3 Discretización de Primitivas Gráficas

Antes de iniciar este capítulo, vale la pena aclarar el concepto de Primitiva Gráfica: Se refiere a las figuras básicas de dibujo vectorial, a partir de las cuales se realizan todas las demás. Estas figuras básicas son los puntos, los segmentos de rectas, círculos, elipses y polígonos.

Por otro lado, antes de comenzar con este capítulo, se sugiere hacer un repaso de los temas de geometría analítica básica. Aquí sólo se mencionarán las formas de las ecuaciones usadas en el resto del libro, sin describirlas formalmente ya que se asume que el lector ha aprobado algún curso básico de geometría analítica vectorial.

3.1. Recordatorio básico

Recuérdese la ecuación *pendiente-intersecto* de las líneas rectas:

$$y = mx + B \tag{3.1}$$

Recuérdese la ecuación *punto-pendiente* de las líneas rectas:

$$y - y_0 = m(x - x_0) \tag{3.2}$$

La equación *dos puntos* de las líneas rectas no verticales:

$$y - y_0 = \left(\frac{y_1 - y_0}{x_1 - x_0}\right)(x - x_0) \tag{3.3}$$

También se dispone de la ecuación canónica o implícita de las líneas rectas:

$$ax + by + c = 0 \tag{3.4}$$

Recuérdese la ecuación paramétrica de las circunferencias centradas en un punto dado:

$$(x - x_0)^2 + (y - y_0)^2 = R^2 \tag{3.5}$$

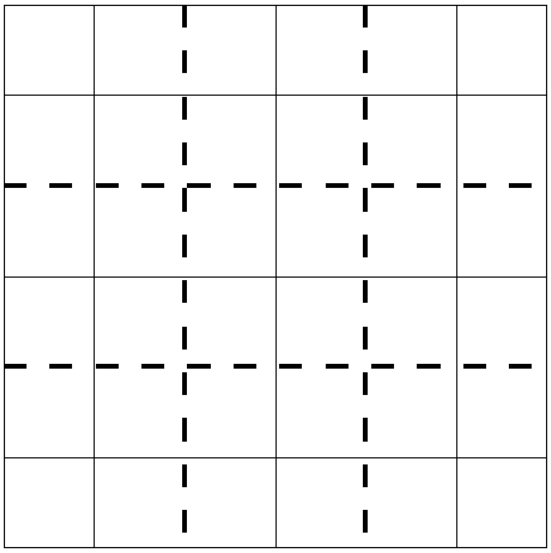

Figura 3.1: Representación abstracta de nueve pixeles

3.2. Simbología

Antes que nada, consideremos que los dispositivos principales sobre los que vamos a trabajar, son monitores cuyas pantallas están compuestas por pixeles. Los pixeles están dispuestos en forma de una matriz rectangular de puntos *casi* perfectamente cuadrados y cada uno puede brillar de un color independiente de los demás.

La figura 3.1 es la imágen que se usará para esquematizar en este capítulo cómo funcionan los algoritmos presentados

La cuadrícula de la figura 3.1 representa un grupo de nueve pixeles de una computadora. Las líneas gruesas de guiones representan las fronteras entre dichos pixeles y las intersecciones de las líneas continuas delgadas representan los centros geométricos de ellos.

Por simplicidad, vamos a suponer, en las deducciones, que la numeración de los pixeles se extiende desde la esquina inferior izquierda, el pixel $(0, 0)$ de las pantallas, hasta la esquina superior derecha. En realidad no es así, pero así es más parecido a los textos básicos de matemática, con los cuales el lector está familiarizado.

Se dice también que, respecto del pixel central, el pixel que está a su derecha, es su pixel «ESTE» o simplemente «E». El pixel de la esquina superior derecha, es el pixel «NOR-ESTE» o «NE» respecto del mismo pixel central. El pixel de la esquina inferior izquierda, es el pixel «SUR-OESTE» o «SO» respecto del pixel central, etc.

3.3. Algoritmo incremental básico

En esta sección, se procederá a describir un algoritmo para dibujar líneas rectas que es sencillo y eficiente desde el punto de vista matemático.

Analicemos el problema de dibujar una línea del punto (x_0,y_0), al (x_1,y_1).

Lo primero a realizar es la definición del modelo: Sea $y_i = mx_i + B$ la ecuación que define la linea, y los pixeles a encender (o a colorear) serán los pares ordenados: $(x_i, redondear(y_i))$. Evidentemente, $m = \frac{y_1-y_0}{x_1-x_0}$ y $B = y_0 - \left(\frac{y_1-y_0}{x_1-x_0}\right) x_0$.

Replanteando el problema en términos de un proceso iterativo, consideramos que $x_{i+1} - x_i = \Delta x$ para iterar sobre el eje horizontal, de izquierda a derecha. Entonces podemos plantear lo siguiente:

$$\begin{aligned} y_{i+1} &= mx_{i+1} + B \\ &= m(x_i + \Delta x) + B \\ &= mx_i + m\Delta x + B \\ &= m\Delta x + (mx_i + B) \\ y_{i+1} &= y_i + m\Delta x \end{aligned}$$

y si asumimos que $\Delta x = 1$ ya que avanzamos de columna en columna de pixeles, entonces:

$$\begin{aligned} y_{i+1} &= y_i + m \\ x_{i+i} &= x_i + 1 \end{aligned} \tag{3.6}$$

Esta idea es la base para el algoritmo que se conoce como Analizador Diferencial Digital (DDA, *Digital Differential Analyzer*). Que está restringido al caso en que $-1 \leq m \leq 1$. Para otros valores de m, se puede usar simetría para modificar el algoritmo.

Esta es su especificación en lenguaje c:

Listing 3.1: Algoritmo Analizador Diferencial Digital

```
1  void dda(int x0, int y0, int x1, int y1){
2      /*
3          Supone que -1 <= m <= 1, x0 < x1.
4          x varía de x0 a x1 en incrementos unitarios
5      */
6      int x;
7      float y, m;
8      m = (float)(y1 - y0) / (x1 - x0);
9      y = y0;
10     for(x = x0; x <= x1, x++){
11         marcarPixel(x, (int) y);
```

```
12              y += m;
13          }
14      }
```

A pesar de que esta solución funciona y se puede adaptar a cualquier pendiente, tiene un detalle indeseable: *Utiliza artimética de coma flotante*. Por ello, en las siguientes secciones se presenta una elegante solución que utiliza únicamente aritmética entera.

3.4. Algoritmo de línea de punto medio

A continuación, se procederá a describir uno de los mejores algoritmos de dibujo de líneas rectas: El *algoritmo de línea de punto medio* (también conocido como *algoritmo de línea de Bresenham*).

Antes de abordar la resolución general del dibujo de segmentos de recta, se abordará un caso particular más sencillo que permite describir la lógica del algoritmo.

> El problema consiste en dibujar un segmento de recta, desde el pixel inferior izquierdo (x_0, y_0), hasta el pixel superior derecho (x_1, y_1) usando únicamente aritmética entera.

Considere la línea que se representa en la figura 3.2, donde el pixel previamente seleccionado aparece sombreado y los dos pixeles candidatos a ser seleccionados en el siguiente paso, son el pixel a la derecha del marcado, el pixel "ESTE" E, y el pixel que está arriba del anterior, el pixel "NOR-ESTE" NE. El punto M es el punto medio entre los centros geométricos de E y de NE.

Acabamos de marcar el pixel en cuestión, el pixel P, en las coordenadas (x_0, y_0) y ahora tenemos que elegir entre el pixel E y el NE. Eso lo hacemos averiguando si la recta ideal pasa más cerca del centro geométrico de E o de NE.

Si el punto medio M, está por encima de la línea, el pixel E es el más cercano a la línea. Si el punto medio está debajo, el pixel NE es el más cercano. En cualquier caso, el error es siempre menor o igual a $\frac{1}{2}$ de pixel.

En el caso de la figura 3.2, el pixel escogido como siguiente es el NE, y en el caso de la figura 3.3 en la página siguiente es el E.

Lo que se necesita entonces, es una manera de averiguar de qué lado de la línea está M. Para ello se usará la siguiente ecuación de la línea recta, la ecuación implícita: $ax + by + c = 0$.

Con ella, podemos construir la siguiente función:

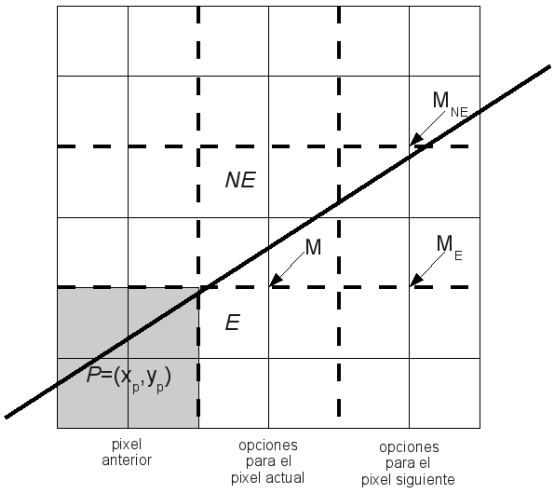

Figura 3.2: Justificación del algoritmo de línea de punto medio básico - 1

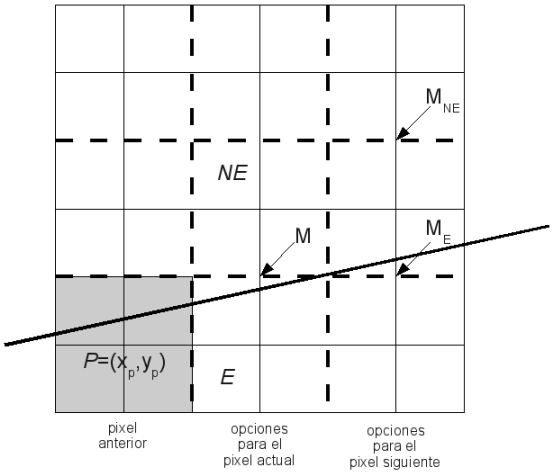

Figura 3.3: Justificación del algoritmo de línea de punto medio básico - 2

$$F(x,y) = ax + by + c \tag{3.7}$$

Esta tiene la siguiente característica: El valor de F es cero para los puntos que pertenecen a la línea, positivo si el punto evaluado está debajo de la línea, y es negativo si el punto evaluado está encima de la línea, siempre y cuando el coeficiente b, el coeficiente de y sea negativo.

$$\ldots\ldots\ldots\ldots F(x,y) < 0 \Longrightarrow \underline{(x,y)} \ldots\ldots\ldots\ldots F(x,y) > 0 \Longrightarrow \overline{(x,y)} \ldots\ldots\ldots\ldots$$

Entonces, tenemos que evaluar el signo de $F(M) = F\left(x_p + 1, y_p + \frac{1}{2}\right)$ para saber si M está arriba o abajo de la recta ideal.

Para ordenar un poco el panorama, definamos una nueva variable, nuestra variable de decisión:

$$d = F\left(x_p + 1, y_p + \frac{1}{2}\right) = a\left(x_p + 1\right) + b\left(y_p + \frac{1}{2}\right) + c \tag{3.8}$$

Con esta variable, podemos establecer el siguiente convenio: Si $d > 0$, se elige el pixel NE; si $d \leq 0$, se elige el pixel E (si $d = 0$, podríamos elegir cualquiera, pero por convención, eligimos E).

Una vez tomada la decisión y marcado el pixel, se pasa al siguiente punto y se hace lo mismo. $d_{nuevo} = F(M_E) = F\left((x_p + 1) + 1, y_p + \frac{1}{2}\right)$ si se eligió el pixel E, y $d_{nuevo} = F(M_{NE}) = F\left((x_p + 1) + 1, (y_p + 1) + \frac{1}{2}\right)$ si se eligió el pixel NE.

Ahora bien, desconocemos los valores de a, b y c. ¿Qué hacemos?. Hagamos una pausa y analicemos el siguiente fenómeno:

Sea $P(x) = mx + B$ y sea $x_{i+1} = x_i + 1$ (ya que consideramos que los sucesivos x_i son las coordenadas de pixeles). Entonces:

$$\begin{aligned} P(x_{i+1}) &= mx_{i+1} + B \\ &= m(x_i + 1) + B \\ &= (mx_i + B) + m \\ P(x_{i+1}) &= P(x_i) + m \\ P(x_{i+1}) - P(x_i) &= m \end{aligned}$$

Ahora veamoslo aplicado a nuestro problema de F (ver la ecuación 3.7):

- En el caso de elegir el pixel E:

$$\begin{aligned} F(x_{i+1}, y_i) &= a(x_i + 1) + by_i + c \\ &= ax_i + a + by_i + c \\ &= (ax_i + by_i + c) + a \\ F(x_{i+1}, y_i) &= F(x_i, y_i) + a \\ F(x_{i+1}, y_i) - F(x_i, y_i) &= a \\ d^E_{nuevo} - d_{viejo} &= a = \Delta_E \end{aligned}$$

- Veamos el caso en que se elige el pixel NE:

$$\begin{aligned} F(x_{i+1}, y_{i+1}) &= a(x_i + 1) + b(y_i + 1) + c \\ &= ax_i + a + by_i + b + c \\ &= (ax_i + by_i + c) + a + b \\ F(x_{i+1}, y_{i+1}) &= F(x_i, y_i) + a + b \\ F(x_{i+1}, y_{i+1}) - F(x_i, y_i) &= a + b \\ d^{NE}_{nuevo} - d_{viejo} &= a + b = \Delta_{NE} \end{aligned}$$

A estas diferencias recién introducidas, las llamaremos Δ_E y Δ_{NE} respectivamente.

Resta entonces el problema del valor inicial de d (equivale al valor d_{viejo} del primer paso):

$$\begin{aligned} d_{inicial} &= F(M) \\ &= F(x_0 + 1, y_0 + \frac{1}{2}) = a(x_0 + 1) + b(y_0 + \frac{1}{2}) + c \\ &= ax_0 + a + by_0 + \frac{1}{2}b + c \\ &= ax_0 + by_0 + c + a + \frac{1}{2}b \\ &= (ax_0 + by_0 + c) + a + \frac{1}{2}b \\ &= F(x_{0,}y_0) + a + \frac{1}{2}b;\ F(x_{0,}y_0) = 0 \\ d_{inicial} &= a + \frac{1}{2}b \end{aligned}$$

Se deben conseguir los parámetros a y b (c no es necesario). Para ello, partimos de $y = \frac{\Delta y}{\Delta x}x + B$, así:

$$\begin{aligned} y &= \frac{\Delta y}{\Delta x}x + B \\ \Delta x \cdot y &= \Delta y \cdot x + \Delta x \cdot B \\ 0 &= \Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot B \\ F(x,y) &= \Delta y \cdot x - \Delta x \cdot y + \Delta x \cdot B \end{aligned}$$

por lo que:

$$\begin{aligned} a &= \Delta y \\ b &= -\Delta x \\ c &= \Delta x \cdot B \end{aligned}$$

Note que el coeficiente b resulta negativo. Note además que el valor de B es desconocido (se puede calcular, porque conocemos al menos dos puntos de la recta), pero como no necesitamos c, no representa ningún problema para nuestro propósito.

Tenemos entonces:

$$\begin{aligned} \Delta_E = \quad & F(x_{i+1}, y_i) - F(x_i, y_i) = a \quad && = \Delta y \\ \Delta_{NE} = \quad & F(x_{i+1}, y_{i+1}) - F(x_i, y_i) = a + b \quad && = \Delta y - \Delta x \\ d_{inicial} = \quad & a + \tfrac{1}{2}b && = \Delta y - \frac{1}{2}\Delta x \end{aligned}$$

Pero esto nos fuerza a usar coma flotante sólo por el valor inicial de d. Todos los demás valores sucesivos son enteros...

Entonces, replanteamos la ecuación inicial, de la siguiente manera:

$$\begin{aligned} y &= \frac{\Delta y}{\Delta x}x + B \\ 2y &= 2\frac{\Delta y}{\Delta x}x + 2B \\ 2\Delta x \cdot y &= 2\Delta y \cdot x + 2\Delta x \cdot B \\ F(x,y) &= 2\Delta y \cdot x - 2\Delta x \cdot y + 2\Delta x \cdot B \end{aligned} \tag{3.9}$$

por lo que ahora:

$$\begin{aligned} a &= 2\Delta y \\ b &= -2\Delta x \\ c &= 2\Delta x \cdot B \end{aligned}$$

y volvemos a calcular los valores que nos interesan[1]. Llegamos a:

$$\begin{aligned} \Delta_E = &\quad F(x_{i+1}, y_i) - F(x_i, y_i) = a &&= 2\Delta y \\ \Delta_{NE} = &\quad F(x_{i+1}, y_{i+1}) - F(x_i, y_i) = a + b &&= 2\Delta y - 2\Delta x \\ d_{inicial} = &\quad a + \tfrac{1}{2}b &&= 2\Delta y - \Delta x \end{aligned}$$

Con esto hemos resuelto[2] el problema del valor inicial flotante para d.

Ahora, el algoritmo usa exclusivamente artimética entera, sin el costoso tiempo para hacer operaciones en coma flotante.

El código en lenguaje C es:

Listing 3.2: Algoritmo de línea de punto medio en la mitad de un cuadrante

```
1  void linea_punto_medio_1(int x0, int y0, int x1, int y1){
2      int delta_x, delta_y, delta_E, delta_NE, d, x, y;
3      /*
4          Supone que x0 < x1, y0 < y1 y que delta_y <= delta_x.
5          x varía de x0 a x1 en incrementos unitarios
6      */
7      delta_x = x1 - x0;
8      delta_y = y1 - y0;
9      d = delta_y * 2 - delta_x;
10     delta_E = delta_y * 2;
11     delta_NE = (delta_y - delta_x) * 2;
12     x = x0;
13     y = y0;
14     marcarPixel(x, y); //marcar el primero
15     while(x < x1){
16         if(d <= 0){
17             d += delta_E;
18         }else{
19             d += delta_NE;
20             y++;
21         }
22         x++;
```

[1] Basta con repetir el proceso con la nueva ecuación F

[2] Note que el valor de b sigue siendo negativo

```
23          marcarPixel(x, y);
24      }
25  }
```

3.4.1. Simetría del algoritmo de línea de punto medio

El problema de generalizar el algoritmo de punto medio para pendientes arbitrarias tiene dos componentes: **(a)** permitir pendientes mayores que 1, y **(b)** poder hacer el recorrido *hacia atrás*, para poder aceptar los puntos en cualquier orden sin tener que intercambiar las variables de entrada.

El primer problema se resuelve, averiguando la relación de orden de Δx y Δy. En base a eso, se decide hacer el recorrido sobre x o sobre y. Así:

Listing 3.3: Algoritmo de línea de punto medio de un cuadrante (o dos mitades de cuadrante)

```
1  void linea_punto_medio_2(int x0, int y0, int x1, int y1){
2
3      int delta_x, delta_y, delta_E, delta_NE, d, x, y;
4      delta_x=x1-x0; delta_y=y1-y0;
5      x=x0, y=y0;
6
7      // Cuando 'horizontal' es verdadero, el ciclo iterará a través de x,
           si no, lo hará a través de y
8      int horizontal = delta_y < delta_x;
9
10     // Inicializaciones
11     if(horizontal){
12         d = (delta_y*2)-delta_x;
13         delta_E = delta_y*2;
14         delta_NE = (delta_y-delta_x)*2;
15     }else{
16         d = (delta_x*2)-delta_y;
17         delta_E = delta_x*2;
18         delta_NE = (delta_x-delta_y)*2;
19     }
20
21     // Dibujar el primer pixel
22     marcarPixel(x,y);
23     if(horizontal)
24         while(x < x1){ // Iterar a través de x
25             if(d<=0)
26                 d+=delta_E;
27             else{
28                 y++;
29                 d+=delta_NE;
30             }
```

```
31              x++;
32              marcarPixel(x,y);
33          }
34      else
35          while(y < y1){ // Iterar a través de y
36              if(d<=0)
37                  d+=delta_E;
38              else{
39                  x++;
40                  d+=delta_NE;
41              }
42              y++;
43              marcarPixel(x,y);
44          }
45  }
```

Ahora bien, el otro problema es un poco más complicado. Consideremos la figura 3.4 y repitamos el análisis a partir de la ecuación 3.9, pero haciendo las modificaciones pertinentes.

Llegamos a lo siguiente[3]:

$$\begin{aligned} d' = \quad & F\left(x_p - 1, y_p - \tfrac{1}{2}\right) && = a\left(x_p - 1\right) + b\left(y_p - \frac{1}{2}\right) + c \\ \Delta_O = \quad & -2\Delta y && = -\Delta_E \\ \Delta_{SO} = \quad & -2\Delta y + 2\Delta x && = -\Delta_{NE} \\ d'_{inicial} = \quad & -2\Delta y + \Delta x && = -d_{inicial} \end{aligned}$$

Lo que significa que el convenio en este caso es: Si $d' \geq 0$, se elige el pixel O; si $d' < 0$, se elige el pixel SO.

Aunque esto parezca bastante complicado, no hace falta darle muchas vueltas al asunto[4]. Basta con tomar conciencia del siguiente razonamiento:

Si el valor inicial de d y todos sus incrementos posibles son del mismo valor absoluto, pero de signo opuesto, y el criterio se basa en el signo de dicha variable, significa que se pueden sustituir los valores Δ_O, Δ_{SO} y $d'_{inicial}$ por Δ_E, Δ_{NE} y $d_{inicial}$ respectivamente sin afectar gravemente el algoritmo. Basta con invertir el criterio de incremento, de tal manera que quede así: Si $d' > 0$, se elige el pixel SO (sumar Δ_{NE}); si $d' \leq 0$, se elige el pixel O (sumar Δ_E). De tal manera que con una ligera modificación al código 3.2 se puedan considerar ambos casos, así:

[3] La demostración de esto se deja como ejercicio
[4] tal como lo hizo el autor de este libro

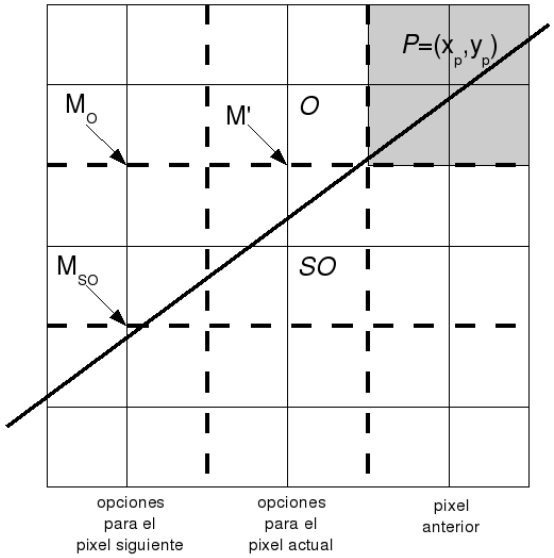

Figura 3.4: Análisis inverso de punto medio

Listing 3.4: Algoritmo de línea de punto medio de dos mitades de cuadrantes opuestos

```
void linea_punto_medio_3(int x0, int y0, int x1, int y1){
    int delta_x, delta_y, delta_E, delta_NE, d, x, y, dir_x, dir_y;
    /*
        x varía de x0 a x1, o de x1 a x0 en incrementos unitarios
    */
    delta_x = abs(x1 - x0);
    delta_y = abs(y1 - y0);
    x = x0;
    y = y0;
    d = delta_y * 2 - delta_x;
    delta_E = delta_y * 2;
    delta_NE = (delta_y - delta_x) * 2;

    // Indican la dirección en que la línea debe seguir
    dir_x = (x1-x0)>0 ? 1 : -1;
    dir_y = (y1-y0)>0 ? 1 : -1;

    marcarPixel(x, y); //marcar el primero
    while(x != x1){
        if(d <= 0){
            d += delta_E;
        }else{
            d += delta_NE;
            y+=dir_y;
        }
```

```
26          x+=dir_y;
27          marcarPixel(x, y);
28      }
29  }
```

Finalmente, el código en lenguaje C para el algoritmo de línea de punto medio genérico (que considera avance hacia atrás como el 3.4 y pendientes arbitrarias como el 3.3) es:

Listing 3.5: Algoritmo de línea de punto medio genérico

```
1  void linea_punto_medio(int x0, int y0, int x1, int y1){
2
3      int delta_x, delta_y, delta_E, delta_NE, d, x, y, dir_x, dir_y;
4      delta_x=abs(x1-x0); delta_y=abs(y1-y0);
5      x=x0, y=y0;
6
7      // Cuando 'horizontal' es verdadero, el ciclo iterará a través de x,
           sino, lo hará a través de y
8      int horizontal = delta_y < delta_x;
9
10     // Indican la dirección en que la línea debe seguir
11     dir_x = (x1-x0)>0 ? 1 : -1;
12     dir_y = (y1-y0)>0 ? 1 : -1;
13
14     // Inicializaciones
15     if(horizontal){
16         d=(delta_y*2)-delta_x;
17         delta_E=delta_y*2;
18         delta_NE=(delta_y-delta_x)*2;
19     }else{
20         d=(delta_x*2)-delta_y;
21         delta_E=delta_x*2;
22         delta_NE=(delta_x-delta_y)*2;
23     }
24
25     // Dibujar el primer pixel
26     marcarPixel(x,y);
27     if(horizontal)
28         while(x != x1){
29             if(d <= 0){
30                 d += delta_E;
31             }else{
32                 d += delta_NE;
33                 y+=dir_y;
34             }
35             x+=dir_x;
36             marcarPixel(x, y);
37         }
38     else
```

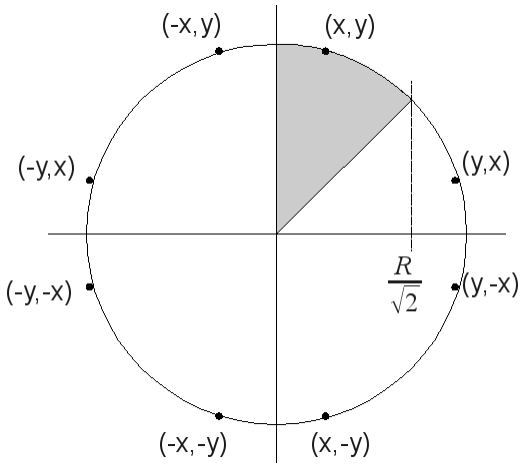

Figura 3.5: Simetría de las circunferencias

```
39          while(y != y1){
40              if(d <= 0){
41                  d += delta_E;
42              }else{
43                  d += delta_NE;
44                  x+=dir_x;
45              }
46              y+=dir_y;
47              marcarPixel(x, y);
48          }
49  }
```

3.5. La simetría de la circunferencia

El círculo se divide en ocho partes iguales, y si se calculan los puntos de un octante, se pueden calcular, por simetría, los puntos de los demás octantes, como puede apreciarse en la figura 3.5

Podría implementarse un procedimiento como el siguiente para aprovechar este fenómeno:

Listing 3.6: Función de aprovechamiento de la simetría de las circunferencias

```
1  void punto_circunferencia_simetria(int x, int y){
2      marcarPixel( x, y);
3      marcarPixel( y, x);
4      marcarPixel( y,-x);
```

```
5       marcarPixel( x,-y);
6       marcarPixel(-x,-y);
7       marcarPixel(-y,-x);
8       marcarPixel(-y, x);
9       marcarPixel(-x, y);
10  }
```

3.6. Algunas ideas sobre circunferencias

Existen varias alternativas para dibujar un círculo. Entre ellas, dibujarla a partir de la ecuación:

$$x^2 + y^2 = R^2 \Rightarrow y = \pm\sqrt{R^2 - x^2} \tag{3.10}$$

Pero esta alternativa provoca algunos efectos no deseables cuando $x \rightarrow R$, además del enorme consumo de recursos para ejecutar las multiplicaciones y la raíz cuadrada.

Otra alternativa, es dibujarla en su forma paramétrica: $(r\cos\theta, r\sin\theta), 0 \leq \theta \leq 2\pi$. Esta es menos ineficiente, pero aún demasiado, debido al cálculo de las funciones trigonométricas (que se realizan con series de potencias).

3.7. Algoritmo de circunferencia de punto medio

La alternativa más eficiente es el *algoritmo de punto medio para circunferencia* o *algoritmo de círculo de Bresenham*. A continuación se presentará la solución básica en el caso de circunferencias centradas en el origen para comprender la idea general.

En la figura 3.6 aparece un diagrama para explicar la idea, que es la misma del algortimo de punto medio para líneas.

Para construir el código se procederá de manera similar. Se usará la siguiente ecuación del círculo, la ecuación implícita:

$$F(x, y) = x^2 + y^2 - R^2 = 0 \tag{3.11}$$

El valor de F es cero en el círculo, positivo fuera, y negativo dentro, siempre y cuando los coeficientes de x^2 y y^2 sean positivos.

Como se hizo con las líneas, la decisión se basa en la variable d:

$$d_{viejo} = F(M) = F(x_p + 1, y_p - \frac{1}{2}) = (x_p + 1)^2 + (y_p - \frac{1}{2})^2 - R^2$$

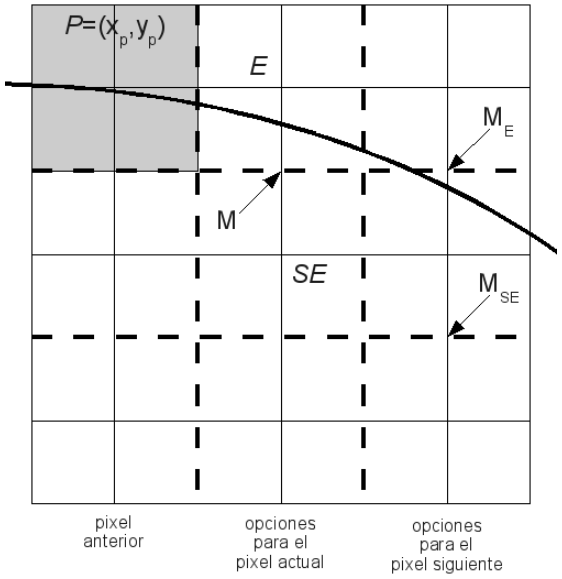

Figura 3.6: Algoritmo de circunferencia de punto medio

Si $d_{viejo} < 0$, elegimos avanzar al pixel E:

$$
\begin{aligned}
d^E_{nuevo} &= F\left((x_p+1)+1, y_p - \frac{1}{2}\right) \\
d^E_{nuevo} &= d_{viejo} + \Delta_E \\
\Delta_E &= d^E_{nuevo} - d_{viejo} \\
&= F(M_E) - F(M) \\
&= F(x_p+2, y_p - \frac{1}{2}) - F(x_p+1, y_p - \frac{1}{2}) \\
&= (x_p+2)^2 + (y_p - \frac{1}{2})^2 - R^2 \\
&\quad -(x_p+1)^2 - (y_p - \frac{1}{2})^2 + R^2 \\
\Delta_E &= 2x_p + 3
\end{aligned}
$$

Si $d_{viejo} \geq 0$, elegimos el pixel SE:

$$
\begin{aligned}
d^{SE}_{nuevo} &= F\left((x_p+1)+1,(y_p-1)-\frac{1}{2}\right)\\
d^{SE}_{nuevo} &= d_{viejo}+\Delta_{SE}\\
\Delta_{SE} &= d^{SE}_{nuevo}-d_{viejo}\\
&= F(M_{SE})-F(M)\\
&= F(x_p+2,y_p-\frac{3}{2})-F(x_p+1,y_p-\frac{1}{2})\\
&= (x_p+2)^2+(y_p-\frac{3}{2})^2-R^2\\
&\quad -(x_p+1)^2-(y_p-\frac{1}{2})^2+R^2\\
&= (2x_p+3)+(-2y_p+2)\\
\Delta_{SE} &= 2x_p-2y_p+5
\end{aligned}
$$

Lo que falta es entonces, calcular la condición inicial. Sabemos, que el punto inicial es $(0,R)$ y por ende, el siguiente punto medio es $(1,R-\frac{1}{2})$:

$$
\begin{aligned}
d_{inicial} &= F(1,R-\frac{1}{2})\\
&= 1+(R-\frac{1}{2})^2-R^2\\
&= 1+(R^2-R+\frac{1}{4})-R^2\\
d_{inicial} &= \frac{5}{4}-R
\end{aligned}
$$

El algoritmo resultante en lenguaje C es:

Listing 3.7: Primera propuesta de algoritmo de círculo de punto medio

```
1 void circunferencia_punto_medio_1(int radio){
2     int x, y;
3     float d;
4 
5     x = 0;
6     y = radio;
7     d = 5.0 / 4 - radio;
8     marcarPixel( x, y);
9     marcarPixel( y, x);
```

```
10      marcarPixel( y,-x);
11      marcarPixel( x,-y);
12      marcarPixel(-x,-y);
13      marcarPixel(-y,-x);
14      marcarPixel(-y, x);
15      marcarPixel(-x, y);
16
17      while(y > x){
18          if(d < 0){
19              d += x * 2.0 + 3;
20          }else{
21              d += (x - y) * 2.0 + 5;
22              y--;
23          }
24          x++;
25          marcarPixel( x, y);
26          marcarPixel( y, x);
27          marcarPixel( y,-x);
28          marcarPixel( x,-y);
29          marcarPixel(-x,-y);
30          marcarPixel(-y,-x);
31          marcarPixel(-y, x);
32          marcarPixel(-x, y);
33      }
34  }
```

Sin embargo, persiste el desagradable asunto de tener que operar con una variable de coma flotante sólo por el valor inicial de d. Eso puede solucionarse de la siguiente manera:

Definimos una nueva variable de decisión: $h = d - \frac{1}{4}$ y sustituimos este valor en lugar de d en el código 3.7. El valor es ahora $h = 1 - R$ y la comparación $d < 0$ es $h < -\frac{1}{4}$. Pero como h es incrementada únicamente en valores enteros, se puede dejar la comparación como $h < 0$ sin afectar el resultado. Y por razones de consistencia, se dejará la variable d en lugar de h.

El algoritmo resultante, usando sólo aritmética entera para dibujar círculos en lenguaje C es:

Listing 3.8: Algoritmo de circunferencia de punto medio sólo con aritmética entera

```
1   void circunferencia_punto_medio_2(int radio){
2       int x, y, d;
3
4       x = 0;
5       y = radio;
6       d = 1 - radio;
7       marcarPixel( x, y);
8       marcarPixel( y, x);
9       marcarPixel( y,-x);
```

```
10      marcarPixel( x,-y);
11      marcarPixel(-x,-y);
12      marcarPixel(-y,-x);
13      marcarPixel(-y, x);
14      marcarPixel(-x, y);
15
16      while(y > x){
17          if(d < 0){
18              d += x * 2 + 3;
19          }else{
20              d += (x - y) * 2 + 5;
21              y--;
22          }
23          x++;
24          marcarPixel( x, y);
25          marcarPixel( y, x);
26          marcarPixel( y,-x);
27          marcarPixel( x,-y);
28          marcarPixel(-x,-y);
29          marcarPixel(-y,-x);
30          marcarPixel(-y, x);
31          marcarPixel(-x, y);
32      }
33  }
```

3.7.1. Versión sin multiplicaciones

El algoritmo 3.8 se puede agilizar aún más si se considera que así como se han calculado *diferencias parciales de primer* orden para d, se pueden calcular diferencias de primer orden para los incrementos de d, es decir, las *diferencias parciales de segundo orden* de d. Así:

Si elegimos E en la iteración actual, el punto de evaluación se mueve de (x_p, y_p) a $(x_p + 1, y_p)$. Tenemos entonces el siguiente panorama:

$$\Delta_{Eviejo} = 2x_p + 3$$
$$\Delta_{Enuevo} = 2(x_p + 1) + 3$$
$$\Delta_{Enuevo} - \Delta_{Eviejo} = 2$$

$$\Delta_{SEviejo} = 2x_p - 2y_p + 5$$

y tenemos además: $\Delta_{SEnuevo} = 2(x_p + 1) - 2y_p + 5$.

$$\Delta_{SEnuevo} - \Delta_{SEviejo} = 2$$

Pero si elegimos SE en la iteración actual, el punto de evaluación se mueve de (x_p, y_p) a $(x_p + 1, y_p - 1)$. Tenemos lo siguiente:

$$\Delta_{Eviejo} = 2x_p + 3$$
$$\Delta_{Enuevo} = 2(x_p + 1) + 3$$
$$\Delta_{Enuevo} - \Delta_{Eviejo} = 2$$

$$\Delta_{SEviejo} = 2x_p - 2y_p + 5$$

y tenemos además: $\Delta_{SEnuevo} = 2(x_p + 1) - 2(y_p - 1) + 5.$

$$\Delta_{SEnuevo} - \Delta_{SEviejo} = 4$$

De este análisis, surge una tercera versión del algoritmo (mucho más rápida que las anteriores, ya que sólo contiene una multiplicación):

Listing 3.9: Algoritmo de línea de punto medio sin multiplicaciones

```
void circunferencia_punto_medio_3(int radio){
    int x, y, d, delta_E, delta_SE;

    x = 0;
    y = radio;
    d = 1 - radio;
    delta_E = 3;
    delta_SE = 5 - radio * 2;
    marcarPixel( x, y);
    marcarPixel( y, x);
    marcarPixel( y,-x);
    marcarPixel( x,-y);
    marcarPixel(-x,-y);
    marcarPixel(-y,-x);
    marcarPixel(-y, x);
    marcarPixel(-x, y);

    while(y > x){
        if(d < 0){
            d += delta_E;
            delta_E += 2;
            delta_SE += 2;
        }else{
            d += delta_SE;
            delta_E += 2;
            delta_SE += 4;
            y--;
        }
        x++;
        marcarPixel( x, y);
        marcarPixel( y, x);
        marcarPixel( y,-x);
        marcarPixel( x,-y);
        marcarPixel(-x,-y);
        marcarPixel(-y,-x);
        marcarPixel(-y, x);
```

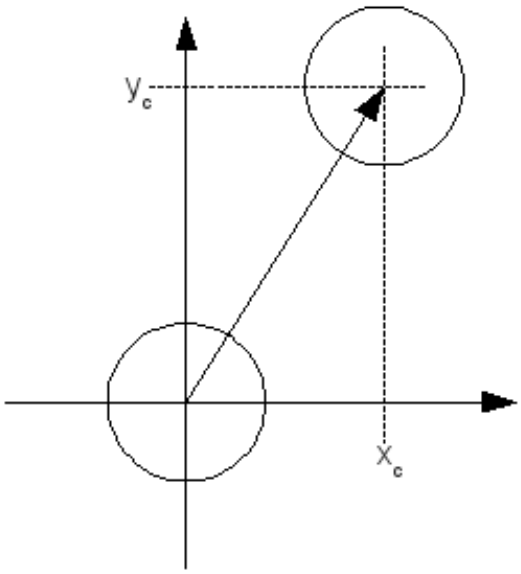

Figura 3.7: Esquema para algorito de circunferencias con centro arbitrario

```
37            marcarPixel(-x, y);
38        }
39    }
```

3.7.2. Circunferencias con centro arbitrario

Para construir un algoritmo de dibujo de circunferencias con centro arbitrario, a partir del algoritmo de punto medio de esta sección, básicamente hay que hacer un desplazamiento vectorial de cada uno de los puntos a marcar.

Consideremos la figura 3.7. En ella aparecen dos circunferencias del mismo radio. La diferencia vectorial entre cada uno de los puntos de la circunferencia centrada en el origen y la centrada en (x_c, y_c), es precísamente el vector $\overrightarrow{C} = (x_c, y_c)$.

> Otra forma de plantearlo es así: Sean (x', y') los puntos de la circunferencia con centro en (x_c, y_c). Sean (x, y) los puntos de la circunferencia con centro en el origen.
>
> Entonces $(x', y') = (x + x_c, y + y_c)$.

En base a este sencillo análisis, presentamos el siguiente algoritmo de dibujo de circunferencias con centro arbitrario, basado en el algoritmo 3.8:

Listing 3.10: Algoritmo de circunferencia de punto medio para centros arbitrarios

```
// Función auxiliar
void marcarPixelesCicunferencia(int x, int y, int xc, int yc){
    marcarPixel( x+xc, y+yc);
    marcarPixel( x+xc,-y+yc);
    marcarPixel(-x+xc, y+yc);
    marcarPixel(-x+xc,-y+yc);
    marcarPixel( y+xc, x+yc);
    marcarPixel( y+xc,-x+yc);
    marcarPixel(-y+xc, x+yc);
    marcarPixel(-y+xc,-x+yc);
}
void circunferencia_punto_medio(int xc, int yc, int radio){
    int x,y,d;
    x=0;
    y=radio;
    d=1-radio;
    marcarPixelesCicunferencia(x,y, xc,yc);
    while(y>x){
        if(d<0){
            d += x * 2 + 3;
        }else{
            d += (x - y) * 2 + 5;
            y--;
        }
        x++;
        marcarPixelesCicunferencia(x,y, xc,yc);
    }
}
```

3.8. Relleno de rectángulos

Veamos primero un par de algoritmos de dibujo de rectángulos huecos antes de pasar al de relleno:

```
void dibujar_rectangulo_1(int x1, int y1, int x2, int x2){
    /* Se asume que x1 < x2 y y1 < y2
    */
    int i;
    if(x1>x2){
        i=x1;
        x1=x2;
        x2=x1;
    }
    for(i=x1; i<=x2; i++){
        marcarPixel(i, y1);
        marcarPixel(i, y2);
```

```
    }
    if(y1>y2){
        i=y1;
        y1=y2;
        y2=y1;
    }
    for(i=y1; i<=y2; i++){
        marcarPixel(x1, i);
        marcarPixel(x2, i);
    }
}
```

```
void dibujar_rectangulo_2(int x1, int y1, int ancho, int alto){
    int i;
    for(i=x1; i< x1+ancho; i++){
        marcarPixel(i, y1,      );
        marcarPixel(i, y1+alto-1);
    }
    for(i=y1; i< y1+alto; i++){
        marcarPixel(x1,         i);
        marcarPixel(x1+ancho-1, i);
    }
}
```

El primer algoritmo recibe como parámetros dos puntos opuestos del rectángulo (nótese que no importa el orden en que se especifiquen). El segundo recibe el punto más cercano al origen y el ancho y alto del rectángulo (nótese que tanto `ancho` como `alto`, deben ser positivos).

Una de las alternativas más usuales entre las bibliotecas gráficas, es ofrecer al programador una función de relleno de rectángulos que requiere del punto inicial del rectángulo y su ancho y su alto. Esta es una propuesta de implementación en lenguaje C:

Listing 3.11: Relleno de rectángulos

```
void dibujar_rectangulo_relleno(int x1, int y1, int ancho, int alto){
    int i, j;
    for(i=x1; i< x1+ancho; i++)
        for(j=y1; j< y1+alto; j++)
            marcarPixel(i, j);
}
```

3.9. Relleno de circunferencias

Para el relleno de circunferencias, también existen diferentes propuestas. Una de ellas es aprovechar el algoritmo de punto medio para circunferencias y la simetría propia de

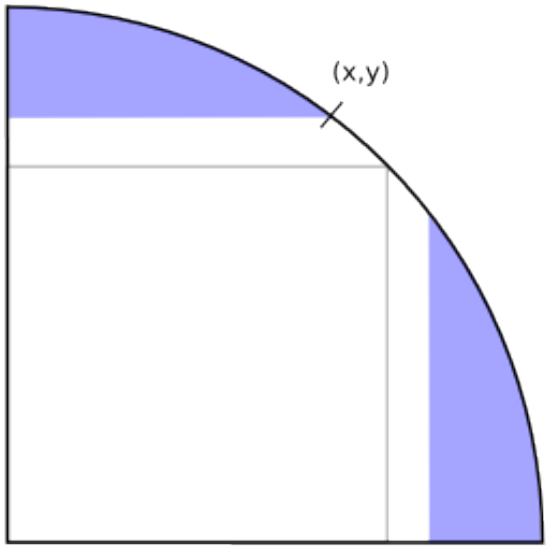

Figura 3.8: Relleno de circunferencias

esta hermosa figura geométrica.

La idea básica es hacer una modificación al algoritmo de punto medio para circunferencias de la siguiente manera: Cuando se haga un avance hacia el pixel SE, aprovechar para marcar todos los pixeles desde $(0, y)$ hasta (x, y) (y al mismo tiempo aprovechar para marcar los pixeles correspondientes con la misma idea de la simetría de ocho octantes). A continuación se presenta el código fuente de dicha solución[5]:

Listing 3.12: Algoritmo de relleno de circunferencias de centro arbitrario (basado en el código 3.10 y 3.11)

```
1  void circunferencia_rellena(int x0, int y0, int radio){
2      int r_x, r_y, r_ancho, r_alto;
3      int x,y,d,i;
4      x=0;
5      y=radio;
6      d=1-radio;
7      marcarPixelesCicunferencia(x,y, x0,y0);
8      while(y>x){
9          if(d<0){
10             d += x * 2 + 3;
11             x++;
12             marcarPixelesCicunferencia(x,y, x0,y0);
13         }else{
14             d += (x - y) * 2 + 5;
```

[5] comparar con el código de la función `circunferencia_punto_medio` de la subsección en la página 101

```
15              y--;
16              x++;
17              for(i=0;i<=x; i++){
18                  marcarPixelesCicunferencia(i,y, x0,y0);
19                  marcarPixelesCicunferencia(y,i, x0,y0);
20              }
21          }
22      }
23      r_x = x0-x;
24      r_y = y0-y;
25      r_ancho = 2*x;
26      r_alto = 2*y;
27      dibujar_rectangulo_relleno(r_x, r_y, r_ancho, r_alto);
28  }
```

3.10. Ejercicios

1. Construya un algoritmo que use la idea del DDA (código 3.1 en la página 83) para pendientes arbitrarias.
2. Describa cómo funciona el algoritmo de dibujo de líneas conocido como algoritmo de línea de punto medio.
3. Implementar un programa de dibujo usando la ecuación 3.10 en la página 95.
4. En la cuadrícula de la figura 3.9, marque (sombree, tache, repinte, etc.) los pixeles que encendería el algoritmo de línea 3.5, al unir a los puntos (2,2) y (9,5) y a los puntos (0,2) y (3,9). El centro geométrico de los pixeles está en el centro de los cuadritos negros. Las líneas grices son sólo guías.
5. En la cuadrícula de la figura 3.9, marque (sombree, tache, repinte, etc.) los pixeles que encendería el algoritmo de círculo 3.10 en la página 102, con centro (4,4) y radio 4. El centro geométrico de los pixeles está en el centro de los cuadritos negros. Las líneas grices son sólo guías.
6. Modificar el algoritmo 3.9 en la página 100 para dibujar círculos con centro arbitrario.
7. Modificar el algoritmo del ejercicio anterior para implementar el dibujo de círculos rellenos.
8. Considere los códigos siguientes para rellenar círculos y explique por qué el primero es preferible frente al segundo:

```
1  void marcarPixelesCicunferencia(int x, int y){
2      marcarPixel( x, y);
```