Computer Graphics HW1 – Draw Line Based on OpenGL

1552746 Cui Hejie

April 30, 2018

Abstract

This assignment's object is to realize the drawing line algorithms in 3D space, using cube to replace pixel. Here I implemented two drawing line algorithms, DDA and Bresenham.

1 Introduction

1.1 Project Structure

- main.cpp
- glad.c
- headers (a number of header files)
- shaders (vertex shader and fragment shader)
- resources (textures)
- * Environment : Xcode 9.3, Glew2.1.0, Glfw 3.2.1, Glad, GLM

1.2 Running Operations

1. Run the main.cpp
2. Type in the coordinates of the start points and end points.
3. Choose a draw line algorithm(1 for DDA, 2 for Bresenham).
4. The 3D line will show immediately.

2 Algorithm Description

2.1 3D DDA Algorithm[4]

Digital Differential Analyzer (DDA) algorithm is the simple line generation algorithm which is explained step by step here.

- **Step1** - Get the input of two end points start and end.

- **Step2** - Calculate the difference between two points.

```
  dx = end.x-start.x;
  dy = end.y-start.y;
  dz = end.z-start.z;
```

- **Step3** - Based on the calculated difference in step-2, identify the number of steps to put pixel. If dx > dy, then we need more steps in x coordinate; otherwise in y coordinate, dz is similar.

```
int step = abs(dx);
 if(abs(dy) > step)
 {
     step = abs(dy);
 }
 if(abs(dz) > step)
 {
     step = abs(dz);
 }
```

- **Step4** - Calculate the increment in x, y, and z coordinate.

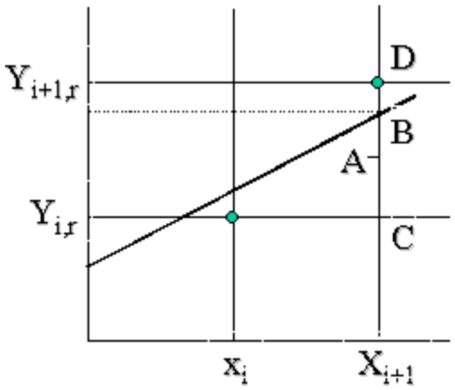
```
xincre = (float)(end.x - start.x)/step;
yincre = (float)(end.y - start.y)/step;
zincre = (float)(end.z - start.z)/step;
```

- **Step5** - Put the pixel by successfully incrementing x, y and z coordinates accordingly and store the coordinates in an array cube Position. The cube number here counts how many points are used to draw lines.

```
x = start.x;
y = start.y;
z = start.z;
glm::vec3 temp;
 for (int i = 1; i < step; i++)
 {
     temp = glm::vec3(x,  y,  z);
     cubePositions[cube_num] = temp;
     x = x + xincre;
     y = y + yincre;
     z = z + zincre;
     cube_num++;
 }
```

2.2 3D Bresenham Algorithm[5]

The Bresenham algorithm is another incremental scan conversion algorithm. The big advantage of this algorithm is that, it uses only integer calculations. Moving across the x axis in unit intervals and at each step choose between two different y coordinates. The basic idea of Bresenham algorithm in 2D is as below:



Assume that $(x_i, y_{i,r})$ has been used on column x_i in the figure as a point on the line, and point B is on the straight line, the coordinates of it is(x_{i+1}, y_{i+1}). It is clearly that the next point of straight line $(x_{i+1}, y_{i+1,r})$ can only be chosen from C or D. Let's say A is the midpoint of the CD. If B is above point A, we should take D points as $(x_{i+1}, y_{i+1,r})$, otherwise C points should be taken.

In order to determine whether B is above or below A, let

$$\epsilon(x_{i+1}) = y_{i+1} - y_{i,r} - 0.5(2) \ ,$$

if B is below A, then $\epsilon(x_{i+1}) < 0$, else $\epsilon(x_{i+1}) > 0$.

From the picture we know that

$$y_{i+1,r} = y_{i,r} + 1, if \epsilon(x_{i+1}) >= 0(3)$$
$$y_{i+1,r} = y_{i,r}, if \epsilon(x_{i+1}) <= 0(3)$$

From (2) and (3) we know that,

$$\epsilon(x_{i+2}) = y_{i+2} - y_{i+1}, r - 0.5 = y_{i+1} + k - y_{i+1,r} - 0.5;$$

Since,

$$y_{i+1} - y_{i,r} - 0.5 + k - 1, if \epsilon(x_{i+1}) \geqslant 0$$
$$y_{i+1} - y_{i,r} - 0.5 + k, if \epsilon(x_{i+1}) \leqslant 0$$

Then,

$$\epsilon(x_{i+2}) = \epsilon(x_{i+1}) + k - 1, if \epsilon(x_{i+1}) \geqslant 0$$
$$\epsilon(x_{i+2}) = \epsilon(x_{i+1}) + k, if \epsilon(x_{i+1}) \leqslant 0$$

From (2) we know that the initial value is $\epsilon(x_2) = y_2 - y_r - 0.5 = d - 0.5 = -0.5$

The above Bresenham algorithm has two disadvantages: Division is used to calculate the slope, and decimal is used to calculate the error term.

The solution is as follows: $E' = 2 * E * DX$, because the algorithm only uses the symbol of the error term to judge, so, as the substitution does not change the nature of the algorithm.

Extend the above idea to 3D situation, the whole process is like this:

- **Step1** - Get the input of two end points start and end.

- **Step2** - Calculate the difference between two points.

```
    dx = end.x-start.x;
    dy = end.y-start.y;
    dz = end.z-start.z;
```

- **Step3**- Initial e1, e2, x, y, z

```
     e1 = e2 = (-dx);

      x = start.x;
      y = start.y;
      z = start.z;
```

- **Step4** - Based on the algorithm's idea talked above, put the pixel by successfully incrementing x, y and z coordinates accordingly and store the coordinates in an array cube Position. The cube number here counts how many points are used to draw lines.

```
     glm::vec3 temp;
      for (x = start.x; x < end.x; x++)
      {
          temp = glm::vec3(x, y, z);
          cubePositions[cube_num] = temp;
```

```
        e1 = e1 + 2*dy;
        e2 = e2 + 2*dz;
        if(e1 > 0)
        {
            y = y + 1;
            e1 = e1 - 2*dx;
        }
        if(e2 > 0)
        {
            z = z + 1;
            e2 = e2 - 2*dx;
        }
        cube_num++;
    }
```

3 Code and Annotations

The code is implemented in C++, under the OpenGL. It is divided in three parts: Main function, Vertex Shader, and Fragment Shader.

3.1 Main.cpp[2]

```
#include <glad/glad.h>
#include <GLFW/glfw3.h>

#include "headers/stb_image.h"
#include "headers/shader_m.h"

#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

#include <iostream>

void framebuffer_size_callback(GLFWwindow* window, int width, int height);
void processInput(GLFWwindow *window);

// two draw line algorithms
void dda_draw_line(glm::vec3 start, glm::vec3 end);
void bresenham_draw_line(glm::vec3 start, glm::vec3 end);

// settings
const unsigned int SCR_WIDTH = 800;
const unsigned int SCR_HEIGHT = 600;
// init cube number
unsigned int cube_num = 0;

//the line's start point and end point
glm::vec3 start = glm::vec3( -10.0f,  -5.0f,  -5.0f);
glm::vec3 end = glm::vec3(8.0f,  5.0f, 3.0f);

// init a vec3 array to store cubePositions
glm::vec3 cubePositions[10000];

int main()
{
    // glfw: initialize and configure
    // ------------------------------
    glfwInit();
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    // uncomment this statement to fix compilation on OS X
#ifdef __APPLE__
    glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
#endif

    // glfw window creation
    // --------------------
    GLFWwindow* window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT,
    "3DDrawLine", NULL, NULL);
    if (window == NULL)
    {
        std::cout << "Failed␣to␣create␣GLFW␣window" << std::endl;
        glfwTerminate();
        return -1;
    }
    glfwMakeContextCurrent(window);
    glfwSetFramebufferSizeCallback(window, framebuffer_size_callback);

    // glad: load all OpenGL function pointers
    // ---------------------------------------
    if (!gladLoadGLLoader((GLADloadproc)glfwGetProcAddress))
    {
        std::cout << "Failed␣to␣initialize␣GLAD" << std::endl;
        return -1;
    }

    // configure global opengl state
    // -----------------------------
    glEnable(GL_DEPTH_TEST);

    // build and compile our shader zprogram
    // ------------------------------------
    Shader ourShader("shaders/coordinate_system_vec.glsl",
    "shaders/coordinate_system_frag.glsl");


    // set up vertex data (and buffer(s)) and configure vertex attributes
    // Vertex sets, draw two triangles to form a rectangle
    // (OpenGL mainly deals with triangles), six surfaces form a cube.
    // ------------------------------------------------------------------
    float vertices[] = {
        // The first triangles
        -0.5f, -0.5f, -0.5f,  0.0f, 0.0f, // Top right corner
        0.5f, -0.5f, -0.5f,  1.0f, 0.0f, // Bottom right corner
        0.5f,  0.5f, -0.5f,  1.0f, 1.0f, // Top left corner
        // The second triangles
        0.5f,  0.5f, -0.5f,  1.0f, 1.0f, // Bottom right corner
        -0.5f,  0.5f, -0.5f,  0.0f, 1.0f, // Bottom left corner
        -0.5f, -0.5f, -0.5f,  0.0f, 0.0f, // Top left corner

        -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
        0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
        0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
        0.5f,  0.5f,  0.5f,  1.0f, 1.0f,
        -0.5f,  0.5f,  0.5f,  0.0f, 1.0f,
        -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,

        -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
        -0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
        -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
        -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
        -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
        -0.5f,  0.5f,  0.5f,  1.0f, 0.0f,

        0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
        0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
        0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
        0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
        0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
        0.5f,  0.5f,  0.5f,  1.0f, 0.0f,

        -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,
        0.5f, -0.5f, -0.5f,  1.0f, 1.0f,
        0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
        0.5f, -0.5f,  0.5f,  1.0f, 0.0f,
        -0.5f, -0.5f,  0.5f,  0.0f, 0.0f,
        -0.5f, -0.5f, -0.5f,  0.0f, 1.0f,

        -0.5f,  0.5f, -0.5f,  0.0f, 1.0f,
        0.5f,  0.5f, -0.5f,  1.0f, 1.0f,
        0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
        0.5f,  0.5f,  0.5f,  1.0f, 0.0f,
        -0.5f,  0.5f,  0.5f,  0.0f, 0.0f,
        -0.5f,  0.5f, -0.5f,  0.0f, 1.0f
    };

    unsigned int algorithm_type;
    float a, b, c;

    // User sets the start point coordinates
    std::cout << "Please␣type␣in␣the␣coordinates␣of␣the␣start␣points
␣␣␣␣(three␣floats␣with␣blank␣space␣spliting␣them):␣x␣y␣z" << std::endl;
    std::cin >> a >> b >> c;
    glm::vec3 start = glm::vec3(a,  b,  c);

    // User sets the end point coordinate
    std::cout << "Please␣type␣in␣the␣coordinates␣of␣the␣end␣points
␣␣␣␣(three␣floats␣with␣blank␣space␣spliting␣them):␣x␣y␣z"<< std::endl;
    std::cin >> a >> b >> c;
    glm::vec3 end = glm::vec3(a,  b, c);

    // User choose draw line algorithm
    std::cout << "Please␣type␣in␣a␣number␣to␣choose␣draw␣line␣algorithm:
␣␣␣␣1(DDA),␣2(Bresenham)"<< std::endl;
    std::cin >> algorithm_type;
    if(algorithm_type==1)
    {
        dda_draw_line(start, end);
    }else
    {
        bresenham_draw_line(start, end);
    }

    unsigned int VBO, VAO;
    // Generate a VAO object
    glGenVertexArrays(1, &VAO);
    // Generate a VBO object
    glGenBuffers(1, &VBO);

    glBindVertexArray(VAO);//Bind VAO

    // Copy the vertex array to buffer memory for OpenGL usage
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices),
    vertices, GL_STATIC_DRAW);

    // Link vertex properties, using the glVertexAttribPointer function to
    // tell OpenGL how to parse vertex data (applied to one vertex attribute)
    // position attribute
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 5 * sizeof(float), (void*)0);
    // Use vertex positions attribute value as a parameter
    // enable the vertex attribute;
    glEnableVertexAttribArray(0);

    // texture coord attribute
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE,
    5 * sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    // load and create a texture
    // -------------------------
    unsigned int texture1, texture2;
    // texture 1
    // ---------
    glGenTextures(1, &texture1);
    glBindTexture(GL_TEXTURE_2D, texture1);
    // set the texture wrapping parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // set texture filtering parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // load image, create texture and generate mipmaps
    int width, height, nrChannels;
    // stbi_set_flip_vertically_on_load(true);
    // tell stb_image.h to flip loaded texture's on the y-axis.
    unsigned char *data = stbi_load("resources/textures/container.jpg",
    &width, &height, &nrChannels, 0);
    if (data)
    {
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,
        0, GL_RGB, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else
    {
        std::cout << "Failed␣to␣load␣texture" << std::endl;
    }
    stbi_image_free(data);
    // texture 2
    // ---------
    glGenTextures(1, &texture2);
    glBindTexture(GL_TEXTURE_2D, texture2);
    // set the texture wrapping parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    // set texture filtering parameters
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    // load image, create texture and generate mipmaps
    data = stbi_load("resources/textures/awesomeface.png",
    &width, &height, &nrChannels, 0);
    if (data)
    {
        // note that the awesomeface.png has transparency
        // and thus an alpha channel, so make sure to tell OpenGL
        // the data type is of GL_RGBA
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,
        0, GL_RGBA, GL_UNSIGNED_BYTE, data);
        glGenerateMipmap(GL_TEXTURE_2D);
    }
    else
    {
        std::cout << "Failed␣to␣load␣texture" << std::endl;
    }
    stbi_image_free(data);

    // tell opengl for each sampler to which texture unit it belongs to
    // (only has to be done once)
    // ---------------------------------------------------
    ourShader.use();
    ourShader.setInt("texture1", 0);
    ourShader.setInt("texture2", 1);

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

        // render
        // ------
        glClearColor(0.2f, 0.3f, 0.3f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        // bind textures on corresponding texture units
        glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, texture1);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, texture2);

        // activate shader
        ourShader.use();

        // create transformations
        glm::mat4 view;
        glm::mat4 projection;
        projection = glm::perspective(glm::radians(45.0f),
        (float)SCR_WIDTH / (float)SCR_HEIGHT, 0.1f, 100.0f);
        view       = glm::translate(view, glm::vec3(0.0f, 0.0f, -20.0f));
        // pass transformation matrices to the shader
        ourShader.setMat4("projection", projection);
        // note: currently we set the projection matrix each frame,
        // but since the projection matrix rarely changes
        // it's often best practice to set it outside the main loop only once.
        ourShader.setMat4("view", view);

        // render boxes
        glBindVertexArray(VAO);
        for (unsigned int i = 0; i < cube_num; i++)
        {
            // calculate the model matrix for each object and
            // pass it to shader before drawing
            glm::mat4 model;
            //The space position of the cube.
            model = glm::translate(model, cubePositions[i]);
            // float angle = 20.0f * i;
            // model = glm::rotate(model, glm::radians(angle),
            // glm::vec3(1.0f, 0.3f, 0.5f));
            ourShader.setMat4("model", model);

            glDrawArrays(GL_TRIANGLES, 0, 36);
        }

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

    // optional: de-allocate all resources once they've outlived their purpose:
    // --------------------------------------------------------
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);

    // glfw: terminate, clearing all previously allocated GLFW resources.
    // -----------------------------------------------------
    glfwTerminate();
    return 0;
}

// process all input: query GLFW whether relevant keys are pressed/released
// this frame and react accordingly
// -----------------------------------------------
void processInput(GLFWwindow *window)
{
    if (glfwGetKey(window, GLFW_KEY_ESCAPE) == GLFW_PRESS)
        glfwSetWindowShouldClose(window, true);
}

```

```
// glfw: whenever the window size changed (by OS or user resize)
// this callback function executes
// -----------------------------------------------
void framebuffer_size_callback(GLFWwindow* window, int width, int height)
{
    // make sure the viewport matches the new window dimensions;
    // note that width and height will be significantly larger than
    // specified on retina displays.
    glViewport(0, 0, width, height);
}

// 3D DDA draw line algorithm
void dda_draw_line(glm::vec3 start, glm::vec3 end)
{
    float x, y, z, xincre, yincre, zincre;
    int k = abs(end.x-start.x);
    if(abs(end.y-start.y) > k)
    {
        k = abs(end.y-start.y);
    }
    if(abs(end.z-start.z) > k)
    {
        k = abs(end.z-start.z);
    }

    xincre = (float)(end.x - start.x)/k;
    yincre = (float)(end.y - start.y)/k;
    zincre = (float)(end.z - start.z)/k;

    x = start.x;
    y = start.y;
    z = start.z;

    glm::vec3 temp;
    for (int i = 1; i < k; i++)
    {
        temp = glm::vec3(x,  y,  z);
        cubePositions[cube_num] = temp;
        x = x + xincre;
        y = y + yincre;
        z = z + zincre;
        cube_num++;
    }
}

// 3D Bresenham draw line algorithm
void bresenham_draw_line(glm::vec3 start, glm::vec3 end)
{
    int x, y, z, dx, dy, dz;
    float k1, k2, e1, e2;

    dx = end.x - start.x;
    dy = end.y - start.y;
    dz = end.z - start.z;

    e1 = e2 = (-dx);

    x = start.x;
    y = start.y;
    z = start.z;

    glm::vec3 temp;
    for (x = start.x; x < end.x; x++)
    {
        temp = glm::vec3(x, y, z);
        cubePositions[cube_num] = temp;

        e1 = e1 + 2*dy;
        e2 = e2 + 2*dz;

        if(e1 > 0)
        {
            y = y + 1;
            e1 = e1 - 2*dx;
        }
        if(e2 > 0)
        {
            z = z + 1;
            e2 = e2 - 2*dx;
        }
        cube_num++;
    }
}
```

3.2 Vertex Shader[3]

```
#version 330 core
// The attribute position value of the location variable is 0.
layout (location = 0) in vec3 aPos;
// The property position value of the material variable is 1.
layout (location = 1) in vec2 aTexCoord;

out vec2 TexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    gl_Position = projection * view * model * vec4(aPos, 1.0f);
    TexCoord = vec2(aTexCoord.x, 1.0 - aTexCoord.y);
}
```

3.3 Fragment Shader[3]

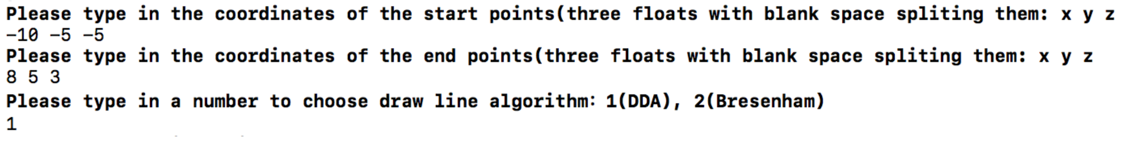
```
#version 330 core
// Specifies a color output for the fragment shader
out vec4 FragColor;
// Input variable from vertex shader (same name, same type)
in vec2 TexCoord;

uniform sampler2D texture1;
uniform sampler2D texture2;

void main()
{
    FragColor = mix(texture(texture1, TexCoord),
    texture(texture2, TexCoord), 0.2);
}
```

4 Result Samples

4.1 DDA Algorithm



```
Please type in the coordinates of the start points(three floats with blank space spliting them: x y z
-10 -5 -5
Please type in the coordinates of the end points(three floats with blank space spliting them: x y z
8 5 3
Please type in a number to choose draw line algorithm: 1(DDA), 2(Bresenham)
1
```

Figure 1: Parameter Settings

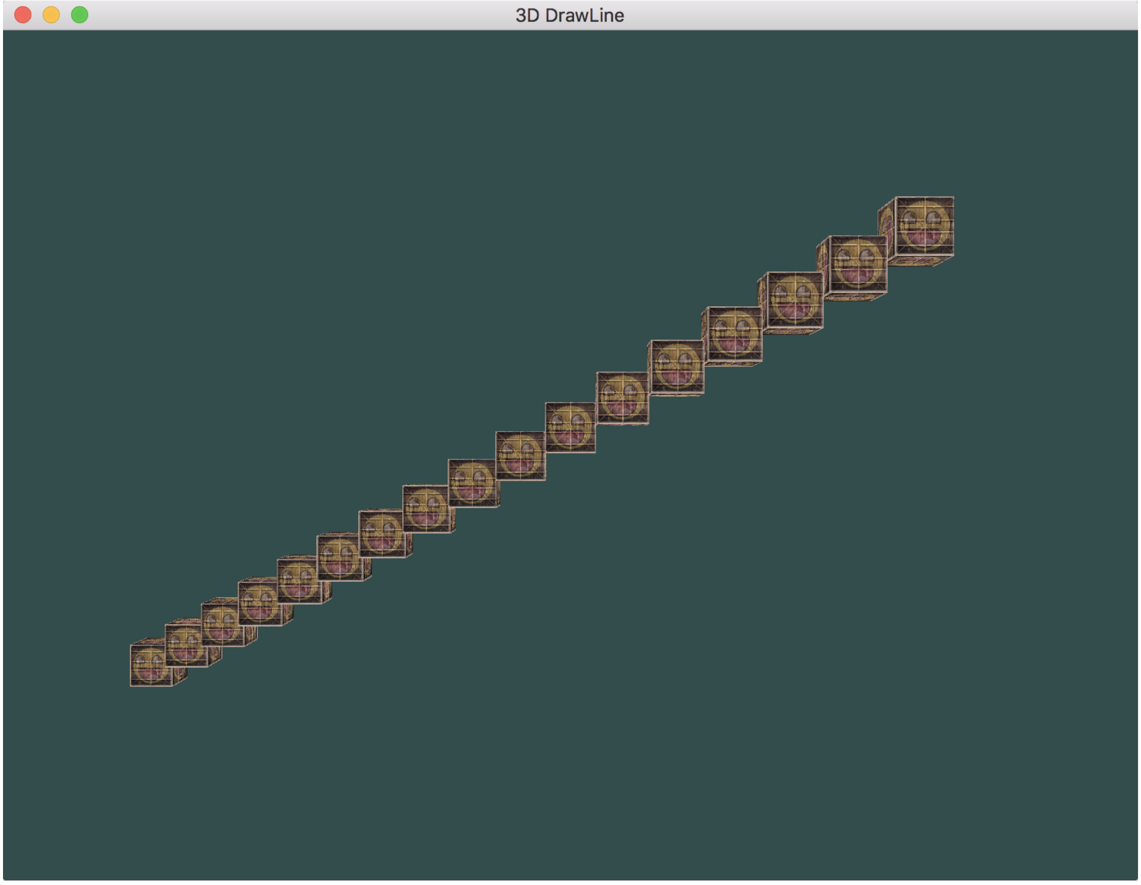
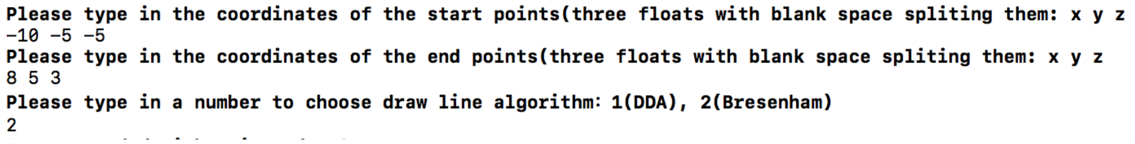


Figure 2: 3D line by DDA Algorithm

4.2 Bresenham Algorithm



```
Please type in the coordinates of the start points(three floats with blank space spliting them: x y z
-10 -5 -5
Please type in the coordinates of the end points(three floats with blank space spliting them: x y z
8 5 3
Please type in a number to choose draw line algorithm: 1(DDA), 2(Bresenham)
2
```

Figure 3: Parameter Settings

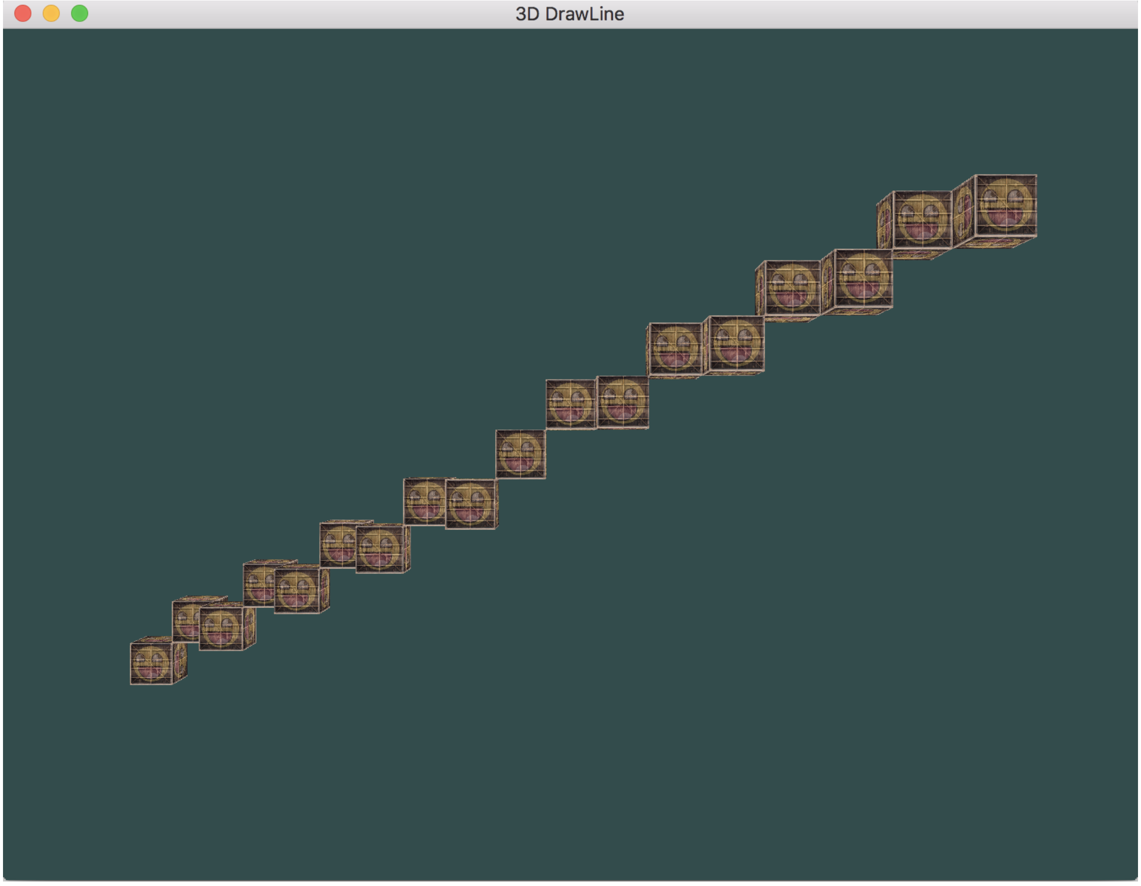


Figure 4: 3D line by Bresenham Algorithm

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

[2] OpenGL: LearnOpenGL CN,
`https://learnopengl-cn.github.io`

[3] OpenGL Code,
`https://github.com/JoeyDeVries/LearnOpenGL`

[4] DDA: 2D DDA Algorithm,
`https://blog.csdn.net/mni2005/article/details/8772803`

[5] Bresenham: 2D Bresenham Algorithm,
`https://www.cs.helsinki.fi/group/goa/mallinnus/lines/bresenh.html`