计算机图形学实验报告

学号：16340054

姓名：戴馨乐

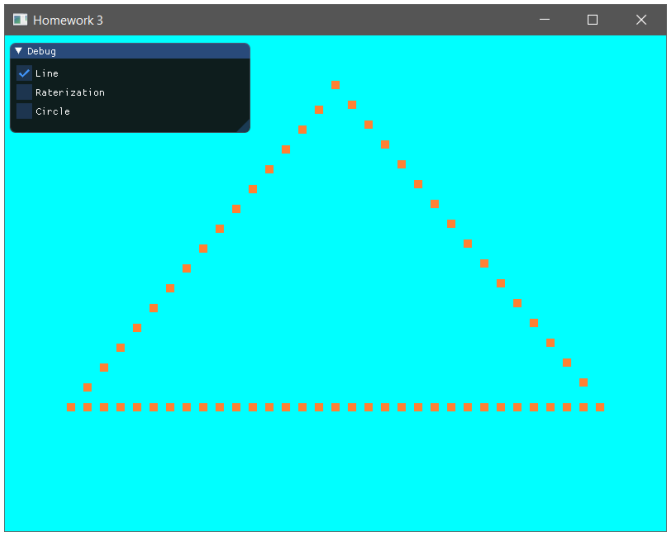
学院：数据科学与计算机学院

作业：Homework3

Basic:

1. 使用 Bresenham 算法画一个三角形边框

   a) 运行截图:



   b) 核心代码:

   将 3 个点两两组合，通过 Bresenham 算法来分别画出 3 条直线

```
void generateVertices(Point p1, Point p2, Point p3) {
    vector<Point> v;
    bresenham(p1, p2, v);
    bresenham(p1, p3, v);
    bresenham(p2, p3, v);
    Point head[3] = { p1, p2, p3 };
    if (need_raterization) triangle_resterization(head, v);
    s = v.size() * 3 * 4;
    if (vertices) delete vertices;
    vertices = new float[s/4];
    int index = 0;
    for (int i = 0; i < v.size(); i++, index+=3) {
        vertices[index] = v[i].x/float(width/2);
        vertices[index + 1] = v[i].y/float(height/2);
        vertices[index + 2] = 0.0f;
    }
}
```

Bresenham 画直线（部分）

```
// 初始化Δx, Δy
float delta_x, delta_y;
delta_x = e.x - s.x;
delta_y = e.y - s.y;
int x_step = 20, y_step = x_step*abs(delta_y/delta_x);
float t, p0, x, y, p;
if (delta_x > 0 && delta_y > 0) {
    t = 2 * delta_y - 2 * delta_x;
    p0 = 2 * delta_y - delta_x;
    x = s.x;  y = s.y;  p = p0;
    vertices.push_back(s);
    while (x + x_step < e.x) {
        if (p <= 0) {
            Point point(x + x_step, y);
            vertices.push_back(point);
            p = p + 2 * delta_y;
            x = x + x_step;
        }
        else {
            Point point(x + x_step, y + y_step);
            vertices.push_back(point);
            p = p + t;
            x = x + x_step;
            y = y + y_step;
        }
    }
    vertices.push_back(e);
}
```

c) 实现思路：

Bresenham 算法描述：

对于一条从(x_0, y_0)到(x_1, y_1)的直线，初始化如下变量：

$$\Delta x = x_1 - x_0, \Delta y = y_1 - y_0$$

$$p_0 = 2 \times \Delta y - \Delta x$$

定义一个 x 变化的步长x_{step}和一个 y 变化的步长y_{step}，那么：

初始化 x 为x_0，y 为y_0，p 为p_0：

不断进行如下过程：

若$p \leq 0$：

选取的下一个点为$(x + x_{step}, y)$

更新 x 为$x + x_{step}$

更新 p 为$p + \Delta y$

若$p > 0$：

选取的下一个点为$(x + x_{step}, y + y_{step})$

更新 x 为$x + x_{step}$

更新 y 为$y + y_{step}$

更新 p 为$p + 2 \times \Delta y - 2 \times \Delta x$

直到$x \geq x_1$

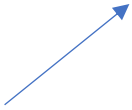
实现思路：

可以看到，上面的算法仅仅考虑了斜率为 0 到45°的直线，对于其他直线，后来我查到可以通过直线变换来得到。但是当时实现的时候没有这样做，而是做了一个分类讨论。

首先确认起点和终点。规则是：选择 x 坐标小的为起点，大的为终点；若两点的 x 坐标一样，则选择 y 坐标小的为起点，大的为终点，假如一样，则这两点是同一点，不需要画直线。

然后将直线根据Δx和Δy来区分出 4 类直线：

i. 斜角为0到45°



ii. 斜角为45°到180°

iii. 斜角为0

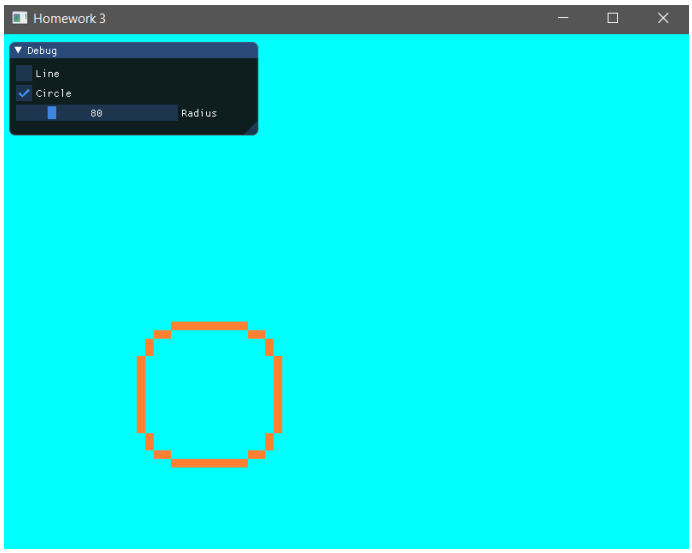
iv. 斜率不存在（垂直）

其中，第二种直线需要重新推导一下，稍微有些不一样，但是还是类似的。

可以看到代码是蛮累赘的，假如通过直线变换将会简洁很多。

2. 使用 Bresenham 算法画一个圆

a) 运行截图：



b) 核心代码：

```
void bresenham_circle(Point center, int radius) {
    vector<Point> v;
    float d = 3 - 2 * radius;
    int x = 0, y = radius;
    while (x <= y) {
        v.push_back(Point(x, y));
        v.push_back(Point(y, x));
        v.push_back(Point(-y, x));
        v.push_back(Point(-x, y));
        v.push_back(Point(-x, -y));
        v.push_back(Point(-y, -x));
        v.push_back(Point(y, -x));
        v.push_back(Point(x, -y));
        if (d < 0) {
            d += 4 * x + 6;
        }
        else {
            d += 4 * (x - y) + 10;
            y-=10;
        }
        x+=10;
    }
    for (int i = 0; i < v.size(); i++) {
        v[i].x += center.x;
        v[i].y += center.y;
    }
    s = v.size() * 3 * 4;
    if (vertices) delete vertices;
    vertices = new float[s / 4];
    int index = 0;
    for (int i = 0; i < v.size(); i++, index+=3) {
        vertices[index] = v[i].x / float(width/2);
        vertices[index + 1] = v[i].y / float(height/2);
        vertices[index + 2] = 0.0f;
    }
}
```

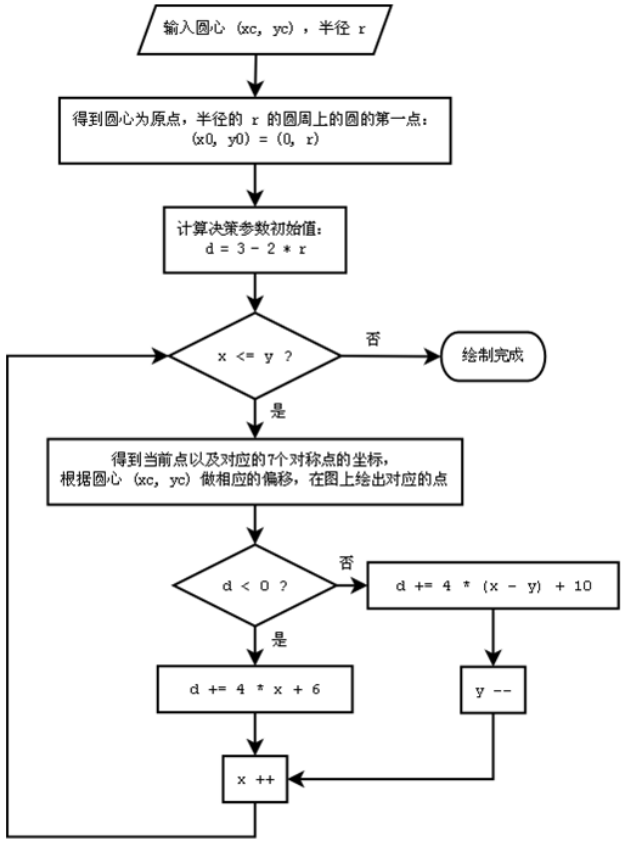
c) 实现思路：

算法描述：

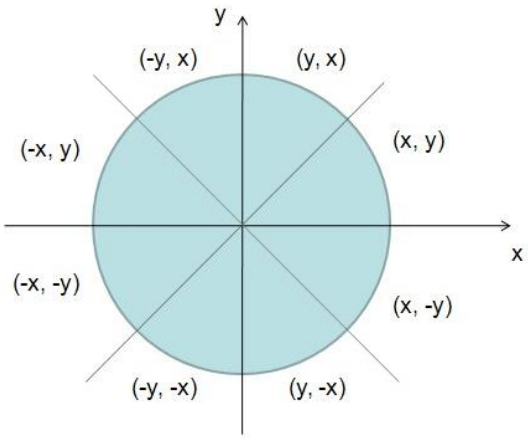
Bresenham 算法画圆的算法可以通过下面这张流程图可以很清楚的展示出来：

假设：

圆心为(x_c, y_c)，半径为r



其中 7 个相对于圆心的对称点是如下 7 个:



为什么需要对称点？这是因为圆的对称性，所以我们只需要计算 $\frac{1}{8}$ 的圆周，其余的部分都可以通过对称得到。

实现思路：

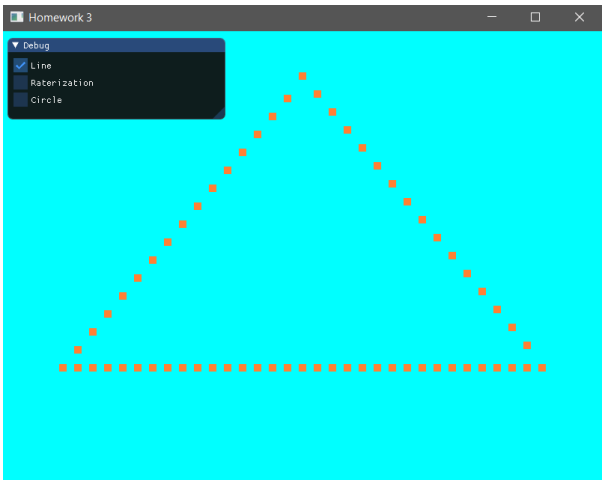
首先，为了简化对称点的计算，我们可以假设圆心为$(0,0)$，按照算法流程图来求出圆周上的所有点

然后，将坐标系的原点$(0,0)$移动到(x_c, y_c)。将每一个点的 x 坐标加上x_c，y 坐标加上y_c

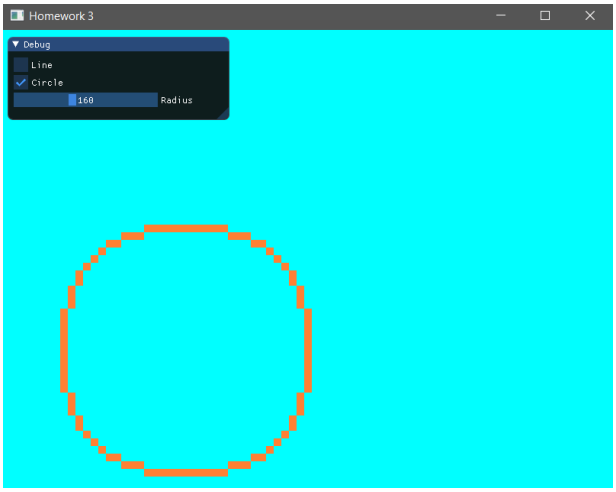
3. 添加菜单栏，可以选择是三角形边框还是圆，以及能调整圆的大小

a) 运行截图：

i. 选择直线还是圆



ii. 调整圆的半径的大小



b) 核心代码及实现思路：

初始化 ImGui：

```
void init_imgui(GLFWwindow* window) {
#if __APPLE__
    const char* version = "#version 150";
#else
    const char* version = "#version 130";
#endif
    // 设置ImGui的上下文
    IMGUI_CHECKVERSION();
    ImGui::CreateContext();
    ImGuiIO& io = ImGui::GetIO(); (void)io;
    // 设置ImGui的样式
    ImGui::StyleColorsDark();
    ImGui_ImplGlfw_InitForOpenGL(window, true);
    ImGui_ImplOpenGL3_Init(version);
}
```

ImGui 控件的设置和渲染：

这里我们需要的是 3 个 Checkbox 和一个 Slide 来调整圆的半径。其中，是否光栅化的 Checkbox 由 line 是否选中来决定是否显示，Slide 由 circle 是否选中来决定是否显示

```
ImGui_ImplOpenGL3_NewFrame();
ImGui_ImplGlfw_NewFrame();
static int r = 80, pre_r = 80;
ImGui::NewFrame();
{
    ImGui::Checkbox("Line", &line);
    if (line)
        ImGui::Checkbox("Raterization", &need_raterization);
    ImGui::Checkbox("Circle", &circle);
    if (circle)
        ImGui::SliderInt("Radius", &r, 0, 400);
}
ImGui::Render();
```

CheckBox 选择决定的逻辑：

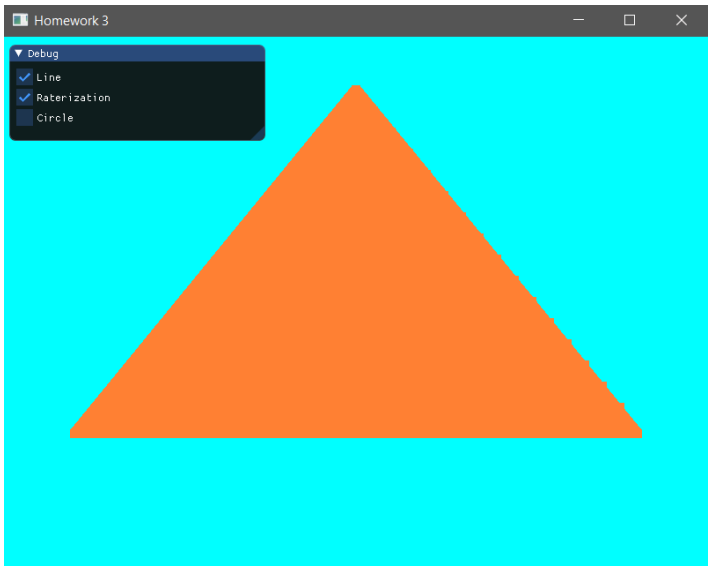
```
if (init || (!pre_line && line)) {
    Point p1(-0.8f*width / 2, -0.5f*height / 2), p2(0.8f*width / 2, -0.5f*height / 2), p3(0.0f*width / 2, 0.8f*height / 2);
    generateVertices(p1, p2, p3);
    pre_line = line = true;
    pre_circle = circle = false;
    if (init) init = false;
}
else if (pre_need_rat != need_raterization) {
    Point p1(-0.8f*width / 2, -0.5f*height / 2), p2(0.8f*width / 2, -0.5f*height / 2), p3(0.0f*width / 2, 0.8f*height / 2);
    generateVertices(p1, p2, p3);
    pre_line = line = true;
    pre_circle = circle = false;
    pre_need_rat = need_raterization;
    if (init) init = false;
}
else if (pre_r != r || (!pre_circle && circle)) {
    bresenham_circle(Point(-0.2f*width, -0.2*height), r);
    pre_line = line = false;
    pre_circle = circle = true;
    pre_r = r;
}
```

其中 init 是表示是否是第一次打开，假如是，则画的是 Bresenham 画直线画出来的三角形边框。另外，pre_line 和 pre_circle 是为了记录上一次 line 和 circle 的选择值，来判断选项是否改变，从而避免了每次渲染都需要重新计算点，避免不必要的运算。need_raterization 和 pre_need_rat 同理。

Bonus：

1. 使用三角形光栅转换算法，用和背景不同的颜色，填充你的三角形

   a) 运行截图：



   b) 核心代码：

   三角形光栅转换算法：

```
void triangle_resterization(Point p[3], vector<Point>& v) {
    Point min, max;
    get_triangle_bbox(p, min, max);
    for (int y = min.y; y < max.y; y++) {
        for (int x = min.x; x < max.x; x++) {
            if (is_in_triangle(Point(x, y), p)) {
                v.push_back(Point(x, y));
            }
        }
    }
}
```

辅助函数：

求出三角形的包围盒

```
void get_triangle_bbox(Point p[3], Point& min, Point& max) {
    min = p[0];
    max = p[0];
    for (int i = 1; i < 3; i++) {
        if (p[i].x > max.x) max.x = p[i].x;
        else if (p[i].x < min.x) min.x = p[i].x;
        if (p[i].y > max.y) max.y = p[i].y;
        else if (p[i].y < min.y) min.y = p[i].y;
    }
}
```

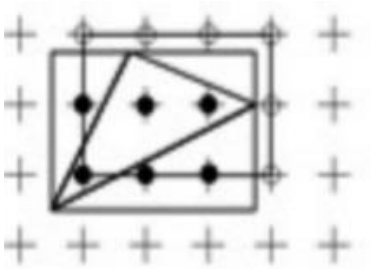
判断一个点是否在三角形内

```
bool is_in_triangle(Point point, Point p[3]) {
    float x0 = p[2].x - p[0].x, y0 = p[2].y - p[0].y;   // v0
    float x1 = p[1].x - p[0].x, y1 = p[1].y - p[0].y;   // v1
    float x2 = point.x - p[0].x, y2 = point.y - p[0].y; // v2

    float t00 = x0 * x0 + y0 * y0; // v0*v0
    float t01 = x0 * x1 + y0 * y1; // v0*v1
    float t02 = x0 * x2 + y0 * y2; // v0*v2
    float t11 = x1 * x1 + y1 * y1; // v1*v1
    float t12 = x1 * x2 + y1 * y2; // v1*v2

    // u = ((v1*v1) * (v0*v2) - (v0*v1) * (v1*v2)) / ((v0*v0) * (v1*v1) - (v0*v1) * (v0*v1))
    float u = float(t11*t02 - t01 * t12) / (t00*t11 - t01 * t01);
    // v = ((v0*v0) * (v1*v2) - (v0*v1) * (v0*v2)) / ((v0*v0) * (v1*v1) - (v0*v1) * (v0*v1))
    float v = float(t00*t12 - t01 * t02) / (t00*t11 - t01 * t01);
    if (u + v <= 1 && u >= 0 && v >= 0) return true;
    else return false;
}
```

c) 实现思路：

三角形光栅算法思路如下：

首先，对于一个三角形，求解其包围盒



可以理解为一个可以包含了整个三角形的最小矩形。求解包围盒的目的是避免扫描整个图片，仅仅扫描尽量少的点

然后遍历包围盒内的每一个点：

若该点在三角形内，则填充该点

否则，继续