计算机图形学 Homework3

15331416 数字媒体技术方向 赵寒旭

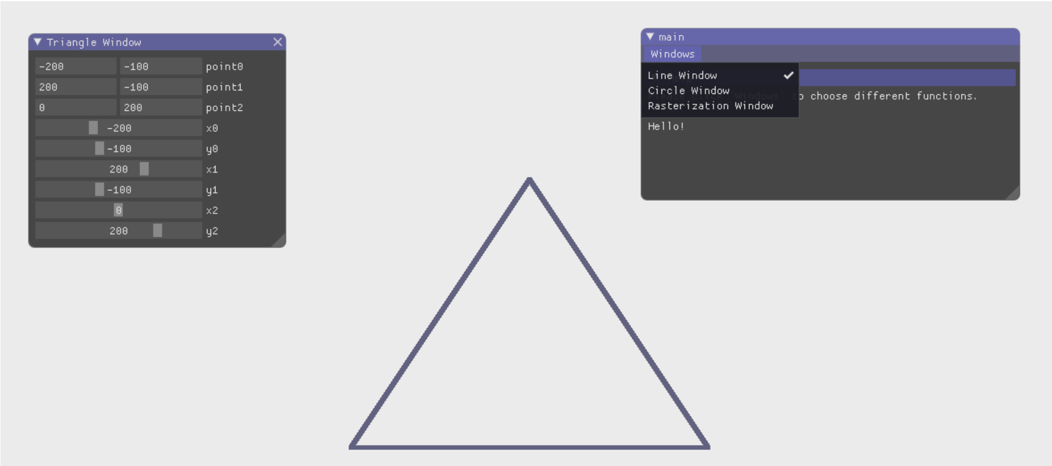
目录

1. 运行结果

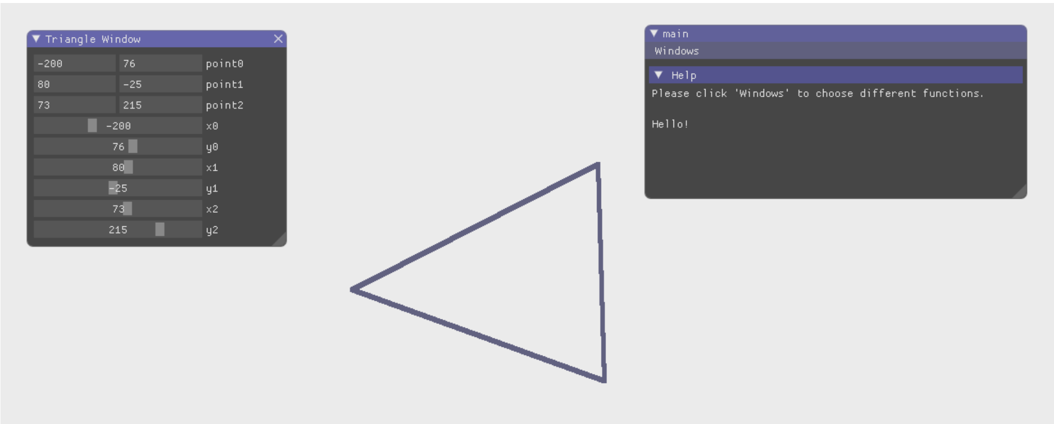
窗口大小初始化为 1200*800。

1.1 三角形边框绘制

初始状态：point0 (-200, -100), point1 (200, -100), point2 (0, 200)

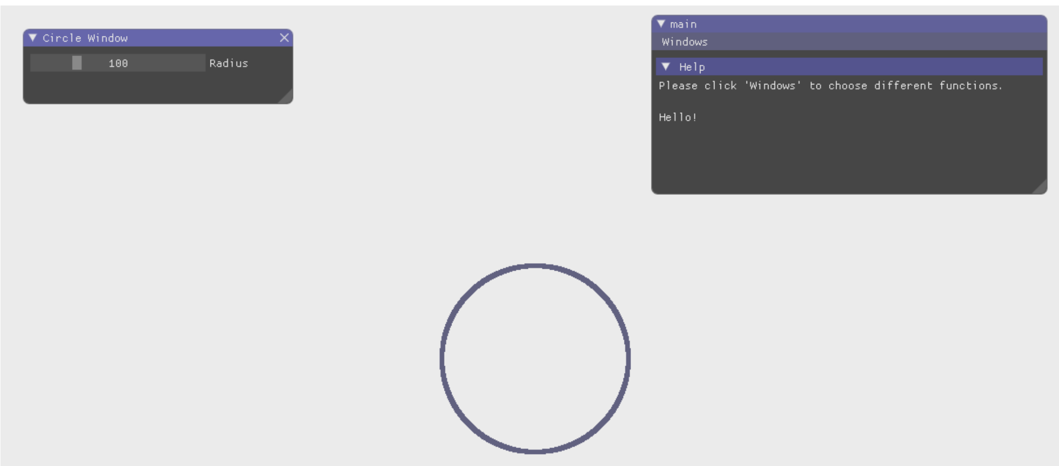


随意调整三角形顶点位置：

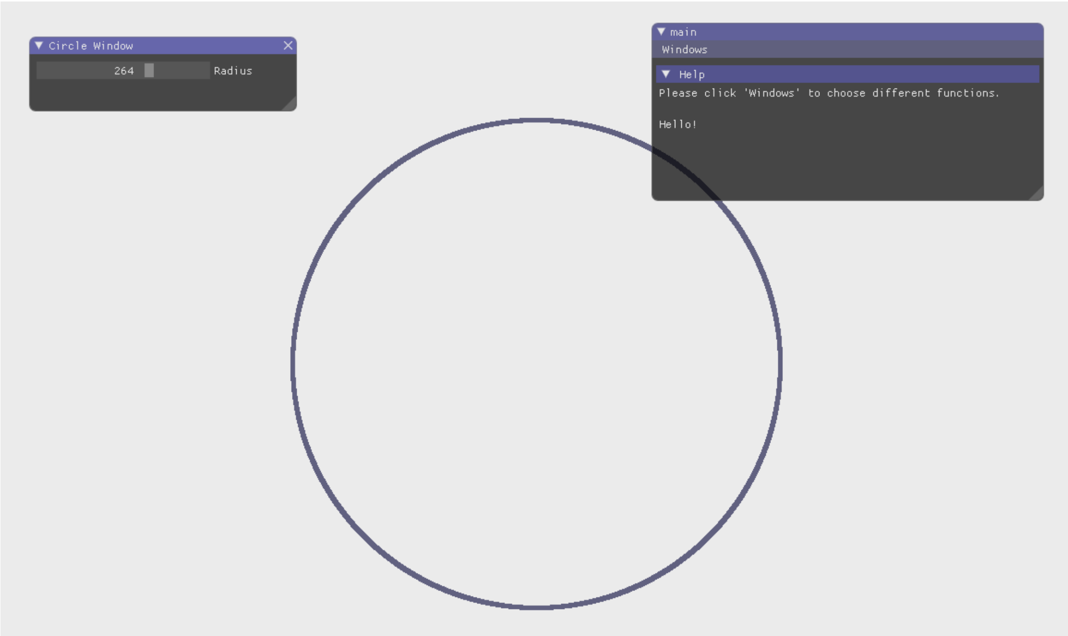


1.2 画圆及调整大小

初始状态：圆心(0, 0) 半径 100

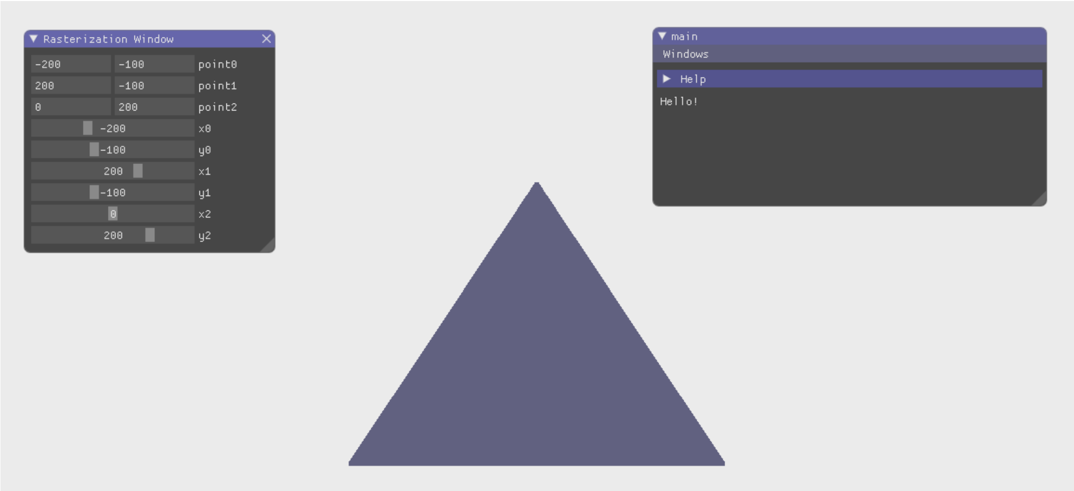


随意拉动滑块至 Radius=264

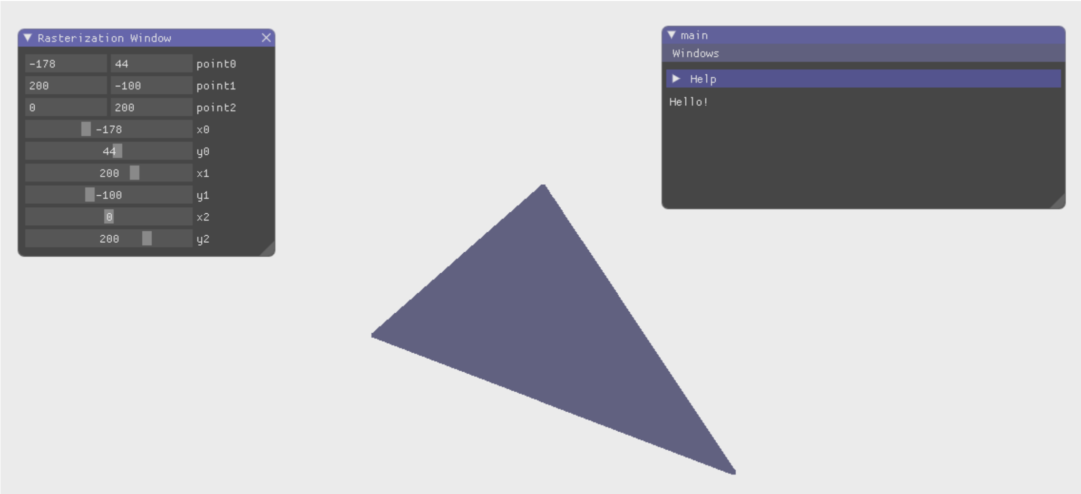


1.3 用区别于背景色的颜色填充三角形

初始状态：point0 (-200, -100), point1 (200, -100), point2 (0, 200)



随意调整三角形顶点位置：



另有演示视频在 doc/目录下。

2. 实现思路

2.1 三角形边框的绘制

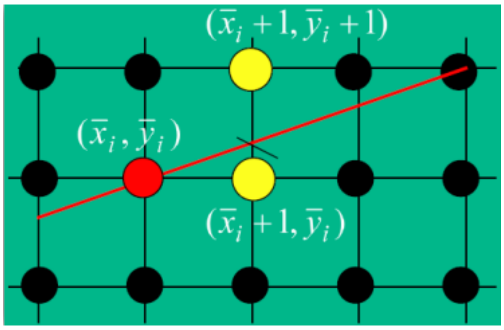
2.1.1 算法描述

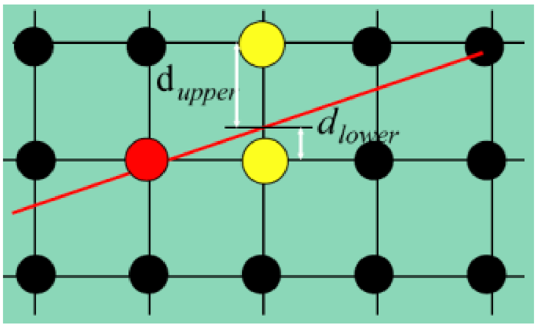
1）Bresenham 直线算法

（1）一种典型的情况

输入起点终点的坐标 A(x_0, y_0)和 B(x_1, y_1)，我们的目标是求出两点连线上的所有像素点的坐标。首先考虑一种典型的情况：$x_0 < x_1$且$0 < \mathrm{k_{AB}} < 1$：

连线如下图：





$$x_{i+1} = x_i + 1$$
$$y_{i+1} = mx_{i+1} + B$$
$$y_{i+1} = m(x_i + 1) + B$$

$$d_{upper} = \bar{y}_i + 1 - y_{i+1}$$
$$d_{lower} = y_{i+1} - \bar{y}_i$$

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

division operation

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

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

选择方法：根据y_{i+1}距离上方和下方点的距离取最接近的一个

$$\text{If } p_i \succ 0, \text{ then } \quad (\bar{x}_i + 1, \bar{y}_i + 1) \text{ is selected}$$

$$\text{If } p_i \prec 0, \text{ then } \quad (\bar{x}_i + 1, \bar{y}_i) \text{ is selected}$$

$$\text{If } p_i = 0, \ \textit{arbitrary one}$$

此时要计算p_i仍然需要做多次数乘和求和运算，为了简化运算，我们可以使用迭代的方法根据之前的数据计算出当前的p_i值。

首先计算初始值p_0：

$$p_0 = 2\Delta y \bullet x_0 - 2\boxed{\Delta x \bullet \bar{y}_0} + (2B - 1)\Delta x + 2\Delta y$$
$$= 2\Delta y \bullet x_0 - 2\boxed{(\Delta y \bullet x_0 + B \bullet \Delta x)} + (2B - 1)\Delta x + 2\Delta y$$
$$= 2\Delta y - \Delta x$$

$$y_{i+1} = mx_{i+1} + B$$

后项减前项可得：

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

可知迭代公式如下：

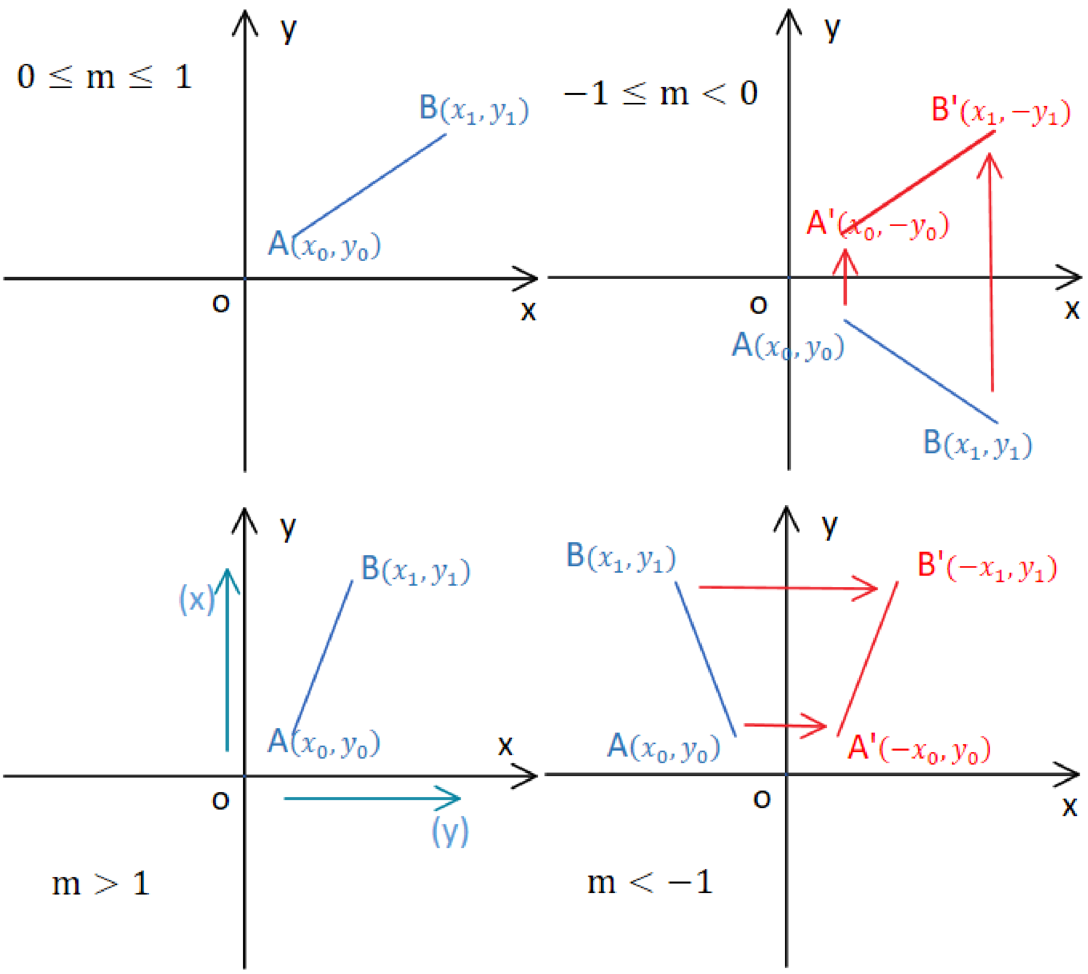
当$p_i \le 0$，有$\bar{y}_{i+1}-\bar{y}_i=0$，此时$p_{i+1}=p_i+2\Delta y$

当$p_i > 0$，有$\bar{y}_{i+1}-\bar{y}_i=1$，此时$p_{i+1}=p_i+2\Delta y-2\Delta x$

综上所述可得如下算法，用于获得连线上所有点的坐标：

① 存入起点(x_0,y_0)的 x，y 坐标，计算$length=x_1-x_0+1$，在末尾存入终点坐标(x_1,y_1)
② 计算Δx，Δy，$2\Delta x$，$2\Delta y$，$p_0=2\Delta y-\Delta x$
③ 若$p_i \le 0$，存入$(x_{i+1},\bar{y}_{i+1})=(x_i+1,\bar{y}_i)$，更新$p_i=p_i+2\Delta y$
④ 若$p_i > 0$，存入$(x_{i+1},\bar{y}_{i+1})=(x_i+1,\bar{y}_i+1)$，更新$p_i=p_i+2\Delta y-2\Delta x$
⑤ 重复③④，直到算至终点坐标前一点（$i == length - 1$时停止）

（2）其他情况的推广



$-1 \le m < 0$时，先做关于 x 轴的镜像翻转，使得符合典型情况，在计算完一系列 y 坐标后再取反得到实际值。

与前两种情况不同，在$|m|>1$时，y 坐标递增，x 坐标根据迭代关系逐步得到。此时观察图像可知，若把 y 轴看作典型情况中的 x 轴，起点 A 到终点 B 的连线符合典型情况。

特殊地，当斜率不存在时，我们可以不通过 Bresenham 算法直接递增得到 y 坐标。

2）由顶点连线组成三角形边框

由 1）所述，我们可以由两个端点生成线段上所有点的坐标，对三角形三个点两两求出连线上点的坐标即可构造出三角形。

2.1.2 代码实现

1）着色器设置

因为本次任务不涉及颜色变化，故在顶点着色器中直接定义了颜色值。

```
#version 440 core
layout (location = 0) in vec3 Position;
out vec4 vetexColor;\n"
void main()
{
   gl_Position = vec4(Position, 1.0);
   vetexColor = vec4(0.38, 0.38, 0.50, 1.0);
};
```

2）迭代坐标生成函数 getYcoordinate

```
void getYcoordinate(int* Ycoordinate, int pi, int dx, int dy, int length)
```

输入：待获得的迭代数组 Ycoordinate，初始p_0值 pi，dx，dy，数组长度 length

输出：迭代赋值完成的数组 Ycoordinate

作用：根据 Bresenham 算法循环迭代得到坐标值

```
int parm1 = 2 * dy;
    int parm2 = 2 * dy - 2 * dx;
    for (int i = 0; i < length; i++) {
        if (i == length - 1) {
            // 计算完成，此处为终点坐标
            return;
        }
        if (pi <= 0) {
            Ycoordinate[i + 1] = Ycoordinate[i];
            pi += parm1;
        }
        else {
            Ycoordinate[i + 1] = Ycoordinate[i] + 1;
            pi += parm2;
        }
    }
```

3）二维坐标生成函数 BresenhamLine

```
vector<int> BresenhamLine(int x0, int y0, int x1, int y1)
```

输入：起点 A 和终点 B 的坐标x_0，y_0，x_1，y_1

输出：逐个存放线段上所有点 x，y 坐标值的数组 p

作用：根据输入两点坐标得到线段上所有点的二维坐标

具体实现分 5 种情况分别处理，参见 2.1.1 里 Bresenham 直线算法描述中（2）其他情况的推广所描述的 4 种斜率取值范围及斜率不存在的情况。

4）归一化函数 normalize

```
vector<float> normalize(vector<int>p, int width, int height)
```

输入：存有全部 x,y 坐标值的数组 p，窗口宽度 width，窗口高度 height
输出：归一化后的 vector<float> q
作用：用于将生成的坐标数组中所有的值根据窗口大小归一到-1 到 1
由 int 转到 float 的方法：

```
q.push_back(float(p[i]) / float(width / 2));
q.push_back(float(p[i + 1]) / float(height / 2));
```

5）顶点数组生成函数 getRealCoordinate

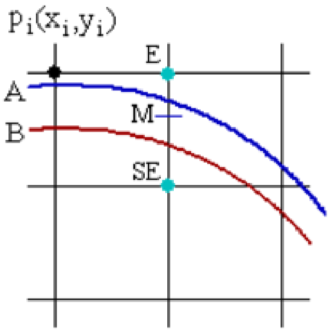
```
float* getRealCoordinate(vector<float> p, int length)
```

输入：由 BresenhamLine 并经 normalize 归一化得到的 vector<float> p 和数组长度 lengtjh
输出：含有每个点三维坐标的动态数组
作用：把含有 x,y 坐标的 vector 转换成 float*顶点数组(填充 z 值 0.0f)

```
for (int i = 0; i < length; i = i + 2) {
        line[index] = p[i];
        line[index + 1] = p[i + 1];
        line[index + 2] = 0.0f;
        index = index + 3;
    }
```

2.2 圆的绘制

2.2.1 算法描述

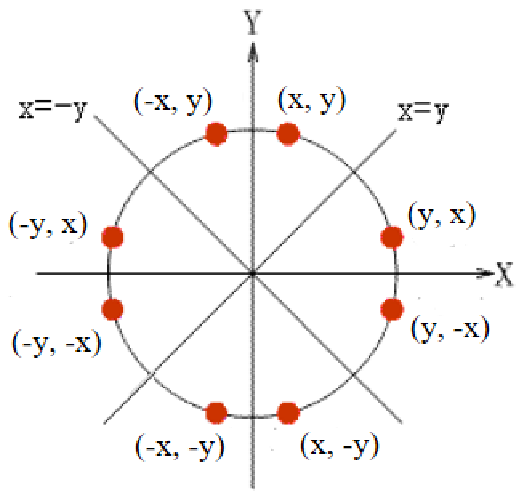


1）Bresenham 算法绘制 1/8 圆
和画直线的算法相似，取右边点和右下点的中点 M，判断点在圆内还是圆外，再判断选取 E 还是 SE。
构造判别函数$F(x,y) = x^2 + y^2 - r^2$
有判别式d：
$d = F(x_i + 1, y_i - 0.5) = (x_i + 1)^2 + (y_i - 0.5)^2 - r^2$
① 若$d < 0$，取 E 为下一个点，此时下一个判别式：
$d' = F(x_i + 2, y_i - 0.5) = (x_i + 2)^2 + (y_i - 0.5)^2 - r^2$
将d带入可得：$d' = d + 2x_i + 3$
② 若$d > 0$，取 SE 为下一个点，此时下一个判别式：
$d' = F(x_i + 2, y_i - 1.5) = (x_i + 2)^2 + (y_i - 1.5)^2 - r^2$
将 d 带入可得：$d' = d + 2(x_i - y_i) + 5$
特别地，在$(0, r)$点，可以推出判别式d的初始值d_0：

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

为了避免浮点运算，可以将 d 的计算放大两倍，同时将初始值改为$3 - 2r$，乘二运算也可以用左移一位快速代替。

2）8 分法生成整圆



只要获得圆 1/8 一段圆弧上点的坐标，就可以根据和圆心的相对位置得到整个圆的坐标。

2.2.2 代码实现

（1）圆二维坐标生成函数

```
vector<int> BresenhamCircle(int xc, int yc, int r)
```

输入：圆心坐标和半径 r

输出：逐个存放圆弧上所有点 x，y 坐标值的数组 Circle

作用：根据输入得到圆上所有点的二维坐标

```
vector<int> Circle;
int x = 0;
int y = r;
int d = 3 - (r<<1);
Circle.push_back(xc + x);
Circle.push_back(xc + y);
Circle.push_back(xc + x);
Circle.push_back(xc - y);
Circle.push_back(xc - x);
Circle.push_back(xc + y);
Circle.push_back(xc - x);
Circle.push_back(xc - y);
Circle.push_back(xc + y);
Circle.push_back(xc + x);
Circle.push_back(xc + y);
Circle.push_back(xc - x);
Circle.push_back(xc - y);
Circle.push_back(xc + x);
Circle.push_back(xc - y);
Circle.push_back(xc - x);
while (x < y) {
    if (d < 0) {
        d = d + (x<<2) + 6;
    }
    else {
        d = d + ((x - y)<<2) + 10;
        y--;
    }
    x++;
    Circle.push_back(xc + x);
    ……
```

循环中仅产生 1/8 的圆弧，其余都相对圆心自动生成。

（2）归一化和顶点数组生成

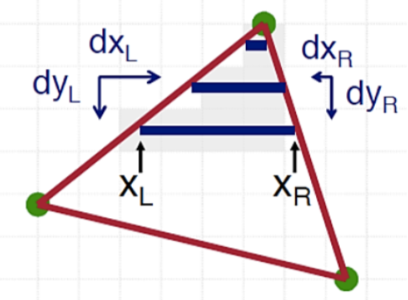
同 2.1 中方法。

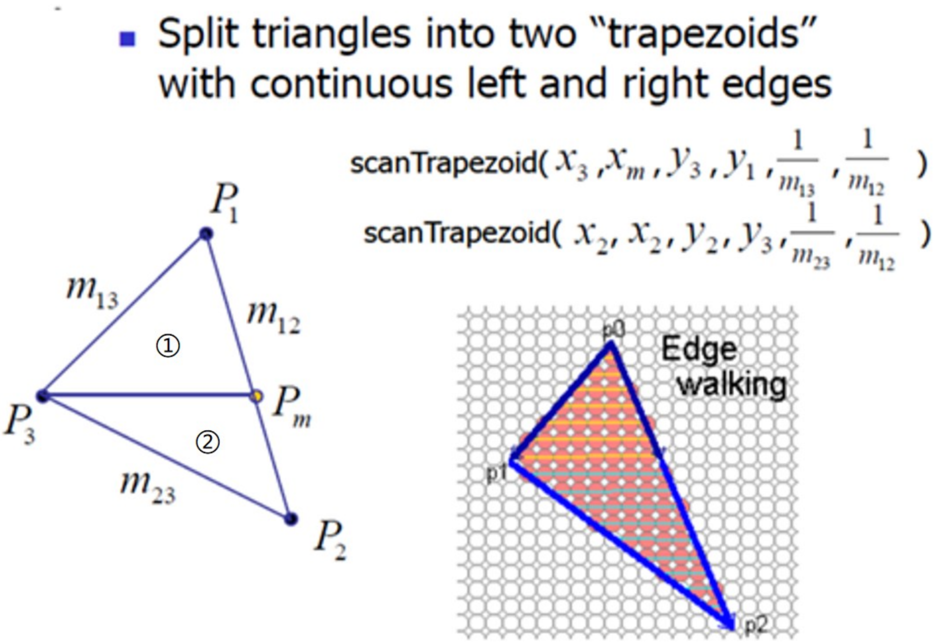
2.3 三角形光栅转换算法

2.3.1 算法描述

参考 ppt 中提到的 edge walking 算法，根据三角形的形状特点分情况绘制：

```
void edge_walking(vertices T[3])
{
  for each edge pair of T {
    initialize x_L, x_R;
    compute dx_L/dy_L and dx_R/dy_R;
    for scanline at y {
      for (int x = x_L; x <= x_R; x++) {
        set_pixel(x, y);
      }
    }
    x_L += dx_L/dy_L;
    x_R += dx_R/dy_R;
  }
}
```



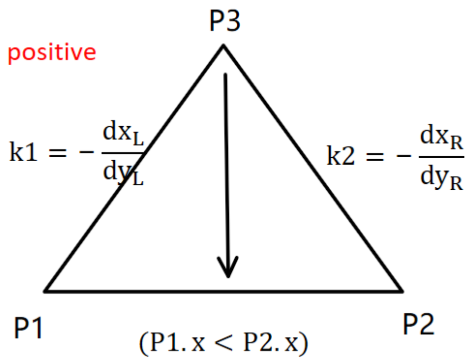


所有的三角形都可以看作是如上图①和②这两种形式三角形的组合，用 positive 形容如①形式的三角形，用 negative 来形容如②形式的三角形。

以绘制 positive 三角形为例：

$$k1 = \frac{dx_L}{dy_L} > 0$$

$$k2 = \frac{dx_R}{dy_R} < 0$$



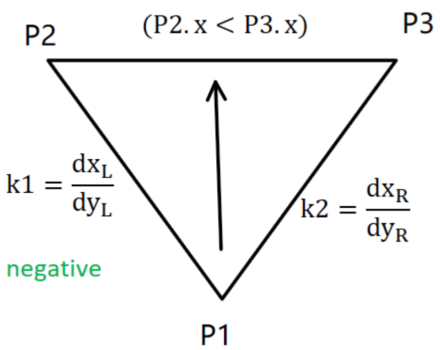
将x_L和x_R初始化为 P3.x，综合考虑客观坐标系和方向问题，取k1，k2的相反数为其赋值，我们可以得到迭代公式：

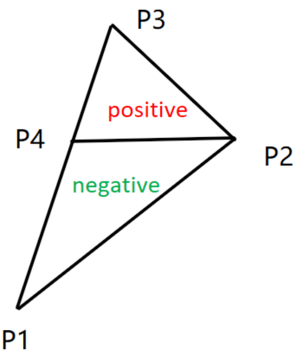
$$x_L += k1$$

$$x_R += k2$$

随着 y 值从 P3.y 到 P1.y 逐步减小，x_L逐步减小，x_R逐步增大。

每次将x_L和x_R中间的 x 坐标和对应的 y 坐标放入数组，最终得到三角形内部所有点的坐标。





绘制 negative 三角形的方法同样，此时从 P1.y 到 P2.y，k1，k2 不必取反。

绘制普通三角形时，可以看作两个特殊三角形的组合，先求出根据 P1P3 的直线方程，计算 y=P2.y 与 P1P3 的交点，求出 P4 的坐标。

$$y1 = \frac{y3 - y1}{x3 - x1}x1 + B$$

$$B = y1 - \frac{y3 - y1}{x3 - x1}x1$$

$$\text{P1P3:} \quad y = \frac{y3 - y1}{x3 - x1}x + \ y1 - \frac{y3 - y1}{x3 - x1}x1$$

代入$y = y2$：

$$x4 = \frac{(x3 - x1)(y2 - y1)}{y3 - y1} + x1$$

P4 坐标：$(x4, y2)$

2.3.2 代码实现

1) 结构体 Point

```
struct Point {
    int x;
    int y;
    Point() {
        x = 0;
        y = 0;
    }
    Point(int x0, int y0) {
        x = x0;
        y = y0;
    }
};
```

x，y 是 Point 对应的坐标值，函数 compare 使可以按照 Point 的数据成员 y 的大小用 sort 升序排序。

```
bool cpmpare(Point a, Point b) {
    return a.y < b.y;
}
```

2）positive 及 negative 三角形二维坐标生成函数

以 positive 为例：

```
vector<int> drawPositive(Point p1, Point p2, Point p3)
```

输入：3 个点

输出：逐个存放线段上所有点 x，y 坐标值的数组

作用：根据输入三点坐标得到 positive 三角形内部所有点的二维坐标

```
float xL = p3.x;
float xR = p3.x;
vector<int> coordinate;
for (int y = p3.y; y >= p1.y; y--) {
    for (int x = (int)xL; x <= (int)xR; x++) {
        coordinate.push_back(x);
        coordinate.push_back(y);
    }
    xL += k1;
    xR += k2;
}
```

3）三角形二维坐标生成函数 drawTriangle

```
vector<int> drawTriangle(Point p1, Point p2, Point p3)
```

输入：3 个点

输出：逐个存放线段上所有点 x，y 坐标值的数组

作用：根据输入三点坐标得到三角形内部所有点的二维坐标

首先按照 y 值升序排序：

```
sort(points.begin(), points.end(), cpmpare);
```

```
if (p1.y == p2.y) { // 正放三角形
    trianglePoints = drawPositive(p1, p2, p3);
}
else if (p2.y == p3.y) { // 倒放三角形
    trianglePoints = drawNegative(p1, p2, p3);
}
else { // 一正一倒拼接而成
    int x4 = (int)((float)((p3.x - p1.x)*(p2.y - p1.y)) / (float)(p3.y - p1.y)) + p1.x;
    Point p4(x4, p2.y);
    v1 = drawPositive(p4, p2, p3);
    v2 = drawNegative(p1, p4, p2);
```

分三种情况讨论，具体算法参考前文。