My First OpenGL Project

Project Description

Filip Vuković

Osijek 2021.

Introduction

Task

My task is to show a raw image (either RGB, YUV or similar format) on screen using OpenGL.

Once that works properly, the next task is to optimize the process for my own graphics card. If no relevant optimizations are available, investigate graphics card optimizations in general and compare their advantages over others.

Environment

The solution should be implemented in C language using gcc compiler, OpenGL loader library (GLAD) and multiplatform OpenGL library (GLFW).

GLFW is free, an Open Source, multi-platform library for OpenGL that provides a simple API for creating windows, contexts and surfaces, receiving input and events. On the other hand, GLAD is a loading library that generates a loader for your exact needs based on the official specifications.

Optimizing the process for a GPU requires a set of extensions defined in *glext.h* header file. There are a lot of extension types: vendor-specific, generic, and ARB-approved. For optimization of my graphic card *NVIDIA mx250*, I used a lot of ARB-approved as well as vendor-specific GL_NV (NVIDIA hardware) extensions. Besides *glext.h*, I have generated addiotional extensions libraries using GLAD

Solution

"How to display an image in OpenGL?"

Short answer: By texturing a rectangle, or even better two triangles.

I used Visual Studio Code in order to solve this task. The solution consists of both C and C++ solution files named "learning".

In order to display a raw image on the screen I used the following include files:

- GLAD
- GLFW3
- stb_image – basically an image-loading C library
- glext – extension header file used for GPU optimization
- stdbool

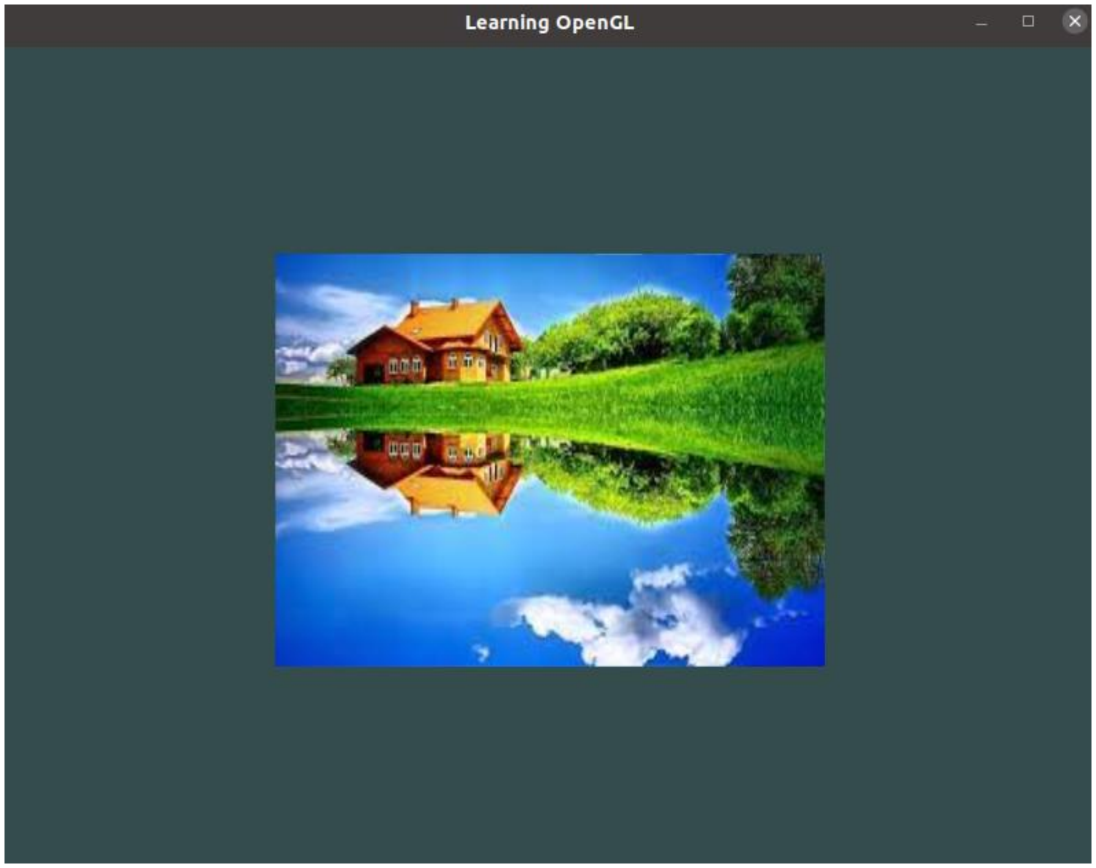
First, we initialize GLFW, GLAD and window object that stores all the windowing data. Then, we create and compile necessary shaders. A key thing to remember is that modern OpenGL requires at least vertex and fragment shaders if we want to do some rendering. The Vertex Shader handles the processing of individual vertices. The fragment shader on the other hand takes care of how the pixels between the vertices look. Shaders are written in the C-like language GLSL. GLSL is tailored for use with graphics and contains useful features specifically targeted at vector and matrix manipulation.

Next, we define vertices, indices and generate special OpenGL objects that communicate with our GPU. Those objects manipulate with vertex and buffer data, make required calculations and store results.

In the final stage we load our image, generate texture and bound that texture to current texture object.

Lastly, render loop. Since we don't want the application to draw a single image and then immediately close the window. We want the application to keep drawing images and handling user input until the program has been explicitly told to stop. For this reason, we must create a render loop. In a render loop we handle inputs, rendering, drawing, buffer swapping and possibly some pool events as well.

And that is pretty much it! The program I just presented displays this:



For more detailed description of the source code please follow the comments in *learning.c* file.