Graphics Assignment No: 8

Shikhar Jaiswal
1601CS44

April 25, 2019

1 Assignment Description

Implement Z-Buffer algorithm for visible surface detection.

2 Procedure

Installation

Install the following packages from the Ubuntu repository:
- freeglut3-dev
- mesa-common-dev
- libglm-dev

```
sudo apt-get install freeglut3 freeglut3-dev mesa-common-dev
libglm-dev
```

Check your */usr/include/GL* folder to verify the installation of the OpenGL headers that you intend to use.

Compiling and Linking

We will have to use the $-lglut$ linker option with $gcc/g++$ to compile a program with *glut* library.

For example, to compile the program, use the following to get the binary executable code:

```
g++ z_buffer.cpp -lGL -lGLU -lglut -o z_buffer
```

3 Discussion

The primary objective of the assignment is to implement an algorithm capable of differentiating between visible and hidden surfaces in 3D space. For this purpose, Z-Buffer algorithm is implemented. For a detailed explanation, please refer the code.

OpenGL Code

```
#include <GL/glut.h>
#include <iostream>

#define max_vertices 8000
#define max_height 800
#define max_width 600

using namespace std;

// EdgeBucket structure created for storing edge information.
typedef struct EdgeBucket
{
        int ymax;
        float xofymin;
        float inv_slope;
} EdgeBucket;

typedef struct Background
{
        int color;
        float depth;
} Background;

// EdgeTableTuple structure for storing different edges for a
// scanline.
typedef struct edgetabletup
{
        int cnt_ed_bucks;
        EdgeBucket ed_buckets[max_vertices];
} EdgeTableTuple;

// EdgeTable stores EdgeTableTuple for various scanlines.
EdgeTableTuple EdgeTable[max_height], AEL;
Background background[max_height][max_width];

// Function for displaying the points.
void display()
```

```
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glBegin(GL_POINTS);

        for (int i = 0; i < max_height; i++)
        {
                for (int j = 0; j < max_width; j++)
                {
                        int color = background[i][j].color;
                        if (color == 0)
                        {
                                glColor3f(0.0f, 0.0f, 0.0f);
                        } else if (color == 1)
                        {
                                glColor3f(0.3f, 0.23f, 0.75f);
                        } else
                        {
                                glColor3f(1.0f, 0.8f, 0.0f);
                        }
                        glVertex2i(i, j);
                }
        }

        glEnd();
    glFlush();
}

// Points with depth 1.
// The depth for each point in the diagram is
// later computed using plane parameters.
int coordinate[3][5][3] = {{{20, 20, 1},
                        {200, 40, 2},
                        {300, 550, 3},
                        {80, 520, 1},
                        {20, 20, 1}},
                             {{10, 10, 2},
                        {100, 50, 4},
                        {120, 500, 0},
                        {10, 540, 2},
                        {10, 10, 2}}};

// For drawing two polygons.
int ind = 0;
int colors[2] = {2, 1};
float n[3], d;
```

```
// Function for computing the cross product.
void cross_product(int *a, int *b)
{
    n[0] = a[1] * b[2] - a[2] * b[1];
    n[1] = a[2] * b[0] - a[0] * b[2];
    n[2] = a[0] * b[1] - a[1] * b[0];
}

// Function for computing the dot product.
void dot_product(int *a)
{
    d = a[0] * n[0] + a[1] * n[1] + a[2] * n[2];
}

// Function for initializing the plane.
void initialise_plane()
{
        int ab[3], bc[3];
        for (int i = 0; i < 3; i++)
        {
                ab[i] = coordinate[ind][1][i] -
                coordinate[ind][0][i];
                bc[i] = coordinate[ind][2][i] -
                coordinate[ind][1][i];
        }
        cross_product(ab, bc);
        dot_product(coordinate[ind][0]);
}

// Function for initializing the edge table.
void initEdgeTable()
{
        for (int i = 0; i < max_height; i++)
        {
                EdgeTable[i].cnt_ed_bucks = 0;
        }

        AEL.cnt_ed_bucks = 0;
}

// Function that sorts edges according to their xofymin.
void sort(EdgeTableTuple *ett)
{
        EdgeBucket temp;
```

```
        for (int i = 1; i < ett->cnt_ed_bucks; i++)
        {
                temp.ymax = ett->ed_buckets[i].ymax;
                temp.xofymin = ett->ed_buckets[i].xofymin;
                temp.inv_slope = ett->ed_buckets[i].inv_slope;
                int j = i - 1;

                while ((temp.xofymin <
                ett->ed_buckets[j].xofymin) && (j >= 0))
                {
                        ett->ed_buckets[j + 1].ymax =
                        ett->ed_buckets[j].ymax;
                        ett->ed_buckets[j + 1].xofymin =
                        ett->ed_buckets[j].xofymin;
                        ett->ed_buckets[j + 1].inv_slope =
                        ett->ed_buckets[j].inv_slope;
                        j = j - 1;
                }
                ett->ed_buckets[j + 1].ymax = temp.ymax;
                ett->ed_buckets[j + 1].xofymin = temp.xofymin;
                ett->ed_buckets[j + 1].inv_slope =
                temp.inv_slope;
        }
}

// Function for storing the edge in the tuple.
void storeEdgeinTuple(EdgeTableTuple *rec, int y_max, int x_max,
float minv)
{
        (rec->ed_buckets[(rec)->cnt_ed_bucks]).ymax = y_max;
        (rec->ed_buckets[(rec)->cnt_ed_bucks]).xofymin =
        (float)x_max;
        (rec->ed_buckets[(rec)->cnt_ed_bucks]).inv_slope = minv;
        sort(rec);
        (rec->cnt_ed_bucks)++;
}

// Function for storing the edge in the table.
void storeEdgeInTable(int x1, int y1, int x2, int y2)
{
        float m, minv;
        int ymax1, xwithymin1, scanline;

        if (x2 == x1)
        {
                minv = 0.000000;
```

```
        } else
        {
                m = ((float)(y2 - y1)) / ((float)(x2 - x1));

                if (y2 == y1)
                {
                        return;
                }

                minv = (float)1.0 / m;
        }

        if (y1 > y2)
        {
                scanline = y2;
                ymax1 = y1;
                xwithymin1 = x2;
        } else
        {
                scanline = y1;
                ymax1 = y2;
                xwithymin1 = x1;
        }

        storeEdgeinTuple(&EdgeTable[scanline], ymax1, xwithymin1,
        minv);
}

// Function for removing an edge.
void removeEdgeByYmax(EdgeTableTuple *Tup, int yy)
{
        for (int i = 0; i < Tup->cnt_ed_bucks; i++)
        {
                if (Tup->ed_buckets[i].ymax == yy)
                {
                        for (int j = i; j < Tup->cnt_ed_bucks
                        - 1; j++)
                        {
                                Tup->ed_buckets[j].ymax =
                                        Tup->ed_buckets[j + 1].
                                        ymax;
                                Tup->ed_buckets[j].xofymin =
                                        Tup->ed_buckets[j + 1].
                                        xofymin;
                                Tup->ed_buckets[j].inv_slope =
                                        Tup->ed_buckets[j + 1].
```

```
                                        inv_slope;
                        }

                        Tup->cnt_ed_bucks--;
                        i--;
                }
        }
}

// Function for updating the value of x.
void update_x(EdgeTableTuple *Tup)
{
        for (int i = 0; i < Tup->cnt_ed_bucks; i++)
        {
                (Tup->ed_buckets[i]).xofymin =
                        (Tup->ed_buckets[i]).xofymin +
                        (Tup->ed_buckets[i]).inv_slope;
        }
}

// Function for filling the scanlines.
void scanline_fill()
{
        int x1, ymax1, x2, ymax2, FillFlag = 0, coordCount;

        for (int i = 0; i < max_height; i++)
        {
                for (int j = 0; j < EdgeTable[i].cnt_ed_bucks;
                j++)
                {
                        storeEdgeinTuple(&AEL,
                        EdgeTable[i].ed_buckets[j].ymax,
                        EdgeTable[i].ed_buckets[j].xofymin,
                        EdgeTable[i].ed_buckets[j].inv_slope);
                }

                removeEdgeByYmax(&AEL, i);
                sort(&AEL);

                // Fill lines between different edges in order.
                int j = 0;
                FillFlag = 0;
                coordCount = 0;
                x1 = 0;
                x2 = 0;
                ymax1 = 0;
```

```
ymax2 = 0;

while (j < AEL.cnt_ed_bucks)
{
        if (coordCount % 2 == 0)
        {
                x1 =
                (int)(AEL.ed_buckets[j].xofymin);
                ymax1 = AEL.ed_buckets[j].ymax;
                if (x1 == x2)
                {
                        if (((x1 == ymax1) &&
                        (x2 != ymax2)) ||
                                ((x1 != ymax1) &&
                                (x2 == ymax2)))
                        {
                                x2 = x1;
                                ymax2 = ymax1;
                        } else
                        {
                                coordCount++;
                        }
                } else
                {
                        coordCount++;
                }
        } else
        {
                x2 = (int)AEL.ed_buckets[j].
                xofymin;
                ymax2 = AEL.ed_buckets[j].ymax;
                FillFlag = 0;

                if (x1 == x2)
                {
                        if (((x1 == ymax1) &&
                        (x2 != ymax2)) ||
                                ((x1 != ymax1) &&
                                (x2 == ymax2)))
                        {
                                x1 = x2;
                                ymax1 = ymax2;
                        } else
                        {
                                coordCount++;
                                FillFlag = 1;
```

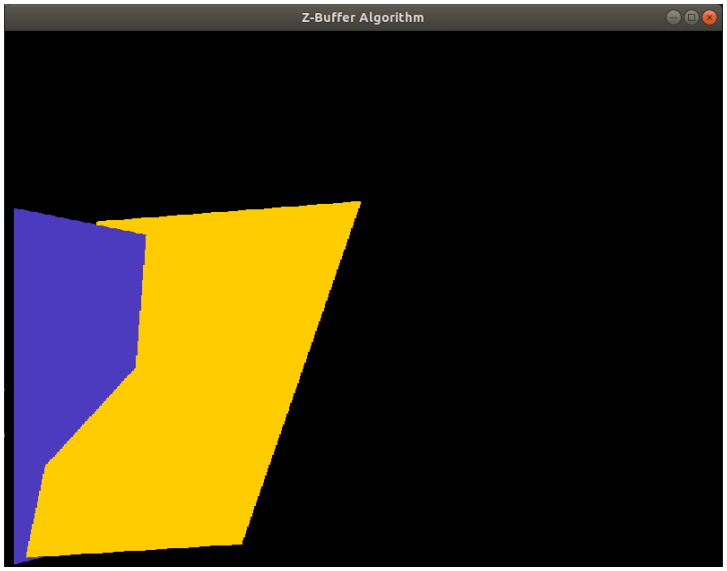
```
                                        }
                                } else
                                {
                                        coordCount++;
                                        FillFlag = 1;
                                }

                                if (FillFlag)
                                {
                                        if (x2 < x1)
                                        {
                                                int temp = x2;
                                                x2 = x1;
                                                x1 = temp;
                                        }

                                        float di;

                                        for (int j = x1; j <= x2;
                                        j++)
                                        {
                                                di = (d - n[0]
                                                * j - n[1] * i)
                                                / n[2];
                                                if
                                                (background[j][i]
                                                .depth > di)
                                                {
                                        background[j][i].depth =
                                        di;
                                        background[j][i].color =
                                        colors[ind];
                                                }
                                        }
                                }
                        }
                        j++;
                }

                update_x(&AEL);
        }
}

// Function for drawing the polygons from the edges.
void drawPolygon()
{
```

```
        int count = 0, x1, y1, x2, y2;
        while (count < 5)
        {
                if (count == 0)
                {
                        x2 = coordinate[ind][count][0];
                        y2 = coordinate[ind][count][1];
                } else
                {
                        x1 = x2;
                        y1 = y2;
                        x2 = coordinate[ind][count][0];
                        y2 = coordinate[ind][count][1];
                        storeEdgeInTable(x1, y1, x2, y2);
                }
                count++;
        }
}

int main(int argc, char** argv)
{
    for (int i = 0; i < max_height; i++)
    {
                for (int j = 0; j < max_width; j++)
                {
                        background[i][j].color = 0;
                        background[i][j].depth = 1000;
                }
        }

        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(max_height, max_width);
        glutInitWindowPosition(0, 0);
        glutCreateWindow("Z-Buffer Algorithm");
    glClearColor(0, 0, 0, 0);
        glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, max_width, 0.0, max_height);
    glPointSize(6.0);

    for (ind = 0; ind < 2; ind++)
    {
                // Initializing the plane.
                initialise_plane();
                // Initializing the edge table.
```

```
                initEdgeTable();
                // Drawing the Polygon edges.
                drawPolygon();
                // Filling the polygon through scanline_fill.
                scanline_fill();
    }
        glutDisplayFunc(display);
        glutMainLoop();

        return 0;
}
```

4 Result

OpenGL



5 References

[1] How to install OpenGL/GLUT libraries

[2] An Introduction to OpenGL Programming

[3] OpenGL Mathematics