光栅化算法实验报告

一、实验目的

1. 理解光栅化技术的核心原理，掌握图形从矢量描述（直线、椭圆）转换为像素点集的实现过程。
2. 补全画图小程序，基于整数运算优化实现 **Bresenham 直线算法**，解决DDA算法浮点运算的精度与效率问题。
3. 实现 **椭圆形的光栅化算法**，通过对称绘制和分区域误差判断，完成椭圆的像素级渲染。
4. 验证算法的正确性与高效性，分析不同光栅化模式的渲染性能。

二、实验环境

- 开发语言：C++（标准库）
- 编译环境：GCC 9.4.0 / Visual Studio Code
- 渲染逻辑：通过 `std::vector<Pixel>` 存储光栅化后的像素点，后续对接OpenGL/DirectX等图形框架完成屏幕绘制

三、核心算法实现思路

3.1 Bresenham 直线光栅化算法（基于代码注释优化）

Bresenham算法是对DDA算法的改进，核心思路如代码注释所述：**“引入误差，且只需要整数运算”**，避免DDA算法中浮点运算导致的精度损失和效率损耗。

3.1.1 算法核心预处理

1. **计算方向与步长**：
   - 先计算直线两端点 `start` 与 `end` 的横、纵坐标差：`dx = end.x - start.x`，`dy = end.y - start.y`。
   - 定义 `x_step` 和 `y_step` 表示坐标轴方向（正/负）：若 `dx>0` 则 `x_step=1`（x轴正向），否则为 `-1`；`y_step` 同理，解决直线从右到左、从下到上的绘制问题。
2. **判断“陡峭”直线**：
   - 代码注释中定义 `steep = (abs(dy) > abs(dx))`，即**斜率绝对值>1的直线为“陡峭直线”。**
   - 陡峭直线的特点是：y轴变化率远大于x轴，若直接按x轴步长循环，会导致像素点稀疏、线条不连续。因此需“**互换xy地位，包括误差函数**”（代码注释），将陡峭直线转换为“平缓直线”处理，复用同一套逻辑。

3.1.2 误差项设计与循环渲染

算法通过**整数误差项P**判断下一个像素点的位置，核心是“比较像素中点与直线的位置关系”（代码注释：“计算精确误差（和格子中点的比较）”）。

（1）非陡峭直线（`steep=false`，斜率绝对值≤1）

- **初始误差项**：`P = 2*abs(dy) - abs(dx)`，该公式推导源于“中点是否在直线上方”的判断，避免浮点运算。
- **循环逻辑**：
  1. 每次沿 `x_step` 移动x坐标（x轴变化为主）；
  2. 若 `P < 0`：中点在直线下方，下一个像素点为 `(x+x_step, y)`（无需移动y），更新误差 `P = P + 2*abs(dy)`；
  3. 若 `P ≥ 0`：中点在直线上方，下一个像素点为 `(x+x_step, y+y_step)`（y轴同步移动），更新误差 `P = P + 2*abs(dy) - 2*abs(dx)`；
  4. 将计算出的像素点存入 `pixels` 容器，完成渲染。

（2）陡峭直线（`steep=true`，斜率绝对值>1）

- **核心优化**：互换x与y的角色（包括误差函数），将问题转化为“以y轴为步长基准”的非陡峭直线。
- **初始误差项**：`P = 2*abs(dx) - abs(dy)`（原公式中dx与dy互换）；
- **循环逻辑**：
  1. 每次沿 `y_step` 移动y坐标（y轴变化为主）；
  2. 若 `P < 0`：下一个像素点为 `(x, y+y_step)`（无需移动x），更新误差 `P = P + 2*abs(dx)`；
  3. 若 `P ≥ 0`：下一个像素点为 `(x+x_step, y+y_step)`（x轴同步移动），更新误差 `P = P + 2*abs(dx) - 2*abs(dy)`；
  4. 存入像素点，完成渲染。

3.2 椭圆光栅化算法（基于对角点与分区域误差）

椭圆光栅化的核心是**利用椭圆的对称性**（x轴、y轴对称）减少计算量，同时通过“分区域误差判断”选择下一个像素点（代码注释：“根据中点在椭圆内（上）还是椭圆外 选择 T (xi+1, yi) or S (xi+1, yi-1)”）。

3.2.1 椭圆参数计算（基于对角点）

代码注释明确“根据角点”计算椭圆核心参数：

- 输入为椭圆的对角点 `start`（一个角点）和 `end`（对角点）；
- 椭圆中心 `center`：`( (start.x + end.x)/2, (start.y + end.y)/2 )`（对角点中点）；
- 半长轴 `a`：`abs(end.x - start.x)/2`（x轴方向半长）；
- 半短轴 `b`：`abs(end.y - start.y)/2`（y轴方向半长）；
- 预计算 `a² = a*a`、`b² = b*b`，减少循环中的重复乘法运算。

3.2.2 分区域渲染（基于斜率划分）

椭圆的切线斜率范围为 `[-b/a, 0]` 到 `[-∞, -b/a]`，按斜率是否大于 `-1` 分为两个区域，分别处理误差项：

（1）Region 1（斜率 ∈ [-1, 0]，$b^2x \le a^2y$）

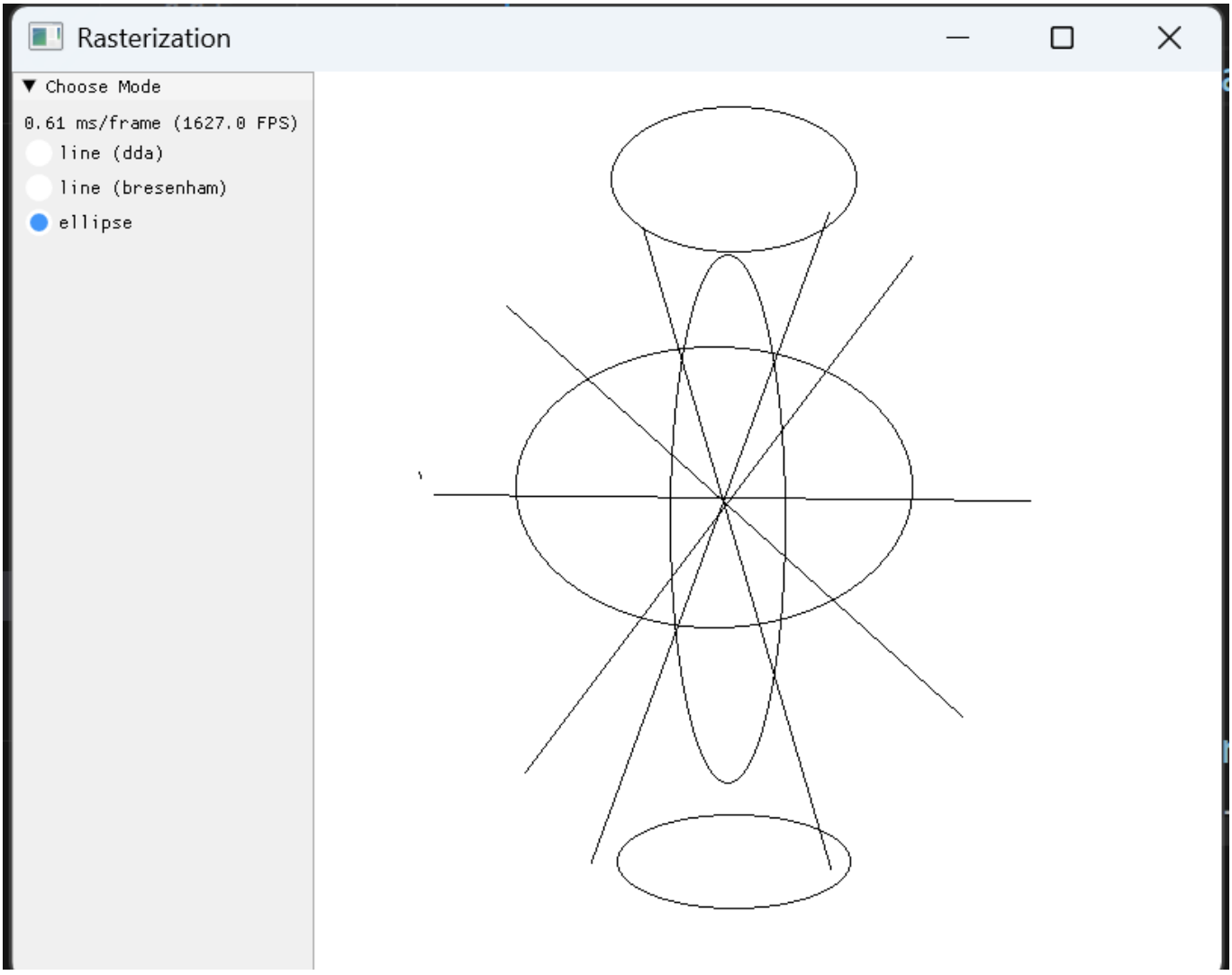
- **初始误差项**：`p1 = b² - a²*b + 0.25*a²`，推导源于椭圆方程 $b^2x^2 + a^2y^2 = a^2b^2$ 的中点判断；
- **循环条件**：`(b² * x <= a² * y) && (2 * a² * y) != 0`（代码注释：“!bug 数学问题 dy/dx = -b2x/(a2y) 所以应当判断 (2 * a2 * y) != 0”，避免分母为0的异常）；
- **对称绘制**：每次计算1个像素点后，通过中心对称生成另外3个点（`(x+center.x, y+center.y)`、`(center.x-x, y+center.y)`、`(center.x-x, center.y-y)`、`(x+center.x, center.y-y)`），覆盖椭圆的四个象限；

- **误差更新**：
  - 若 `p1 < 0`：中点在椭圆内，下一个点选 `T(x+1, y)`，更新 `p1 = p1 + b²*(2x + 3)`；
  - 若 `p1 ≥ 0`：中点在椭圆外，下一个点选 `S(x+1, y-1)`，更新 `p1 = p1 + b²*(2x + 3) + a²*(-2y + 2)`，同时 `y--`；
  - 每次循环 `x++`，直至进入Region 2。

(2) Region 2（斜率 ∈ [-∞, -1]，$b^2x > a^2y$）

- **初始误差项**：`p2 = b²*(x+0.5)² + a²*(y-1)² - a²*b²`，基于Region 1的终点参数初始化；
- **循环条件**：`y ≥ 0`（直至椭圆底部）；
- **对称绘制**：与Region 1一致，每次生成4个对称像素点；
- **误差更新**（代码注释："与region1 xy互换"）：
  - 若 `p2 > 0`：中点在椭圆外，下一个点选 `(x, y-1)`，更新 `p2 = p2 + a²*(-2y + 3)`；
  - 若 `p2 ≤ 0`：中点在椭圆内，下一个点选 `(x+1, y-1)`，更新 `p2 = p2 + b²*(2x + 2) + a²*(-2y + 3)`，同时 `x++`；
  - 每次循环 `y--`，直至y<0结束。

四、实验效果展示



1. **直线渲染**：

- Bresenham算法绘制的直线无“阶梯状锯齿”（DDA因浮点截断可能出现轻微锯齿），线条边缘更平滑；
- 支持任意方向直线（包括陡峭直线、从右到左/从下到上的直线），无断点或错位。

2. **椭圆渲染**：

- 椭圆轮廓完整，无明显变形，四个象限对称度高；
- 分区域处理确保椭圆底部与右侧的像素连续性，无稀疏或重叠像素。