HW3

16340220 王培钰 电子政务

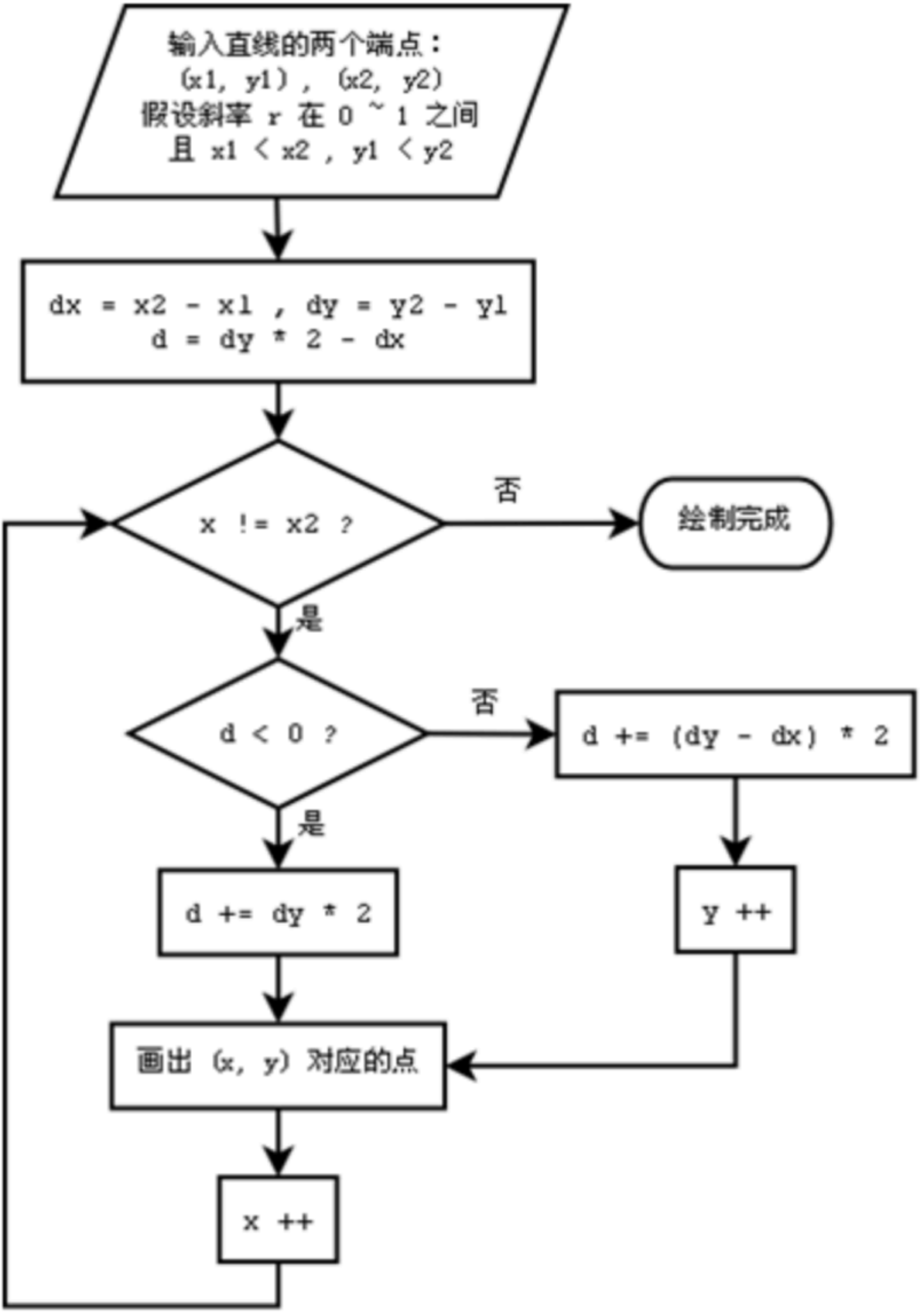
Bresenham算法在进行直线的扫描转换时，由于不涉及浮点数运算，只是整数类型的运算，所以大大提高了计算速率。

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)。

Bresenham画直线的算法主要步骤是判断下一点的位置。

Bresenham画直线的算法的流程如下：



核心算法代码实现如下：

```
// 异或交换两个变量的值
inline void swap(int *a, int *b)
{
```

```
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}

vector<point> Bresenham_Line (point p1, point p2)
{
    vector<point> temp;
    //temp.push_back(p1);
    int dx = abs(p2.x - p1.x);
    int dy = abs(p2.y - p1.y);
    // 弦值大于45度, 进行坐标变换
    bool direction = 1;
    if (dx < dy)
    {
        direction = 0;
        swap(&p1.x, &p1.y);
        swap(&p2.x, &p2.y);
        swap(&dx, &dy);
    }
    int ix = (p2.x - p1.x) > 0 ? 1 : -1;
    int iy = (p2.y - p1.y) > 0 ? 1 : -1;
    int d = dy * 2 - dx;
    int x = p1.x;
    int y = p1.y;
    if (direction)
    {
        while (x != p2.x)
        {
            if (d < 0)
            {
                d += dy * 2;
            }
            else
            {
                d += (dy - dx) * 2;
                y+=iy;
            }
            temp.push_back(point(x, y));
            x+=ix;
        }
    }
    else
    {
        while (x != p2.x)
        {
            if (d < 0)
            {
                d += dy * 2;
```

```
            }
            else
            {
                d += (dy - dx) * 2;
                y+=iy;
            }
            temp.push_back(point(y, x));
            x+=ix;
        }
    }
    //temp.push_back(p2);
    return temp;
}
```

之后再根据题目要求根据三个点绘制三条直线组成三角形：

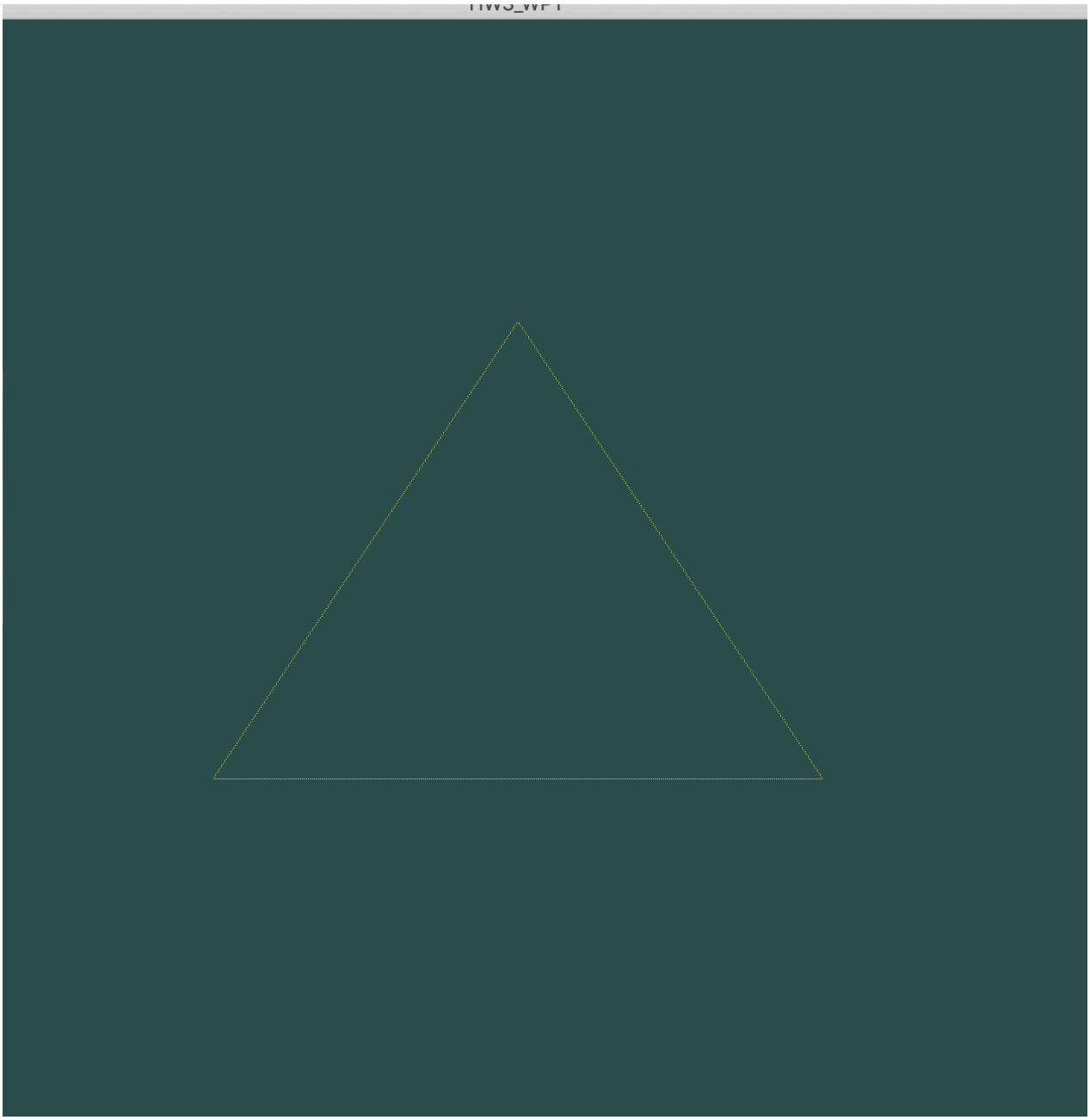
```
// 根据三个点画出三条直线并获取这三条直线组成的三角形的x所有的点
vector<point> Draw_Triangle(point p1, point p2, point p3)
{
    vector<point> vec, vec1, vec2, vec3;
    vec1 = Bresenham_Line(p1, p2);
    vec2 = Bresenham_Line(p1, p3);
    vec3 = Bresenham_Line(p2, p3);
    vec.insert(vec.end(), vec1.begin(), vec1.end());
    vec1.clear();
    vec.insert(vec.end(), vec2.begin(), vec2.end());
    vec2.clear();
    vec.insert(vec.end(), vec3.begin(), vec3.end());
    vec3.clear();
    vec.push_back(p1);
    vec.push_back(p2);
    vec.push_back(p3);
    return vec;
}
```

之后再将存储的组成三角形所需要的所有点传入到vertices数组中，便于之后和openGL的GL_PINTS图元进行绑定。

```
float* getVec(vector<point> points)
{
    int num = points.size();
    int N = num * 6;
    float *vertices = new float [N];
    for (int i = 0; i < num; i++) {
        // 位置，归一化操作
        vertices[i * 6 + 0] = (float)points[i].x / ((float)SCR_WIDTH / 2);
        vertices[i * 6 + 1] = (float)points[i].y / ((float)SCR_HEIGHT / 2);
        vertices[i * 6 + 2] = 0.0f;
```

```
            // 颜色
            vertices[i * 6 + 3] = 1.0f;
            vertices[i * 6 + 4] = 1.0f;
            vertices[i * 6 + 5] = 0.0f;
        }
        return vertices;
    }
```

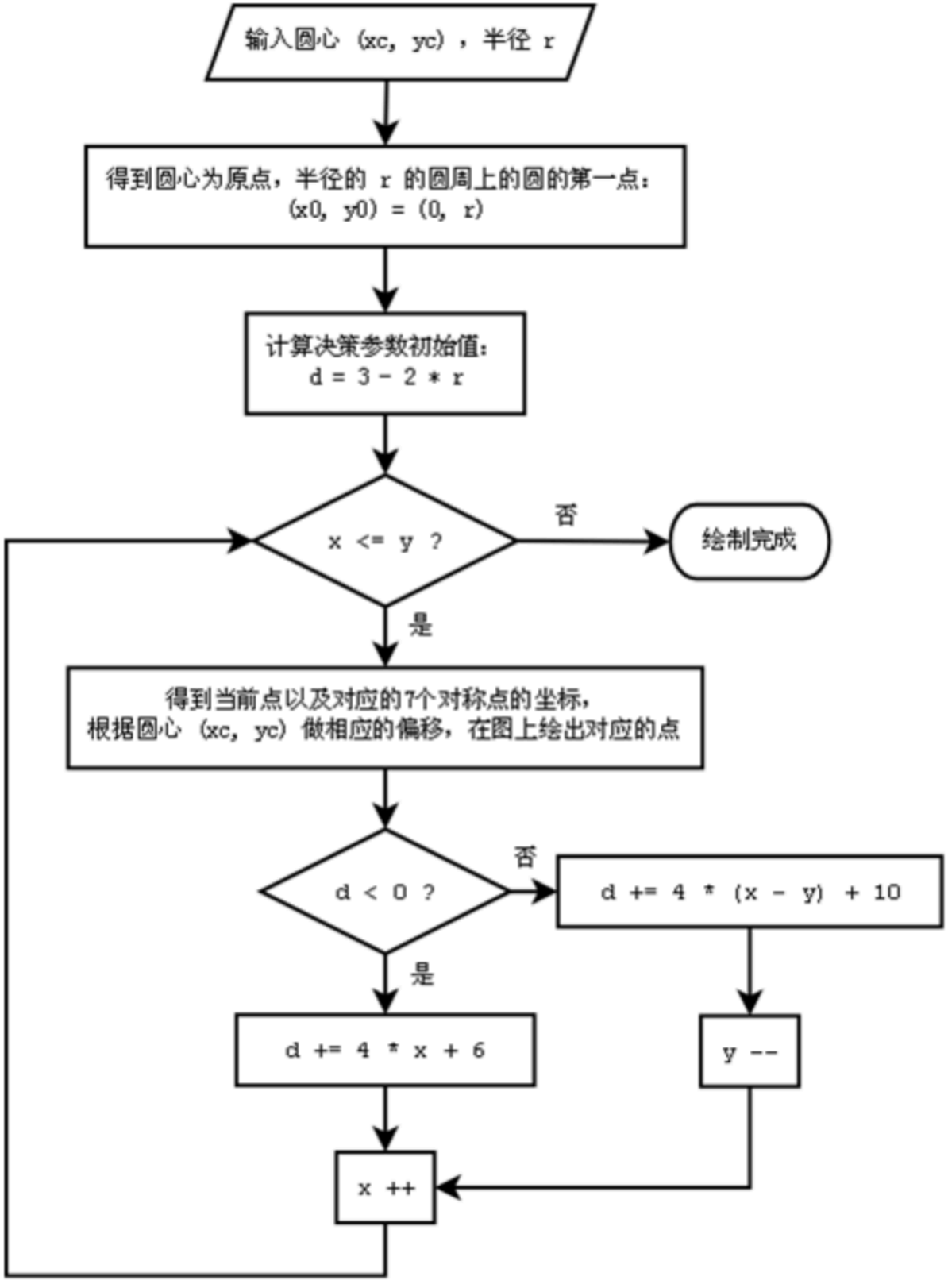
画出的效果如下：



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。

Bresenham画圆算法又称中点画圆算法，与Bresenham 直线算法一样，其基本的方法是利用判别变量来判断选择最近的像素点，判别变量的数值仅仅用一些加、减和移位运算就可以计算出来。为了简便起见，考虑一个圆心在坐标原点的圆，而且只计算八分圆周上的点，其余圆周上的点利用对称性就可得到。

算法流程如图所示：

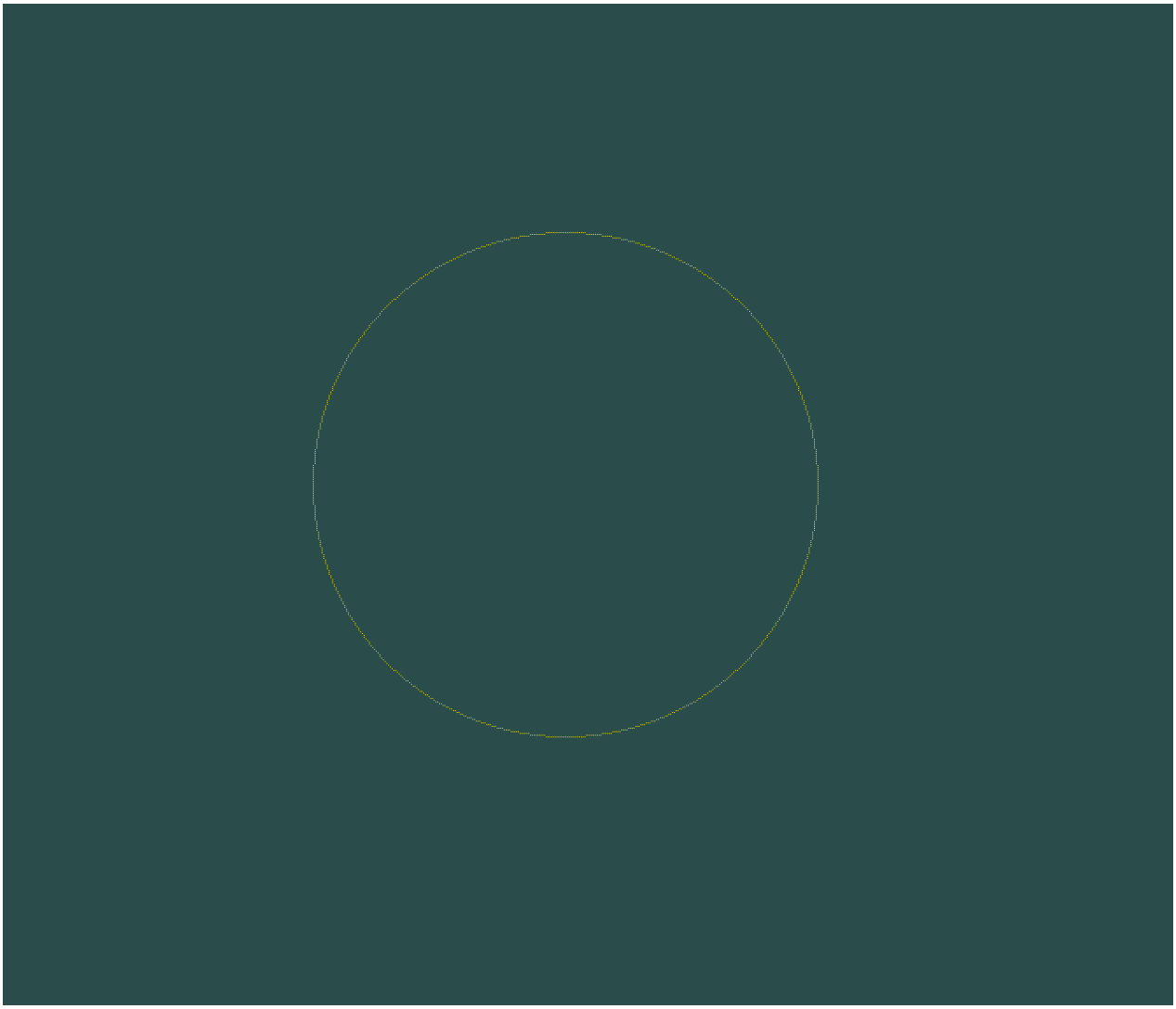


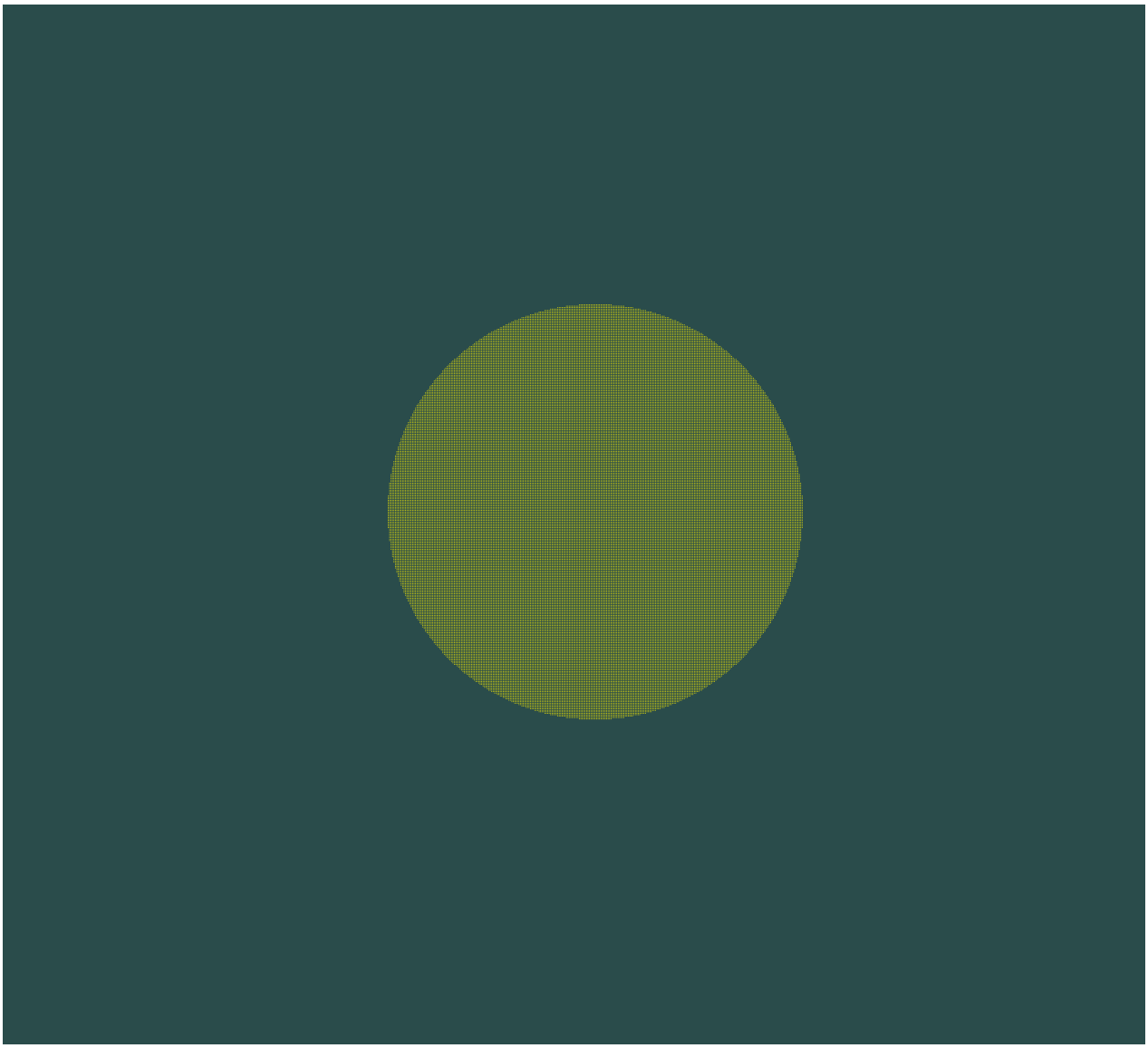
核心代码代码实现如下：

- 其中加了一步fill填充，圆的填充比较简单，只需要趋近半径来迭代即可。

```
// p0为圆心，r是半径，fill代表是否进行填充
vector<point> Draw_Circle(point p0, int r, bool fill)
{
    vector<point> points;
    int x = 0, y = r, yi, d;
    d = 3 - 2 * r;
    if (fill)
    {
        // 填充的话画实心圆
        while (x <= y)
        {
            for (yi = x; yi <= y; yi++)
            {
                circle_8_points(points, p0, point(x, yi));
            }
            if (d < 0)
            {
                d = d + 4 * x + 6;
            }
            else
            {
                d = d + 4 * (x - y) + 10;
                y--;
            }
            x++;
        }
    }
    // 如果不填充
    else
    {
        while (x <= y)
        {
            circle_8_points(points, p0, point(x, y));
            if (d < 0)
            {
                d = d + 4 * x + 6;
            }
            else
            {
                d = d + 4 * (x - y) + 10;
                y--;
            }
            x++;
        }
    }
    return points;
}
```

填充和不填充的圆效果分别如下：





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)。

运用ImGui里面的`Menu`模块，设计实现了画三角形，画圆，关闭窗口的功能。

```
ImGui::Begin("HW3_WPY", &active, ImGuiWindowFlags_MenuBar);
ImGui::SetWindowSize(ImVec2(400, 180));
if (ImGui::BeginMenuBar())
{
    if (ImGui::BeginMenu("Menu"))
    {
        if (ImGui::MenuItem("Triangle")) {
            circle = false;
        }
        if (ImGui::MenuItem("Circle")) {
            circle = true;
        }
        if (ImGui::MenuItem("Close")) {
            active = false;
            glfwSetWindowShouldClose(window, true);//关闭窗口
```

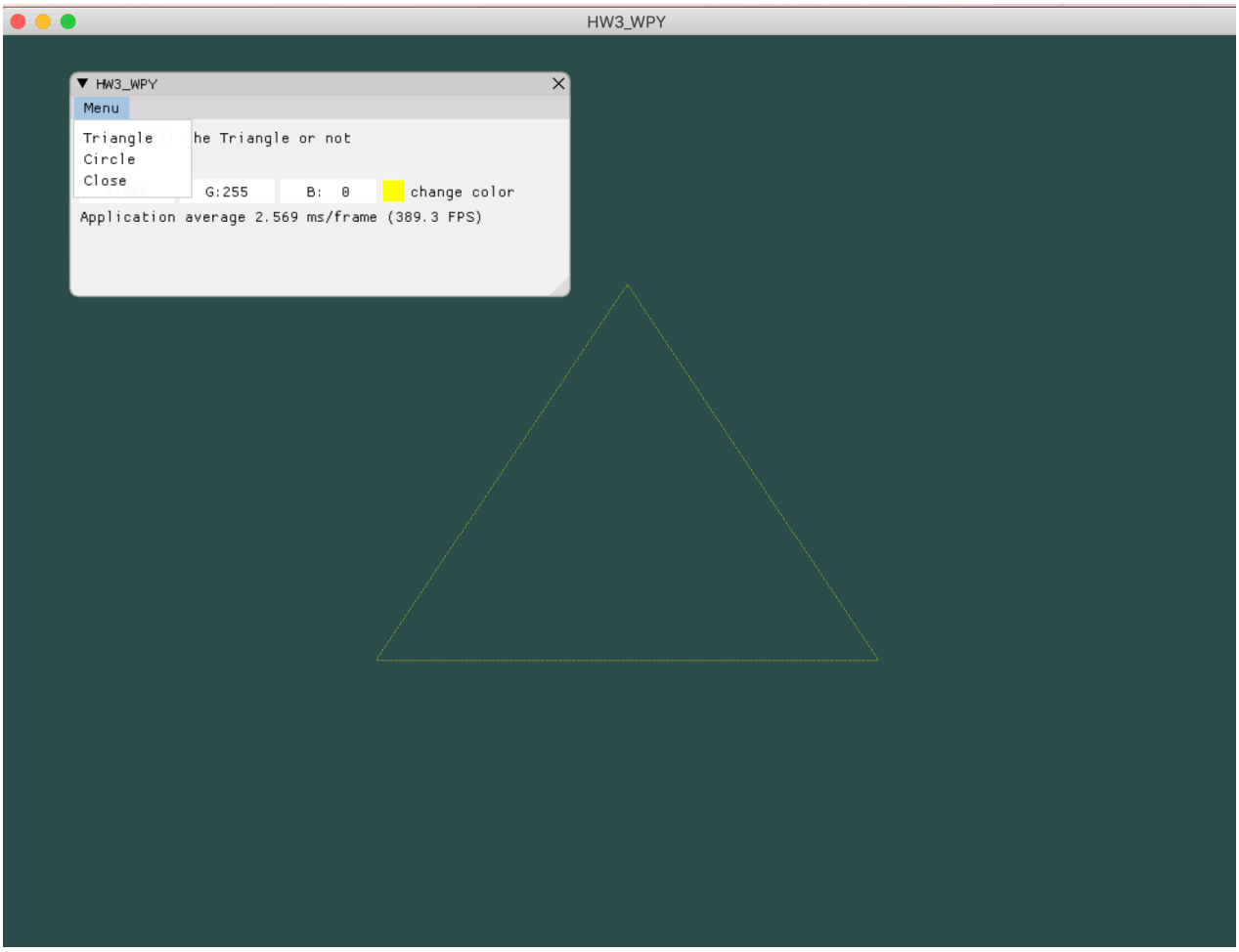
```
            }
            ImGui::EndMenu();
        }
        ImGui::EndMenuBar();
    }
```

然后利用`SliderInt`滑动条来进行半径赋值的拖动操作。又实现了对圆的颜色更改以及选择是否填充的操作

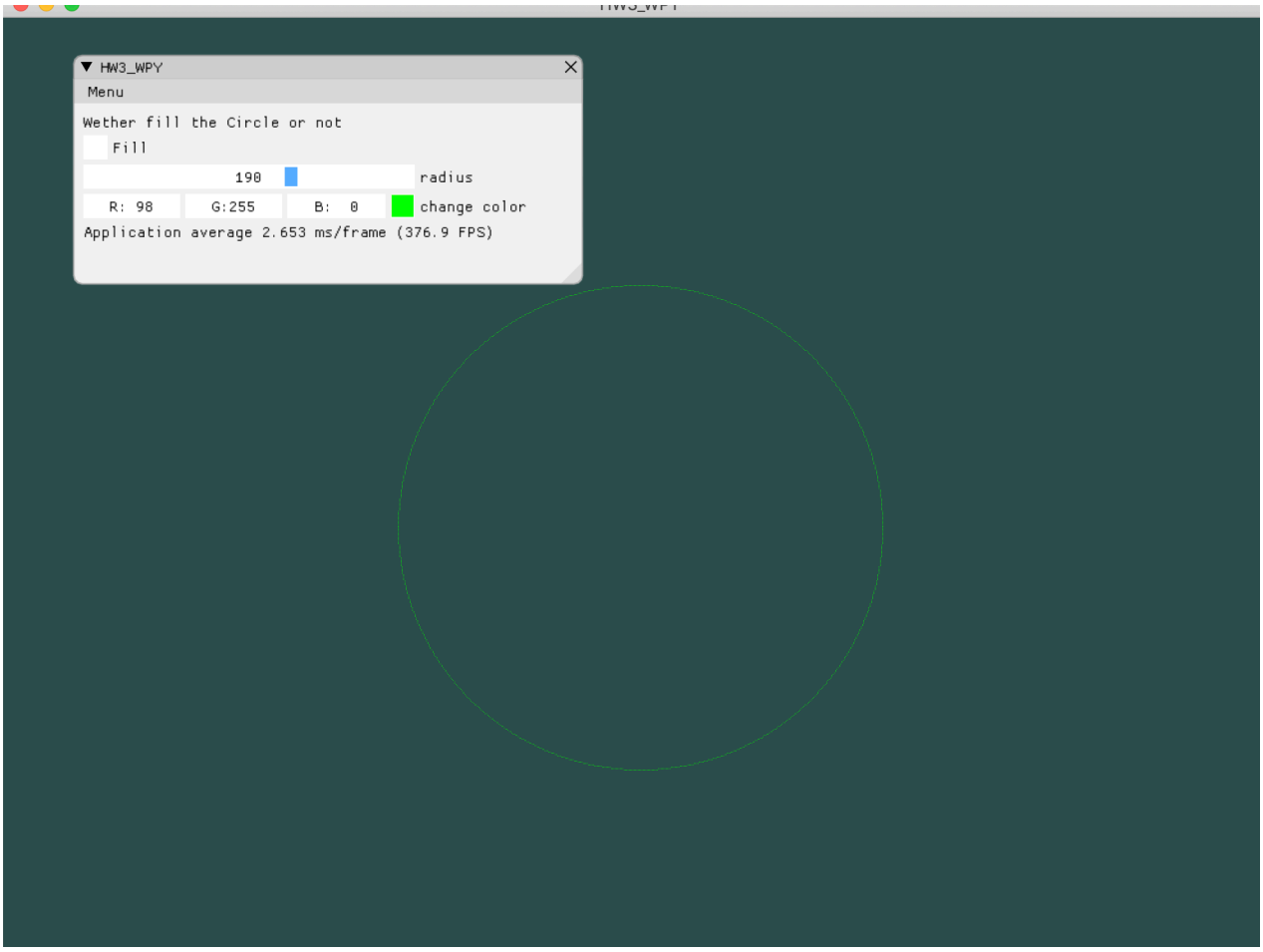
```
if (circle)
{
    ImGui::Text("Wether fill the Circle or not");
    ImGui::Checkbox("Fill", &fill);
    ImGui::SliderInt("radius", &r, 0, 300);
    ImGui::ColorEdit3("change color", (float*)&change_rgb); // 设置圆的填充色
}
```

实现效果如下：

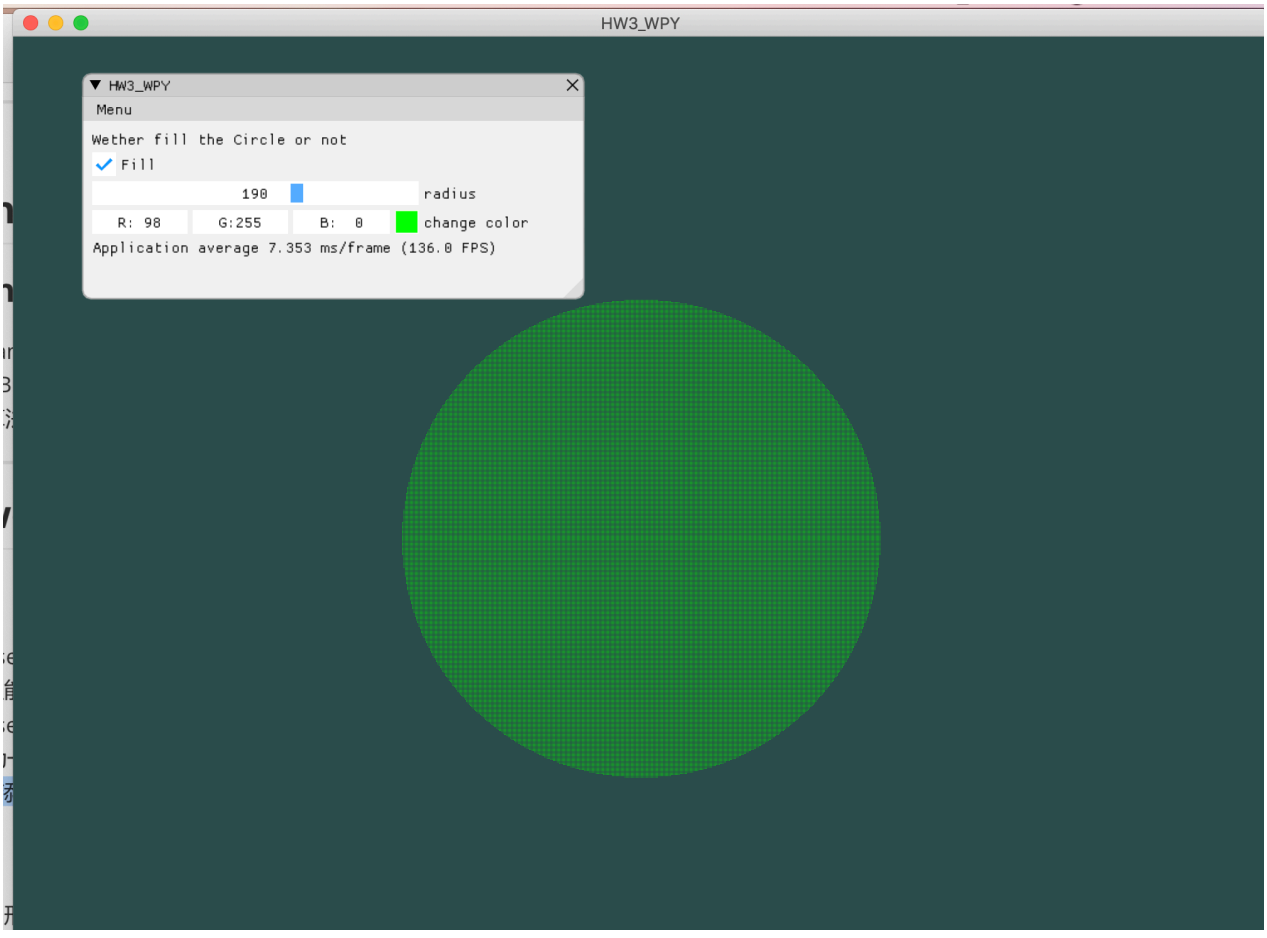
- 三角形：



- 圆非填充


HW3_WPY
Menu
Wether fill the Circle or not
Fill
190
radius
R: 98
G:255
B: 0
change color
Application average 2.653 ms/frame (376.9 FPS)

- 圆的填充


HW3_WPY
HW3_WPY
Menu
Wether fill the Circle or not
Fill
190
radius
R: 98
G:255
B: 0
change color
Application average 7.353 ms/frame (136.0 FPS)

Bonus:

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。

对于扫描法我们的方法只需先判断三角形对应的最小包围矩阵：

```
int min_x = min(p1.x, min(p2.x, p3.x));
int max_x = max(p1.x, max(p2.x, p3.x));
int min_y = min(p1.y, min(p2.y, p3.y));
int max_y = max(p1.y, max(p2.y, p3.y));
```

然后在这个矩阵内进行循环扫描判断扫描的点是否在三角形内：

其判断条件如下：

- 若点P在三角形ABC内部，可以通过以下三个条件判断：
  - 点P和点C在直线AB同侧
  - 点P和点B在直线AC同侧
  - 点P和点A在直线BC同侧

三个条件都满足，则该点是我们需要的点

```
// 判断三角形内的点和直线对的角点在直线的同侧
float t1 = (i * k1 + b1 - j) * (p1.x * k1 + b1 - p1.y);
float t2 = (i * k2 + b2 - j) * (p2.x * k2 + b2 - p2.y);
float t3 = (i * k3 + b3 - j) * (p3.x * k3 + b3 - p3.y);
if (t1 >= 0 && t2 >= 0 && t3 >= 0)
{
    points.push_back(point(i, j));
}
```

函数的完整代码如下：

```
// 三角形的光栅化(扫描法)
vector<point> Rasterize(point p1, point p2, point p3)
{
    vector<point> points;
    float k1 = (float)(p3.y - p2.y) / (float)(p3.x - p2.x);
    float k2 = (float)(p1.y - p3.y) / (float)(p1.x - p3.x);
    float k3 = (float)(p2.y - p1.y) / (float)(p2.x - p1.x);
    float b1 = (float)p2.y - k1 * p2.x;
    float b2 = (float)p3.y - k2 * p3.x;
    float b3 = (float)p1.y - k3 * p1.x;
    int min_x = min(p1.x, min(p2.x, p3.x));
    int max_x = max(p1.x, max(p2.x, p3.x));
    int min_y = min(p1.y, min(p2.y, p3.y));
    int max_y = max(p1.y, max(p2.y, p3.y));
```

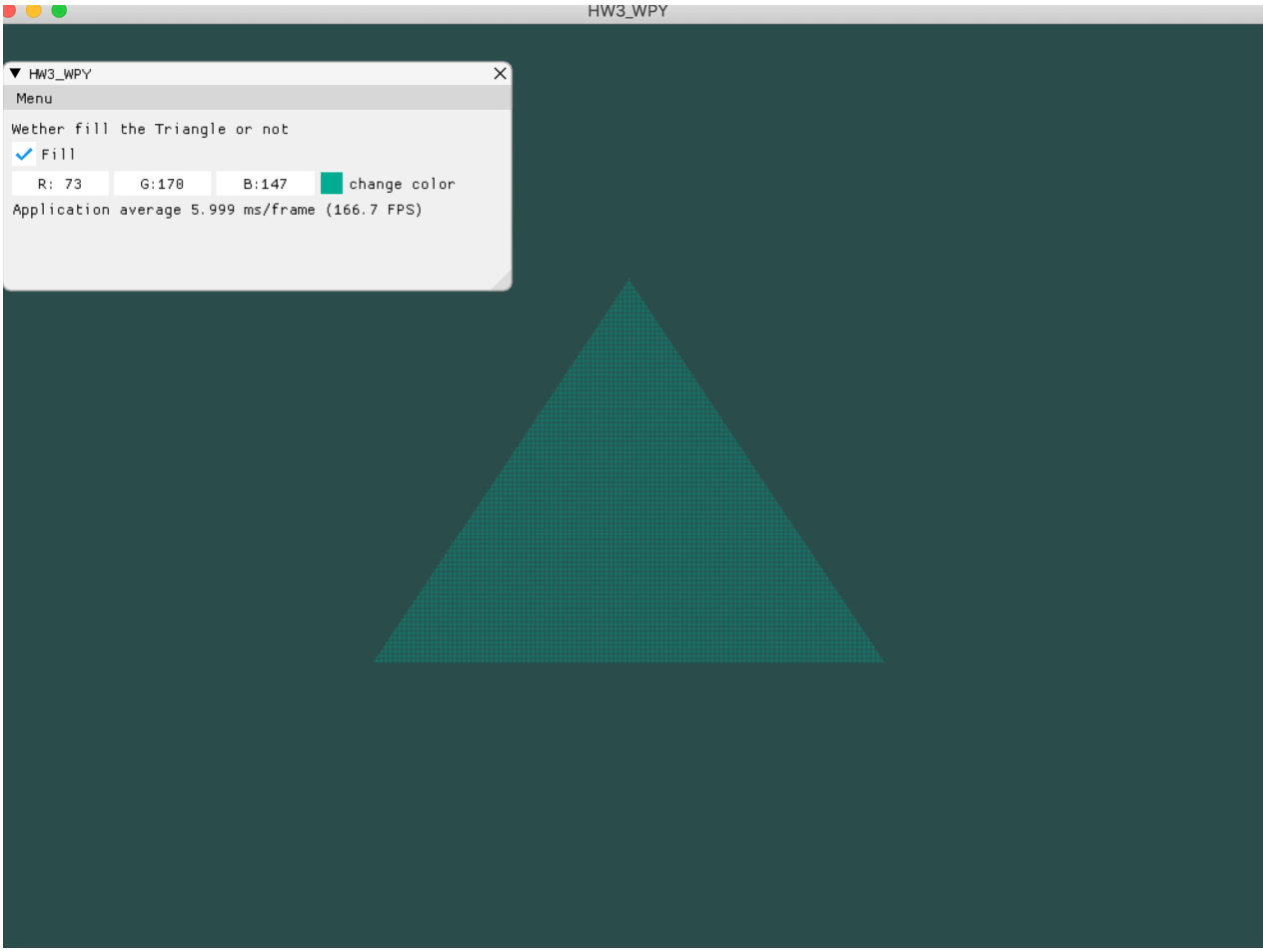
```
    for (int i = min_x; i <= max_x; i++)
    {
        for (int j = min_y; j <= max_y; j++)
        {
            // 判断三角形内的点和直线对的角点在直线的同侧
            float t1 = (i * k1 + b1 - j) * (p1.x * k1 + b1 - p1.y);
            float t2 = (i * k2 + b2 - j) * (p2.x * k2 + b2 - p2.y);
            float t3 = (i * k3 + b3 - j) * (p3.x * k3 + b3 - p3.y);
            if (t1 >= 0 && t2 >= 0 && t3 >= 0)
            {
                points.push_back(point(i, j));
            }
        }
    }
    return points;
}
```

然后将其绑定到gui上选择是否填充三角形以及填充色：

```
ImGui::Text("Wether fill the Triangle or not");
ImGui::Checkbox("Fill", &fill1);
ImGui::ColorEdit3("change color", (float*)&change_rgb); // 设置三角形颜色
```

```
if (fill1)
{
    points = Rasterize(point(0, 200), point(200, -100), point(-200, -100));
}
else
{
    points = Draw_Triangle(point(0, 200), point(200, -100), point(-200,
-100));
}
```

实现效果如下：



Bresenham算法参考自博客：https://www.cnblogs.com/wlzy/p/8695226.html