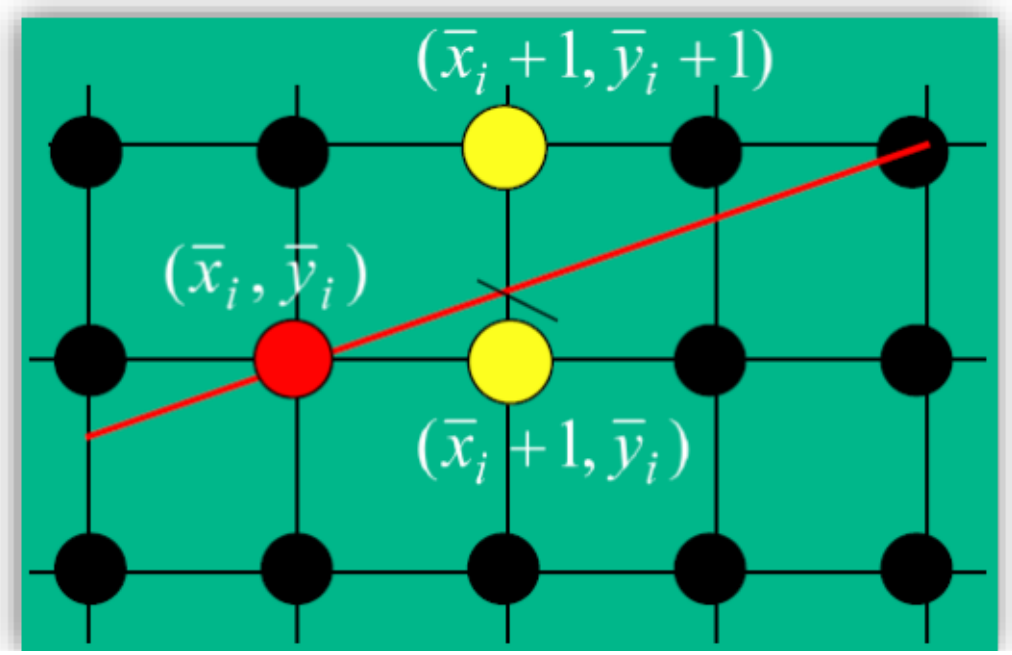
郑国林 1634304

1. 使用 Bresenham 算法(只使用 integer arithmetic)画一个三角形边框

   算法的主要思想就是当前一个点确定的时候，这时候下一个点由于一定要在格点地上，所以只会有两种选择，（下图是当斜率小于 1 且方向为正向的情况），要么是（x+1，y）的点，要么是(x+1,y+1)的点，然后直线上真实的点应该是（x+1, m(x+1)+b），这时候就看这个真实点距离哪个格点比较近就行了。



两个距离的计算公式如下：

$$
\begin{aligned}
d_{upper} &= \bar{y}_i + 1 - y_{i+1} \\
&= \bar{y}_i + 1 - mx_{i+1} - B \\
d_{lower} &= y_{i+1} - \bar{y}_i \\
&= mx_{i+1} + B - \bar{y}_i
\end{aligned}
$$

又由于我们只需要知道哪个大，就直接求差看正负符号

$$d_{lower} - d_{upper} = m(x_i + 1) + B - \bar{y}_i - (\bar{y}_i + 1 - m(x_i + 1) - B)$$
$$= 2m(x_i + 1) - 2\bar{y}_i + 2B - 1$$

division operation

M 是斜率，要用到除法，为了减少计算量，我们可以乘以 dx(这里假设 dx 是正的，负的就是换一个方向而已，后面会讲到)，乘 dx 不改变符号

$$p_i = \Delta x \bullet (d_{lower} - d_{upper}) = 2\Delta y \bullet (x_i + 1) - 2\Delta x \bullet \bar{y}_i + (2B - 1)\Delta x$$
$$= 2\Delta y \bullet x_i - 2\Delta x \bullet \bar{y}_i + (2B - 1)\Delta x + 2\Delta y$$
$$= 2\Delta y \bullet x_i - 2\Delta x \bullet \bar{y}_i + c$$

here

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

然后就是要找出 p 的递推式减少计算量，如下

$$p_{i+1} - p_i = (2\Delta y \bullet x_{i+1} - 2\Delta x \bullet \bar{y}_{i+1} + c) - (2\Delta y \bullet x_i - 2\Delta x \bullet \bar{y}_i + c)$$
$$= 2\Delta y - 2\Delta x(\bar{y}_{i+1} - \bar{y}_i)$$

- **If** $p_i \le 0$ **then** $\bar{y}_{i+1} - \bar{y}_i = 0$ **therefore**

$$p_{i+1} = p_i + 2\Delta y$$

- **If** $p_i > 0$ **then** $\bar{y}_{i+1} - \bar{y}_i = 1$ **therefore**

$$p_{i+1} = p_i + 2\Delta y - 2\Delta x$$

算法的最终步骤如下

- **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** $(x_{i+1}, \bar{y}_{i+1}) = (x_i + 1, \bar{y}_i)$

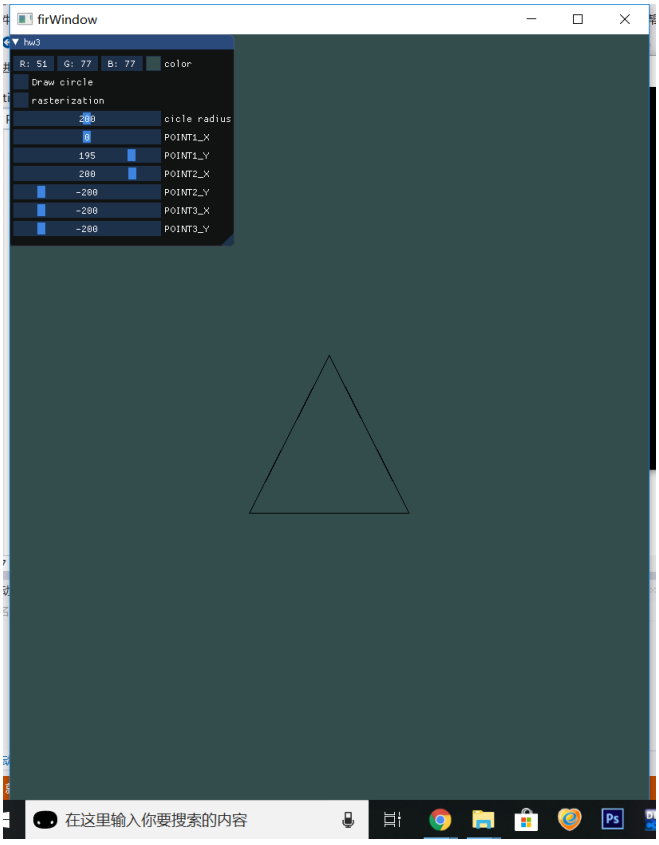
  and compute $p_{i+1} = p_i + 2\Delta y$
- **If** $p_i > 0$ **draw** $(x_{i+1}, \bar{y}_{i+1}) = (x_i + 1, \bar{y}_i + 1)$

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

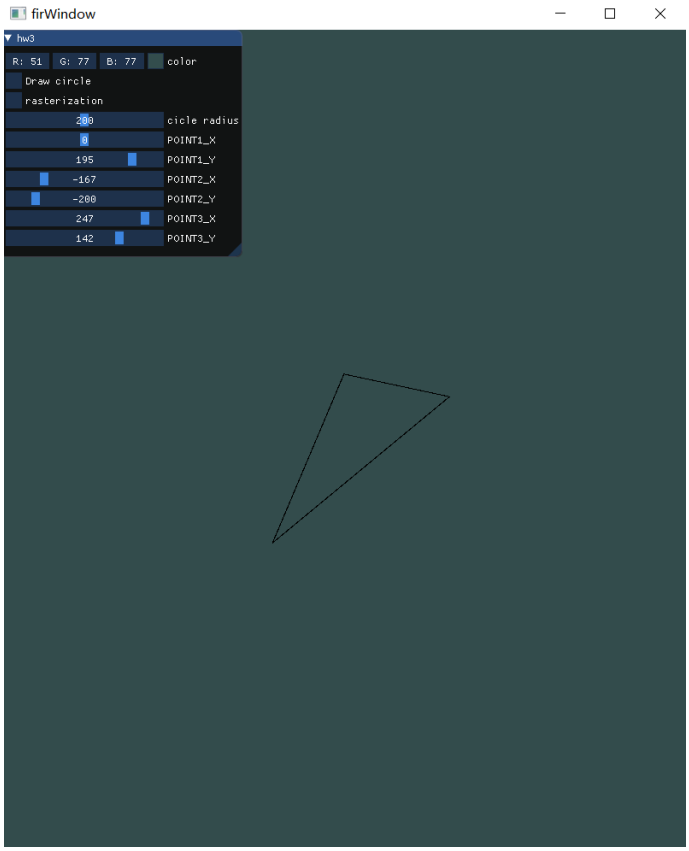
至于如果 dx 或者 dy 小于 0，只是说明他直线前进的方向不一样，这时候只要把上面公式中的+1 全部换成-1 就行了，然后如果直线的斜率大于 1，也就是和直线斜率小于 1 的那部分直线是关于 y=x 对称的，所以只要将公式中的 x 和 y 换一下就行了。

采用这个算法画三角形，就是用三个点分别两两使用上面的算法求出直线上的所有模拟点，然后像上一次作业那样，按照格式 x，y，depth，r，g，b 放在一个数组里，接着就调用 `glDrawArrays(GL_POINTS, 0, point_num);` 来将这些点全部画出来就行。(这里就略过那些 VAO,VBO 那些了，和之前是一样的)。

结果截图:



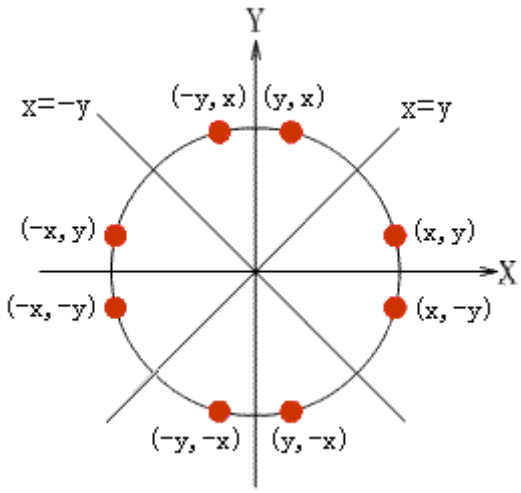
调整输入坐标



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

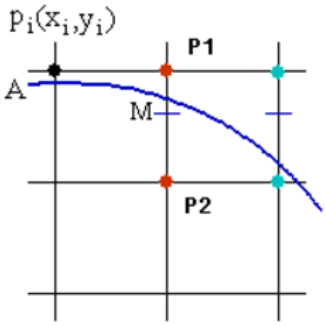
参考博客 https://blog.csdn.net/mmogega/article/details/53055625

首先，是要知道通过中点的圆的对称性，分别关于 x 轴，y 轴，y = x，y = -x 对称，所以只要得到八分之一的那部分坐标，其余都能通过这个对称性求得（x, -y）、(-x, y)、(-x, -y)、(y, x)、(y, -x)、(-y, x)、(-y, -x)。而如果是圆心不在原点的就直接通过平移移到原点就行了。



中点画圆法：在八块的第一象限第一块中，如果上一个点确定了，下一个点由于要在格点上，所以只能是（x+1,y）或者是(x+1,y-1)这两个中的一个，类似上面算法的思想，

F(x, y）= x^2 + y^2 – R^2（到圆心的距离）
当 F(x, y）= 0，表示点在圆上，当 F(x, y）> 0，表示点在圆外，当 F(x, y）< 0，表示点在圆内。如果 M 是 P1 和 P2 的中点，则 M 的坐标是（xi + 1, yi – 0.5），当 F（xi + 1, yi – 0.5）< 0 时，M 点在圆内，说明 P1 点离实际圆弧更近，应该取 P1 作为圆的下一个点。同理分析，当 F（xi + 1, yi – 0.5）> 0 时，P2 离实际圆弧更近，应取 P2 作为下一个点。当 F（xi + 1, yi – 0.5）= 0 时，P1 和 P2 都可以作为圆的下一个点，算法约定取 P2 作为下一个点。



然后又是和之前那样对这个中点到圆心距离 d 的公式推到过程了，目的是求出一个递推式

现在将M点坐标（$x_i + 1, y_i - 0.5$）带入判别函数F(x, y），得到判别式d：

$$d = F(x_i + 1, y_i - 0.5) = (x_i + 1)^2 + (y_i - 0.5)^2 - R^2$$

若d < 0，则取P_1为下一个点，此时P_1的下一个点的判别式为：

$$d' = F(x_i + 2, y_i - 0.5) = (x_i + 2)^2 + (y_i - 0.5)^2 - R^2$$

展开后将d带入可得到判别式的递推关系：

$$d' = d + 2x_i + 3$$

若d > 0，则取P_2为下一个点，此时P_2的下一个点的判别式为：

$$d' = F(x_i + 2, y_i - 1.5) = (x_i + 2)^2 + (y_i - 1.5)^2 - R^2$$

展开后将d带入可得到判别式的递推关系：

$$d' = d + 2(x_i - y_i) + 5$$

而且根据这个第一个八分之一块的第一个点是（0，r）可以推出 d0

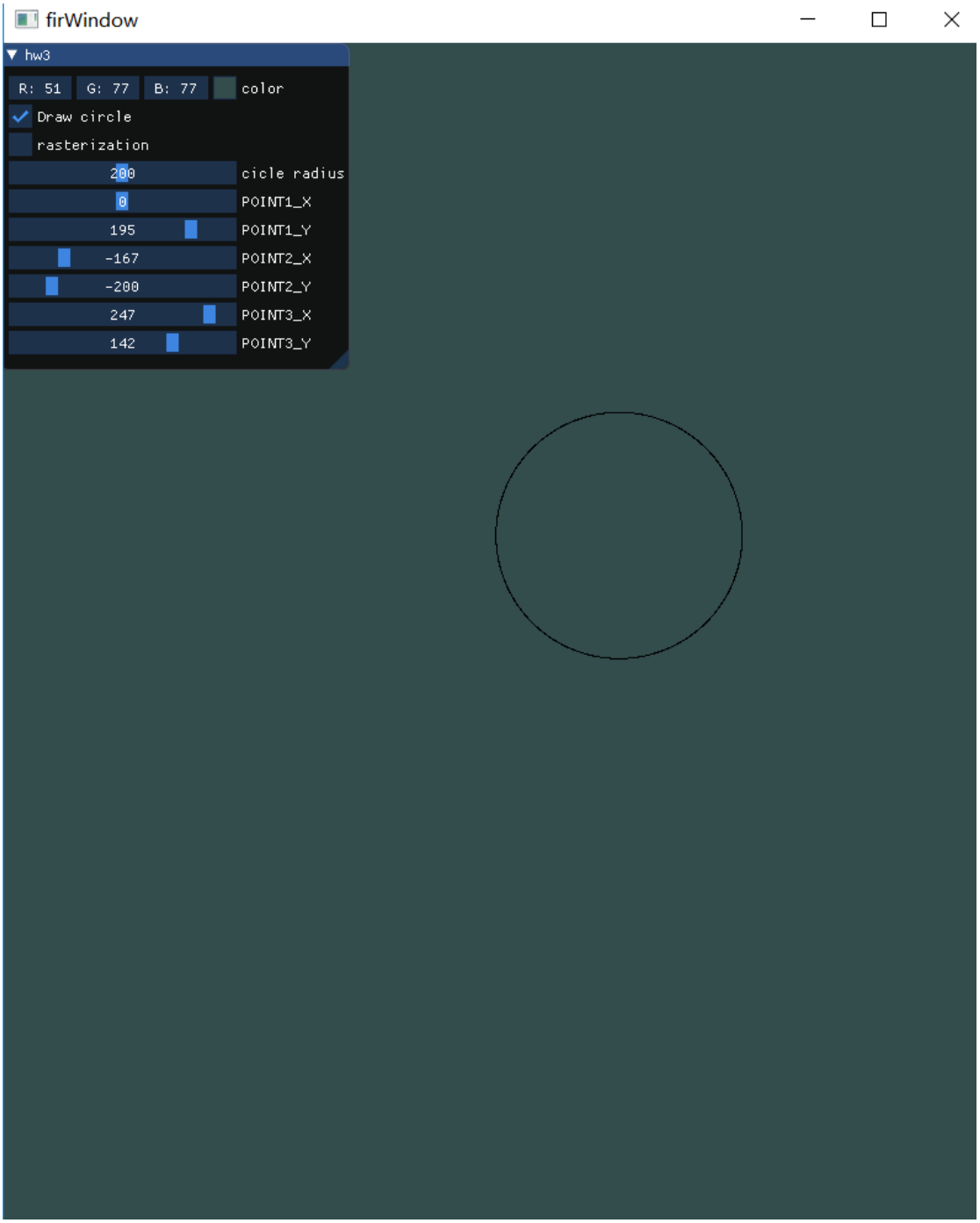
$$d_0 = F(1, R - 0.5) = 1 - (R - 0.5)^2 - R^2 = 1.25 - R$$

至此好像已经很完美了，但是为了减少使用浮点数运算，我们还要将 1.25 这个东东变成整数，由于我们只是要看 d 的符号，一种方法是将 d 的计算放大两倍，同时将初始值改成 3－2R，这样避免了浮点运算，乘二运算也可以用移位快速代替，采用 3－2R 为初始值的改进算法，又称为 Bresenham 算法，有了初值以后就可以像之前那个算法那样一步步迭代了。

```
52 void Bresenham_Circle(int xc , int yc , int r)
53 {
54    int x, y, d;
55
56    x = 0;
57    y = r;
58    d = 3 - 2 * r;
59    CirclePlot(xc , yc , x , y);
60    while(x < y)
61    {
62       if(d < 0)
63       {
64          d = d + 4 * x + 6;
65       }
66       else
67       {
68          d = d + 4 * ( x - y ) + 10;
69          y--;
70       }
71       x++;
72       CirclePlot(xc , yc , x , y);
73    }
```

具体的实现的话还是就是用上面的算法，求出那八分之一块点的坐标，然后用对称性和平移性求出所有坐标，按格式组成一个 vertice 数组，接着的画法就是把所有点画出来就行了，和画第一问三角形是一样的。（点的坐标还要进行归一化之类的操作）

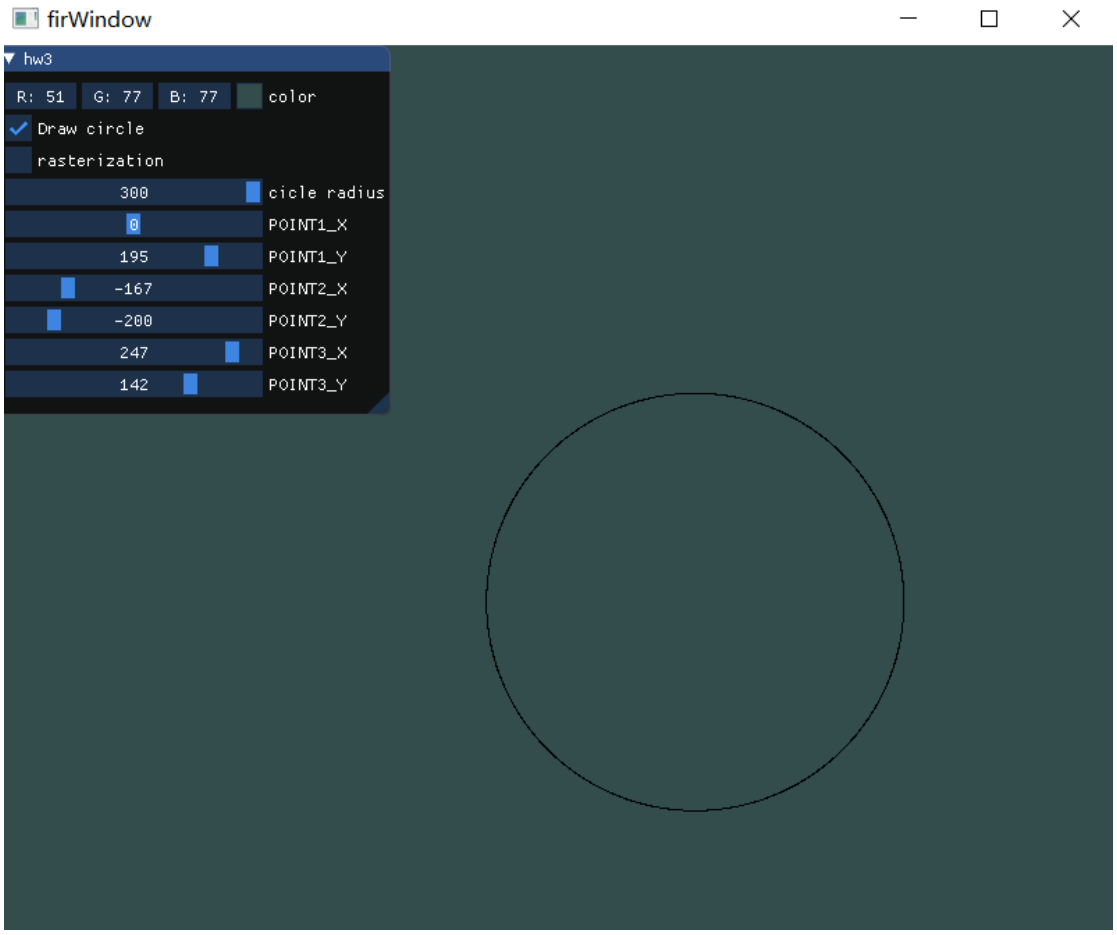
截图如下：



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)。

   这个就在 imgui 里调用 checkbox 来控制一个 bool 变量，然后分别调用不同的算法来获取点的集合然后用同样的方法画就行了。至于调整圆的大小 SliderInt 来获取半径再调用算法就行了

   截图：


firWindow
hw3
R: 51
G: 77
B: 77
color
Draw circle
rasterization
300
cicle radius
0
POINT1_X
195
POINT1_Y
-167
POINT2_X
-200
POINT2_Y
247
POINT3_X
142
POINT3_Y


firWindow
hw3
R: 51
G: 77
B: 77
color
Draw circle
rasterization
100
cicle radius
0
POINT1_X
195
POINT1_Y
-167
POINT2_X
-200
POINT2_Y
247
POINT3_X
142
POINT3_Y

Bonus:

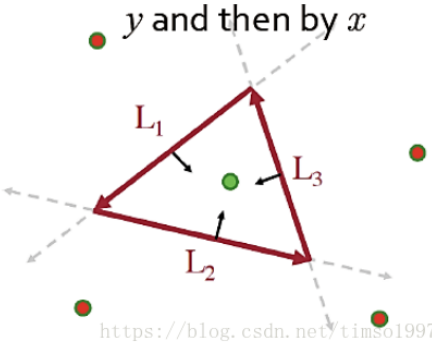
1. 使用三角形光栅转换算法，用和背景不同的颜色，填充你的三角形。

   我用的方法是 Edge Equation 的方法

Edge Equations

```
void edge_equations(vertices T[3])
{
  bbox b = bound(T);
  foreach pixel(x, y) in b {
    inside = true;
    foreach edge line Li of Tri {
      if (Li.A*x+Li.B*y+Li.C < 0) {
        inside = false;
      }
    }
    if (inside) {
      set_pixel(x, y);
    }
  }
}
```

can be rewritten to update the L's incrementally by y and then by x



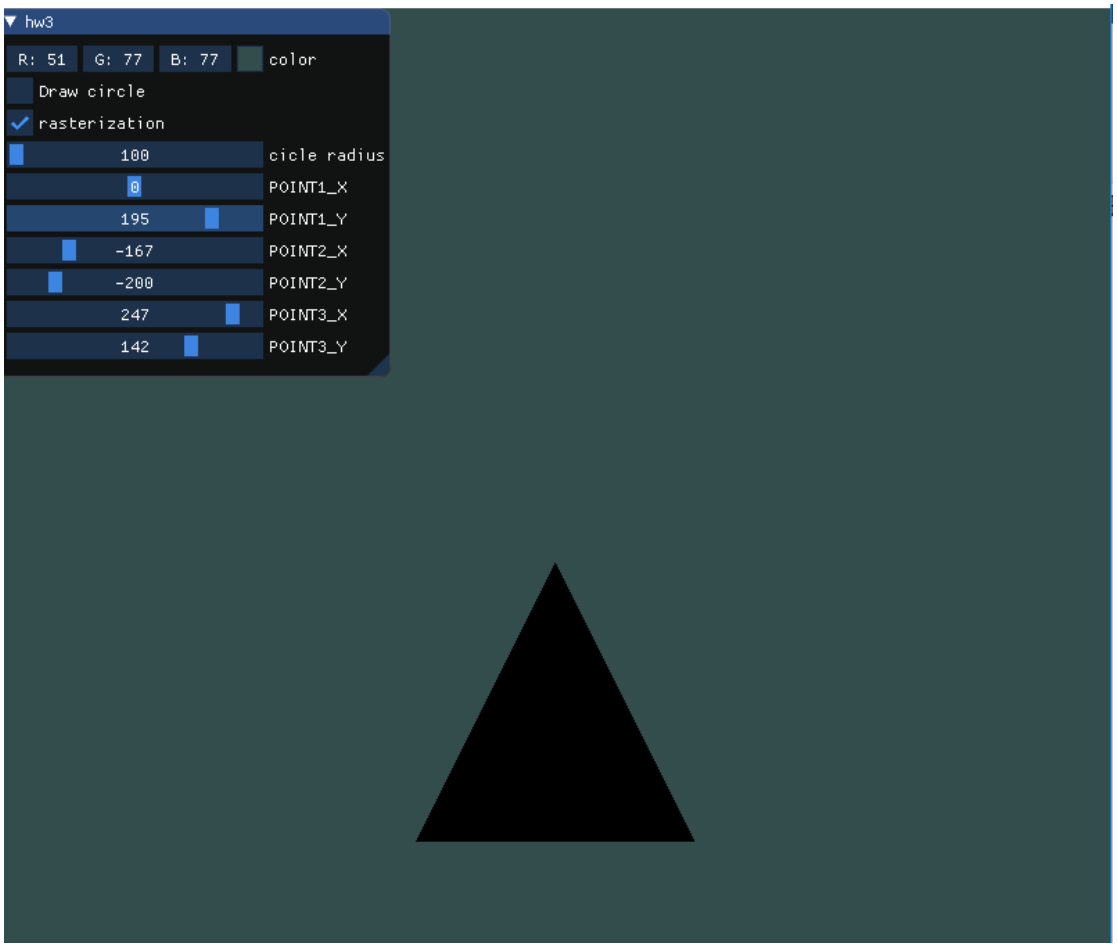
大概的思路就是如下：先通过三个顶点计算出包含三角形的一个矩形，然后扫描矩形里的每一个点，如果他在计算每个直线方程的结果都大于 0，即是在三角形里面的。在求直线方程时，仅通过两个点进行计算不能保证里面的点计算是大于 0，所以可以通过第三个点进行验证，因为第三个点肯定是在三角形里的，所以就把第三个点带入已计算出的直线方程中，如果大于 0 就满足，小于 0 只要把整个直线方程乘以-1，就是把系数乘以-1 就行了。

1. compute a bounding box:
   $x_{min}, y_{min}, x_{max}, y_{max}$ of triangle
2. compute edge equations from vertices
   - orient edge equations: let negative halfspaces be on the triangle's exterior (multiply by -1 if necessary)
   - can be done incrementally per scan line
3. scan through *each* pixel in bounding box and evaluate against all edge equations
4. set pixel if all three edge equations > 0

在实际计算的时候，我感觉用这个方法有点慢，因为要计算每一个点，很多点是没用的，所以就直接把三角形三个点写死了，然后三角形内部每个点都已经一次算好了，不在每一帧的时候进行计算，不然太慢了。

实际操作就是用上面的算法把点都求出来，然后在按格式放到数组 vertice 里，接着的画法就和之前一样了。

截图：


hw3
R: 51
G: 77
B: 77
color
Draw circle
rasterization
100
cicle radius
0
POINT1_X
195
POINT1_Y
-167
POINT2_X
-200
POINT2_Y
247
POINT3_X
142
POINT3_Y