作业

任意斜率直线画线法

DDA 画线法

算法原理

对于任意一直线段$L(P_0, P_1) \quad (P_0 \neq P_1)$，其两端点坐标为$P_0(x_0, y_0)$和$P_1(x_1, y_1)$，若斜率存在，则直线的微分方程为：

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \frac{\Delta y}{\Delta x} = k$$

DDA 是通过在某一位置(x_i, y_i)分别加上增量Δx和Δy，求得下一点(x_{i+1}, y_{i+1})的坐标，不断迭代从而生成直线。
具体原理如下：

- 若斜率存在，且当$x_1 - x_0$绝对值较大($x_0 \neq x_1, |k| \leq 1$)，以x方向作为步进方向，且每步增量为单位步长（$\Delta x = 1$，即一个像素），计算y方向的增量（$\Delta y = \Delta x \cdot k = k$），把每次计算出的$(x_{i+1}, y_{i+1})$经取整后送到显示器输出，则得到扫描转换后的直线。
- 若斜率存在，且当$y_1 - y_0$绝对值较大($x_0 \neq x_1, |k| \geq 1$)，以y方向作为步进方向，且每步增量为单位步长（$\Delta y = 1$，即一个像素），计算x方向的增量（$\Delta x = \Delta y \cdot k = k$），把每次计算出的$(x_{i+1}, y_{i+1})$经取整后送到显示器输出，则得到扫描转换后的直线。
- 若斜率不存在（$x_0 = x_1$），以y方向作为步进方向，且每步增量为单位步长（$\Delta y = 1$，即一个像素），x方向不存在增量（$\Delta x = 0$），把每次计算出的(x_{i+1}, y_{i+1})经取整后送到显示器输出，则得到扫描转换后的直线。

以上两种情况可以转换为如下通法：

> 分别计算两端点坐标$P_0(x_0, y_0), P_1(x_1, y_1)$的横纵坐标差值$\Delta x = x_1 - x_0, \Delta y = y_1 - y_0$，选绝对值最大的值作为迭代次数$steps$，则每次迭代的增量为$\mathrm{d}x = \frac{\Delta x}{steps}, \mathrm{d}y = \frac{\Delta y}{steps}$。然后令初始点为$P = P_0$，每次在点$P$处放置像素点后将$P$的横纵坐标加上相应增量，迭代$steps$次后即可生成直线。

算法表示

伪代码：

Algorithm 1: 数值微分法 (DDA)

Input: 直线 L 两端点坐标 $P_0(x_0, y_0), P_1(x_1, y_1)$, 画线颜色 $color$

$\Delta x \leftarrow x_1 - x_0$, $\Delta y \leftarrow y_1 - x_0$;
$steps \leftarrow \max(|\Delta x|, |\Delta y|)$;
$\mathrm{d}x \leftarrow \frac{\Delta x}{steps}$, $\mathrm{d}y \leftarrow \frac{\Delta y}{steps}$;
$P \leftarrow P_0$;
for $i \leftarrow 1$ **to** $steps + 1$ **do**
    putpixel (round(P), $color$);
    $P \leftarrow P + (\mathrm{d}x, \mathrm{d}y)$;
end

代码：

```
void Line_DDA(int x0, int y0, int x1, int y1, Color color)
{
    int delta_x, delta_y, steps, dx, dy;
    // 计算delta_x, delta_y, 确定steps, 并计算dx, dy
    delta_x = x1 - x0, delta_y = y1 - y0;
    steps = max(abs(delta_x), abs(delta_y));
    dx = delta_x / steps, dy = delta_y / steps;
    for (i = 1; i <= steps + 1; i++)
    {
        putpixel((int)(x + 0.5), (int)(y + 0.5), color); // 四舍五入生成像素点
        x += dx, y += dy;
    }
}
```

中点画线法

算法原理

对于任意一直线段$L(P_0, P_1) \quad (P_0 \neq P_1)$，其两端点坐标为$P_0(x_0, y_0)$和$P_1(x_1, y_1)$（保证$x_0 < x_1$），先讨论直线斜率$0 \leq k \leq 1$的情况。

令直线方程为为如下形式：

$$F(x, y) : ax + by + c = 0$$

可计算得到相应的参数为：$a = y_0 - y_1, b = x_1 - x_0, c = x_0y_1 - x_1y_0$，且可知$b > 0$。

则对于平面上任何点$P_i(x_i, y_i)$，与直线的相对位置有且只有三种情况：

$$F(x_i, y_i) \begin{cases} < 0 & \text{点 } (x_i, y_i) \text{ 位于直线下方} \\ = 0 & \text{点 } (x_i, y_i) \text{ 位于直线上} \\ > 0 & \text{点 } (x_i, y_i) \text{ 位于直线上方} \end{cases}$$

从直线左侧向右生成直线，设当前已确定与直线最近的像素点坐标为$P_i(x_i, y_i)$，则下一个与直线最近的右方像素，只能是正右方像素$P_{RM}(x_i + 1, y_i)$或右上方像素$P_{RU}(x_i + 1, y_i + 1)$，称二者为“候选像素”。
记P_{RM}和P_{RU}的中点为M，易知其坐标为$M(x_i + 1, y_i + 0.5)$。

再假设Q为直线L与垂直线$x = x_i + 1$的交点，则M与Q有三种关系：

- 若$F(M) > 0$，即M在Q(直线L)上方，说明P_{RM}离直线最近，应作为下一个像素。
- 若$F(M) = 0$，即M与Q重合(在直线L上)，说明P_{RM}与P_{RU}离直线等距离，两者均可作为下一个像素，这里约定P_{RM}作为下一个像素。
- 若$F(M) < 0$，即M在Q(直线L)下方，说明P_{RU}离直线最近，应作为下一个像素。

故可用中点M的函数值$F(M)$作为决策变量d_i，根据d_i的符号确定下一个像素的选取。

$$d_i \begin{cases} < 0 & \text{取右上方像素 } P_{RU}\ (x_i + 1, y_i + 1) \text{ 作为下一个像素} \\ \geq 0 & \text{取正右方像素 } P_{RM}\ (x_i + 1, y_i) \text{ 作为下一个像素} \end{cases}$$

选择相应像素后，将下两个候选像素的中点M'带入，可推导出下一个决策变量d_{i+1}的情况为：

$$d_i \begin{cases} < 0 & d_{i+1} = d_i + a + b \\ \geq 0 & d_{i+1} = d_i + a \end{cases}$$

注意到初始的决策变量为$d_0 = a + 0.5b$，会造成d_0为浮点数，进行浮点数加法，
但d只需要关注与0的关系，所以可以将d_0乘2，转换为整数加法，即$d_0 = 2a + b$，$d_{i+1} = d_i + 2a$或$d_{i+1} = d_i + 2(a + b)$。

对于$k > 1$的情况，其情况相当于$0 \le k \le 1$情况按直线$y = k$对称处理。
故可将点$P_i(x_i, y_i)$与直线的三种相对位置情况中的下方改称为右方，上方改称为左方，判断条件仍相同；
两个候选像素变为正上方像素$P_{MU}(x_i, y_i + 1)$或右上方像素$P_{RU}(x_i + 1, y_i + 1)$，中点坐标为$M(x_i + 0.5, y_i + 1)$。

仍用中点M的函数值$F(M)$作为决策变量d_i，决策变量d_i的每次迭代处理为：

$$d_i \begin{cases} < 0 & \text{取正上方像素 } P_{MU}\left(x_i, y_i + 1\right) \text{ 作为下一个像素 }, d_{i+1} = d_i + 2b \\ \ge 0 & \text{取右上方像素 } P_{RU}\left(x_i + 1, y_i + 1\right) \text{ 作为下一个像素 }, d_{i+1} = d_i + 2(a + b) \end{cases}$$

其中：$d_0 = a + 2b$。
当$P_i = P_1$时终止迭代。

对于其他情况也采用类似分析方法，故只给出决策变量d_i每次迭代的处理如下表，不再给出详细分析过程。

k情况	$k \in [0, 1]$	$k \in (1, +\infty)$	$k \in [-1, 0)$	$k \in (-\infty, -1)$
d_0	$d_0 = 2a + b$	$d_0 = a + 2b$	$d_0 = 2a - b$	$d_0 = a - 2b$
$d_i < 0$情况	`x++, y++, d+=2(a+b)`	`y++, d+=2b`	`x++, d+=2a`	`y--, x++, d+=2(a-b)`
$d_i \ge 0$情况	`x++, d+=2a`	`y++, x++, d+=2(a+b)`	`x++, y--, d+=2(a-b)`	`y--, d+=-2b`

算法表示

伪代码：

Algorithm 2: 中点画线法

Input: 直线 L 两端点坐标 $P_0(x_0, y_0), P_1(x_1, y_1)$, 画线颜色 *color*

if $x_0 > x_1$ **then**
    `swap` (P_0, P_1);
end
$a \leftarrow y_0 - y_1, b \leftarrow x_1 - x_0, c \leftarrow x_0 y_1 - x_1 y_0$;
$P_i \leftarrow P_0$;
if $k \in [0, 1]$ **then**
    $d \leftarrow 2a + b$;
    $\Delta P_L \leftarrow (1, 1), \Delta d_L \leftarrow 2(a + b)$;
    $\Delta P_G \leftarrow (1, 0), \Delta d_G \leftarrow 2a$;
else if $k \in (1, +\infty)$ **then**
    $d \leftarrow a + 2b$;
    $\Delta P_L \leftarrow (0, 1), \Delta d_L \leftarrow 2b$;
    $\Delta P_G \leftarrow (1, 1), \Delta d_G \leftarrow 2(a + b)$;
else if $k \in [-1, 0)$ **then**
    $d \leftarrow 2a - b$;
    $\Delta P_L \leftarrow (1, 0), \Delta d_L \leftarrow 2a$;
    $\Delta P_G \leftarrow (1, -1), \Delta d_G \leftarrow 2(a - b)$;
else
    $d \leftarrow a - 2b$;
    $\Delta P_L \leftarrow (1, -1), \Delta d_L \leftarrow 2(a - b)$;
    $\Delta P_G \leftarrow (0, -1), \Delta d_G \leftarrow -2b$;
end
while $P \neq P_1$ **do**
    `putpixel` (P, color);
    if $d < 0$ **then**
        $P \leftarrow P + \Delta P_L, d \leftarrow d + \Delta d_L$;
    else
        $P \leftarrow P + \Delta P_G, d \leftarrow d + \Delta d_G$;
    end
end
`putpixel` (P_1, color);

代码:

```
void Line_Midpoint(int x0, int y0, int x1, int y1, Color color)
{
    if (x0 > x1) // 保证x0 <= x1
        swap(x0, x1), swap(y0, y1);
    int a = y0 - y1, b = x1 - x0,   // 直线L的参数（c因为没有用到不用计算）
        d,                          // 决策变量d
        dd_L, dd_G,                 // 决策变量d的增量(L代表小于0的情况，G代表大于0的情况)
        x = x0, y = y0,             // 初始P坐标
        dPx_L, dPy_L, dPx_G, dPy_G; // P坐标的增量(L代表小于0的情况，G代表大于0的情况)
    // 根据k值分情况生成决策变量和增量
    if (-b <= a && a <= 0) // k ∈ [0, 1]
    {
        d = 2 * a + b;                            // d0 = 2a + b
        dPx_L = 1, dPy_L = 1, dd_L = 2 * (a + b); // d < 0情况
        dPx_G = 1, dPy_G = 0, dd_G = 2 * a;       // d >= 0 情况
    }
    else if (a < -b) // k ∈ (1, +∞)
    {
        d = a + 2 * b;                            // d0 = a + 2b
        dPx_L = 0, dPy_L = 1, dd_L = 2 * b;       // d < 0情况
        dPx_G = 1, dPy_G = 1, dd_G = 2 * (a + b); // d >= 0 情况
    }
    else if (0 < a && a <= b) // k ∈ [-1, 0)
    {
        d = 2 * a - b;                             // d0 = 2a - b
        dPx_L = 1, dPy_L = 0, dd_L = 2 * a;        // d < 0情况
        dPx_G = 1, dPy_G = -1, dd_G = 2 * (a - b); // d >= 0 情况
    }
    else // k ∈ (-∞, -1)
    {
        d = a - 2 * b;                             // d0 = a - 2b
        dPx_L = 1, dPy_L = -1, dd_L = 2 * (a - b); // d < 0情况
        dPx_G = 0, dPy_G = -1, dd_G = -2 * b;      // d >= 0 情况
    }
    // 迭代生成直线
    while (x != x1 || y != y1)
    {
        putpixel(x, y, color);
        if (d < 0)
            x += dPx_L, y += dPy_L, d += dd_L;
        else
            x += dPx_G, y += dPy_G, d += dd_G;
    }
    putpixel(x, y, color);
}
```

Bresenham 画线法

算法原理

其与中点画线算法的思想基本一致，只是在判断选P_{RM}和P_{RU}方法不同，
这里把用实际位置与中点的位置关系，变为了与P_{RM}和P_{RU}两距离的大小关系，
且约定$d_1 = d_2$时选择右上方像素P_{RU}。
仍先考虑$0 \le k \le 1$的情况。

用d_1和d_2来表示正右P_{RM}和右上两个候选像素的y值，与线段上理想y值（Q点y值）的差值，
即$d_1 = |P_{RM}Q|, d_2 = |P_{RU}Q|$，可得$d_1 - d_2 = 2k(x_i + 1) - 2y_i + 2b - 1$，
其中$k = \frac{\Delta y}{\Delta x}$，代入并在两边同乘$\Delta x$后得：

$$\Delta x(d_1 - d_2) = 2x_i\Delta y - 2y_i\Delta x + C$$

其中C为一常量，$C = 2\Delta y + \Delta x(2b-1)$。

设决策变量$d_i = \Delta x(d_1 - d_2)$，其符号与$d_1 - d_2$相同，起判别作用。
判别情况如下：

$$d_i \begin{cases} < 0 & \text{取正上方像素 } P_{MU}\ (x_i, y_i+1) \text{ 作为下一个像素} \\ \geq 0 & \text{取右上方像素 } P_{RU}\ (x_i+1, y_i+1) \text{ 作为下一个像素} \end{cases}$$

而对于决策变量的迭代如下：

$$\begin{gathered} d_{i+1} = 2x_{i+1}\Delta y - 2y_{i+1}\Delta x + C \\ d_{i+1} - d_i = 2(x_{i+1} - x_i)\Delta y - 2(y_{i+1} - y_i)\Delta x \\ d_{i+1} = d_i + 2\Delta y - 2\Delta x(y_{i+1} - y_i) \end{gathered}$$

- 当$d_i < 0$时，$y_{i+1} = y_i$，故$d_{i+1} = d_i + 2\Delta y$
- 当$d_i \geq 0$时，$y_{i+1} = y_i + 1$，故$d_{i+1} = d_i + 2(\Delta y - \Delta x)$

总结如下：

$$d_i \begin{cases} < 0 & \text{取正上方像素 } P_{MU}\ (x_i, y_i+1) \text{ 作为下一个像素 }, d_{i+1} = d_i + 2\Delta y \\ \geq 0 & \text{取右上方像素 } P_{RU}\ (x_i+1, y_i+1) \text{ 作为下一个像素 }, d_{i+1} = d_i + 2(\Delta y - \Delta x) \end{cases}$$

而对于决策变量初始值d_0，仍取左端点$P_0(x_0, y_0)$，计算得到d_0为：

$$\begin{gathered} d_0 = 2x_0\Delta y - 2y_0\Delta x \\ C = 2\Delta y + \Delta x(2b-1) \\ b = y_0 - \frac{\Delta y}{\Delta x}x_0 \end{gathered}$$

整理得：

$$d_0 = 2\Delta y - \Delta x$$

当$P_i = P_1$时终止迭代。

对于其他情况按中点画线法类似分析，可得到决策变量d_i每次迭代的处理如下表：

k情况	$k \in [0, 1]$	$k \in (1, +\infty)$	$k \in [-1, 0)$	$k \in (-\infty, -1)$
d_0	$d_0 = 2\Delta y - \Delta x$	$d_0 = 2\Delta x - \Delta y$	$d_0 = -2\Delta y - \Delta x$	$d_0 = 2\Delta x + \Delta y$
$d_i < 0$情况	`x++, d+=2Δy`	`y++, d+=2Δx`	`x++, d+=-2Δy`	`y--, d+=2Δx`
$d_i \geq 0$情况	`x++, y++, d+=2(Δy-Δx)`	`x++, y++, d+=2(Δx-Δy)`	`x++, y--, d+=-2(Δy+Δx)`	`x++, y--, d+=2(Δx+Δy)`

算法表示

伪代码：

Algorithm 3: Bresenham 画线法

Input: 直线 L 两端点坐标 $P_0(x_0, y_0), P_1(x_1, y_1)$, 画线颜色 *color*

if $x_0 > x_1$ **then**
  `swap` (P_0, P_1);
end
$\Delta x = x_1 - x_0, \Delta y = y_1 - y_0$;
$P_i \leftarrow P_0$;
if $k \in [0, 1]$ **then**
  $d \leftarrow 2\Delta y - \Delta x$;
  $\Delta P_L \leftarrow (1, 0), \Delta d_L \leftarrow 2\Delta y$;
  $\Delta P_G \leftarrow (1, 1), \Delta d_G \leftarrow 2(\Delta y - \Delta x)$;
else if $k \in (1, +\infty)$ **then**
  $d \leftarrow 2\Delta x - \Delta y$;
  $\Delta P_L \leftarrow (0, 1), \Delta d_L \leftarrow 2\Delta x$;
  $\Delta P_G \leftarrow (1, 1), \Delta d_G \leftarrow 2(\Delta x - \Delta y)$;
else if $k \in [-1, 0)$ **then**
  $d \leftarrow -2\Delta y - \Delta x$;
  $\Delta P_L \leftarrow (1, 0), \Delta d_L \leftarrow -2\Delta y$;
  $\Delta P_G \leftarrow (1, -1), \Delta d_G \leftarrow -2(\Delta y + \Delta x)$;
else
  $d \leftarrow 2\Delta x + \Delta y$;
  $\Delta P_L \leftarrow (0, -1), \Delta d_L \leftarrow 2\Delta x$;
  $\Delta P_G \leftarrow (1, -1), \Delta d_G \leftarrow 2(\Delta x + \Delta y)$;
end
while $P \neq P_1$ **do**
  `putpixel` (P, color);
  if $d < 0$ **then**
    $P \leftarrow P + \Delta P_L, d \leftarrow d + \Delta d_L$;
  else
    $P \leftarrow P + \Delta P_G, d \leftarrow d + \Delta d_G$;
  end
end
`putpixel` (P_1, color);

代码：

```
void Line_Bresenham(int x0, int y0, int x1, int y1, Color color)
{
    if (x0 > x1) // 保证x0 <= x1
        swap(x0, x1), swap(y0, y1);
    int Delta_x = x1 - x0, Delta_y = y1 - y0,
        d,                          // 决策变量d
        dd_L, dd_G,                 // 决策变量d的增量(L代表小于0的情况, G代表大于0的情况)
        x = x0, y = y0,             // 初始P坐标
        dPx_L, dPy_L, dPx_G, dPy_G; // P坐标的增量(L代表小于0的情况, G代表大于0的情况)
    // 根据k值分情况生成决策变量和增量
    if (0 <= Delta_y && Delta_y <= Delta_x) // k ∈ [0, 1]
    {
        d = 2 * Delta_y - Delta_x;                          // d0 = 2Δy - Δx
        dPx_L = 1, dPy_L = 0, dd_L = 2 * Delta_y;           // d < 0情况
        dPx_G = 1, dPy_G = 1, dd_G = 2 * (Delta_y - Delta_x); // d >= 0 情况
    }
    else if (Delta_x < Delta_y) // k ∈ (1, +∞)
    {
        d = 2 * Delta_x - Delta_y;                          // d0 = 2Δx - Δy
        dPx_L = 0, dPy_L = 1, dd_L = 2 * Delta_x;           // d < 0情况
        dPx_G = 1, dPy_G = 1, dd_G = 2 * (Delta_x - Delta_y); // d >= 0 情况
    }
    else if (-Delta_x <= Delta_y && Delta_y < 0) // k ∈ [-1, 0)
    {
        d = -2 * Delta_y - Delta_x;                           // d0 = -2Δy - Δx
        dPx_L = 1, dPy_L = 0, dd_L = -2 * Delta_y;            // d < 0情况
        dPx_G = 1, dPy_G = -1, dd_G = -2 * (Delta_y + Delta_x); // d >= 0 情况
    }
    else // k ∈ (-∞, -1)
    {
        d = 2 * Delta_x + Delta_y;                           // d0 = 2Δx + Δy
        dPx_L = 0, dPy_L = -1, dd_L = 2 * Delta_x;           // d < 0情况
        dPx_G = 1, dPy_G = -1, dd_G = 2 * (Delta_x + Delta_y); // d >= 0 情况
    }
    // 迭代生成直线
    while (x != x1 || y != y1)
    {
        putpixel(x, y, color);
        if (d < 0)
            x += dPx_L, y += dPy_L, d += dd_L;
        else
            x += dPx_G, y += dPy_G, d += dd_G;
    }
    putpixel(x, y, color);
}
```

三种算法的比较分析

- 算法精度
  对于三种算法生成的直线精度，在这里采用残差平方和SSE作为评判标准。
  对于同样两端点分别调用三种算法生成直线并计算其SSE，多次运行比较后发现值均相同，如下图所示：

```
Error of DDA:2019.831243
Error of MID:2019.831243
Error of BSH:2019.831243
--------------------------------
Error of DDA:259971.735867
Error of MID:259971.735867
Error of BSH:259971.735867
--------------------------------
Error of DDA:261101.235978
Error of MID:261101.235978
Error of BSH:261101.235978
--------------------------------
Error of DDA:2499123.602321
Error of MID:2499123.602321
Error of BSH:2499123.602321
--------------------------------
Error of DDA:2500079.602452
Error of MID:2500079.602452
Error of BSH:2500079.602452
--------------------------------
```

故可以判断三种算法的精度基本一致。

- 算法速度
  由算法代码可分析：
  对于DDA算法，其存在浮点数运算；
  而对于中点画线法和 Bresenham 画线法，两者均把浮点数运算转换为了整数运算，而这两种算法的代码操作相近。
  故DDA算法的速度慢于中点画线法和 Bresenham 画线法，而中点画线法和 Bresenham 画线法两算法速度相近。

画圆法

中点画圆法

算法原理

对于任意一圆$C(P,r)$，其中$P(x_P,y_P)$为圆心，r为半径。根据平移不变性，可以先按照圆心在原点的情况生成圆$C'(O,r)$，然后将所有生成的点按向量$\vec{OP}$平移即可。
而对于生成圆的操作，由于圆自身的对称性，可以将其以$x=0,y=0,y=x,y=-x$这四条对称轴平均分成八份，每次只用考虑生成某一部分的八分圆弧，其余部分按照对称性生成即可。若已生成圆O'某一八分圆弧上一点$Q(x,y)$，则生成圆O八个部分对应对称的点的代码如下：

```
void CirclePoints(int xP, int yP, int x, int y, int color)
{
    putpixel(xP+x, yP+y, color);
    putpixel(xP+y, yP+x, color);
    putpixel(xP-x, yP+y, color);
    putpixel(xP+y, yP-x, color);
    putpixel(xP+x, yP-y, color);
    putpixel(xP-y, yP+x, color);
    putpixel(xP-x, yP-y, color);
    putpixel(xP-y, yP-x, color);
}
```

由于平移不变性和八分对称性，接下来只考虑中心在原点$(0,0)$，半径为R（整数）的圆C'中，从点$A(0,R)$顺时针到点$B(\frac{R}{\sqrt{2}},\frac{R}{\sqrt{2}})$的八分圆弧$\overset{\frown}{AB}$上点的生成方式。

圆C'函数形式如下：

$$F(x,y)=x^2+y^2-R^2$$

则对于平面上任何点$P_i(x_i, y_i)$，与圆C'的相对位置有且只有三种情况：

$$F(x_i, y_i)\begin{cases} <0 & 点\ (x_i, y_i)\ 位于圆内 \\ =0 & 点\ (x_i, y_i)\ 位于圆上 \\ >0 & 点\ (x_i, y_i)\ 位于圆外 \end{cases}$$

从圆弧左侧向右侧生成圆弧，设当前已确定与圆弧最近的像素点坐标为$P_i(x_i, y_i)$，则下一个与圆弧最近的右方像素，只能是正右方像素$P_{RM}(x_i+1, y_i)$或右下方像素$P_{RB}(x_i+1, y_i-1)$，称二者为“候选像素”。
记P_{RB}和P_{RM}的中点为M，易知其坐标为$M(x_i+1, y_i-0.5)$。

再假设Q为理想圆弧$\overset{\frown}{AB}$与垂直线$x=x_i+1$的交点，则M与Q有三种关系：

- 若$F(M)>0$，即M在Q上方(圆弧$\overset{\frown}{AB}$外)，说明P_{RB}离直线最近，应作为下一个像素。
- 若$F(M)=0$，即M与Q重合(圆弧$\overset{\frown}{AB}$上)，说明P_{RB}与P_{RM}离圆弧等距离，两者均可作为下一个像素，这里约定P_{RB}作为下一个像素。
- 若$F(M)<0$，即M在Q(圆弧$\overset{\frown}{AB}$内)，说明P_{RM}离直线最近，应作为下一个像素。

故可用中点M的函数值$F(M)$作为决策变量d_i，根据d_i的符号确定下一个像素的选取。

$$d_i\begin{cases} <0 & 取正右方像素\ P_{RM}\ (x_i+1, y_i)\ 作为下一个像素 \\ \geq 0 & 取右下方像素\ P_{RB}\ (x_i+1, y_i-1)\ 作为下一个像素 \end{cases}$$

选择相应像素后，将下两个候选像素的中点M'带入，可推导出下一个决策变量d_{i+1}的情况为：

$$d_i\begin{cases} <0 & d_{i+1}=d_i+2x_i+3 \\ \geq 0 & d_{i+1}=d_i+2(x_i-y_i)+5 \end{cases}$$

初始的决策变量为$d_0=1.25-R$，会造成d_0为浮点数，进行浮点数加法，
但d只需要关注与0的关系，注意到迭代式中d_{i+1}与d_i等系数，且R以及之后的迭代运算中只会出现整数，则任一决策变量d_i的所有取值集合$D=\{x|x=0.25+k, k\in Z\}$，
故可用$d_0-0.25$代替d_0，即$d_0'=1-R$。

由于迭代中d_{i+1}与d_i等系数，迭代过程中每一项的偏移值不变，则对于任意d_i'其值均为$d_i-0.25$，即对应所有取值集合变为$D'=\{x|x=k, k\in Z\}$。
对于d_i'仍与0进行大小关系判断，d_i'相当于d_i沿x负半轴平移了0.25个单位，而d_i原本取值集合为$D=\{x|x=0.25+k, k\in Z\}$，向负半轴平移0.25个单位后，对于$k\neq 0$的每个元素与0的大小关系不会发生变化；而$k=0$即$d_i=0.25$平移后$d_i'=0$，两者均≥ 0执行取正右方像素的决策，
因此平移后每个决策变量的执行的决策仍一致，故可以通过该平移变化使得浮点数运算转换为整数运算。

将上述原理总结为：

$$d_i\begin{cases} <0 & 取正右方像素\ P_{RM}\ (x_i+1, y_i)\ 作为下一个像素\ , d_{i+1}=d_i+2x_i+3 \\ \geq 0 & 取右下方像素\ P_{RB}\ (x_i+1, y_i-1)\ 作为下一个像素\ , d_{i+1}=d_i+2(x_i-y_i)+5 \end{cases}$$

其中：$d_0=1-R$。
当$x>y$时终止迭代。

算法表示

伪代码：

Algorithm 4: 中点画圆法

Input: 圆 C 圆心 $C(x, y)$, 半径 R, 画线颜色 $color$

$P_1 \leftarrow (0, R)$;
$d_1 \leftarrow 1 - R$;
$i \leftarrow 1$;
while $P_i \neq (\frac{R}{\sqrt{2}}, \frac{R}{\sqrt{2}})$ **do**
  `DrawCirclePoints` (P_i, color);
  if $d < 0$ **then**
    $P_{i+1} \leftarrow (x_i + 1, y_i), d_{i+1} \leftarrow d_i + 2x_i + 3$;
  else
    $P_{i+1} \leftarrow (x_i + 1, y_i - 1), d_{i+1} \leftarrow d_i + 2(x_i - y_i) + 5$;
  end
  $i \leftarrow i + 1$;
end
`DrawCirclePoints` (P_i, color);

代码：

```
void DrawCirclePoints(int xc, int yc, int x, int y, Color color)
{
    putpixel(xc + x, yc + y, color);
    putpixel(xc + y, yc + x, color);
    putpixel(xc - x, yc + y, color);
    putpixel(xc + y, yc - x, color);
    putpixel(xc + x, yc - y, color);
    putpixel(xc - y, yc + x, color);
    putpixel(xc - x, yc - y, color);
    putpixel(xc - y, yc - x, color);
}
void Circle_Midpoint(int xc, int yc, int R, Color color)
{
    int x = 0, y = R, // 初始P坐标
        d = 1 - R;    // 决策变量d
    while (x < y)
    {
        DrawCirclePoints(xc, yc, x, y, color);
        if (d < 0)
            d += 2 * x + 3, x++;
        else
            d += 2 * (x - y) + 5, x++, y--;
    }
    DrawCirclePoints(xc, yc, x, y, color);
}
```

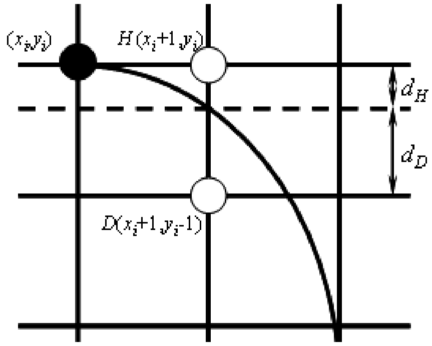
Bresenham 画圆法

算法原理

其跟中点画线法与 Bresenham 画线法的区别类似，也是在选择P_{RM}和P_{RU}时，用P_{RM}和P_{RU}两距离的大小关系判断。
仍考虑跟中点画圆法一样的八分圆弧，候选像素仍是正右方像素$H(x_i + 1, y_i)$和右下方像素$D(x_i + 1, y_i - 1)$，此时两点H, D与理想圆弧存在四种位置关系：

- H, D均在圆内
- H在圆外（或圆上），D在圆内
- H在圆外，D在圆上
- H, D均在圆外

对于上第一种情况可确定候选像素应选择H；第三、四种情况候选像素应选择D；而对于第二种情况需要考虑H, D点和垂线HD与理想圆弧的交点M的距离，若$|HM| < |DM|$则选择H，若$|HM| < |DM|$选择D（若$|HM| = |DM|$则可约定选择H），如下图所示：



然而对于交点M直接求得较为复杂，这里采用近似的方法来代替d_H与d_D，将H, D分别带入圆C'函数$F(x, y)$中，记为Δ_H, Δ_D，则$\Delta_H = |OH|^2 - |OM|^2, \Delta_D = |OD|^2 - |OM|^2$，记$\Delta_{HD} = \Delta_H + \Delta_D$，则$\Delta_{HD} \approx |OH| - |OM| + |OD| - |OM| \approx d_H - d_M$，故若$\Delta_{HD} \leq 0$，则应选择$H$，否则选择$D$。

故对于当前已确定点$P_i(x_i, y_i)$，设决策变量$d_i = \Delta D = (x_i + 1)^2 + (y_i - 1)^2 - R^2$，综合整理后可得迭代情况如下：

$$\begin{cases} d_i < 0 \ \& \ 2(d_i + y_i) - 1 <= 0 & \text{取正右方像素 } H(x_i + 1, y_i) \text{ 作为下一个像素}, d_{i+1} = d_i + 2x_i + 3 \\ \text{否则} & \text{取右下方像素 } D(x_i + 1, y_i - 1) \text{ 作为下一个像素}, d_{i+1} = d_i + 2(x_i - y_i + 3) \end{cases}$$

其中：$d_0 = 2(1 - R)$。
当$x > y$时终止迭代。

算法表示

伪代码：

Algorithm 5: Bresenham 画圆法

Input: 圆 C 圆心 $C(x, y)$, 半径 R, 画线颜色 $color$

$P_1 \leftarrow (0, R)$;
$d_1 \leftarrow 2(1 - R)$;
$i \leftarrow 1$;
while $P_i \neq (\frac{R}{\sqrt{2}}, \frac{R}{\sqrt{2}})$ **do**
  `DrawCirclePoints` (P_i, color);
  if $d < 0$ & $2(d + y) - 1 <= 0$ **then**
    $P_{i+1} \leftarrow (x_i + 1, y_i), d_{i+1} \leftarrow d_i + 2x_i + 3$;
  else
    $P_{i+1} \leftarrow (x_i + 1, y_i - 1), d_{i+1} \leftarrow d_i + 2(x_i - y_i + 3)$;
  end
  $i \leftarrow i + 1$;
end
`DrawCirclePoints` (P_i, color);

代码：

```
// DrawCirclePoints()函数已在中点画圆法给出，这里不再给出
void Circle_Bresenham(int xc, int yc, int R, COLORREF color)
{
    int x = 0, y = R,  // 初始P坐标
        d = 2 - 2 * R; // 决策变量d
    while (x < y)
    {
        DrawCirclePoints(xc, yc, x, y, color);
        if (d < 0 && 2 * (d + y) - 1 <= 0)
            d += 2 * x + 3, x++;
        else
            d += 2 * (x - y + 3), x++, y--;
    }
    DrawCirclePoints(xc, yc, x, y, color);
}
```

多边形逼近画圆法

算法推导

对于一正多边形，若其边数足够多时，则可以近似看作圆。利用平移不变性，仍考虑生成中心为原点O，各顶点距中心距离为R的正n边形，可以先通过计算各顶点的坐标，然后利用直线生成算法进行临近顶点的连线，从而生成正多边形。

由于是正n边形，临近顶点与中心连线形成的夹角均为$\frac{2\pi}{n}$，记为θ。对于已生成的圆上一顶点$P_i(x_i, y_i)$，其与x轴正方向的夹角为α，那么存在如下关系：

$$\begin{cases} x_i = R\cos\alpha \\ y_i = R\sin\alpha \end{cases}$$

则下一个顶点可表示为：

$$\begin{cases} x_{i+1} = R\cos(\alpha+\theta) = x_i\cos\theta - y_i\sin\theta \\ y_{i+1} = R\sin(\alpha+\theta) = y_i\cos\theta + x_i\sin\theta \end{cases}$$

假设初始顶点为正上方顶点$P_1(0, R)$，按照上述递推式依次生成n个顶点，然后采用 Bresenham 画线法连接相邻顶点，从而绘制出多边形完成多边形逼近画圆法。

经过个人测试，当多边形边数定位$3\sqrt{R}$时，在$R \geq 25$时均有较好的生成效果（对于较小的R，任意n其生成效果均不理想）。

算法表示

伪代码：

Algorithm 6: 多边形逼近画圆法

Input: 圆 C 圆心 $C(x, y)$, 半径 R, 画线颜色 $color$

$n \leftarrow 3\sqrt{R}$;
$\theta \leftarrow \frac{2\pi}{n}$;
$P_1 \leftarrow (0, R)$;
$i \leftarrow 2$;
while $i <= n$ **do**
    $P_i = (x_{i-1}\cos\theta - y_{i-1}\sin\theta, y_{i-1}\cos\theta + x_{i-1}\sin\theta)$;
    `DrawLine` (P_{i-1},P_i, color);
    $i \leftarrow i + 1$;
end
`DrawLine` (P_n,P_1, color);

代码:

```
const double PI = 3.141592653589793;
void Circle_Polygon(int xc, int yc, int R, COLORREF color)
{
    int n = 3 * sqrt(R); // 正多边形边数
    double cosTheta = cos(2.0 * PI / n), sinTheta = sin(2.0 * PI / n),
           xl = 0, yl = R, // 上一个生成的顶点
           x = 0, y = R;   // 当前生成的顶点
    for (int i = 2; i <= n; i++)
    {
        xl = x, yl = y;
        x = xl * cosTheta - yl * sinTheta, y = yl * cosTheta + xl * sinTheta;
        Line_Midpoint(xc + (int)(xl + 0.5), yc + (int)(yl + 0.5), xc + (int)(x + 0.5), yc + (int)(y + 0.5), color);
    }
    Line_Midpoint(xc + (int)(x + 0.5), yc + (int)(y + 0.5), xc, yc + R, color);
}
```

三种算法的比较分析

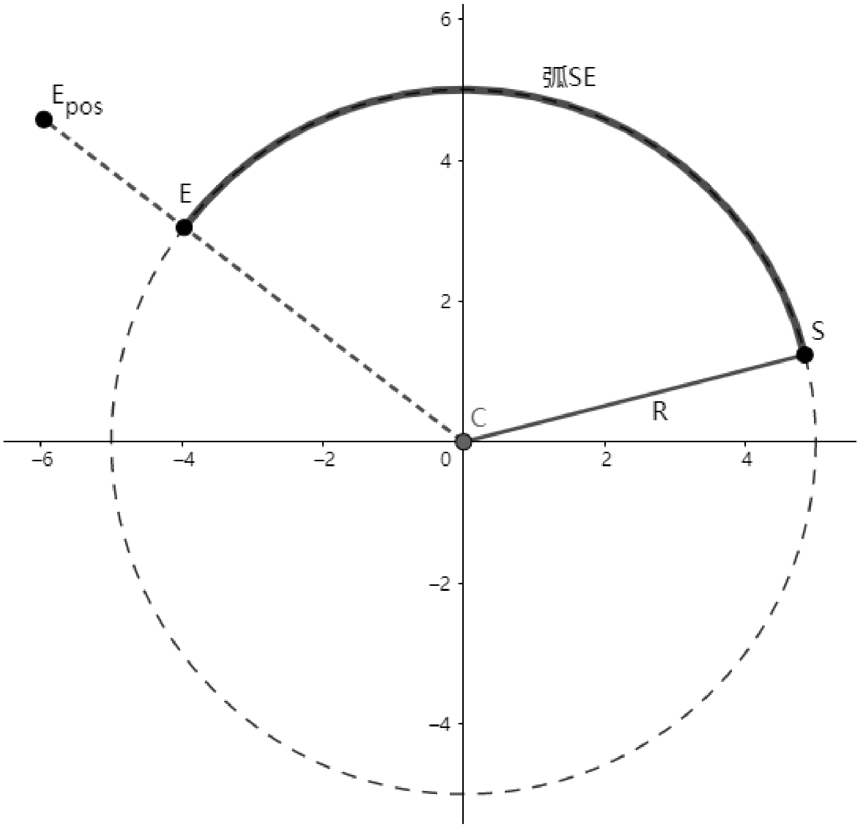
- 算法精度
  对于三种算法生成的圆形精度，由于中点画圆法与 Bresenham 画圆法几乎相近，故两者精度相同；
  而对于多边形逼近法，其精度较低，尤其是当圆半径过大或过小时精度更低。
- 算法速度
  中点画圆法与 Bresenham 画圆法两者运算速度大致一致，Bresenham 画圆法由于多了一步乘法运算会略慢于中点画圆法；
  而对于多边形算法，其直线相连的速度远快于等价的圆弧生成的速度，故其速度快于中点画圆法和 Bresenham 画圆法。

任意角度画弧法

算法推导

对于圆弧的绘制，由于失去了八分对称性，若直接生成过于复杂。其可以在画圆算法的基础上，在以八分对称性绘制像素点时，判断该像素点是否在需要绘制的圆弧上再绘制即可。称需要绘制的像素点集合为绘制区域D，若简单的以角度判断像素点是否在D中则复杂度过高，需尽量采用原有的直角坐标信息来判断，因此这里给出一种复杂度较低的判断方法。

对于圆弧的描述，这里采用三点描述法，即给定圆弧所在圆的圆心$C(x_c, y_c)$、圆弧的起点$S(x_s, y_s)$和终点方位点$E_{pos}(x_{epos}, y_{epos})$，其中圆心和圆弧起点的距离$CS$即为圆的半径$R$。需注意由于用户很难在确定圆心和起点后，给出恰好在圆上的圆弧终点，因此圆弧的实际终点并不是终点方位点，而是终点方位点和圆心的连线CE_{pos}与圆C的交点，记为$E(x_e, y_e)$，如下图所示。

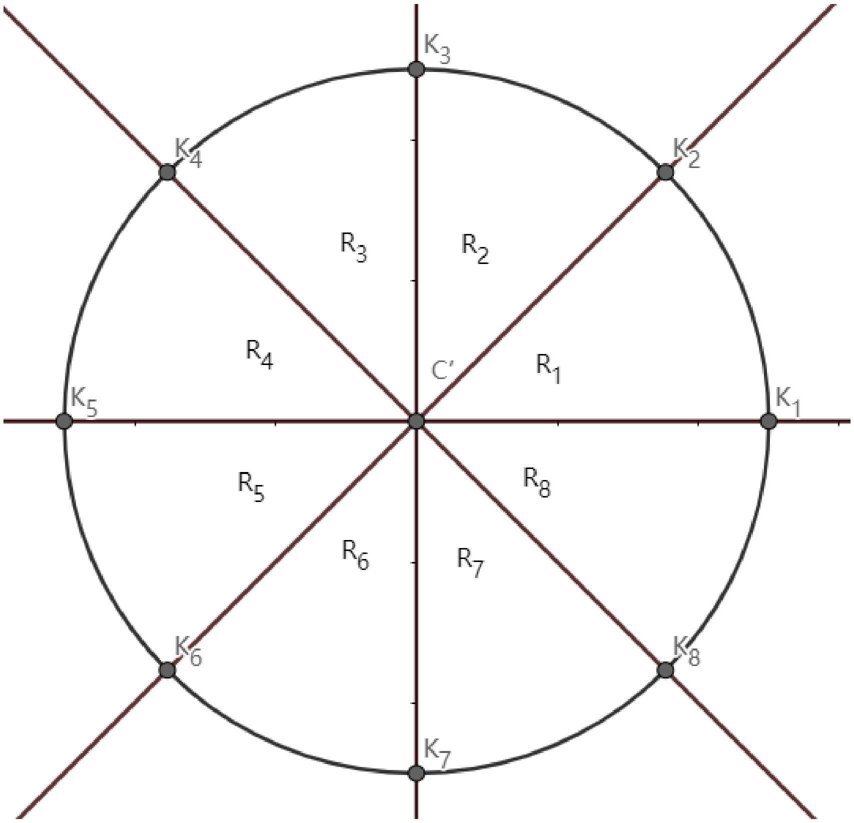


仍考虑平移不变性，将圆C平移到圆心为原点的位置变换为圆C'，将平移后所有标识都加上$'$表示（如$E'(x_e', y_e')$，其中$x_e' = x_e - x_c, y_e' = y_e - y_c$）。首先需要根据终点方位点确定圆$C'$上的实际圆弧终点，可通过$C'E_{pos}'$的线段与圆的交点确定，若直线$C'E_{pos}'$斜率不存在，则根据$y_{epos}'$的大小可确定$E$点：若$E_{pos}'$在$x$轴上方则$E$为$(0, R)$，否则为$(0, -R)$；若斜率存在，可先求得斜率为$k = \frac{y_{epos}'}{x_{epos}'}$，联立直线$y = kx$与圆方程$x^2 + y^2 = R^2$可得：

$$x_e' = \pm\frac{R}{\sqrt{k^2+1}}$$

若E_{pos}'在y轴右侧则E为(x_e', kx_e')，否则为$(-x_e', -kx_e')$。在求得坐标后可对坐标进行四舍五入化整处理。

然后根据八分对称性，将圆C'划分成八个区域，将最右侧点$(0, R)$逆时针第一个区域作为区域1（记作R_1），其余区域按逆时针顺序依次编号为R_i。同时将四条对称轴与圆C'形成的八个点，称为界点，并将最右侧的界点$(0, R)$作为界点1（记作K_1），其余界点按逆时针顺序依次编号为K_i。如下图：



对于圆C'上任意一点$P_i(x_i, y_i)$，根据其与四条对称轴$x = 0, y = 0, y = x, y = -x$的位置关系，可以唯一确定其属于哪一个区域或在哪一个界点上（如若$y_i < x_i \;\&\; y > 0$，则$P_i \in R_1$；若$y = 0 \;\&\; x > 0$，则$P_i = K_1$。

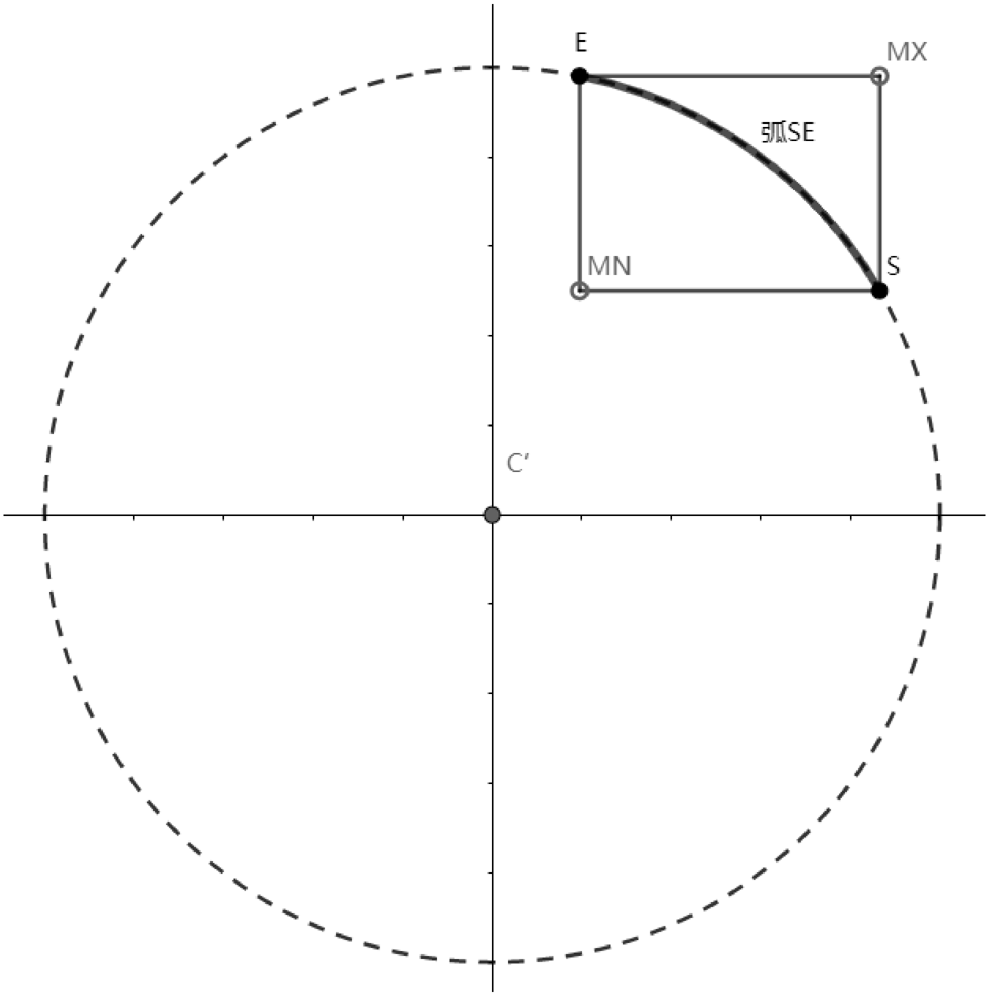
对于任一区域R_i，其包含的像素点集存在以下三种状态：

- R_i中的所有像素点均不会被绘制，称为非绘制区域，即$R_i \cap D = \emptyset$
- R_i中的部分像素点会被绘制，部分像素点不会被绘制，称为部分绘制区域，即$R_i \cap D \neq \emptyset$
- R_i中的所有像素点均会被绘制，称为全绘制区域，即$R_i \subseteq D$

确定出S和E所属的区域或界点后，则可根据其逆时针的相对顺序，确定出八个区域的性质，从而确定绘制区域。共存在以下四种情况：

- 若S, E分别在界点K_i, K_j上：则其之间（S逆时针到E之间的区域，下同）的区域$R_i \sim R_{j-1}$均为全绘制区域；其之外（S顺时针到E之间的区域，下同）的其余区域均为非绘制区域。
- 若S在界点K_i上、E在区域R_j中：则其之间的区域$R_i \sim R_{j-1}$为全绘制区域（若$i > j - 1$则不存在）；K_j到E之间的区域为部分绘制区；其之外的其余区域为非绘制区域。
- 若S在区域R_i中、E在界点K_j上：则其之间的区域$R_{i+1} \sim R_j$为全绘制区域（若$i + 1 > j$则不存在）；S到K_{i+1}之间的区域为部分绘制区；其之外的其余区域为非绘制区域。
- 若S在区域R_i中、E在区域R_j上：则其之间的区域$R_{i+1} \sim R_{j-1}$为全绘制区域（若$i + 1 > j - 1$则不存在）；S到K_{i+1}与K_j到E之间（若K_{i+1}在E逆时针侧则为S到E之间）的区域为部分绘制区；其之外的其余区域为非绘制区域。

对于全绘制区域和非绘制区域，可采用0和1二元值标记它们的状态，在判断某像素点是否属于绘制区域时只需要读取该状态值，若其值为0则不在绘制区域停止绘制；若其值为1则在绘制区域继续绘制。而对于部分绘制区域，根据上述情况可知最多不超过2个，且边界端点均属于同一象限，因此可以记录两端点坐标形成的矩形区域中，最左下的坐标$MN(x_{MN}, y_{MN})$和最右上坐标$MX(x_{MX}, y_{MX})$，若像素点在该矩形区域$Rect$范围内，则属于绘制区域，否则不属于，如下图：



须同时将该区域的状态标记为另一值-1，表示需要根据像素坐标与矩形区域的位置关系进一步判断。同时对于上述第四种情况，若S,E属于同一象限且S在E的顺时针侧，则可将S到K_{i+1}与K_j到E之间的区域合并为S到E之间的区域，以减少判断区域数。

之后就按照中点画圆法的方法，生成第二区域的八分圆弧以生成整个圆，在绘制像素点的时候判断该像素点P_i是否属于绘制区域D再进行绘制。

算法表示

伪代码:

Algorithm 7: 任意角度画弧法

Input: 圆弧 $\widehat{SE}$ 所属圆的圆心 $C(x_c, y_c)$, 圆弧起点 $S(x_s, y_s)$, 圆弧终点方位点 $E_{pos}(x_{epos}, y_{epos})$, 画线颜色 $color$

// 求得平移到原点的圆其对应圆心、起点、终点方位点和实际终点

$C' \leftarrow (0,0), S' \leftarrow (x_s - x_c, y_s - y_c)$;

$E'_{pos} \leftarrow (x_{epos} - x_c, y_{epos} - y_c), E' \leftarrow C'E'_{pos} \cap \odot C'$;

// 确定起点和终点所属区域或界点

$region_s$ = `GetRegion`(S), $region_e$ = `GetRegion`(E);

// 划分完全绘制区域、部分绘制区域和非绘制区域

`Fill` (state,0);

if $region_s = K_i$ & $region_e = K_j$ **then**

  for $R = R_i \rightarrow R_{j-1}$ **do**

    state$[R] \leftarrow 1$;

  end

else if $region_s = K_i$ & $region_e = R_j$ **then**

  for $R = R_i \rightarrow R_{j-1}$ **do**

    state$[R] = 1$;

  end

  $MN \leftarrow (\min(x_{K_j}, x_E), \min(y_{K_j}, y_E)), MX \leftarrow (\max(x_{K_j}, x_E), \max(y_{K_j}, y_E))$;

  $Rect \leftarrow (MN, MX)$, state$[region_e] \leftarrow -1$;

else if $region_s = R_i$ & $region_e = K_j$ **then**

  for $R = R_{i+1} \rightarrow R_j$ **do**

    state$[R] = 1$;

  end

  $MN \leftarrow (\min(x_{K_{i+1}}, x_S), \min(y_{K_{i+1}}, y_S)), MX \leftarrow (\max(x_{K_{i+1}}, x_S), \max(y_{K_{i+1}}, y_S))$;

  $Rect \leftarrow (MN, MX)$, state$[region_s] \leftarrow -1$;

else

  for $R = R_{i+1} \rightarrow R_{j-1}$ **do**

    state$[R] = 1$;

  end

  if S, E 属于同一象限 & S 在 E 顺时针侧 **then**

    $MN \leftarrow (\min(x_S, x_E), \min(y_S, y_E)), MX \leftarrow (\max(x_S, x_E), \max(y_S, y_E))$;

    $Rect \leftarrow (MN, MX)$;

  else

    $MN_1 \leftarrow (\min(x_{K_j}, x_E), \min(y_{K_j}, y_E)), MX_1 \leftarrow (\max(x_{K_j}, x_E), \max(y_{K_j}, y_E))$;

    $MN_2 \leftarrow (\min(x_{K_{i+1}}, x_S), \min(y_{K_{i+1}}, y_S)), MX_2 \leftarrow (\max(x_{K_{i+1}}, x_S), \max(y_{K_{i+1}}, y_S))$;

    $Rect \leftarrow (MN_1, MX_2) \cup (MN_2, MX_2)$;

  end

  state$[region_s] \leftarrow -1$, state$[region_e] \leftarrow -1$;

end

// 八方对称遍历圆像素点，每次生成的像素点需要判断是否在绘制区域内

`DrawCircleOfArc`$(C, R, color, \text{state}, Rect)$;

代码：

```
struct Vector2
{
    int x, y;
    Vector2() {}
    Vector2(const int &_x, const int &_y) : x(_x), y(_y) {}
};

// 返回在哪个八分区域或界点上
int getRegion(int x, int y)
{
    if (y < x && y > 0)
        return 1;
    else if (y > x && x > 0)
        return 2;
    else if (y > -x && x < 0)
        return 3;
    else if (y < -x && y > 0)
        return 4;
    else if (y > x && y < 0)
        return 5;
    else if (y < x && x < 0)
        return 6;
    else if (y < -x && x > 0)
        return 7;
    else if (y > -x && y < 0)
        return 8;
    // 恰好在界点上的情况（一般情况不会让x==y==0)
    // 为了与区域区分，Ki记录为i+10（如K1记录为11)
    else if (x == 0)
        return y > 0 ? 13 : 17;
    else if (y == 0)
        return x > 0 ? 11 : 15;
    else if (y == x)
        return x > 0 ? 12 : 16;
    else if (y == -x)
        return y > 0 ? 14 : 18;
    else
        return -1;
}

int drawRegionState[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0}; // 某个区域的绘制状态（0 - 非绘制区域; 1 - 全绘制区域; -1 - 部分绘制区域)
int judgeCount = 0; // 部分绘制区域个数
Vector2 judgeRegion[2][2]; // 部分绘制区域的矩形区域

// 判断像素点(x, y)是否属于绘制区域
bool isInDrawRegion(int x, int y, int region)
{
    if (drawRegionState[region] != -1) // 如果所处区域不需要具体判断，则直接返回预判断结果
        return drawRegionState[region];
    for (int i = 0; i < judegCount; i++)
        if (judgeRegion[i][0].x <= x && x <= judgeRegion[i][1].x && judgeRegion[i][0].y <= y && y <= judgeRegion[i][1].y) /
            return true;
    return false;
}

// 八分画圆过程中根据是否在绘制区域进行对称绘制
void DrawArcPoints(int xc, int yc, int x, int y, COLORREF color)
{
    isInDrawRegion(x, y, 2) ? putpixel(xc + x, yc + y, color) : (void)0;
    isInDrawRegion(y, x, 1) ? putpixel(xc + y, yc + x, color) : (void)0;
    isInDrawRegion(-x, y, 3) ? putpixel(xc - x, yc + y, color) : (void)0;
    isInDrawRegion(y, -x, 8) ? putpixel(xc + y, yc - x, color) : (void)0;
    isInDrawRegion(x, -y, 7) ? putpixel(xc + x, yc - y, color) : (void)0;
    isInDrawRegion(-y, x, 4) ? putpixel(xc - y, yc + x, color) : (void)0;
```

```
    isInDrawRegion(-x, -y, 6) ? putpixel(xc - x, yc - y, color) : (void)0;
    isInDrawRegion(-y, -x, 5) ? putpixel(xc - y, yc - x, color) : (void)0;
}

void Arc(int xc, int yc, int _xs, int _ys, int _xe, int _ye, COLORREF color)
{
    // 确定对于圆心平移到原点的圆C'，其对应终点的在圆C'上的实际位置
    int xs = _xs - xc, ys = _ys - yc, xe = _xe - xc, ye = _ye - yc;
    double R = sqrt(xs * xs + ys * ys), k, tmp_xe;
    if (xe == 0)
        xe = 0, ye = _ye > 0 ? R + .5 : -R - .5;
    else
        k = (double)ye / xe, tmp_xe = xe > 0 ? R / sqrt(k * k + 1) : -R / sqrt(k * k + 1), xe = round(tmp_xe), ye = round(k
    // 确定起点和终点所属八分区域或界点
    int region_s = getRegion(xs, ys), region_e = getRegion(xe, ye);
    // 划分完全绘制区域、部分绘制区域和非绘制区域
    double a = R / sqrt(2);
    Vector2 baseJudge[9] = {Vector2(), Vector2(R + .5, 0), Vector2(a + .5, a + .5), Vector2(0, R + .5), Vector2(-a - .5, a
    memset(drawRegionState, 0, sizeof(drawRegionState));
    judegCount = 0;
    if (region_s > 10)
        if (region_e > 10) // 起点在界点上，终点也在界点上
            for (int i = (region_e -= 10, region_s -= 10); i != region_e; i = i % 8 + 1)
                drawRegionState[i] = 1;
        else // 起点在界点上，终点不在界点上
        {
            for (int i = region_s -= 10; i != region_e; i = i % 8 + 1)
                drawRegionState[i] = 1;
            judgeRegion[judegCount][0] = Vector2(min(baseJudge[region_e].x, xe), min(baseJudge[region_e].y, ye)), judgeRegi
            drawRegionState[region_e] = -1;
        }
    else if (region_e > 10) // 起点不在界点上，终点在界点上
    {
        for (int i = (region_e -= 10, region_s % 8 + 1); i != region_e; i = i % 8 + 1)
            drawRegionState[i] = 1;
        judgeRegion[judegCount][0] = Vector2(min(baseJudge[region_s % 8 + 1].x, xs), min(baseJudge[region_s % 8 + 1].y, ys)
        drawRegionState[region_s] = -1;
    }
    else // 起点和终点都不在界点上
    {
        for (int i = region_s % 8 + 1; i != region_e; i = i % 8 + 1)
            drawRegionState[i] = 1;
        if ((region_s - 1) / 2 == (region_e - 1) / 2 && region_s < region_e) // 为同一个象限，并且起点在终点顺时针侧（劣弧），
            judgeRegion[judegCount][0] = Vector2(min(xs, xe), min(ys, ye)), judgeRegion[judegCount++][1] = Vector2(max(xs,
        else
            judgeRegion[judegCount][0] = Vector2(min(baseJudge[region_s % 8 + 1].x, xs), min(baseJudge[region_s % 8 + 1].y,
            judgeRegion[judegCount][0] = Vector2(min(baseJudge[region_e].x, xe), min(baseJudge[region_e].y, ye)), judgeRegi
        drawRegionState[region_s] = drawRegionState[region_e] = -1;
    }
    // 八方生成圆(中点画圆法)，每次生成的像素点需要判断是否在绘制区域内
    int x = 0, y = R + 0.5,
        d = 1 - (R + 0.5);
    while (x < y)
    {
        DrawArcPoints(xc, yc, x, y, color);
        if (d < 0)
            d += 2 * x + 3, x++;
        else
            d += 2 * (x - y) + 5, x++, y--;
    }
    DrawArcPoints(xc, yc, x, y, color);
}
```