计算机图形学作业报告

廖蕾 16340135

- 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）。
  这部分的代码都是在上一周的基础上修改的。
  算法部分：
  这里用到的算法就是在上一周中初始化窗口、着色器和渲染等。在绘制三角形边框的时候，新加入了一个函数，代码如下图所示：

```
vector<float> BresenhamLine(int x0, int y0, int x1, int y1) {
    vector<float> result;
    int deltaX = (x1 >= x0 ? x1 - x0 : x0 - x1);
    int deltaY = (y1 >= y0 ? y1 - y0 : y0 - y1);
    int p = 2 * deltaY - deltaX;
    int xi = x0, yi = y0;
    int xinc, yinc;
    xinc = (x1 >= x0 ? 1 : -1);
    yinc = (y1 >= y0 ? 1 : -1);
    result.push_back((float)xi / (float)WINDOW_WIDTH * 2);
    result.push_back((float)yi / (float)WINDOW_HEIGHT * 2);
    if (deltaX > deltaY) {
        while (xi != x1) {
            if (p <= 0) {
                p = p + 2 * deltaY;
            }
            else {
                yi += yinc;
                p = p + 2 * deltaY - 2 * deltaX;
            }
            xi += xinc;
            result.push_back((float)xi / (float)WINDOW_WIDTH * 2);
            result.push_back((float)yi / (float)WINDOW_HEIGHT * 2);
        }
    }
    else {
        while (yi != y1) {
            if (p <= 0) {
                p = p + 2 * deltaX;
            }
            else {
                xi += xinc;
                p = p + 2 * deltaX - 2 * deltaY;
            }
            yi += yinc;
            result.push_back((float)xi / (float)WINDOW_WIDTH * 2);
            result.push_back((float)yi / (float)WINDOW_HEIGHT * 2);
        }
    }
    return result;
}
```

  这里的算法就是Bresenham算法：
  - 这里需要直线两个端点的坐标作为输入，计算Δx和Δy，此时Δx和Δy均取正数。
  - 计算$p = 2 \times \Delta y + \Delta x$，再用一个符号记号：$x_{inc} = x_1 > x_0 ? 1 : -1$和$y_{inc} = y_1 > y_0 ? 1 : -1$
  - 将初始的x_0和y_0放入我们最后的直线点集中。

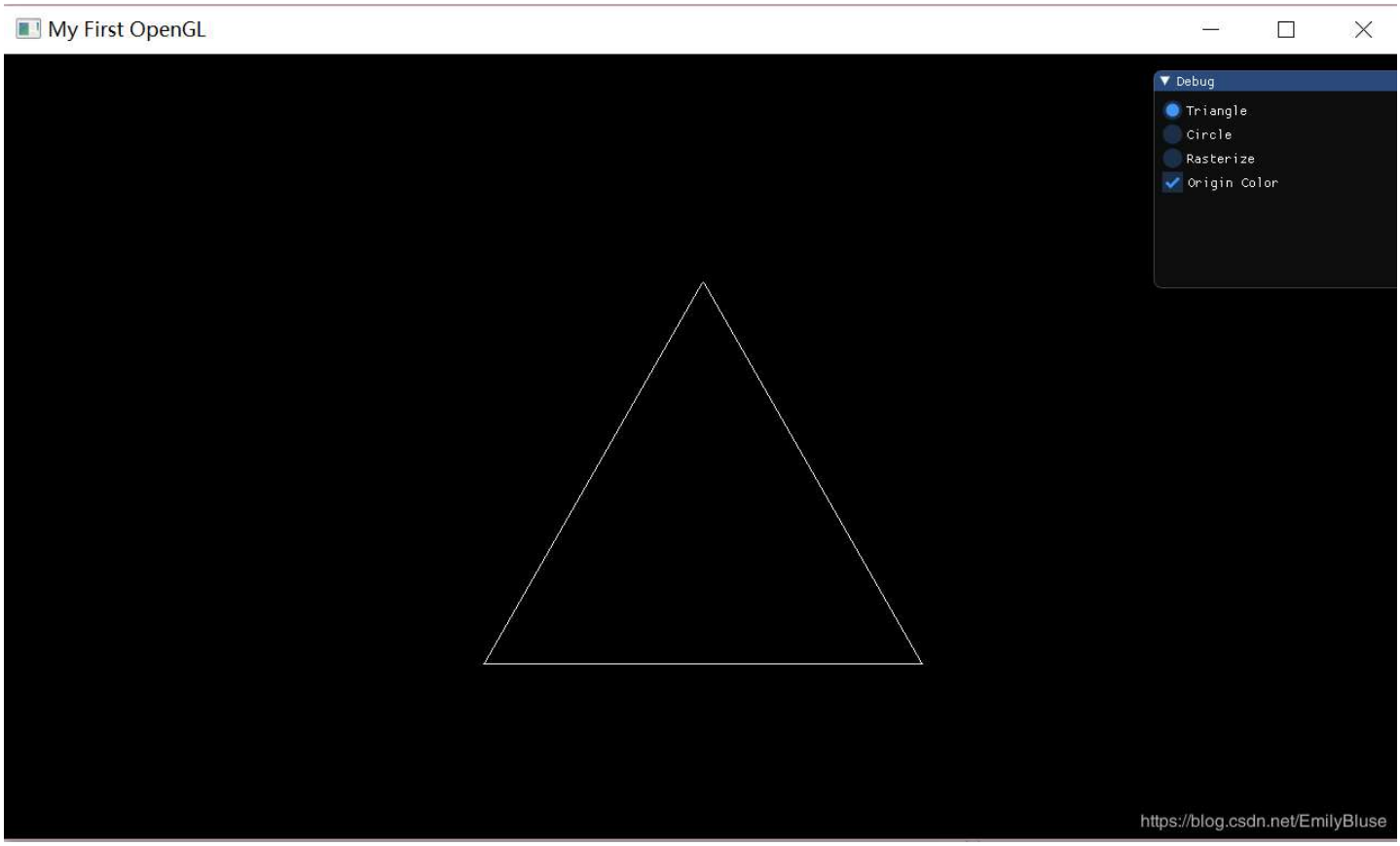
- 当$\Delta x > \Delta y$时，从x_0开始移动到x_1，如果此时$p <= 0$, 那么更新$p = p + 2 \times \Delta y$。否则，沿$y$轴移动一个单位长度，更新$p = p + 2 \times \Delta y - 2 \times \Delta x$。将$x$沿轴移动一个长度之后，将此时的点的坐标存入答案的点集中。
- 当$\Delta x < \Delta y$，则将上述过程沿y轴重复即可。

通过 `BresenhamLine` 函数我们可以得到某条直线中组成这条直线的全部点，在渲染的过程中使用一下语句渲染出来：

```
//获得三条直线中的全部点
triangle.push_back(BresenhamLine(0, 150, -200, -200));
triangle.push_back(BresenhamLine(0, 150, 200, -200));
triangle.push_back(BresenhamLine(-200, -200, 200, -200));
for (size_t i = 0; i < triangle.size(); i++) {
    for (size_t j = 0; j < triangle[i].size(); j += 2) {
        //每次渲染一个点
        float vertices[] = { triangle[i][j], triangle[i][j + 1], 0.0f, tri.x, tri.y, tri.z };
        glBindVertexArray(VAO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
        // position
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
        glEnableVertexAttribArray(0);
        //color
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
        glEnableVertexAttribArray(1);

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
        glUseProgram(shaderProgram);
        glDrawArrays(GL_POINTS, 0, 1);
    }
}
```

得到的结果如下图所示：



- 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。
  仿照上题所示的结果，我们在原有代码的基础上加入下面这个函数：

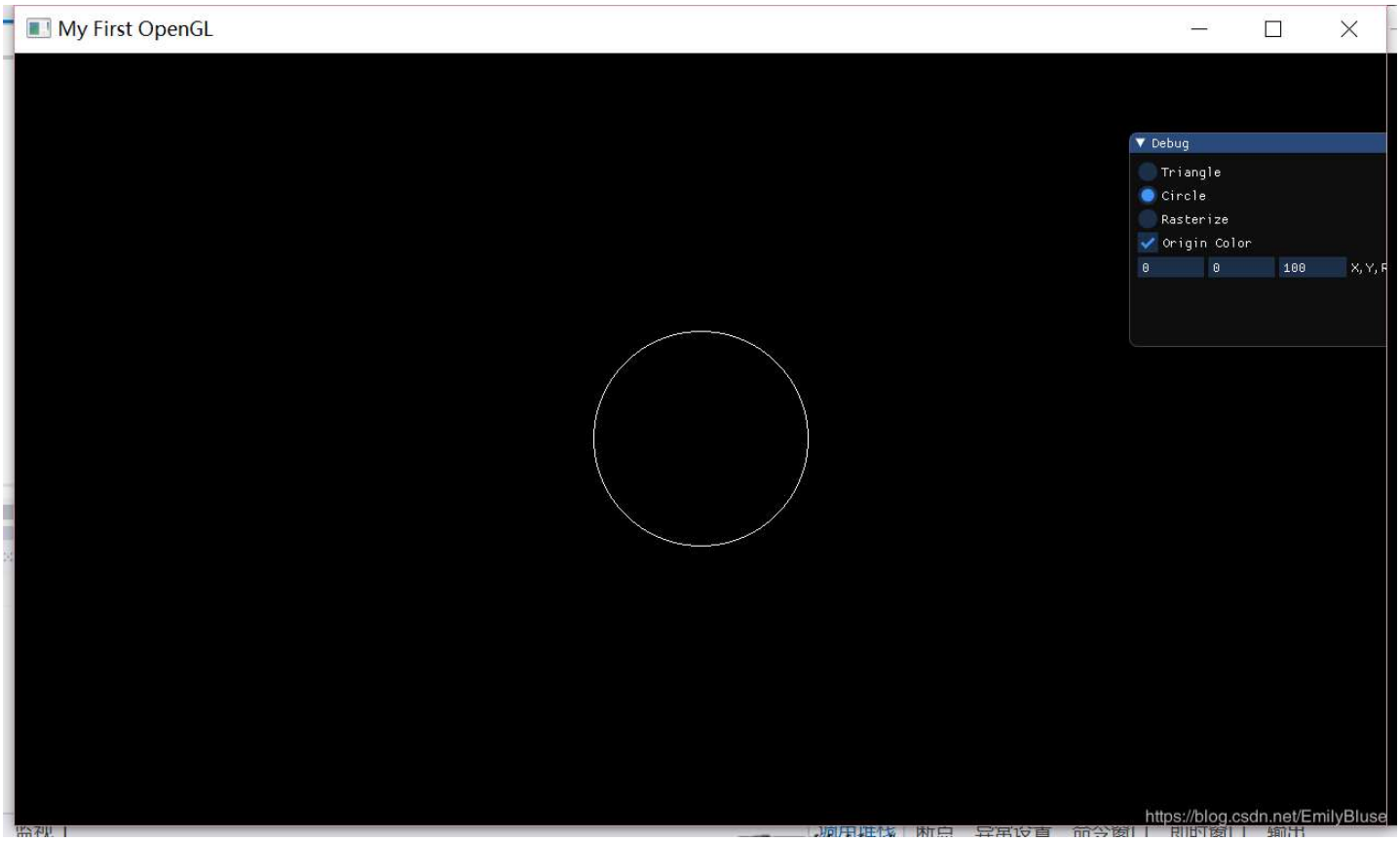
```
vector<float> BresenhamCircle(int x, int y, int radius) {
    vector<float> result;
    int xi = 0, yi = radius, rec = 0;
    int d = 5 - 4 * radius;
    //1
    result.push_back((float)(xi + x) / (float)WINDOW_WIDTH * 2);
    result.push_back((float)(yi + y) / (float)WINDOW_HEIGHT * 2);
    //2
    result.push_back((float)(-xi + x) / (float)WINDOW_WIDTH * 2);
    result.push_back((float)(yi + y) / (float)WINDOW_HEIGHT * 2);
    //3
    result.push_back((float)(xi + x) / (float)WINDOW_WIDTH * 2);
    result.push_back((float)(-yi + y) / (float)WINDOW_HEIGHT * 2);
    //4
    result.push_back((float)(-xi + x) / (float)WINDOW_WIDTH * 2);
    result.push_back((float)(-yi + y) / (float)WINDOW_HEIGHT * 2);
    //5
    result.push_back((float)(yi + x) / (float)WINDOW_WIDTH * 2);
    result.push_back((float)(xi + y) / (float)WINDOW_HEIGHT * 2);
    //6
    result.push_back((float)(-yi + x) / (float)WINDOW_WIDTH * 2);
    result.push_back((float)(xi + y) / (float)WINDOW_HEIGHT * 2);
    //7
    result.push_back((float)(yi + x) / (float)WINDOW_WIDTH * 2);
    result.push_back((float)(-xi + y) / (float)WINDOW_HEIGHT * 2);
    //8
    result.push_back((float)(-yi + x) / (float)WINDOW_WIDTH * 2);
    result.push_back((float)(-xi + y) / (float)WINDOW_HEIGHT * 2);
    while (xi <= yi) {
        if (d <= 0) {
            d += 8 * xi + 12;
        }
        else {
            d += 8 * (xi - yi) + 20;
            yi--;
        }
        xi++;
        //1
        result.push_back((float)(xi + x) / (float)WINDOW_WIDTH * 2);
        result.push_back((float)(yi + y) / (float)WINDOW_HEIGHT * 2);
        //2
        result.push_back((float)(-xi + x) / (float)WINDOW_WIDTH * 2);
        result.push_back((float)(yi + y) / (float)WINDOW_HEIGHT * 2);
        //3
        result.push_back((float)(xi + x) / (float)WINDOW_WIDTH * 2);
        result.push_back((float)(-yi + y) / (float)WINDOW_HEIGHT * 2);
        //4
        result.push_back((float)(-xi + x) / (float)WINDOW_WIDTH * 2);
        result.push_back((float)(-yi + y) / (float)WINDOW_HEIGHT * 2);
        //5
        result.push_back((float)(yi + x) / (float)WINDOW_WIDTH * 2);
        result.push_back((float)(xi + y) / (float)WINDOW_HEIGHT * 2);
        //6
        result.push_back((float)(-yi + x) / (float)WINDOW_WIDTH * 2);
        result.push_back((float)(xi + y) / (float)WINDOW_HEIGHT * 2);
        //7
        result.push_back((float)(yi + x) / (float)WINDOW_WIDTH * 2);
        result.push_back((float)(-xi + y) / (float)WINDOW_HEIGHT * 2);
        //8
        result.push_back((float)(-yi + x) / (float)WINDOW_WIDTH * 2);
        result.push_back((float)(-xi + y) / (float)WINDOW_HEIGHT * 2);
    }
    return result;
}
```

画圆的算法就是先定下开始迭代计算的第一点$(x_0, y_0) = (0, r)$，加入决策参数：$d = 5 - 4 \times r$，然后将当前的点，以及对于7个对称点的坐标加入我们的目标点中。判断之后是用x去循环迭代还是y。
渲染的时候还是一个一个点去渲染：

```
vector<float> circle = BresenhamCircle(cicleInput[0], cicleInput[1], cicleInput[2]);
for (size_t i = 0; i < circle.size(); i += 2) {
    float vertices[] = { circle[i], circle[i + 1], 0.0f, tri.x, tri.y, tri.z };
    glBindVertexArray(VAO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    // position
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
    glEnableVertexAttribArray(0);
    //color
    glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
    glEnableVertexAttribArray(1);

    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
    glUseProgram(shaderProgram);
    glDrawArrays(GL_POINTS, 0, 1);
}
```

得到的结果如下图所示：



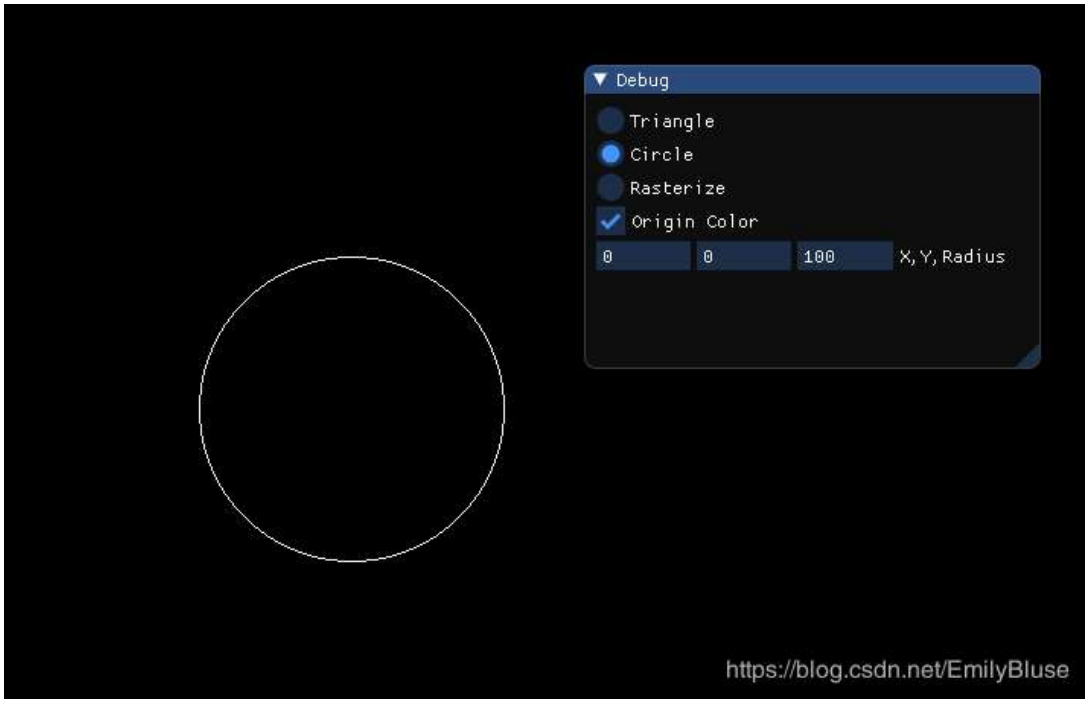
- 在GUI在添加菜单栏，可以选择是三角形边框还是圆，以及能调整圆的大小(圆心固定即可)。
  加入选择菜单的参数，用到ImGui初始化和上周一样，这里只放出渲染时用到的算法：

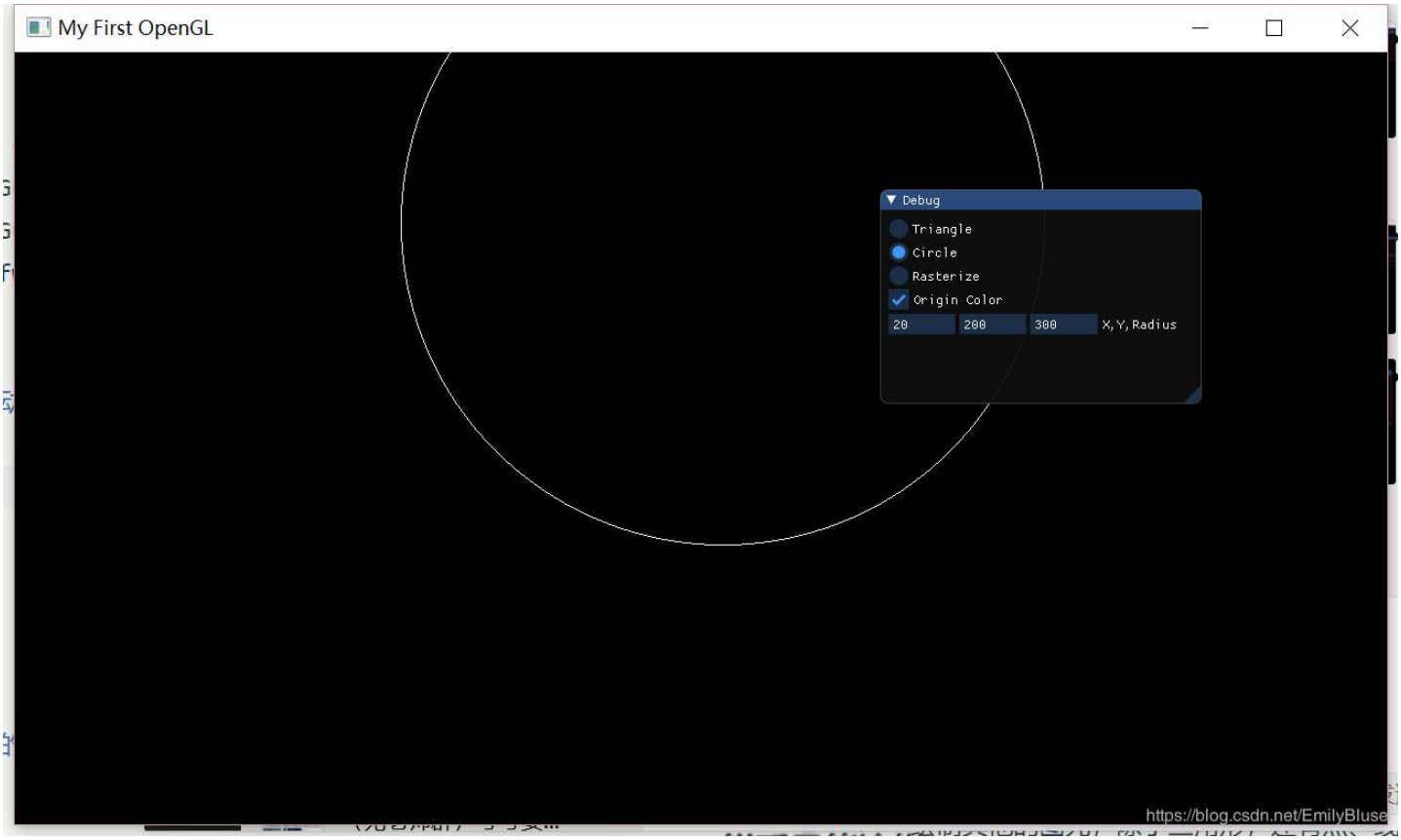
```
int radioMark = 0;
int cicleInput[3];
cicleInput[0] = 0;
cicleInput[1] = 0;
cicleInput[2] = 100;
bool originColor = true;

while (!glfwWindowShouldClose(window)) {
    ImGui_ImplGlfwGL3_NewFrame();
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    // init ImGui
    {
        ImGui::RadioButton("Triangle", &radioMark, 0);
        ImGui::RadioButton("Circle", &radioMark, 1);
        ImGui::RadioButton("Rasterize", &radioMark, 2);
        ImGui::Checkbox("Origin Color", &originColor);
        if (!originColor) {
            ImGui::ColorEdit3("Choose a Color: ", (float*)&tri);
        }
        if (radioMark == 0) {
            //渲染三角形
        }
        else if (radioMark == 1) {
            ImGui::InputInt3("X,Y,Radius", cicleInput, 0);
            //渲染圆
        }
        else if (radioMark == 2) {
            //Bouns
        }
    }
    // check out triggerations & render
    ImGui::Render();
    ImGui_ImplGlfwGL3_RenderDrawData(ImGui::GetDrawData());
    glfwSwapBuffers(window);
}
```

最后的动态结果请见同文件夹下的演示视频。这里只展示几张图片：





Bouns

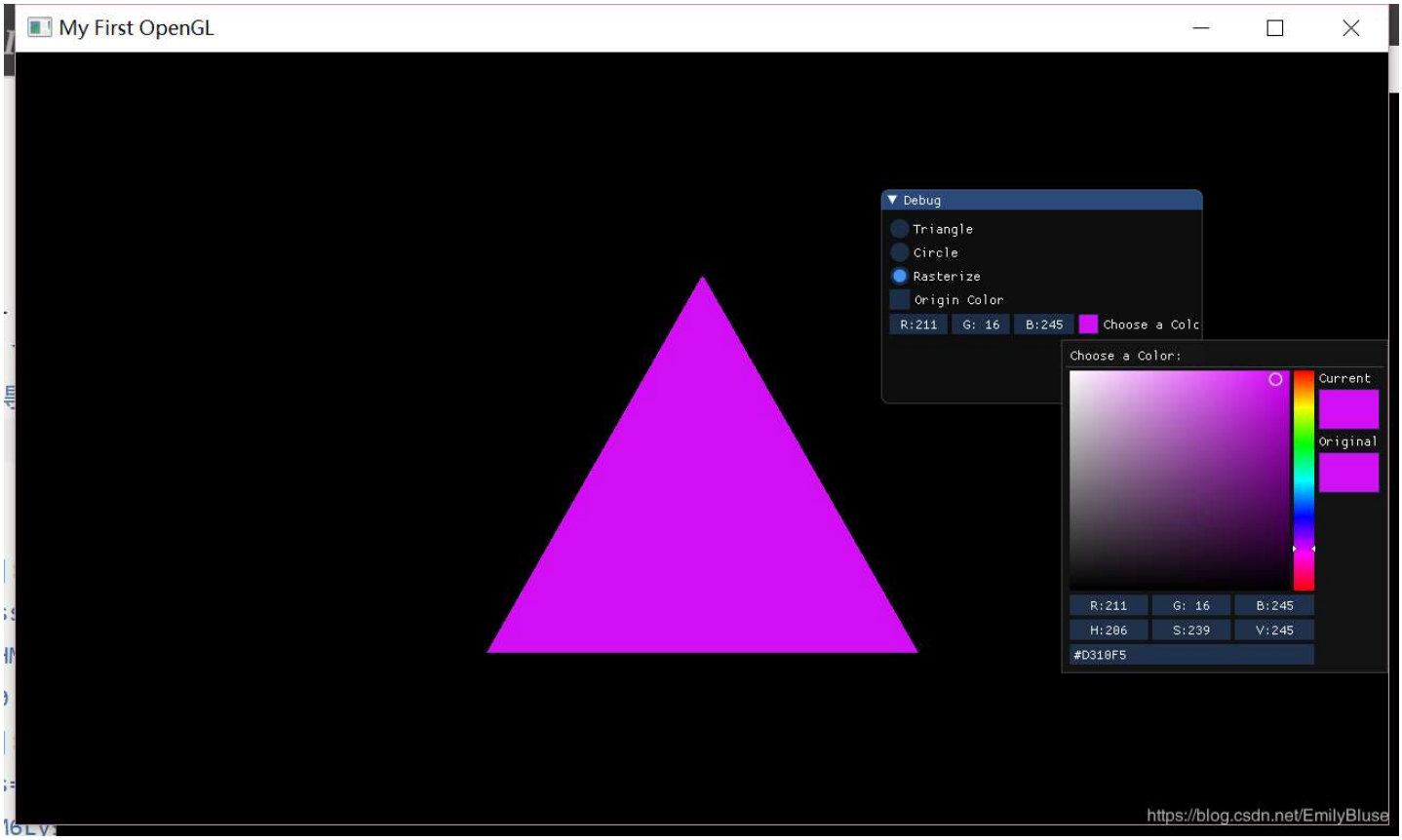
- 使用三角形光栅转换算法，用和背景不同的颜色，填充你的三角形。
  这里加入颜色选择栏，同时用edge walk来填充我的三角形。
  颜色选择用了上周的代码，所以这里不放出来，只需要在渲染直线的时候加入同样的颜色列就好了。
  在绘制三角形边框的基础上，加入edge walk可以填充三角形了，代码如下：

```
// Edge Walking
for (size_t i = 1; i < triangle[0].size() + 1; i += 2) {
    float xi = triangle[0][i - 1];
    while (triangle[1][i - 1] - xi > 0.00001) {
        float vertices[] = { xi, triangle[0][i], 0.0f, tri.x, tri.y, tri.z };
        glBindVertexArray(VAO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
        // position
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0);
        glEnableVertexAttribArray(0);
        //color
        glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)(3 * sizeof(float)));
        glEnableVertexAttribArray(1);

        glBindBuffer(GL_ARRAY_BUFFER, VBO);
        glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
        glUseProgram(shaderProgram);
        glDrawArrays(GL_POINTS, 0, 1);

        xi += (float)1.0 / (float)WINDOW_WIDTH;
    }
}
```

得到结果如下：


My First OpenGL
Debug
Triangle
Circle
Rasterize
Origin Color
R:211
G: 16
B:245
Choose a Colc
Choose a Color:
Current
Original
R:211
G: 16
B:245
H:206
S:239
V:245
#D310F5
https://blog.csdn.net/EmilyBluse