CG 第三次作业

软件工程数字媒体 15331101 洪鹏圳

一．代码实现思路以及主要函数解释：

1.使用 Bresenham 算法画三角形边框

步骤：

（1）我们先默认输入的顶点数据为整型，Bresenham 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**

可见其是对横坐标进行整数采样（每次递增加 1），对纵坐标进行量化（根据 Pi 的正负性来决定取值）。其中输入的两个顶点构成的斜率 m 需要满足 0 <=m<=1。

先将 Bresen 算法转换成代码，我们可以使用递推来实现该算法（第一个点和最后一个点为我们输入的直线的两个顶点）：

```
//Bresenham算法，前提是斜率m满足-1<=m<=1
void bresenham(int y[], int p, int i, int size, int dx, int dy) {
    if (i == size - 1) {
        return;
    }
    int pnext;
    if (p <= 0) {
        y[i + 1] = y[i];
        pnext = p + 2 * dy;
    }
    if (p > 0) {
        y[i + 1] = y[i] + 1;
        pnext = p + 2 * dy - 2 * dx;
    }
    bresenham(y, pnext, i + 1, size, dx, dy);
}
```

由于该算法只有在斜率 m 满足 0 <= m <= 1 的情况下成立，因此我们需要考虑以下几种情况：
①斜率 m 不存在的情况，即直线与 y 轴平行或者重合。
此时要求出直线上的每一个点，只需将每个点的横坐标都设成直线起点的横坐标，纵坐标每次加 1 递增到直线终点（默认起点的纵坐标小于终点的纵坐标）：

```
//斜率不存在的情况，即直线与y轴平行或者重合
if (x0 == x1) {
    //保证y1始终大于或等于y0
    if (y0 > y1) {
        int temp = y0;
        y0 = y1;
        y1 = temp;
    }
    size = y1 - y0 + 1;//计算直线总点数
    //计算直线上每个点的横纵坐标
    for (int i = 0; i < size; i++) {
        x[i] = x0;
        y[i] = y0 + i;
    }
}
```

②斜率 m 满足 0 <= m <= 1 的情况。
此时可以直接使用 Bresenham 算法，默认起点的横坐标值 x0 小于终点的横坐标 x1。先通过 x1 - x0 + 1 求出直线上的总点数，对于直线上的每一点，横坐标的值每次递增加 1。通过 bresenham 算法求出直线上除了起点和终点之外其他点的纵坐标：

```
float m = float(y1 - y0) / float(x1 - x0);
//斜率m满足-1 <= m <= 1
if (fabs(m) <= 1) {
    //默认x1大于x0
    if (x0 > x1) {
        int temp = x0;
        x0 = x1;
        x1 = temp;
        temp = y0;
        y0 = y1;
        y1 = temp;
    }
    size = x1 - x0 + 1;//计算直线总点数
    //计算直线上每点的横坐标
    for (int i = 0; i < size; i++) {
        x[i] = x0 + i;
    }
    //如果斜率m满足0 <= m <= 1
    if (m >= 0 && m <= 1) {
        y[0] = y0;
        y[size - 1] = y1;
        dx = x1 - x0;
        dy = y1 - y0;
        int p0 = 2 * dy - dx;
        bresenham(y, p0, 0, size, dx, dy);
    }
```

③斜率 m 满足-1<=m<0 的情况。

此时只需要把 m 看成 0<=m<=1（先将直线关于 x 轴对称，把直线的起点纵坐标 y0 和直线终点纵坐标 y1 取反），使用 Bresenham 求出所要求的直线关于 x 轴对称的直线 l'，再做一次关于 x 轴对称即可得到所要求的直线（将 l'上所有点的纵坐标取反）：

```
    //如果斜率m满足-1 <= m < 0,只需要把m看成0 <= m <= 1再关于x轴对称即可
    else {
        y[0] = -1 * y0;
        y[size - 1] = -1 * y1;
        dx = x1 - x0;
        dy = -(y1 - y0);
        int p0 = 2 * dy - dx;
        bresenham(y, p0, 0, size, dx, dy);
        //关于x轴对称
        for (int i = 0; i < size; i++) {
            y[i] *= -1;
        }
    }
```

④斜率 m 满足 m>1 的情况。

因为|m|>1，y 轴上的变化从尺度要比 x 轴上的大，所以应该在 y 轴上进行整数取样，在 y 轴上进行量化。此时我们只需要把 x 看成 y，把 y 看成 x，直线上的纵坐标每次递增加 1，将横坐标数组作为 Bresenham 函数第一个参数传入即可求出直线上所有点的横坐标：

```
//斜率m满足m > 1 或 m < -1，此时把x看成y，y看成x即可
else {
    //默认y1大于y0
    if (y0 > y1) {
        int temp = y0;
        y0 = y1;
        y1 = temp;
        temp = x0;
        x0 = x1;
        x1 = temp;
    }
    size = y1 - y0 + 1;
    //计算直线上每点的纵坐标
    for (int i = 0; i < size; i++) {
        y[i] = y0 + i;
    }
    //如果斜率m满足m > 1
    if (m >= 1) {
        x[0] = x0;
        x[size - 1] = x1;
        dx = x1 - x0;
        dy = y1 - y0;
        int p0 = 2 * dx - dy;
        bresenham(x, p0, 0, size, dy, dx);
    }
```

⑤斜率 m 满足 m<-1 的情况，此时和情况③一样将直线关于 x 轴对称即可得到情况④，求出直线上每个点的横坐标后再关于 x 轴对称即可求出直线上每一点的横纵坐标：

```
        //如果斜率m满足m < -1,只需要把m看成m > 1再关于x轴对称即可
        else {
            x[0] = -1 * x0;
            x[size - 1] = -1 * x1;
            dx = -(x1 - x0);
            dy = y1 - y0;
            int p0 = 2 * dx - dy;
            bresenham(x, p0, 0, size, dy, dx);
            //关于x轴对称
            for (int i = 0; i < size; i++) {
                x[i] *= -1;
            }
        }
```

此时我们已经把全部情况讨论完毕，只需要将直线起点横纵坐标和直线终点横纵坐标输入便可求出直线上每一点的坐标。

（2）因为我们输入的点的坐标值为整数，通过 bresenham 算法求出来的直线上的点坐标值也为整数，而 OpenGL 默认显示坐标范围是[-1,1]区间年内的浮点数。因此我们需要把顶点横纵坐标归一化到[-1,1]。我假设输入的顶点坐标范围为[-height,height]（height 为窗口的宽和高），因此可以通过将坐标值除 height 得到[-1,1]范围内的浮点数坐标，归一化函数如下：

```
//归一化函数，将坐标的x值和y值归一化到[-1,1]
float normalize(int input) {
    return float(input) / height;
}
```

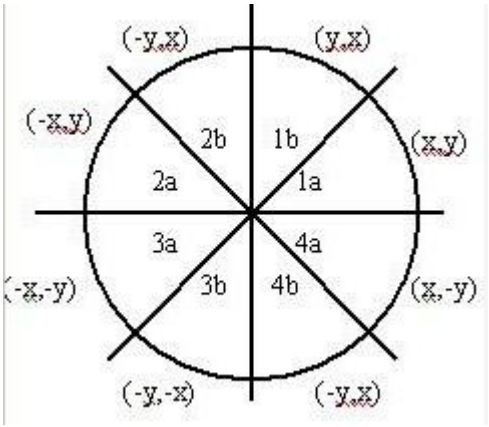
（3）要画出三角形只需要把三角形三个顶点做 bresenham 算法再归一化便可以得到三条直线上的所有点，此时通过调用画点函数和画线函数可以画出三角形的三条边：

```
//画点函数
void drawPoints(float fx, float fy, Shader shader) {
    float vertices[] = {
        fx, fy, 0.0f,   0.0f, 0.0f, 0.0f
    };
    unsigned int points_VBO;//顶点缓冲对象
    unsigned int points_VAO;//顶点数组对象
    glGenVertexArrays(1, &points_VAO);//生成一个VAO对象
    glGenBuffers(1, &points_VBO);//生成一个VBO对象
    glBindVertexArray(points_VAO);//绑定VAO
    //把顶点数组复制到缓冲中供OpengGL使用
    glBindBuffer(GL_ARRAY_BUFFER, points_VBO);//把新创建的缓冲VBO绑定到GL_ARRAY_BUFFER目标上
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);//把之前定义的顶点数据points_vertices

    //链接顶点属性
    //位置属性，值为0
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);//解析顶点数据
    glEnableVertexAttribArray(0);
    //颜色属性，值为1
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));//解析顶点数据
    glEnableVertexAttribArray(1);
    shader.use();//激活着色器程序对象
    glBindVertexArray(points_VAO);//绑定VAO
    glDrawArrays(GL_POINTS, 0, 1);//绘制图元
    glBindVertexArray(0);
}
```

```
//画线函数
void drawLines(int size, float fx[], float fy[], Shader shader) {
    for (int i = 0; i < size; i++) {
        drawPoints(fx[i], fy[i], shader);
    }
}
```

2.使用 Bresenham 算法画圆

假设圆的圆心位于坐标原点（如果圆心不在原点则可以通过坐标平移平移至原点），半径为 R。以原点为圆心的圆 C 有 4 条对称轴:x = 0,y = 0,x = y,x = -y。如果已知圆弧上一点 P1=C(x,y)，利用对称性便可以得到关于四条对称轴的其他 7 个点，即：



这种性质成为八对称性。

因此只需要扫描转换八分之一圆弧，就可以通过圆弧的八对称性得到整个圆。

先算出从(0, r)到(r/√2, r/√2)之间的 1/8 圆弧的点的坐标值，其他部分直接用轴对称和中心对称变换，最后画出完整的圆形。

也就是计算出那特定的 1/8 圆弧上的点的坐标(x,y)，再绘制出相应的(x, -y)，(-x, y)，(-x, -y)，(y, x)，(-y, x)，(y, -x)，(-y, -x)。

而那 1/8 圆弧上因为 x 上的变化比 y 大，所以在 x 方向上取样，在 y 方向上量化。递推思想就是：

d0=1-r

if d<0　　d=d+2*x+3

if d>=0　　d=d+2*(x-y)+5, y=y-1

x=x+1

需要注意的两点是：

（1）需要把坐标值归一化到[-1,1]才能进行点的绘制。

（2）如果圆心不在原点，需要把圆心平移到原点。

实现的 Bresenham 画圆的算法代码如下：

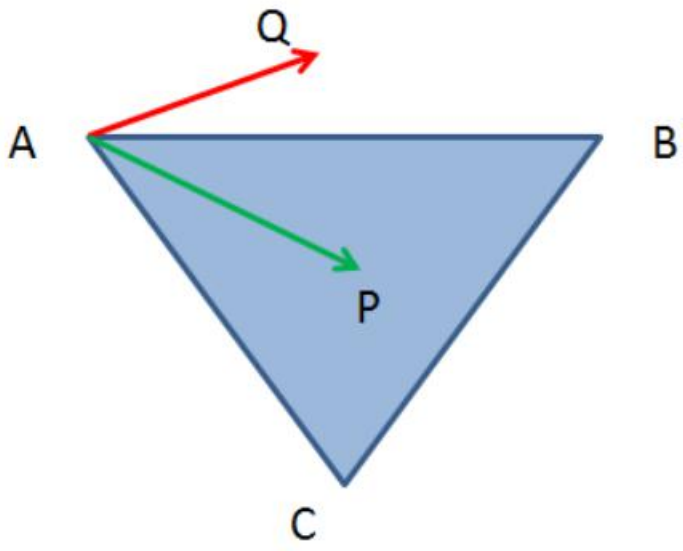
```
//画圆函数
void drawCircle(int r, float cx, float cy, Shader shader) {
    //归一化
    float fcx = normalize(cx);
    float fcy = normalize(cy);

    int x = 0, y = r;
    int d = 1 - r;//起点(0,R),下一中点(1,R - 0.5),d=1*1+(R-0.5)*(R-0.5)-R*R=1.25-R,d只参与整数运算，所以小数部分可省略

    while (y >= x) {
        float fx = normalize(x);
        float fy = normalize(y);

        //绘制点(x,y) (-x,-y) (-x,y) (x,-y)
        drawPoints(fcx + fx, fcy + fy,shader);
        drawPoints(fcx - fx, fcy - fy, shader);
        drawPoints(fcx - fx, fcy + fy, shader);
        drawPoints(fcx + fx, fcy - fy, shader);

        //绘制点(x,y) (-x,-y) (-x,y) (x,-y)
        drawPoints(fcx + fy, fcy + fx, shader);
        drawPoints(fcx - fy, fcy - fx, shader);
        drawPoints(fcx - fy, fcy + fx, shader);
        drawPoints(fcx + fy, fcy - fx, shader);

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

3. 使用三角形光栅转换算法填充三角形

这里我使用的是同向法：

假设点 P 位于三角形内，会有这样一个规律，当我们沿着 ABCA 的方向在三条边上行走时，会发现点 P 始终位于边 AB，BC 和 CA 的右侧。我们就利用这一点来判断点 P 的位置。

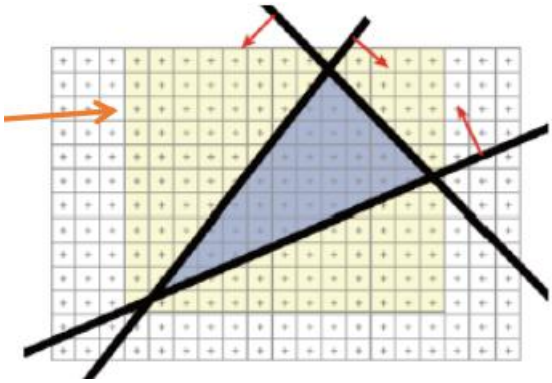
当选定线段 AB 时，点 C 位于 AB 的右侧，同理选定 BC 时，点 A 位于 BC 的右侧，最后选定 CA 时，点 B 位于 CA 的右侧，所以当选择某一条边时，我们只需验证点 P 与该边所对的点在同一侧即可。



（1）我们可以先求出三角形三条边的方程，然后利用同向法来判断。边的方程我记为:
kx - y + b = 0。k 即直线的斜率，假设直线两个顶点分别为 A(x1,y1),B(x2,y2)，因此有:
k = (y2 - y1)/(x2 - x1)
b = y2 - kx2
直线 AB 的方程 fc(x,y)即为(y2 - y1)/(x2 - x1) * x - y + y2 - kx2 = 0
求直线方程的代码如下：

```
//直线方程为kx - y + b = 0
float k1 = (float)(point3[1] - point2[1]) / (float)(point3[0] - point2[0]);
float k2 = (float)(point1[1] - point3[1]) / (float)(point1[0] - point3[0]);
float k3 = (float)(point2[1] - point1[1]) / (float)(point2[0] - point1[0]);

float b1 = (float)point2[1] - k1 * point2[0];
float b2 = (float)point3[1] - k2 * point3[0];
float b3 = (float)point1[1] - k3 * point1[0];
```

（2）为了加快点的遍历，首先我们要求出包围三角形的最小矩形，对这个矩形里的点遍历并判断是否出现在三角形内。



如图，我们只需求出三个顶点最小横纵坐标和最大横纵坐标便可得到最小包围矩形的四个顶点：

```
//找出包围三角形的最小矩形
int xMIN, xMAX, yMIN, yMAX;//最小矩形的四个顶点
xMIN = getMin(point1[0], point2[0], point3[0]);
xMAX = getMax(point1[0], point2[0], point3[0]);
yMIN = getMin(point1[1], point2[1], point3[1]);
yMAX = getMax(point1[1], point2[1], point3[1]);
```

（2）假设三角形的三个顶点分别为 A,B,C，要判断的点为 P。三角形的三条边分别为：
BC: fa(x,y)=0
AC: fb(x,y)=0
AB: fc(x,y)=0
以 AB 为例，在三角形内的点 P 必须与点 C 在 AB 的同侧，即 fc(p1,p2)*fc(c1,c2)>0
记 u、v、w 为下式：
u = fa(x,y)*fa(a1,a2)
v = fb(x,y)*fb(b1,b2)
w = fc(x,y)*fc(c1,c2)
则由 u、v、w 三个数的符号可以确定任意点 P(x,y)与三角形 ABC 的位置关系：

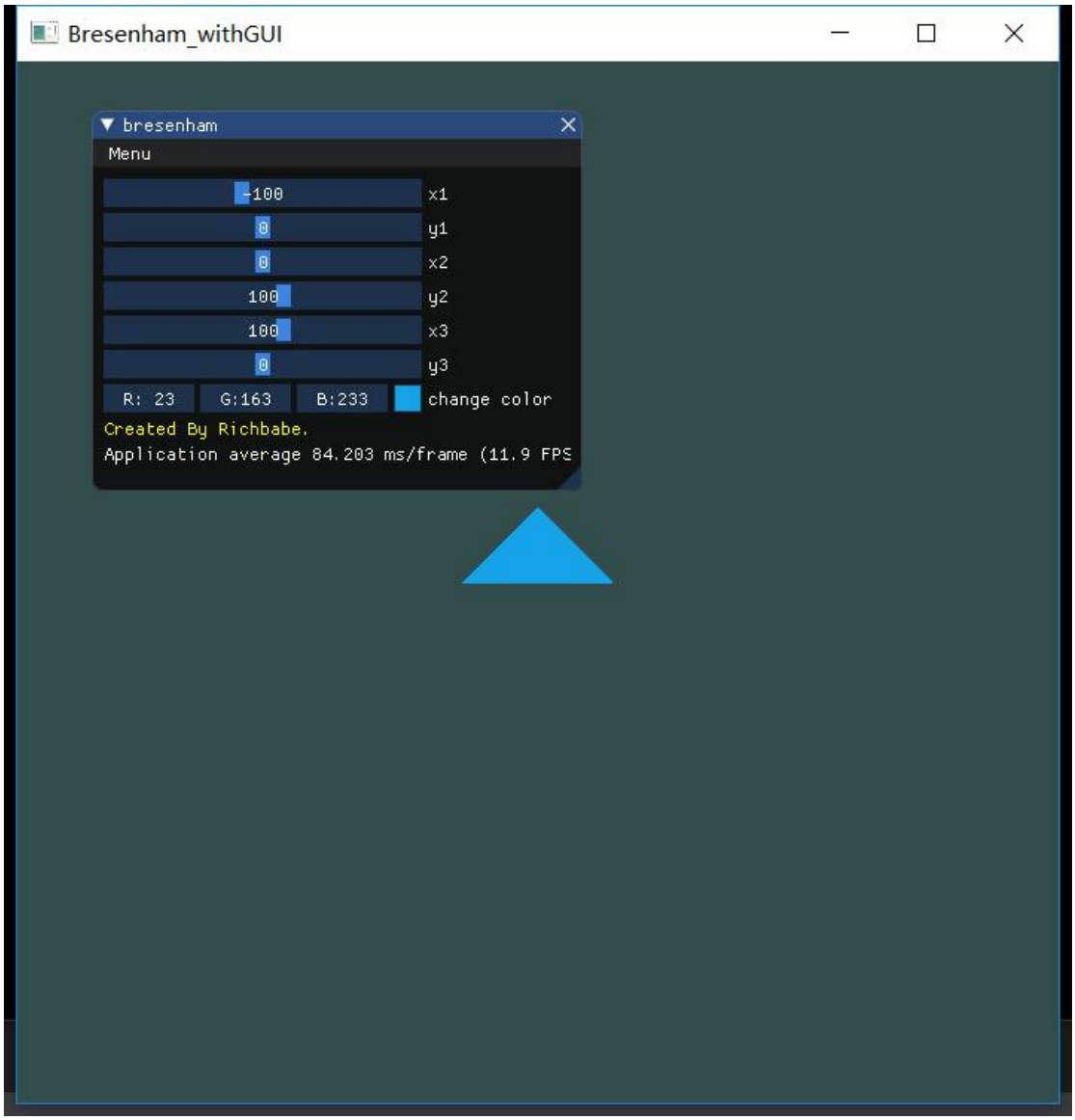
1.三个数都是正数：P 在三角形内；
2.至少有一个负数：P 在三角形外；
3.有且只有一个 0，另两个为正数：P 在三角形边上；
4.有二个 0：在三角形的顶点上。

我们使用关系 2 来进行判断：

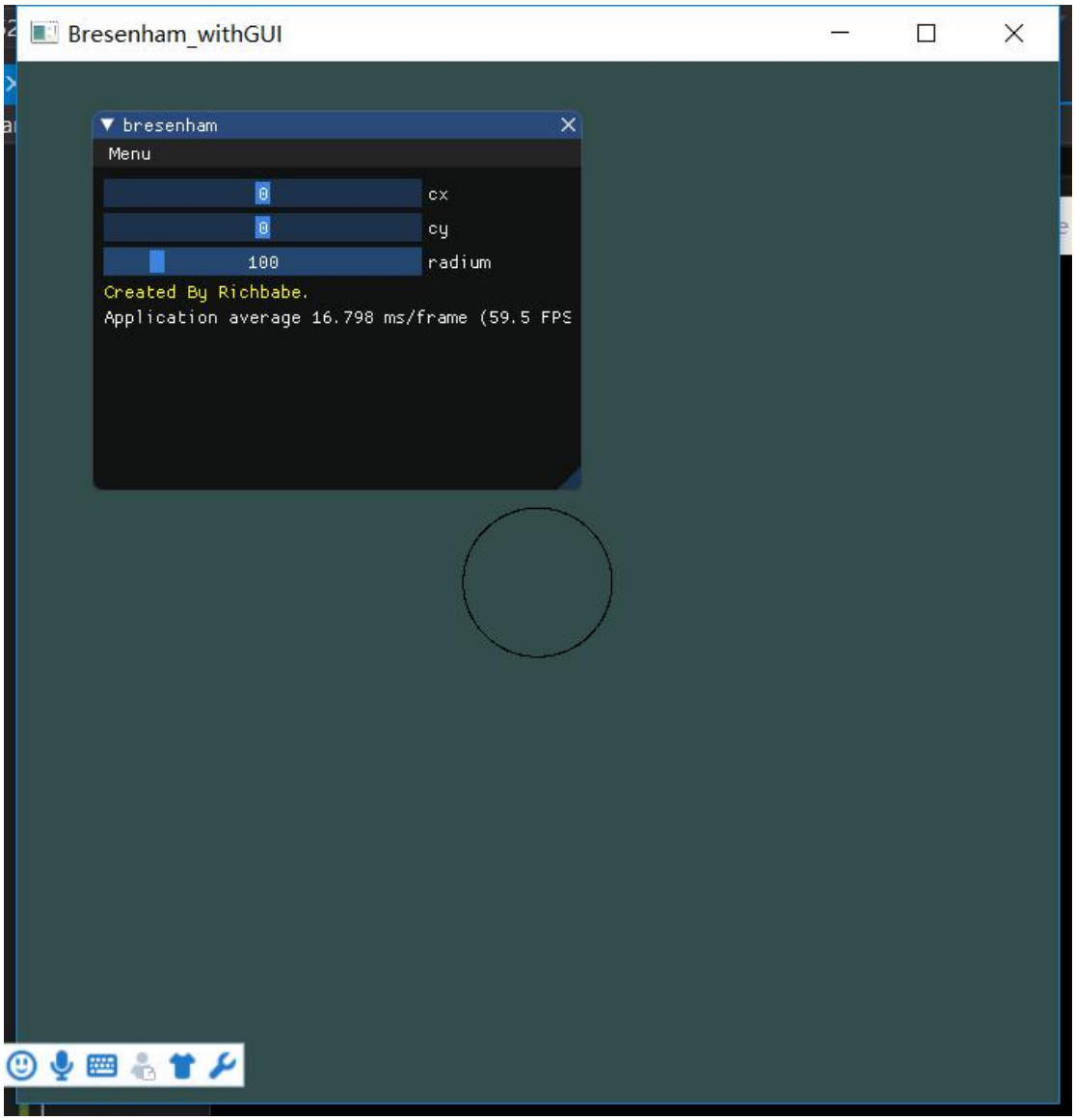
```
//遍历矩形中的每个点，如果点在三角形中则填充颜色
for (int i = xMIN; i <= xMAX; i++) {
    for (int j = yMIN; j <= yMAX; j++) {
        float u = (k1 * i - j + b1) * (k1 * point1[0] - point1[1] + b1);
        float v = (k2 * i - j + b2) * (k2 * point2[0] - point2[1] + b2);
        float w = (k3 * i - j + b3) * (k3 * point3[0] - point3[1] + b3);
        //在三角形中
        if (!(u < 0.0f || v < 0.0f || w < 0.0f)) {
            //归一化
            float fx = normalize(i);
            float fy = normalize(j);

            //填充颜色

            float vertices[] = {
                fx, fy, 0.0f,   color[0], color[1], color[2]
            };
            unsigned int points_VBO;//顶点缓冲对象
            unsigned int points_VAO;//顶点数组对象
            glGenVertexArrays(1, &points_VAO);//生成一个VAO对象
            glGenBuffers(1, &points_VBO);//生成一个VBO对象
            glBindVertexArray(points_VAO);//绑定VAO
            //把顶点数组复制到缓冲中供OpengGL使用
            glBindBuffer(GL_ARRAY_BUFFER, points_VBO);//把新创建的缓冲VBO绑定到GL_ARRAY_BUFFER目标上
            glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);//把之前定义的顶点

            //链接顶点属性
            //位置属性，值为0
            glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);//解析顶点数据
            glEnableVertexAttribArray(0);
            //颜色属性，值为1
```

如果点不在三角形外（即在三角形边上或者三角形内），则绘制该点。
遍历完最小包围矩形的每个点，我们即可填充我们的三角形。

二．运行结果截图
（1）通过 Bresenham 算法和同向法算法绘制填充的三角形



（2）通过 Bresenham 算法绘制圆


Bresenham_withGUI
bresenham
Menu
0
cx
0
cy
100
radium
Created By Richbabe.
Application average 16.798 ms/frame (59.5 FPS