

Homework 1 - OpenGL Basics

Isaiah Martinez
CSUN
Computer Science Department
isaiah.martinez.891@my.csun.edu

02/15/2024

Contents

1	Introduction	2
2	Design Goals	2
3	System Behavior	2
4	Logical View	3
4.1	Mid-Level Design	3
4.1.1	Vertices	3
4.1.2	VBO and VAO	4
4.1.3	Render Loop	4
5	Process View	5
6	Development View	5
7	Use Case View	5

1 Introduction

This document describes the architecture and design for the first assignment with OpenGL. The objective with this assignment was to create a triangle that oscillates between colored and non-colored.

There are two windows that are created when the program is run: window 1 which contains the triangle, and window 2 which is the output for the program. Window 2 displays nothing until window 1 is closed.

There is a single major stakeholder:

1. The professor

2 Design Goals

The priorities for the design are as follows:

- The design should minimize complexity
- The design should be conceptually easy to understand
- The design should be easy to modify

3 System Behavior

The use case view is the prime motivator for the System Behavior. This is because the program is simple with no complex components.

The program starts with the triangle filled in, demonstrated by the left triangle in Figure 1, and then progresses onward infinitely until the Escape Key is pressed. The code describing this behavior will be explained further in Section 4. The Use Case diagram can also be seen for further explanation in figure 4 within section 7.

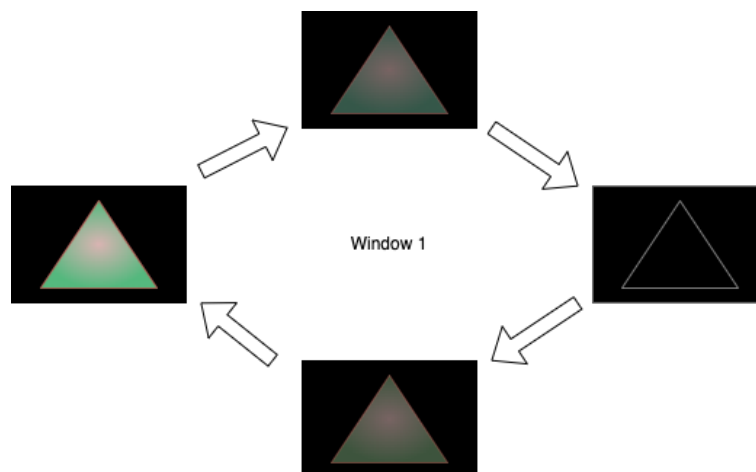


Figure 1: System Behavior for Window 1.

In Figure 1, the right triangle has a white outline and black fill to demonstrate that the triangle exists, but lacks color. When the program is run however, the triangle loses color until it is not visible anymore.

4 Logical View

In this section, the system will be described from a Mid-Level Design View, looking at the code itself. Specifically, we will be looking at the variables and tools used to make the color changing of the triangle work.

4.1 Mid-Level Design

There are several key parts to the code that allow for triangle to be drawn each frame.

4.1.1 Vertices

The first key part is the vertices variable. This variable is defined as an array with length 18.

I utilized a variable `colorValue` to represent the value of the current color for the given vertex. It is initialized to `1.0f` to start with a colored triangle, similar to how the demonstration video starts.

Another variable I created was `deltaColor` which represents the rate of change for `colorValue`. This logic is explained later on in the render loop.

As seen in Figure 2 below, the array is separated into 2 different sections: the positions of the points to generate the triangle, and the colors associated with that part of the triangle. The ordering is x, y, z for the coordinates, eg. `vertices[0] = 0.5f` represents $x = 0.5$ on the cartesian coordinate system. `vertices[1] = -0.5f` represents $y = 0.5$ also on the same plane. Notice that there are no points lying outside of the cartesian plane since all z values = 0.

The colors represent RGB in order, thus `vertices[3] = 1.0f` representing the amount of red to apply. `vertices[4]` and `vertices[5] = 0.0f` representing no values for Green and Blue respectively. Thus, the overall color of this vertex extending towards the other points of the triangle is red.

```
//color value
float colorValue = 1.0f;

//deltaColor = rate of change for color
float deltaColor = -0.002f;

// set up vertex data (and buffer(s)) and configure vertex attributes
// -----
float vertices[] = {
    // positions      // colors
    0.5f, -0.5f, 0.0f, colorValue, 0.0f, 0.0f, // bottom right
    -0.5f, -0.5f, 0.0f, 0.0f, colorValue, 0.0f, // bottom left
    0.0f, 0.5f, 0.0f, 0.0f, 0.0f, colorValue // top
};
```

Figure 2: Important Variables utilized for changing the color of the triangle.

4.1.2 VBO and VAO

Now having described the vertices variable in detail, I will briefly describe another set of key parts of the code: the VBO and VAO.

We use a VBO (Vertex Buffer Object) for managing the vertices variable. The VBO is used for storing the vertex data, which in our case would be the position and color. This is then sent to the GPU for fast rendering.

We also utilize a VAO (Vertex Array Object) for changing states of the VBO quickly and easily. This proves useful for adjusting the state of the VBO from one color to the next in an efficient manner.

4.1.3 Render Loop

The final key part of the code is within the render loop.

The first portion worth mentioning is the function given which processes the user input. This function allows for the user to close the program.

The more important part of the render loop is the logic used to update the triangle. This is done using the VAO and adjusting the previously discussed variables `colorValue` and `deltaColor`.

First, `colorValue` is added to `deltaColor` and subsequently set to the new value. Then, the logic performs a boolean check to see if `colorValue` has hit or passed either extreme value for the color value possible for the system. These values are 1.0 and 0.0 respectively. If `colorValue` has reached either extrema or beyond them, then the sign of `deltaColor` is flipped to repeat the same logic as before. This optimization allows for minimal code to be written and succinctly describing the process.

Afterwards, the newly updated `colorValue` is attached to the respective color value within the vertices variable. Subsequently, we draw the buffer and utilize the VAO to render the triangle once again.

```

// render loop
// -----
while (!glfwWindowShouldClose(window))
{
    // input
    // ----
    processInput(window);

    //increment/decrement colorValue
    colorValue += deltaColor;

    //flip the sign of the deltaColor when at either extreme (1.0f or 0.0f)
    if (colorValue >= 1.0f || colorValue <= 0.0f) {
        deltaColor *= -1;
    }

    //attach the updated colorValue to the RGB of triangle
    vertices[3] = colorValue;
    vertices[10] = colorValue;
    vertices[17] = colorValue;

    //Debugging to see that colorValue has been updated in console
    //std::cout << "color value is:" << colorValue << std::endl;

    //updates the buffer
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // # frames to wait until updating
    //glfwSwapInterval(30);

    // render the triangle
    glBindVertexArray(VAO);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    // glfw: swap buffers and poll IO events (keys pressed/released, mouse moved etc.)
    // -----
    glfwSwapBuffers(window);
    glfwPollEvents();
}

```

Figure 3: Render Loop logic.

5 Process View

How many threads are running? ?? one for each window?

There is a single thread managing the program.

Upon building the program, the triangle is drawn each frame at a slightly different RGB value.

6 Development View

2 threads?

- 1 thread with the console
- 1 thread with the OpenGL rendered triangle

7 Use Case View

The user has the ability to close window 1 by pressing the Escape key. Afterwards, the user has the ability to close window 2 by pressing any key. In fact, there is a prompt telling the user this information.

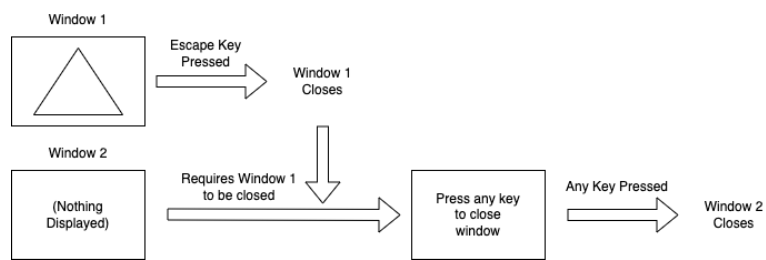


Figure 4: A diagram demonstrating the Use Case Scenario.