Computer Graphic PA0

王子轩
2023011307
wang-zx23@mails.tsinghua.edu.cn

2025/3/15

1 Algorithm

1.1 线段绘制

首先，利用最原始的 Bresenham 画线算法，可以画出一条从起点 (x_0, y_0) 到终点 (x_1, y_1) 的线段, 其中起点需要在终点的左下方。但该过程需要利用除法计算 k，实际上不是必须的。

Algorithm 1 Basic Bresenham Line Drawing Algorithm

Require: Starting point (x_0, y_0), Ending point (x_1, y_1), Image object, and color
Ensure: Draw a line from (x_0, y_0) to (x_1, y_1) in the specified color

Calculate the differences: $dx = x_1 - x_0$, $dy = y_1 - y_0$
Calculate the slope: $k = \frac{dy}{dx}$
Initialize the error term: $e = -0.5$
Set the starting position: $x = x_0$, $y = y_0$
for each i from 0 to dx **do**
    Set the pixel at (x, y) to the given color in the image
    Increment x by 1
    Update the error term: $e = e + k$
    if $e \geq 0$ **then**
        Increment y by 1
        Decrease the error term by 1: $e = e - 1$
    end if
end for

我们将该算法进行改进：通过计算起点和终点的坐标差值 (dx, dy)，确定绘制过程中每一步在 x 和 y 方向上的变化量。根据起点和终点的相对位置，代码决定了每次 x 和 y 坐标的增减步长 (sx 和 sy)。然后，设置一个初始误差值 (err)，并进入循环，根据误差值调整 x 和 y 的坐标。每次迭代中，当前像素点 (x0, y0) 会被绘制，如果当前点已经到达终点，则退出循环。如果没有到达终点，算法根据误差值判断是更新 x 坐标还是 y 坐标。

Listing 1 实现 void Line::draw()

```
void draw(Image &img) override {
    int x0 = xA, y0 = yA, x1 = xB, y1 = yB;
    int dx = abs(x1 - x0), dy = -abs(y1 - y0);
    int sx = (x0 < x1) ? 1 : -1, sy = (y0 < y1) ? 1 : -1;
    int err = dx + dy;
    while (true){
        img.SetPixel(x0, y0, color);
        if (x0 == x1 && y0 == y1) break;
        int e2 = 2 * err;
        if (e2 >= dy){
            if (x0 == x1) break;
            err += dy; x0 += sx;
        }
        if (e2 <= dx) {
            if (y0 == y1) break;
            err += dx; y0 += sy;
        }
    }
}
```

1.2 圆弧绘制

使用中点画圆算法（Midpoint Circle Algorithm）来绘制圆。该算法基于圆的八分对称性，只需要计算圆的八分之一的点，然后通过对称变换得到整个圆的所有点。

Algorithm 2 Midpoint Circle Drawing Algorithm

Require: Circle center (cx, cy), radius r, Image object, and color
Ensure: Draw a circle with center (cx, cy) and radius r in the specified color

Initialize: $x = 0$, $y = r$
Calculate initial decision parameter: $d = 1.25 - r$
Draw points at $(cx \pm x, cy \pm y)$ and $(cx \pm y, cy \pm x)$
while $x \leq y$ **do**
  if $d < 0$ **then**
    Update decision parameter: $d = d + 2x + 3$
  else
    Update decision parameter: $d = d + 2(x - y) + 5$
    Decrement y by 1
  end if
  Increment x by 1
  Draw points at $(cx \pm x, cy \pm y)$ and $(cx \pm y, cy \pm x)$
end while

算法使用一个决策参数 d 来决定下一个像素点的位置，避免了浮点数运算。当 $d < 0$ 时，选择水平方向的像素点；当 $d \geq 0$ 时，选择对角线方向的像素点. 决策参数 d 的推导过程：考虑圆的方程：$F(x, y) = x^2 + y^2 - r^2 = 0$, 在点 (x_k, y_k) 处，我们需要决定下一个点是选择 $(x_k + 1, y_k)$ 还是 $(x_k + 1, y_k - 1)$。中点判别法考虑这两个候选点的中点 $(x_k + 1, y_k - \frac{1}{2})$。定义决策参数：$d_k = F(x_k + 1, y_k - \frac{1}{2}),= (x_k + 1)^2 + (y_k - \frac{1}{2})^2 - r^2$ 当 $d_k < 0$ 时，中点在圆内，选择 $(x_k + 1, y_k)$；当 $d_k \geq 0$ 时，中点在圆外或圆上，选择 $(x_k + 1, y_k - 1)$。下一个决策参数 d_{k+1} 的递推关系：当 $d_k < 0$ 时（选择 E 点）：$d_{k+1} = F(x_k + 2, y_k - \frac{1}{2}) = d_k + 2(x_k + 1) + 1 = d_k + 2x_k + 3$ 2. 当 $d_k \geq 0$ 时（选择 SE 点）：$d_{k+1} = F(x_k + 2, (y_k - 1) - \frac{1}{2}) = d_k + 2(x_k + 1) - 2(y_k - 1) + 1 = d_k + 2(x_k - y_k) + 5$ 这就是代码中更新决策参数 d 的依据。初始值 $d_0 = 1.25 - r$ 是将 $k = 0$ 代入决策参数公式得到的。

Listing 2 实现 void Circle::draw()

```
void draw(Image &img) override {
    int x = 0, y = radius; float d = 1.25 - radius;
    auto drawCirclePoints = [&](int x, int y) {
        img.SetPixel(cx + x, cy + y, color); img.SetPixel(cx - x, cy + y, color);
        img.SetPixel(cx + x, cy - y, color); img.SetPixel(cx - x, cy - y, color);
        img.SetPixel(cx + y, cy + x, color); img.SetPixel(cx - y, cy + x, color);
        img.SetPixel(cx + y, cy - x, color); img.SetPixel(cx - y, cy - x, color);
    };
    drawCirclePoints(x, y);
    while (x <= y) {
        if (d < 0)  d += 2 * x + 3
        else {
            d += 2 * (x - y) + 5;
            y--;
        }
        x++;
        drawCirclePoints(x, y);
    }
}
```

1.3 颜色填充

洪水填充算法（Flood Fill Algorithm）是一种用于填充连通区域的算法。该算法从一个起始点开始，通过广度优先搜索的方式，将所有与起始点颜色相同且相连的像素点都替换为目标颜色。

Algorithm 3 Flood Fill Algorithm

Require: Starting point (cx, cy), target color $color$, Image object img
Ensure: Fill the connected region starting from the given point
1: Get the color at starting point: $targetColor = img(cx, cy)$
2: If $targetColor = color$ then return
3: Create queue Q and add starting point (cx, cy) to Q
4: **while** Q is not empty **do**
5: Get front pixel (x, y) from Q
6: **if** (x, y) is out of bounds OR color at $(x, y) \neq targetColor$ **then**
7: Continue to next iteration
8: **end if**
9: Set color at (x, y) to $color$
10: Add $(x + 1, y)$, $(x - 1, y)$, $(x, y + 1)$, $(x, y - 1)$ to Q
11: **end while**

采用广度优先搜索策略，使用队列存储待处理的像素点。对于每个待处理的像素点，首先检查其是否越界或是否需要填充（颜色是否与目标颜色相同）。如果需要填充，则将其颜色更改为目标颜色，并将其四个相邻像素点（上、下、左、右）加入队列。这个过程会一直持续到队列为空，即所有需要填充的像素都已处理完毕。

Listing 3 实现 void Fill::draw()

```
void draw(Image &img) override {
    Vector3f targetColor = img.GetPixel(cx, cy);
    if (targetColor == color) {
        return;
    }
    std::queue<std::pair<int, int>> pixels;
    pixels.push({cx, cy});
    int width = img.Width();
    int height = img.Height();
    while (!pixels.empty()) {
        auto [x, y] = pixels.front();
        pixels.pop();
        if (x < 0 || x >= width || y < 0 || y >= height ||
            img.GetPixel(x, y) != targetColor) {
            continue;
        }
        img.SetPixel(x, y, color);
        pixels.push({x + 1, y}); pixels.push({x - 1, y});
        pixels.push({x, y + 1}); pixels.push({x, y - 1});
    }
}
```

2 Honor Code

PA0 算法思路部分借鉴了《计算机图形学基础》（第二版）P22-P31 内容，就代码实现时的 bug 与 Claud-Sonnet 3.5 模型进行了交互，并借鉴了其部分做法

3 Problem and Solution

3.1 $k = \frac{dy}{dx}$ 当 $dx = 0$ 以及起止点相对位置讨论

在 1.1 部分的代码实现中，我本来是利用的原版的 Bresenham 画线算法，需要计算直线的斜率，然而没有考虑到除法分母为零的竖直直线情形，从而产生了错误的代码逻辑，产生了如图所示的测试例子；同时对起止点的位置进行讨论因此绘图没有正确处理。解决办法：采用改进后的算法，改用整数避免除法，做 $e_2 = 2 * err$ 的替换，并引入 sx 和 sy 控制起止点相对位置与绘制点自增方向的关系。

4 Suggestion

一个是建议可以给更多的测试例子；第二个是好像这三个算法都比较简单（?），也许可以增加一些反走样的实验。感觉上小作业难度和大作业难度跨越较大（?）上课内容感觉相对前沿，但上课内容和作业似乎线性无关。