Assignment 1: Frame Buffer with Depth Buffer

Mai Trinh

COSC 6372 Computer Graphics*, University of Houston

February 23, 2021

1 Problem

The goal of this assignment is to build the frame buffer with depth buffer (z-buffer) for a small graphics library, *Gz* library. Given an input file containing the information of several rectangles including the coordinates of 2 corners, color and z-coordinate (depth value), the application should be able to draw the image. The pixel with larger depth value (z value) will be on top (i.e. shown in front and closer to the viewer). If the 2 pixels have the same depth value, the pixel drawn later will be on top. Several classes, functions, and data types are already given in the *Gz* library. We were to complete the implementation of the *GzFrameBuffer* class to render the output image with and without depth buffer.

2 Methods

Our goal is to rendered the image of the scene and save the image into BMP files. In order to do so, we need to store the image that is being rendered while it's being rendered by using the frame buffer. A frame buffer stores the final pixels that make up the rendered image. There are 2 different types of buffer: color buffer and depth buffer. A color buffer stores the color value (RGBA value) of each pixel. A depth buffer (or Z-buffer) stores depth value for each pixel.

The pipeline of rendering an image is first create the frame buffer and set all pixels to a default color. At render time, if multiple pixels overlap each other, then depends on its depth value, we store the color of the pixel closest to the viewer (larger z value). When all rectangles have been rasterized, the frame buffer will contain the output image. The Z-buffer method compares the depth value of each pixel. The algorithm for Z-buffer method is given below:

```
Algorithm:
    1. Initialize the depth of all pixels to a default depth value
    2. Initialize the color of all pixels to a default color (i.e. background color)
    3. For each pixel, do the following:
        a. Calculate the depth z at pixel (i,j). In this assignment, z value
        is already given in the input file.
        b. Compare depth value.
            If z > depth(i,j), this pixel is closer to the viewer than the
            value previously recorded for this pixel. Update the depth and color value
            corresponding to this pixel with the new value.
            Otherwise, no action is taken.
```

The space complexity is two times the number of pixels because we need two arrays: one for color buffer and the other for the depth buffer.

*Course assignments

3 Implementation

Assume that the task of reading input from a file and storing it in appropriate vectors are correctly implemented in the main application. The coordinates *(x,y,z)* of each pixel is stored in *vertexQueue*, and the color of each pixel is stored in *colorQueue* in *Gz* library. For each pixel, we pass the coordinate vertex and the color value to the frame buffer in order to render the output image.

To implement the frame buffer for our graphic library, the *GzFrameBuffer* class has the following structures:

```
class GzFrameBuffer {
public:
        void initFrameSize(GzInt width, GzInt height); //Initialize
        GzImage toImage(); //Convert the current rendering result to image
        void clear(GzFunctional buffer); //Clear buffers to preset values
        void setClearColor(const GzColor& color); //Set clear values for color buffer
        void setClearDepth(GzReal depth); //Set clear value for depth buffer
        void drawPoint(const GzVertex& v, const GzColor& c, GzFunctional status);

private:
        GzInt w, h;
        GzColor presetColor;
        GzReal presetDepth;
        vector<vector<GzColor>> colorBuffer;
        vector<vector<GzReal>> depthBuffer;
};
```

3.1 Initialize

First step of the application is to initialize the frame size. We set the size of image using *gz.initFrameSize(width,height)*. We must check the coordinate of each pixel against these boundaries value to remove any out-of-range pixel. Thus, any pixel with x-coordinate outside of the range [0,width] and y-coordinate outside of the range [0,height] will be discarded.

3.2 Buffers

First task is to draw the output image given the coordinates and the color of the rectangle. To do so, we create a 2D array color buffer to hold the color data of each pixel. The color buffer stores the default color (i.e. background color) given in the main application using *clearColor()* function.

3.2.1 Clearing buffers

We must clear the frame buffer (i.e the color buffer and the depth buffer if enabled) before each frame with the default value. To perform the clear operation, we pass in a list of all the buffers to be cleared. For example, using $gz.clear(GZ\_COLOR\_BUFFER \mid GZ\_DEPTH\_BUFFER)$ to clear the color buffer and the depth buffer. To do so, we first specify the default color (i.e. background color) and the default depth value. Then, we just set the color buffer with the default color value and set the depth buffer with default depth value.

3.3 Rendering image

Coloring the rectangles make solid surfaces, which is what we see in the output image and what will be rendered to the BMP file. To do so, we color each pixel within the frame, row-by-row. The pixels in the same rectangle have the same color value (specified in the input file). Any pixel that is not belong to any rectangle has the color value of the default color (i.e. the background color). We use the *set(GzInt x, GzInt y, const GzColor c)* function provided in *GzImage* class to set the color of each pixel. Once every pixel in the frame is set, we can save the image in a BMP file using the provided function in the *GzImage* class.

3.3.1 Depth buffer

The depth testing is enabled using *gz.enable(GZ_DEPTH_TEST)* from the main application. To implement a depth buffer, we create a 2D array with the same size as the color buffer array. In it we store the default depth value set by the main application using *clearDepth()* function. We also must clear the depth buffer before each frame to remove depth value from the previous frame. To do so, we use *gz.clear(GZ_DEPTH_BUFFER)*. Once enabled, the frame buffer tests the depth value of the pixel given in the input file (z-value) against the value in the depth buffer. Note that the requirement of the assignment is to draw the pixel with larger z value on top, so we discard all pixels that have the depth value less than or equal to the current depth buffer's value. If the depth test passes, the color of that pixel is updated in the color buffer (i.e. the pixel is rendered) and the depth buffer is updated with the new depth value. If the depth test fails, the pixel is discarded. A snippet of *drawPoint* function is shown below:

```
if (v[Z] > depthBuffer[v[X]][v[Y]]) { \\depth test passes
        depthBuffer[v[X]][v[Y]] = v[Z];\\update depth buffer with new value
        colorBuffer[v[X]][v[Y]] = c;\\update color buffer for rendering}
```

4 Results

The results of the implementation are shown below, where Figure 1 shows the rendered image without depth buffer and Figure 2 shows the image with depth buffer. We can see that the rectangle with color brown in Figure 1 is no longer rendered in Figure 2. The reason for this particular output is due to its z-value of -23, which is less than the default depth value (z = -20) of the depth buffer set in the main application, thus all the pixels in this rectangle were discarded when the depth test is enabled. Another observation can be made from these images is the changes in depth position of the yellow rectangle and the lighter brown rectangle. Z-value of the yellow rectangle (z = 26) is larger than the z-value of the other rectangle (z = 13), thus the yellow rectangle is shown in front of the other rectangle (i.e. closer to the viewer). Lastly, the case of two pixels having the same z-value is shown between the yellow rectangle and the green rectangle. Both figures have the same z-value of 26. However, the last input is the information of the green rectangle, hence it is drawn later and it appears on top.

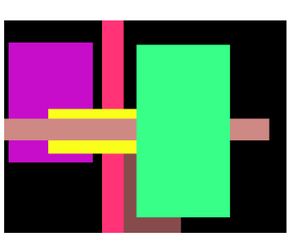

Figure 1: Image without depth buffer

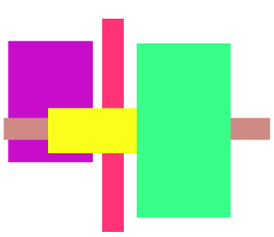

Figure 2: Image with depth buffer