hw3实验报告

实验要求

Basic:

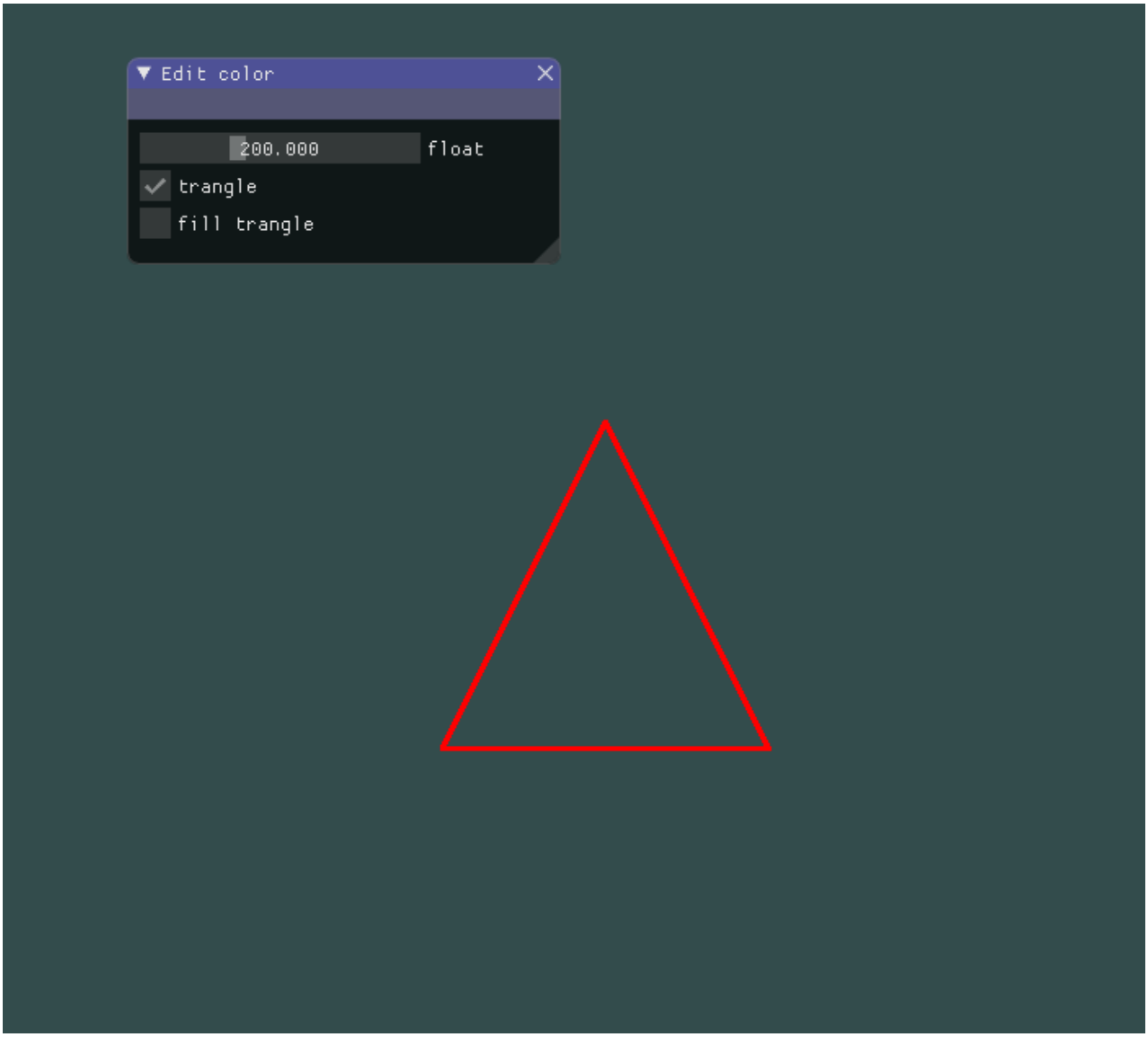
1. 使用Bresenham算法(只使用integer arithmetic)画一个三角形边框：input为三个2D点；output三条直线（要求图元只能用 GL_POINTS ，不能使用其他，比如 GL_LINES 等）。
2. 使用Bresenham算法(只使用integer arithmetic)画一个圆：input为一个2D点(圆心)、一个integer半径；output为一个圆。
3. 在GUI在添加菜单栏，可以选择是三角形边框还是圆，以及能调整圆的大小(圆心固定即可)。

Bonus:

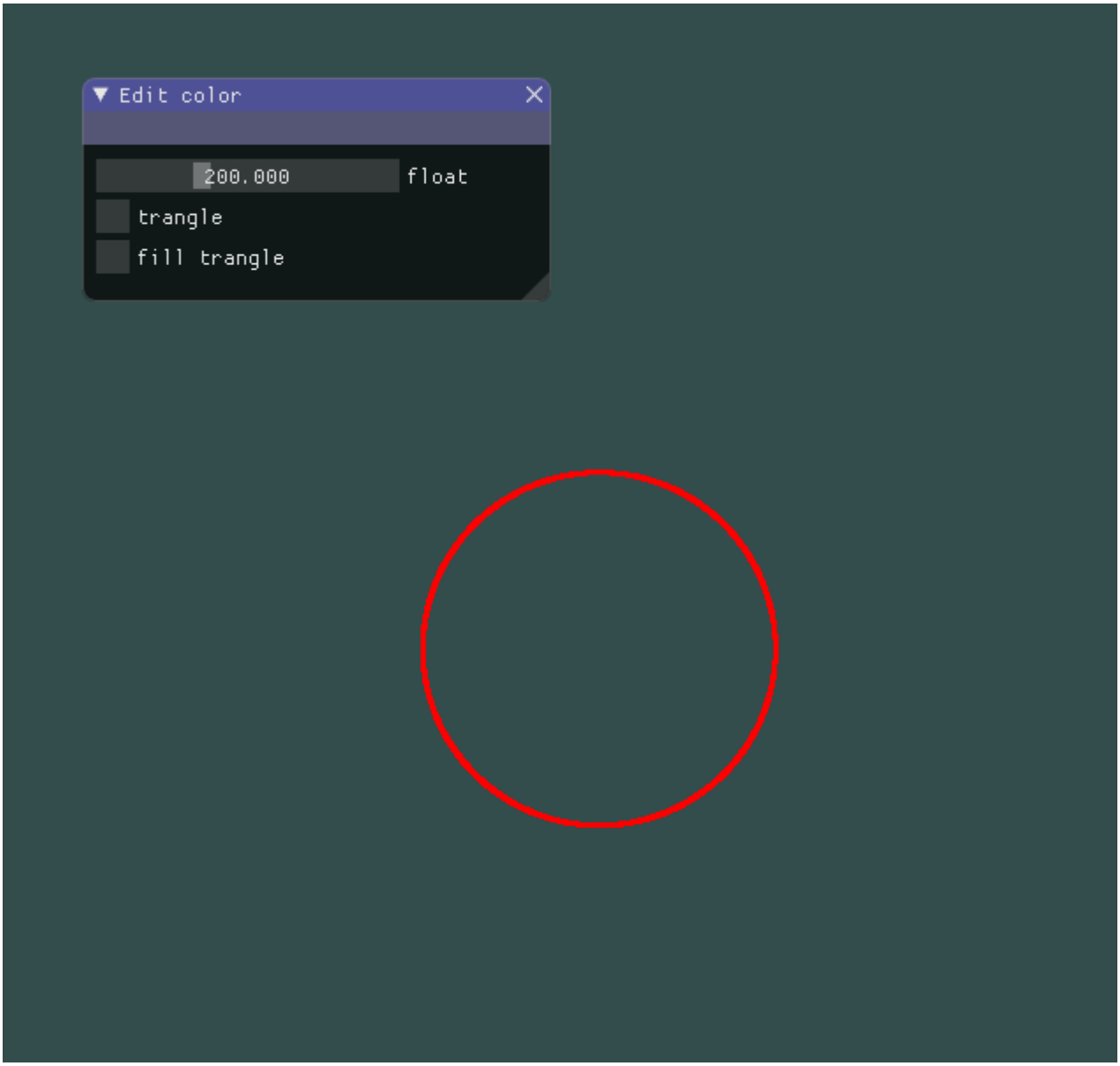
1. 使用三角形光栅转换算法，用和背景不同的颜色，填充你的三角形。

实验截图

1. Bresenham算法画三角形边框：

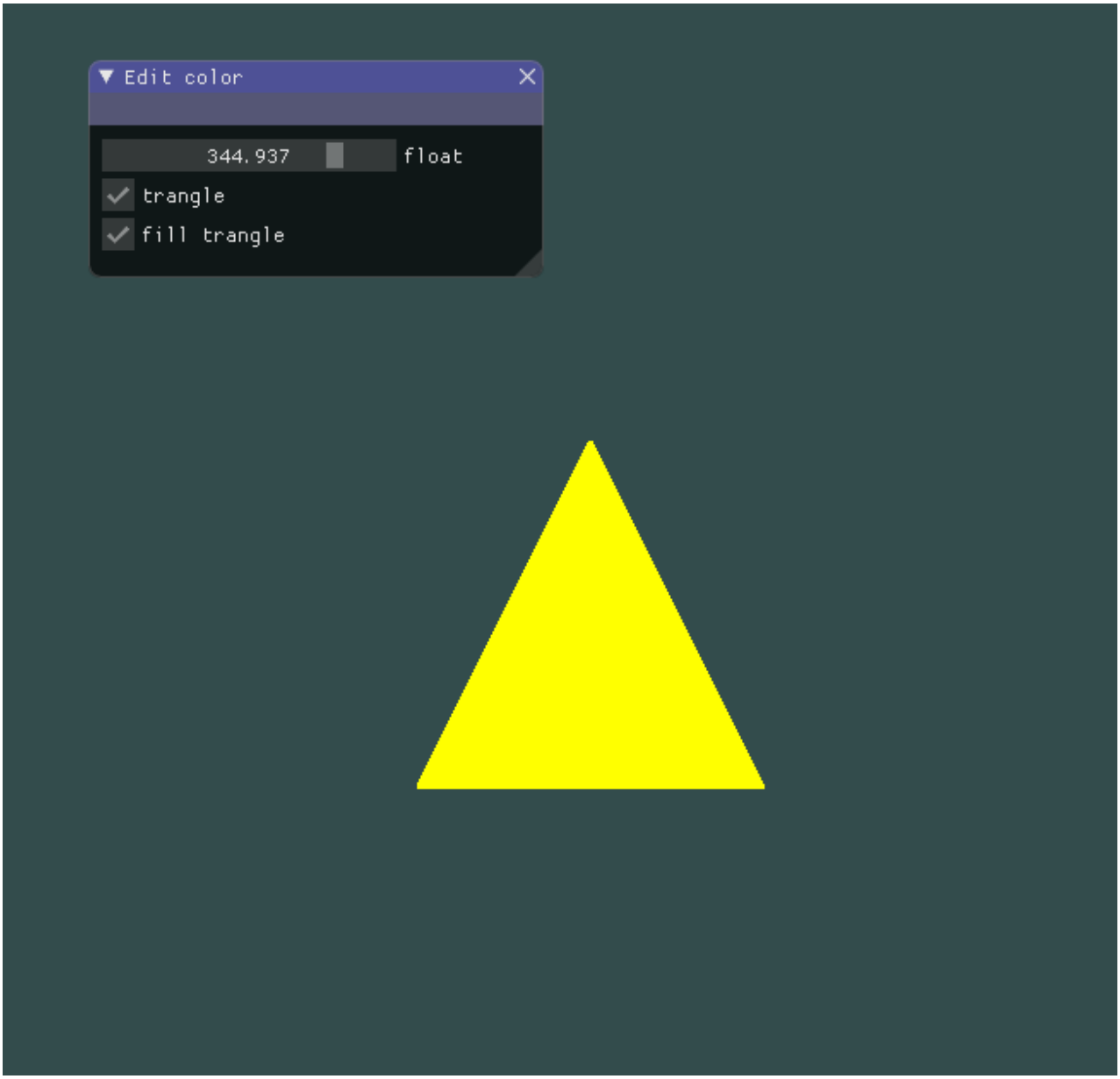


2. Bresenham算法画圆：



3. 在GUI在添加菜单栏，可以选择是三角形边框还是圆，以及能调整圆的大小(圆心固定即可)。

4. bonus：使用三角形光栅转换算法，用和背景不同的颜色，填充三角形



实验主要函数算法介绍

1. Bresenham算法

   Bresenham算法是用来描绘由两点所决定的直线的算法，它会算出一条线段在n维位图上最接近的点。这个算法只会用到较为快速的整数加法、减法和位元移位，常用于绘制电脑画面中的直线。

2. 画三角形：在算法中，输入两个点，根据算法得出直线，由于是三角形，我们根据三个点输出三条直线即可。具体算法如下：

Summary of Bresenham Algorithm

- **draw** (x_0, y_0)
- **Calculate** $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\Delta x, p_0 = 2\Delta y - \Delta x$
- **If** $p_i \le 0$ **draw** $(x_{i+1}, \bar{y}_{i+1}) = (x_i + 1, \bar{y}_i)$

  and compute $p_{i+1} = p_i + 2\Delta y$
- **If** $p_i > 0$ **draw** $(x_{i+1}, \bar{y}_{i+1}) = (x_i + 1, \bar{y}_i + 1)$

  and compute $p_{i+1} = p_i + 2\Delta y - 2\Delta x$
- **Repeat the last two steps**

where

$$\Delta x = x_1 - x_0, \Delta y = y_1 - y_0, \quad m = \Delta y / \Delta x$$
$$c = (2B - 1)\Delta x + 2\Delta y$$

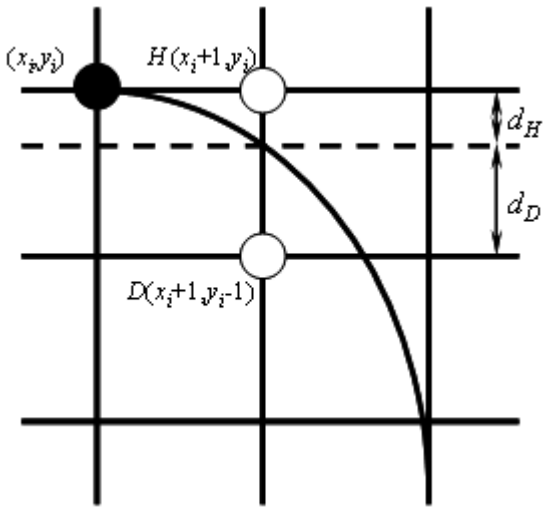
具体C++代码实现如下：

```
vector<vector<float>> Triangle::DrawLine(float x1, float y1, float x2, float y2) {
    vector<vector<float>> points;
    vector<float> pointsX;
    vector<float> pointsY;
    float deltaY = abs(y1 - y2);
    float deltaX = abs(x1 - x2);
    float incX = x1 > x2 ? -1.0f : 1.0f;
    float incY = y1 > y2 ? -1.0f : 1.0f;
    float tempX = x1, tempY = y1;
    pointsX.push_back(tempX);
    pointsY.push_back(tempY);
    if (deltaX > deltaY) {
        float Pi = 2 * deltaY - deltaX;
        for (float i = 0.0f; i < deltaX; i += 1.0f) {
            if (Pi <= 0.0f) {
                Pi += 2 * deltaY;
```

```
            }
            else {
                tempY += incY;
                Pi += 2 * deltaY - 2 * deltaX;
            }
            tempX += incX;
            pointsX.push_back(tempX);
            pointsY.push_back(tempY);
        }
    }
    else {
        float Pi = 2 * deltaX - deltaY;
        for (float i = 0.0f; i < deltaY; i += 1.0f) {
            if (Pi <= 0.0f) {
                Pi += 2 * deltaX;
            }
            else {
                tempX += incX;
                Pi += 2 * deltaX - 2 * deltaY;
            }
            tempY += incY;
            pointsX.push_back(tempX);
            pointsY.push_back(tempY);
        }
    }
    points.push_back(pointsX);
    points.push_back(pointsY);
    return points;
}
```

如果只比较delatX的话，那么则只能实现绘画斜率小于或等于1的直线，所以就把delatY也比较了，使得各个方向的直线都能够绘画。

3. Bresenham算法画圆利用了圆的对称性，如下图，在 0≤x≤y 的 1/8 圆周上，像素坐标 x 值单调增加，y 值单调减少。



1. d_H更小，下一个像素点就选 H（x_i + 1,y_i）
2. d_D 更小，下一个像素点就选 D（x_i + 1,y_i − 1）
3. 一样大，选哪个都行。

类似直线，我们可以使用d_0进行迭代比较。

初始化$d_0 = F（1， R - 1/2） = 1 + （R - 1/2）^2 - R^2 = 5/4 - R;$

若$d_i < 0$:

$d_{i+1} = d_i + 2x_i + 3$

否则：

$d_{i+1} = d_i + 2(x_i - y_i) + 5$

根据对称性每次比较后都记录8个点。

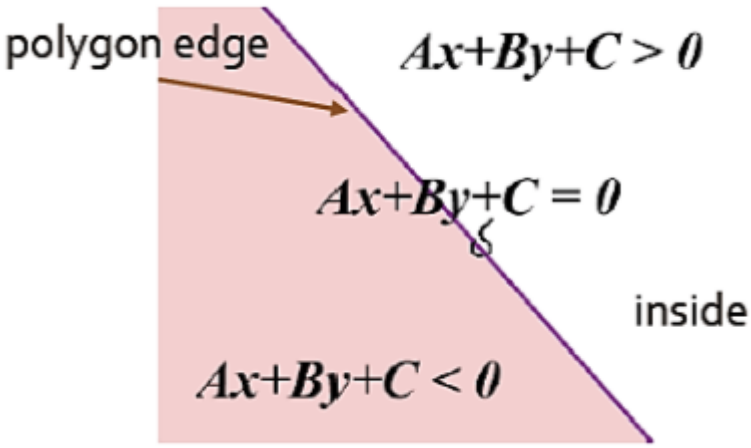
具体C++代码实现如下：

```
float* Circle::getDrawPoints() {
    float d = 1.25f - r;
    float tempX = 0, tempY = r;
    vector<float> pointsX;
    vector<float> pointsY;
    while (tempX < tempY) {
        // 1
        pointsX.push_back(tempX + x);
        pointsY.push_back(tempY + y);
        // 2
        pointsX.push_back(-tempX + x);
        pointsY.push_back(tempY + y);
        // 3
        pointsX.push_back(tempX + x);
        pointsY.push_back(-tempY + y);
        // 4
        pointsX.push_back(-tempX + x);
        pointsY.push_back(-tempY + y);
        // 5
        pointsX.push_back(tempY + y);
        pointsY.push_back(tempX + x);
        // 6
        pointsX.push_back(-tempY + y);
        pointsY.push_back(tempX + x);
        //7
        pointsX.push_back(tempY + y);
        pointsY.push_back(-tempX + x);
        // 8
        pointsX.push_back(-tempY + y);
        pointsY.push_back(-tempX + x);

        if (d < 0) {
            d += 2 * tempX + 3;
        }
        else {
            d += 2 * (tempX - tempY) + 5;
            tempY--;
        }
        tempX++;
    }
```

```
    pointsCount = pointsX.size();
    if (Allpoints != NULL) {
        delete[] Allpoints;
    }
    Allpoints = new float[pointsCount * 6];
    int pos = 0;
    for (int i = 0; i < pointsCount; ++i, pos += 6) {
        Allpoints[pos] = pointsX[i] / 800.0f;
        Allpoints[pos + 1] = pointsY[i] / 800.0f;
        Allpoints[pos + 2] = 0.0f;
        //color
        Allpoints[pos + 3] = 1.0f;
        Allpoints[pos + 4] = 0.0f;
        Allpoints[pos + 5] = 0.0f;
    }
    return Allpoints;
}
```

4. 三角形光栅转换算法
    1. 确定三角形的三条边的直线方程表达式，形如：Ax + By + C = 0；
    2. 计算出能够包围三角形最小矩形；
    3. 扫描矩形中的每个点是否在三角形中，利用：



计算点 Ax + By + C 的指，若小于0 则不再三角形中。

注意：

1. 计算直线方程式时，使用两点式计算时需考虑直线为x = k或者 y = k 的特殊情况，得到对应正确的表达式。
2. 通过两点得到直线的表达式后，需要判断第三个点是否属于Ax + By + C > 0 的情况， 否则需要对直线方程式进行调转，继变为 -Ax - By - C = 0，以此保证每个点计算的准确性。

具体C++代码实现如下：

```
vector<vector<float>> Triangle::fillTriangle() {
    vector<vector<float>> fillPoints;
    vector<float> pointsX;
    vector<float> pointsY;
    vector<vector<float>> lines;
```

```
    lines.push_back(getLine(x1, y1, x2, y2));
    lines.push_back(getLine(x1, y1, x3, y3));
    lines.push_back(getLine(x2, y2, x3, y3));

    for (int i = 0; i < 3; i++) {
        float x, y;
        if (i == 0) {
            x = x3;
            y = y3;
        }
        else if (i == 1) {
            x = x2;
            y = y2;
        }
        else {
            x = x1;
            y = y1;
        }
        if (lines[i][0] * x + lines[i][1] * y + lines[i][2] < 0) {
            for (int j = 0; j < 3; j++) {
                lines[i][j] *= -1;
            }
        }
    }
    float maxX = findMaxInThree(x1, x2, x3);
    float maxY = findMaxInThree(y1, y2, y3);
    float minX = findMinInThree(x1, x2, x3);
    float minY = findMinInThree(y1, y2, y3);

    for (float x = minX; x <= maxX; x += 1.0f) {
        for (float y = minY; y <= maxY; y += 1.0f) {
            bool inside = true;
            for (int i = 0; i < lines.size(); ++i) {
                if (lines[i][0] * x + lines[i][1] * y + lines[i][2] < 0) {
                    inside = false;
                }
            }
            if (inside) {
                pointsX.push_back(x);
                pointsY.push_back(y);
            }
        }
    }
    fillPoints.push_back(pointsX);
    fillPoints.push_back(pointsY);
    return fillPoints;
}
```

5. 优化

由于在设置GUI进行三角形和圆直接的切换后，对三角形进行填充时，每个while循环中都回重新扫描三角形的点，导致程序巨卡无比，所以我自己增加了变量的设置，如果三角形的选择没有变化，那么将不会再重新计算需要渲染的点，而使用原来的数据，大大优化了程序的体验感。

具体代码实现如下：

```
...
bool draw_trangle = true;
bool fill_trangle = false;
bool fill_record = false;
bool draw_record = true;
...
while (!glfwWindowShouldClose(window))
{
    processInput(window);

    if (draw_trangle) {
        if (!fill_trangle || fill_record != fill_trangle || !draw_record) {
            vertices = triangle.getDrawPoints(fill_trangle);
            pointsCount = triangle.getPointsCount();
            fill_record = fill_trangle;
        }
    }
    else {
        circle.setR(circle_r);
        vertices = circle.getDrawPoints();
        pointsCount = circle.getPointsCount();
    }
    draw_record = draw_trangle;
    ...
}
```