, 16 GB of RAM and an Nvidia GeForce GTX 970 Graphics Card. As it can be seen in the console presented above, the project runs around 17 frames per second for the majority of the time. While for playing video games or some other tasks a result around 30 fps may be preferred, this program is still usable and presents a screen tearing free experience.

●5. General Algorithm View