作业 3

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 算法

由于 Bresenham 算法只支持斜率为 0-1 的直线，所以当 abs(y2-y1)>abs(x2-x1)时，将 x 与 y 互换后再进行计算。而当 x2 小于 x1 或 y2 小于 y1 时，对直线的方向进行判定选择。

```
int dx = abs(x2 - x1);
int dy = abs(y2 - y1);
bool isLegal = true;
if (dy > dx) {
    swap(&x1, &y1);
    swap(&x2, &y2);
    swap(&dx, &dy);
    isLegal = false;
}
int xi = (x2 - x1) > 0 ? 1 : -1;
int yi = (y2 - y1) > 0 ? 1 : -1;
```

而 Bresenham 算法的主要内容则是如下图一所示。下图二和三的代码则是参照下图一进行实现。区别在于之前是否有进行 x 和 y 的交换。

- **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**

```
if (isLegal) {
    (*xVec).push_back(x);
    (*yVec).push_back(y);
    while (x != x2) {
        if (d <= 0) {
            d += dy * 2;
        }
        else {
            d += (dy - dx) * 2;
            y += yi;
        }
        x += xi;
        (*xVec).push_back(x);
        (*yVec).push_back(y);
    }
}
```

```
else {
    (*xVec).push_back(y);
    (*yVec).push_back(x);
    while (x != x2) {
        if (d <= 0) {
            d += dy * 2;
        }
        else {
            d += (dy - dx) * 2;
            y += yi;
        }
        x += xi;
        (*xVec).push_back(y);
        (*yVec).push_back(x);
    }
}
```

其余主要代码

normalizeWidth && normalizeHeight

将输入的数值进行标准化，转换为-1.0f~1.0f。

```
float normalizeWidth(int num) {
    float answer = (float)(num - (SCR_WIDTH / 2 - 1)) / (SCR_WIDTH / 2 - 1);
    if (answer > 1.0f)
        answer = 1.0f;
    return answer;
}
```

```
float normalizeHeight(int num) {
    float answer = (float)(num - (SCR_HEIGHT / 2 - 1)) / (SCR_HEIGHT / 2 - 1);
    if (answer > 1.0f)
        answer = 1.0f;
    return answer;
}
```

顶点数组

由于 c++传递的数组会转换成指针类型，所以在 show()函数内声明了一个数组储存 main 函数传递的数据并进行使用。最后使用 GL_POINTS 画点

```
//顶点输入
float vertices[20000] = { 0 };
for (int i = 0; i < length * 6; i++) {
    vertices[i] = points[i];
}
```

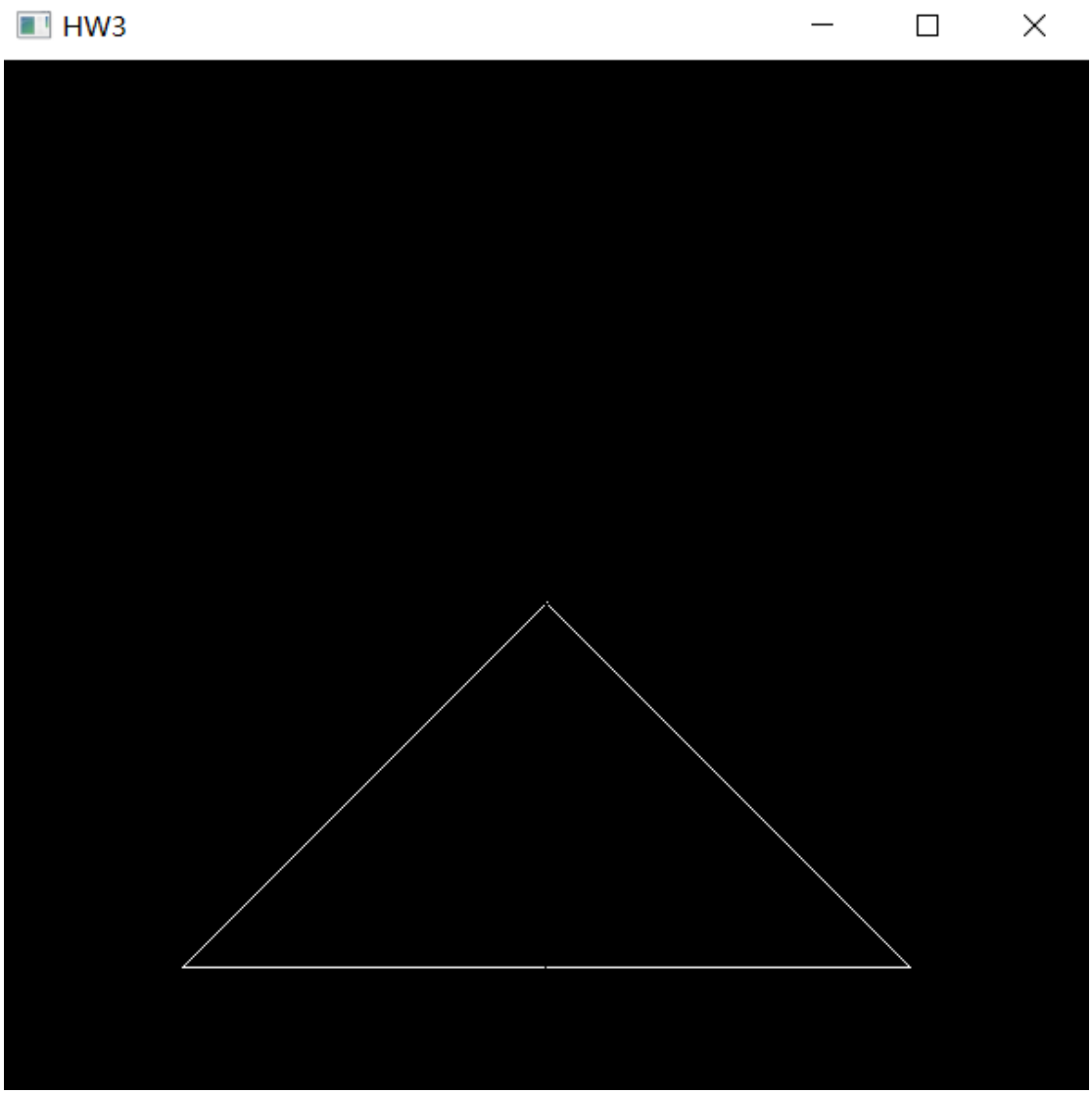
```
//把新创建的缓冲绑定到GL_ARRAY_BUFFER
glBindBuffer(GL_ARRAY_BUFFER, VBO);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

```
//画三角形
glUseProgram(shaderProgram);
glBindVertexArray(VAO);
glDrawArrays(GL_POINTS, 0, length);
```

结果

程序运行一开始需要输入三个点的坐标。输入坐标后即可得到结果。

```
Please input the first point(like:x y)(x in(0,599), y in(0,599))
300 300
Please input the second point(like:x y)(x in(0,599), y in(0,599))
100 100
Please input the third point(like:x y)(x in(0,599), y in(0,599))
500 100
```



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。

本题代码与上题类似，主要变换为修改了 Bresenham 算法进行画圆。

Bresenham 算法

在画圆的算法中，可以利用八对称性，只用计算八分之一圆弧的坐标即可。

```
void bresenhamCircle(vector<int> * xVec, vector<int> * yVec, int x1, int y1, int r) {
    int x = 0, y = r;
    int d = 3 - 2 * r;
    while (x <= y) {
        putPointInCircle(xVec, yVec, x1, y1, x, y, r);
        if (d < 0) {
            d += 4 * x + 6;
        }
        else {
            d += 4 * (x - y) + 10;
            y--;
        }
        x++;
    }
}
```

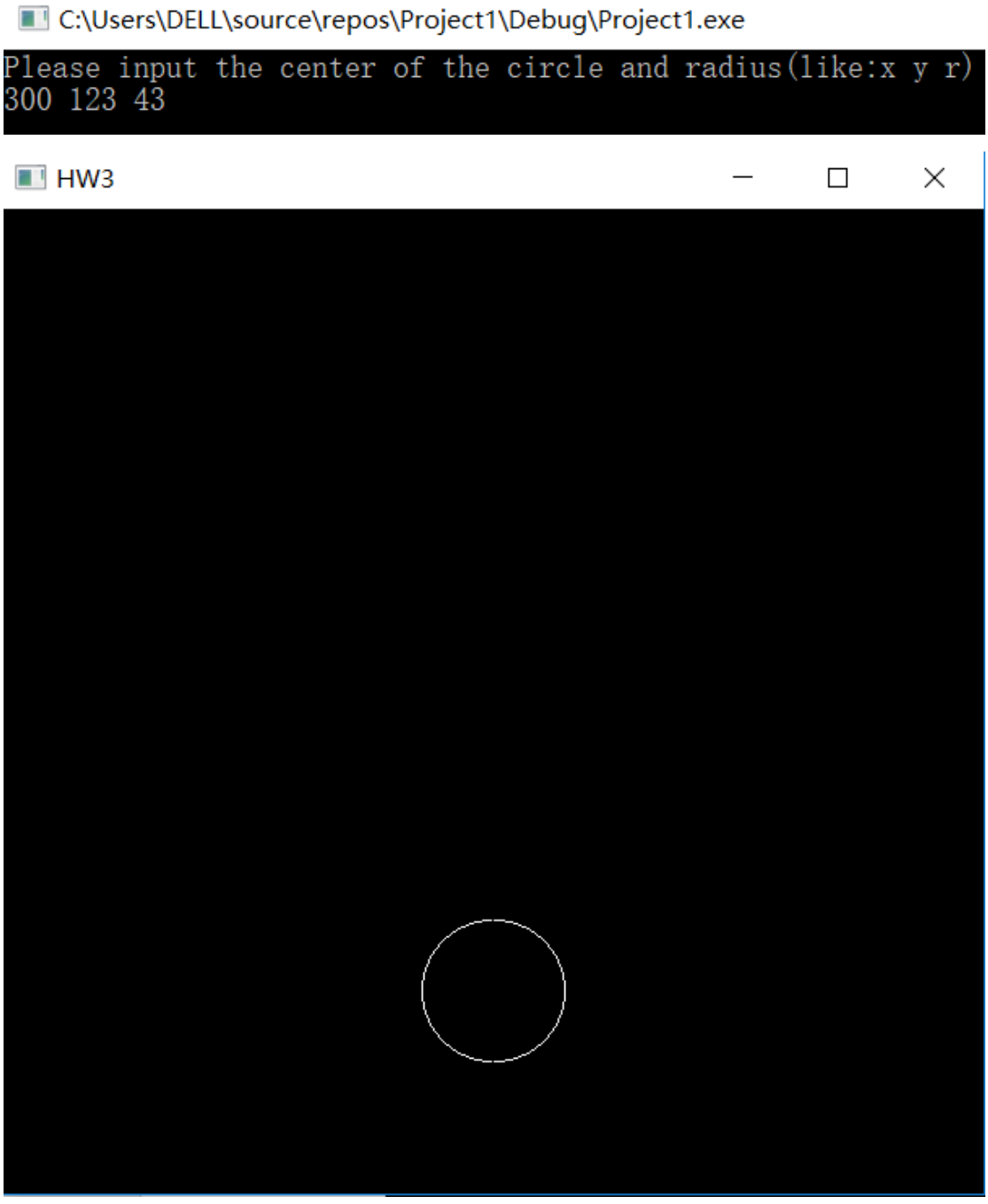
putPointInCircle

因为之前的 Bresenham 算法假设的圆心是(0,0)所以在这里需要按照给定圆心进行偏移，并将所有对称点按顺序按对放入向量保存。

```
void putPointInCircle(vector<int> * xVec,
    (*xVec).push_back(x1 + x);
    (*yVec).push_back(y1 + y);
    (*xVec).push_back(x1 + x);
    (*yVec).push_back(y1 - y);

    (*xVec).push_back(x1 - x);
    (*yVec).push_back(y1 + y);
    (*xVec).push_back(x1 - x);
    (*yVec).push_back(y1 - y);

    (*xVec).push_back(x1 + y);
    (*yVec).push_back(y1 + x);
    (*xVec).push_back(x1 + y);
    (*yVec).push_back(y1 - x);

    (*xVec).push_back(x1 - y);
    (*yVec).push_back(y1 + x);
    (*xVec).push_back(x1 - y);
    (*yVec).push_back(y1 - x);
}
```

结果

程序运行一开始需要输入圆心的坐标和半径。输入坐标后即可得到结果。



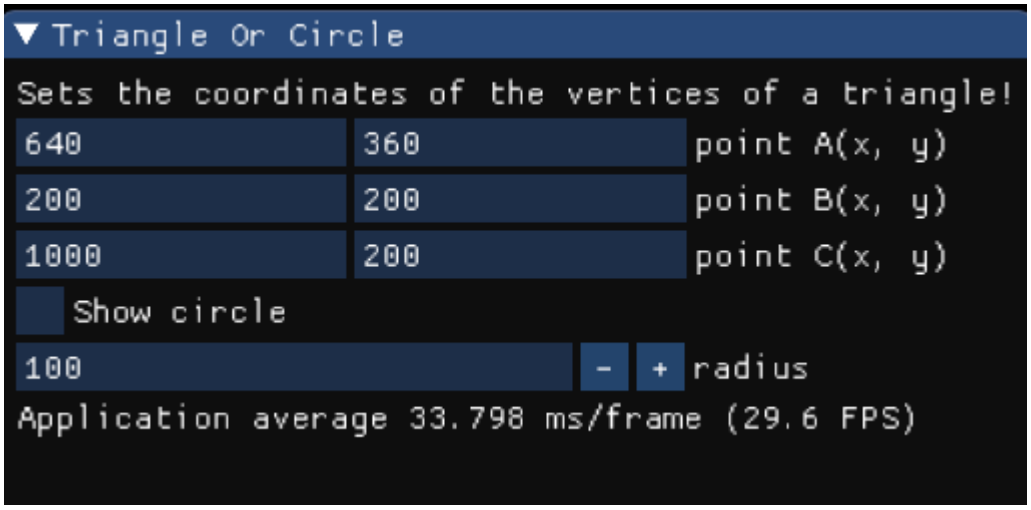
3. 在 GUI 在添加菜单栏，可以选择是三角形边框还是圆，以及能调整圆的大小(圆心固定即可)

本题只需要将前面两题的代码与 GUI 结合即可。这里采用的是 IMGUI。

GUI 界面

这里在 GUI 中可以输入三角形三个点的坐标，当选则 Show Circle 时则可以展示圆，可以通过改变 radius 值，改变圆的半径。具体如下图二所示。在这里输入点的坐标时使用的是 ImGui::InputInt2，这可以接受一个大小为 2 的一维数组并修改其中的数值。

```
ImGui::Begin("Triangle Or Circle");
ImGui::SetWindowFontScale((float)1.4);
static float f = 0.0f;
static int counter = 0;
ImGui::Text("Sets the coordinates of the vertices of a triangle!");

ImGui::InputInt2("point A(x, y)", pointA);
ImGui::InputInt2("point B(x, y)", pointB);
ImGui::InputInt2("point C(x, y)", pointC);

ImGui::Checkbox("Show circle", &isCircle);
ImGui::InputInt("radius", &r);
```



结果

本题具体结果录屏见文件中 HW3 basic-3

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

填充三角形的关键是判断点是否在三角形中。而在这里判断点是否在三角形才用的方法是用同侧法，即通过目标点与三角形的其中一点是否处于另外两点构成直线的同侧来判断点是否处于三角形中。

具体计算方法为当 f(x0,y0)/f(x3,y3)>0 时，目标点与三角形其中一点处于另外两点构成直线的同侧，其中 f(x,y)= (y1-y2)*x+(x2-x1)*y+x1*y2-x2*y1。而当目标点与三个点都处于另外两点构成直线的同侧时则可以认为目标点在三角形中。具体代码如下。

Padding

通过三角形的三个顶点得到包围三角形的最小矩形，在这个矩形中进行扫描，将在三角形中的点放入向量。

```
//得到需要填充的点
void padding(vector<int> * xVec, vector<int> * yVec, int x1, int y1, int x2, int y2, int x3, int y3) {
    int xMax = max(max(x1, x2), x3);
    int yMax = max(max(y1, y2), y3);
    int xMin = min(min(x1, x2), x3);
    int yMin = min(min(y1, y2), y3);

    for (int i = xMin; i <= xMax; i++) {
        for (int j = yMin; j <= yMax; j++) {
            if (checkInTriangle(x1, y1, x2, y2, x3, y3, i, j)) {
                (*xVec).push_back(i);
                (*yVec).push_back(j);
            }
        }
    }
}
```

checkInTriangle&&checkSameSide

在 checkInTriangle 中分布检测目标点与三个顶点是否在同侧，当都处于同侧时，这个点在三角形中。在 checkSameSide 中通过函数计算 f(x0,y0)/f(x3,y3)。

```
//检查点是否在三角形中
bool checkInTriangle(int x1, int y1, int x2, int y2, int x3, int y3, int x, int y) {
    return checkSameSide(x1, y1, x2, y2, x3, y3, x, y) > 0
        && checkSameSide(x2, y2, x3, y3, x1, y1, x, y) > 0
        && checkSameSide(x3, y3, x1, y1, x2, y2, x, y) > 0;
}
```

```
#define checkSameSide(x1, y1, x2, y2, x3, y3, x, y) ((y1 - y2) * x + (x2 - x1) * y + x1 * y2 - x2 * y1) / (float)((y1 - y2) * x3 + (x2 - x1) * y3 + x1 * y2 - x2 * y1)
```

结果

在这里三角形被填充了白色。