Michael McQuade

Dr. Zhigang Deng

Computer Graphics

18 February 2021

Homework 1: Drawing rectangles

In this assignment, the task was to take an input file describing various rectangles and generate bitmap files from it. The input file was defined as the first line containing the number of rectangles to output, and then each following line defined a single rectangle. A rectangle was defined by providing two points, an RGB value, and a z-index.

The two points should that help define the rectangle should be opposite, diagonal corners in order for the filling algorithm to correctly create a rectangle. The RGB values in the input file are on a scale from 0 to 1. It is possible to calculate the traditional RGB value by multiplying 255 by the input RGB value. For instance, if the input file described an input value of 0.78125, 0.05468, 0.79296 this would be equivalent to an RGB value of 199, 14, 202. The depth value, when considered, is used to calculate which rectangle should be on top.

A frame buffer represents a single image that will be drawn to the screen. In order to keep track of color and depth values, the frame buffer implementation has two underlying data stores. Both of these data stores are the same shape. In this implementation, each of these data stores is represented as a vector of vectors, in order to easily represent two-dimensional data. The GZ graphics library that is being implemented has a number of data types defined, such as GzColor and GzReal. GzColor is a data structure that represents RGBA values, and GzReal is a wrapper around the primitive double type. The two data stores for the frame buffer implementation utilize these data types to store color and depth information, accordingly.

In order to populate the data stores with initial values, a clear value is declared. A clear value is the value that will be put in the buffer initially, or any time that the clear method is called. The clear method accepts an integer where each binary digit of the integer represents a different buffer to either clear or not. As an example, the number 2 represents the color buffer, and in binary is represented as 0010. The number 4 represents the depth buffer, and in binary is represented as 0100. Considering you pass the clear function the number 6 (0110), then both the binary digits for color and depth are 1, or enabled, so both would be cleared.

There is also a flag for enabling depth checking altogether. The GZ library accepts a binary flag to the enable function which will disable or enable certain features. At this time, the only accepted flag is to enable depth testing. Depth testing is the act of checking the depth buffer to see if the current value in the frame buffer is behind or in front of the value that is about to be inserted into the color buffer. If the value in the frame buffer is closer to the viewer, then the fragment being inserted should be discarded, because it wouldn't be visible anyways. Without considering the depth buffer, the colors will be overwritten in the order they're written in the file. This means the first rectangle in the file would be covered by all other rectangles and the last rectangle described in the file would be the one on top. Both considerations are visible in the image below.

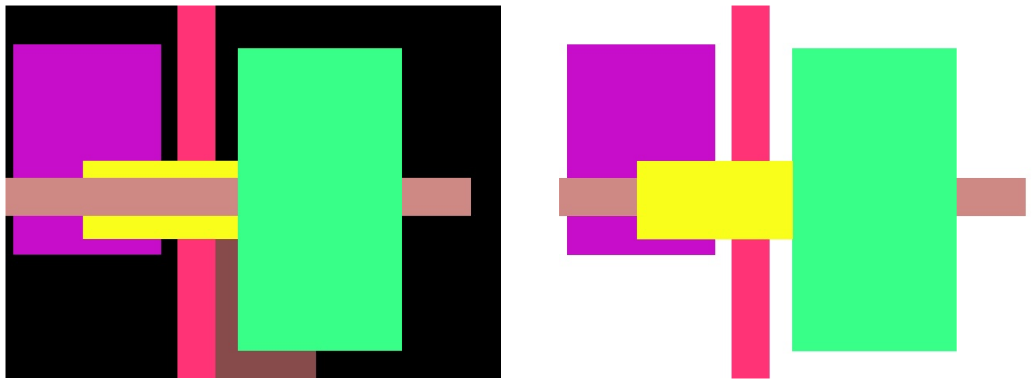

Figure 1 - Without depth testing (left) and with depth testing (right)

While researching this, it was interesting to realize that the way the depth function works could be changed out based on the requirements of the application. For instance, in some cases it might be better to discard items closer to the viewer rather than further away. Because of this, OpenGL's implementation of the depth buffer has the option to accept different functions which will process the fragments differently. In the current implementation of the GZ library, the depth buffer is using the equivalent of OpenGL's GL_GREATER function, which is actually the

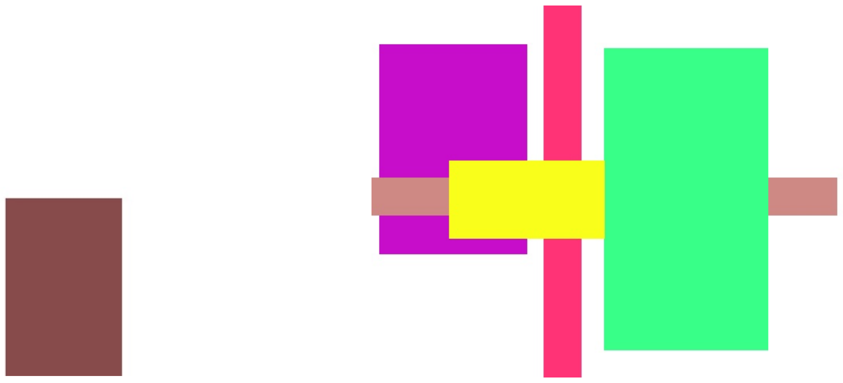
Figure 2 - Depth testing using GL_LESS (left) and using GL_GREATER (right)

opposite of their default function. In OpenGL's default implementation, the lower the number the closer it is considered to be to the viewer or camera. If the GZ library implemented the depth buffer in this way, only a single rectangle would be drawn given the input file for this assignment. See the left picture for how GL_LESS would work, and right picture for how GL_GREATER works in this application.

After creating the frame buffer, exporting the buffer to a bitmap is simple using the provided GzImage class. All that was required was looping through the entire frame buffer and setting the color for each pixel in the bitmap according to the values in the color buffer.

There were great similarities between the work that was done in creating the GZ library and the implementation of the OpenGL. Due to this, I found the OpenGL documentation to be incredibly valuable as a reference material for diving into this topic [1].

[1] OpenGL Documentation: https://github.com/LearnOpenGL-CN/LearnOpenGL-CN/blob/new-theme/docs/04%20Advanced%20OpenGL/01%20Depth%20testing.md