计算机图形学实验报告

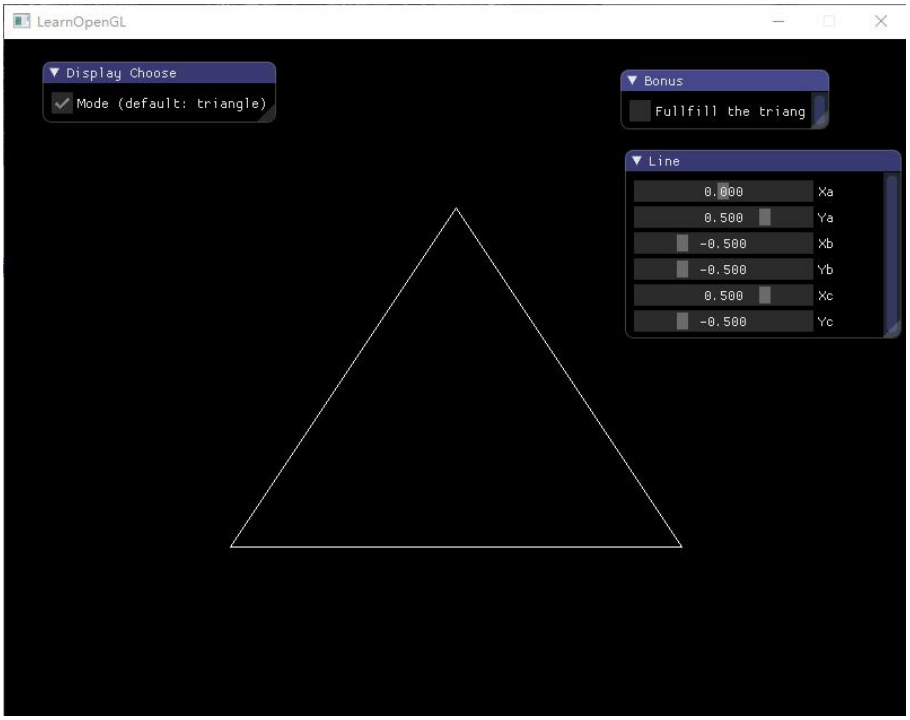
第一部分：实现思路和代码分析

代码截图	代码说明
`// 作图` `if (isDrawLine) {` `  bresenham.drawTriangle(` `  Xa, Ya,` `  Xb, Yb,` `  Xc, Yc,` `  p, fullfill` `  );` `}` `else {` `  bresenham.drawCircle(` `  center_x, center_y,` `  radius, p` `  );` `}` `total = p.size();` `vertices = new GLfloat[total];` `int pos = 0;` `for (unsigned int j = 0; j < p.size(); j++) {` `  vertices[pos++] = p[j];` `}`	这部分代码是主函数调用生成直线和圆的方法。首先使用一个 vector 进行像素点的存储，然后转换成 GLfloat 数组传入到 OpenGL 函数进行图像的渲染。
`// 画三角形` `void drawTriangle(GLfloat Xa, GLfloat Ya,` `  GLfloat Xb, GLfloat Yb,` `  GLfloat Xc, GLfloat Yc,` `  std::vector<GLfloat>& res,` `  bool fullfill = false) {` `  GLint Xa_t = (GLint)std::floor((Xa + 1)*WINDOW_WIDTH) / 2,` `  Ya_t = (GLint)std::floor((1 - Ya)*WINDOW_HEIGHT) / 2,` `  Xb_t = (GLint)std::floor((Xb + 1)*WINDOW_WIDTH) / 2,` `  Yb_t = (GLint)std::floor((1 - Yb)*WINDOW_HEIGHT) / 2,` `  Xc_t = (GLint)std::floor((Xc + 1)*WINDOW_WIDTH) / 2,` `  Yc_t = (GLint)std::floor((1 - Yc)*WINDOW_HEIGHT) / 2;` `  drawLineWithBresenham(Xa_t, Ya_t, Xb_t, Yb_t, res);` `  drawLineWithBresenham(Xb_t, Yb_t, Xc_t, Yc_t, res);` `  drawLineWithBresenham(Xc_t, Yc_t, Xa_t, Ya_t, res);` `  if (fullfill) {` `  fillTriangleUsingEdgeEquation(Xa_t, Ya_t, Xb_t, Yb_t, Xc_t, Yc_t, res);` `  }` `}`	这是用于渲染三角形的接口函数。此函数首先将传入的浮点数（-1 到 1 的比例）转化为对应的整数坐标（坐标系原点为左上角，水平向右为 x 轴，垂直向下为 y 轴），然后根据三点坐标使用 Bresenham 算法作出三角形。
`// Bresenham画直线` `void drawLineWithBresenham(GLint x0, GLint y0,` `  GLint x1, GLint y1,` `  std::vector<GLfloat>& out) {` `  int dx = abs(x1 - x0);` `  int dy = abs(y1 - y0);` `  int direct_x = x1 > x0 ? 1 : -1;` `  int direct_y = y1 > y0 ? 1 : -1;` `  int p = 0;` `  if (dx > dy) {` `  p = 2 * dy - dx;` `  while (x0 != x1) {` `  this->saveLinePoint(x0, y0, out);` `  if (p > 0) {` `  y0 += direct_y;` `  p = p + 2 * (dy - dx);` `  }` `  else {` `  p = p + 2 * dy;` `  }` `  x0 += direct_x;` `  }` `  }` `  else {` `  p = 2 * dx - dy;` `  while (y0 != y1) {` `  this->saveLinePoint(x0, y0, out);` `  if (p > 0) {` `  x0 += direct_x;` `  p = p + 2 * (dx - dy);` `  }` `  else {` `  p = p + 2 * dx;` `  }` `  y0 += direct_y;` `  }` `  }` `}`	产生三角形的 Bresenham 算法。首先判断两点间 x 和 y 坐标的差值 dx 和 dy，然后确定每次迭代移动的位移（-1 或 1）。假如 dx > dy，说明沿 x 轴移动的距离比沿 y 轴的要长，做出的图像更加精确，因此 Bresenham 算法以 x 坐标作为终止条件，每次选择性更新 y 轴坐标；反之，Bresenham 算法以 y 坐标作为终止条件，每次选择性更新 x 轴坐标。
`// 保存直线上的点` `void saveLinePoint(int temp_X, int temp_Y, std::vector<GLfloat>& out) {` `  GLfloat color = 1.0f;` `  out.push_back(GLfloat(float(temp_X) / float(WINDOW_WIDTH / 2)) - 1.0f);` `  out.push_back(1.0f - GLfloat(float(temp_Y) / float(WINDOW_HEIGHT / 2)));` `  out.push_back(0.0f);` `  for (int i = 0; i < 3; i++) {` `  out.push_back(color);` `  }` `}`	将位于直线上的像素点转换成比例尺坐标（-1 到 1），同时将颜色等信息添加上去，组成 OpenGL 的一个点。

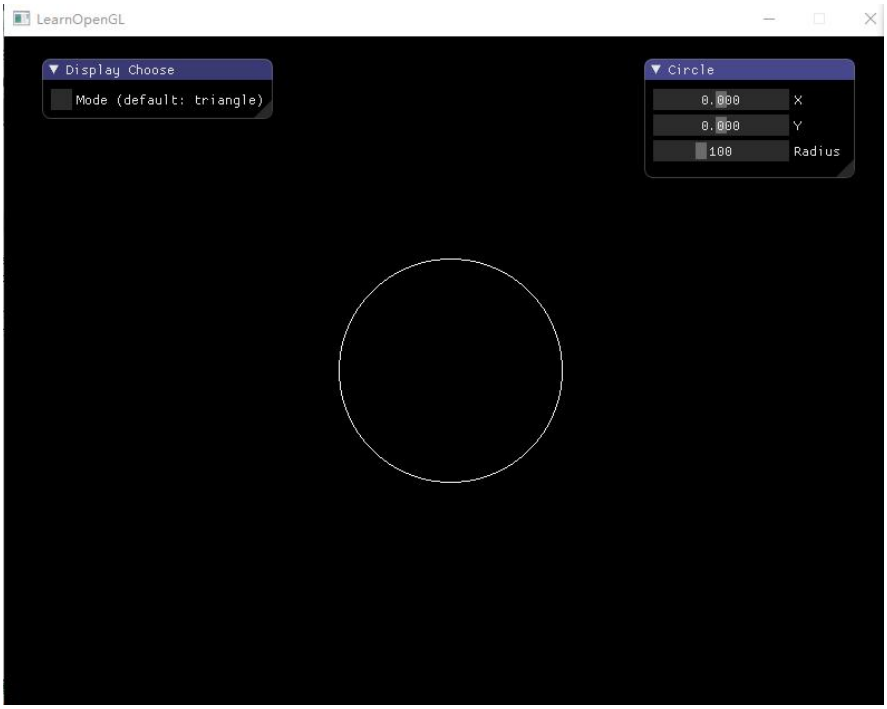
`` // 画圆 void drawCircle(GLfloat x, GLfloat y, int radius, std::vector<GLfloat>& res) { int center_x = std::floor(x * WINDOW_WIDTH / 2), center_y = std::floor(y * WINDOW_HEIGHT / 2); drawCircleWithBresenham(center_x, center_y, radius, res); } ```	这是用于渲染圆形的接口函数。首先将浮点坐标变成整数坐标，再作圆。
`` // Bresenham画圆 void drawCircleWithBresenham(int center_x, int center_y, int radius, std::vector<GLfloat>& out) { int x = 0, y = radius, d = 3 - 2 * radius; while (x <= y) { // 八点法画圆 this->saveCirclePoint(x + center_x, y + center_y, out); this->saveCirclePoint(x + center_x, -y + center_y, out); this->saveCirclePoint(-y + center_x, x + center_y, out); this->saveCirclePoint(-y + center_x, -x + center_y, out); this->saveCirclePoint(-x + center_x, -y + center_y, out); this->saveCirclePoint(-x + center_x, y + center_y, out); this->saveCirclePoint(y + center_x, -x + center_y, out); this->saveCirclePoint(y + center_x, x + center_y, out); if (d < 0) { d = d + 4 * x + 6; } else { d = d + 4 * (x - y) + 10; y--; } x++; } } ```	产生圆形的 Bresenham 算法。由圆的方程 F(x, y）= x^2 + y^2 – R^2 可以推导出当 d < 0 时，d’ = d + 4(xi) + 6；d > 0 时，d’ = d + 4(xi - yi) + 10。初始值设为 d0 = 1.25 - R。但为了避免浮点数运算，将初始值设为 d0 = 3 - 2R。算法只生成八分之一个圆弧，剩下的圆弧利用圆的对称特性进行生成。
`` // 保存圆上的点 void saveCirclePoint(int temp_X, int temp_Y, std::vector<GLfloat>& out) { GLfloat color = 1.0f; out.push_back(GLfloat(float(temp_X) / float(WINDOW_WIDTH / 2))); out.push_back(GLfloat(float(temp_Y) / float(WINDOW_HEIGHT / 2))); out.push_back(0.0f); for (int i = 0; i < 3; i++) { out.push_back(color); } } ```	将位于圆弧上的像素点转换成比例尺坐标（-1 到 1），同时将颜色等信息添加上去，组成 OpenGL 的一个点。
`` // 计算直线一般式：Ax+By+C=0 std::vector<int> lineEquation(GLint X1, GLint Y1, GLint X2, GLint Y2) { std::vector<int> res; int A = Y2 - Y1, B = X1 - X2, C = X2 * Y1 - X1 * Y2; res.push_back(A); res.push_back(B); res.push_back(C); return res; } // 使用Edge Equation方法填充颜色 void fillTriangleUsingEdgeEquation(GLint Xa, GLint Ya, GLint Xb, GLint Yb, GLint Xc, GLint Yc, std::vector<GLfloat>& out) { // 获取外接矩形 int max_x = std::max(Xa, std::max(Xb, Xc)), max_y = std::max(Ya, std::max(Yb, Yc)), min_x = std::min(Xa, std::min(Xb, Xc)), min_y = std::min(Ya, std::min(Yb, Yc)); // 获取三角形三条边 std::vector<std::vector<int>> lines; lines.push_back(this->lineEquation(Xa, Ya, Xb, Yb)); lines.push_back(this->lineEquation(Xb, Yb, Xc, Yc)); lines.push_back(this->lineEquation(Xc, Yc, Xa, Ya)); // 确定直线方程的方向 for (int i = 0; i < 3; i++) { GLint x_t, y_t; if (i == 0) { x_t = Xc; y_t = Yc; } else if (i == 1) { x_t = Xa; y_t = Ya; } else { x_t = Xb; y_t = Yb; } if (lines[i][0] * x_t + lines[i][1] * y_t + lines[i][2] < 0) { for (int k = 0; k < 3; k++) { lines[i][k] *= -1; } } } // 填充颜色 for (GLint x = min_x; x <= max_x; x++) { for (GLint y = min_y; y <= max_y; y++) { bool inside = true; for (int i = 0; i < lines.size(); i++) { if (lines[i][0] * x + lines[i][1] * y + lines[i][2] < 0) { inside = false; break; } } if (inside) { this->saveLinePoint(x, y, out); } } } } ```	这部分是 Bonus 的三角形光栅转换算法的实现代码。这里使用的是 Edge Equation 算法。首先获取待渲染的三角形的最小外接矩形，然后计算得到三角形三条边的一般方程。然后调整三条边的一般方程的方向（这部分计算是为了使得在三角形内部的点代入三条方程时算出的结果都小于 0）。最后遍历外接矩形中的每个像素点，假如将点的坐标代入三条直线方程得到的结果都小于 0，那么可以认为这个点位于三角形中，于是将这个点填充相应颜色。

第二部分：运行结果截图和相关问题解答

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

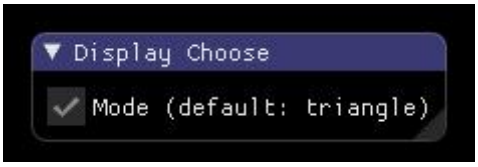


2. 使用 Bresenham 算法(只使用 integer arithmetic)画一个圆



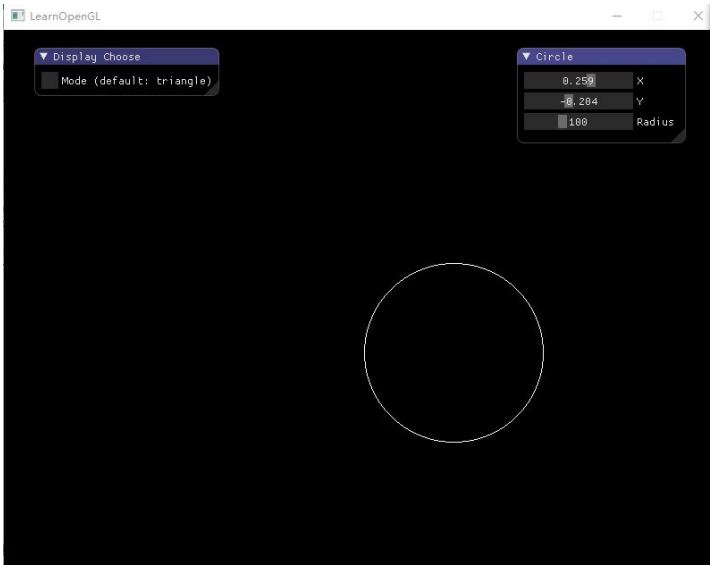
3. 在 GUI 在添加菜单栏，可以选择是三角形边框还是圆，以及能调整圆的大小

（1）菜单栏

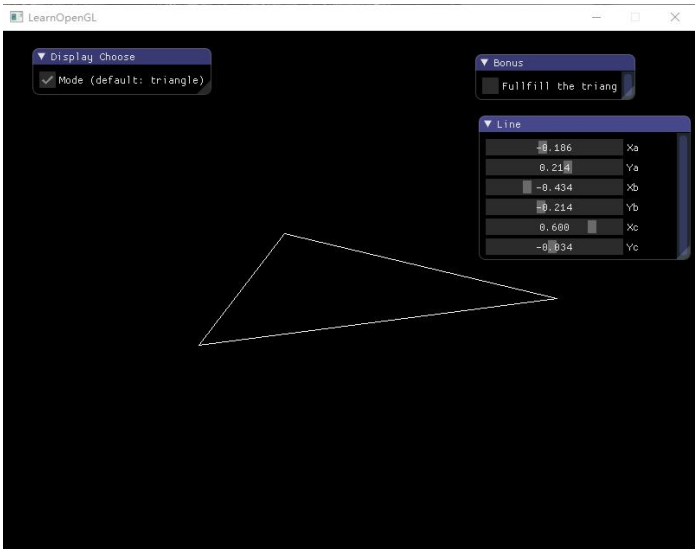


（2）调整圆大小

（3）调整圆位置



（4）调整三角形大小与位置



4. Bonus：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