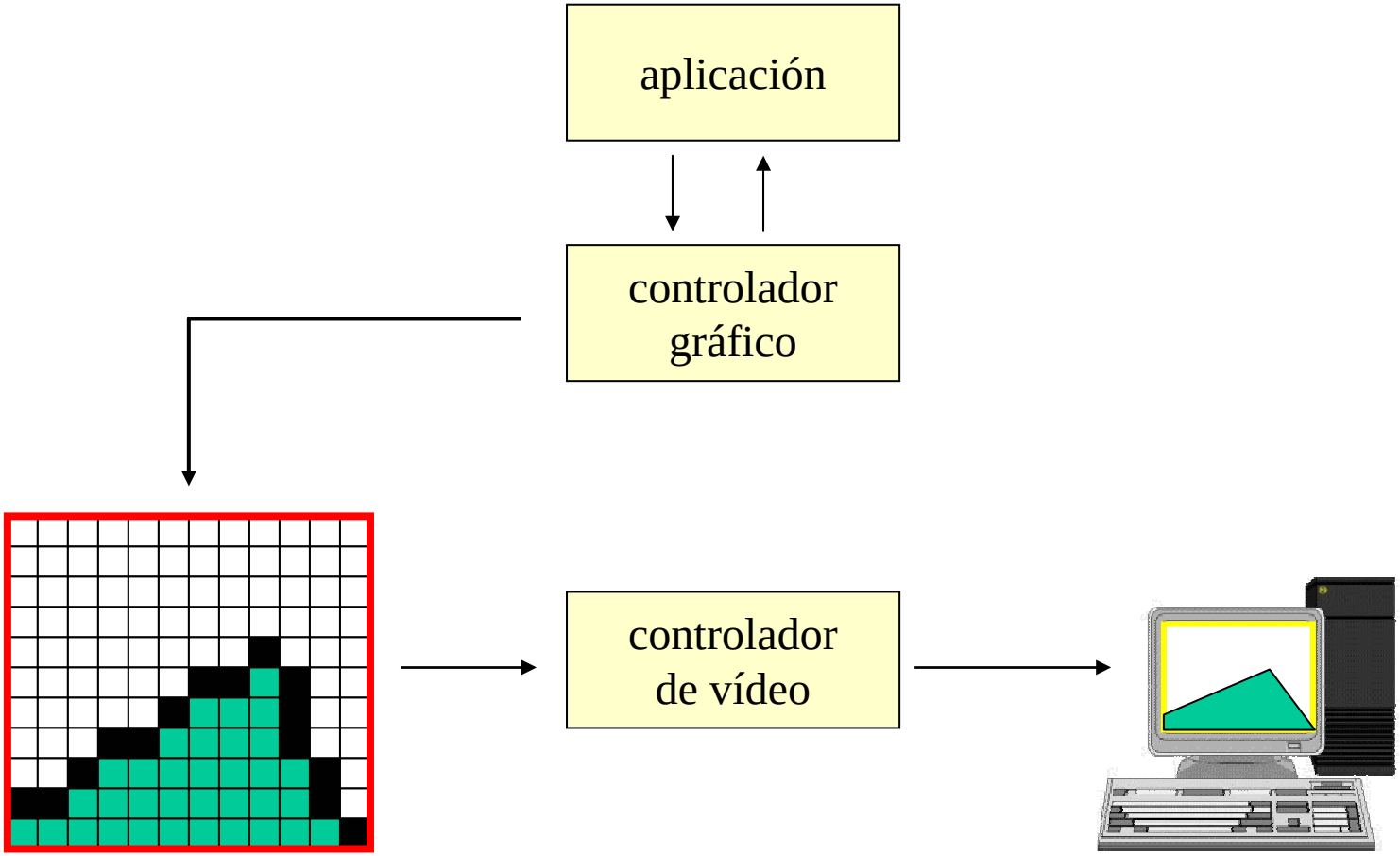
TEMA 2: Primitivas 2D

Índice

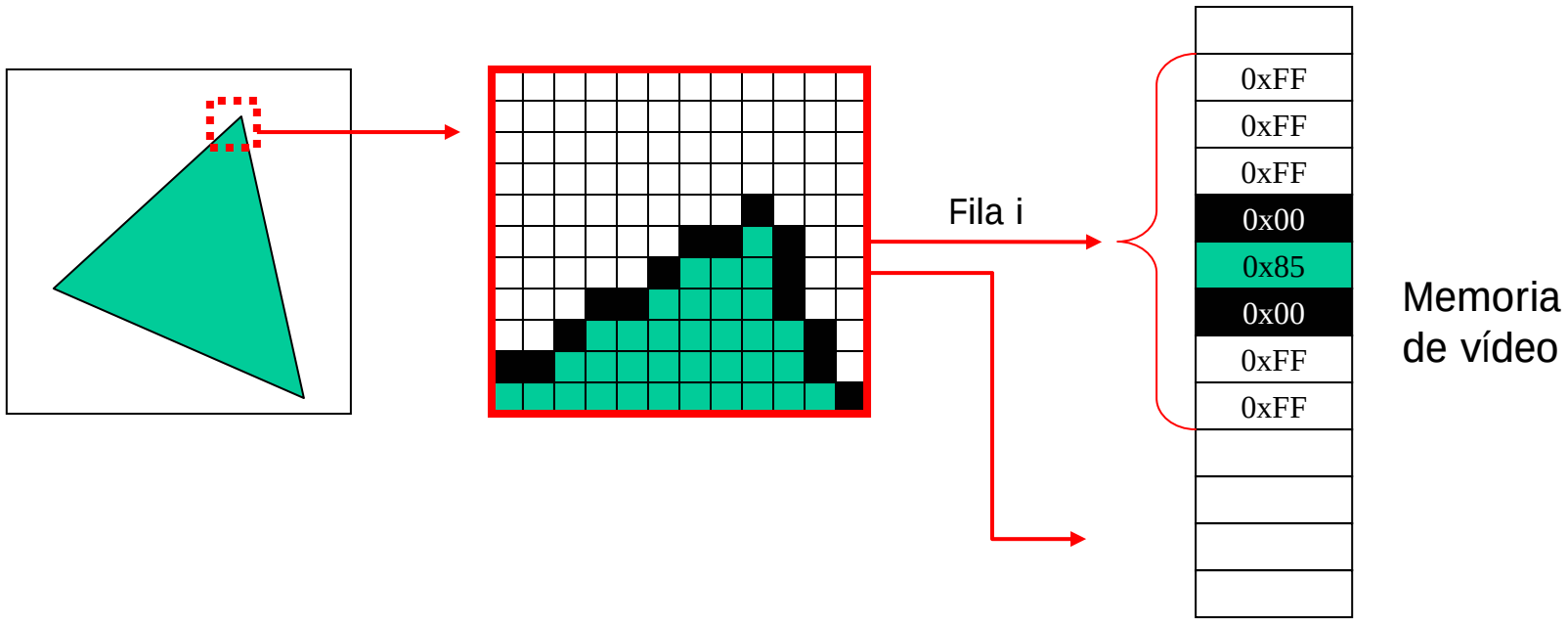
Primitivas 2D

- En los sistemas raster, las imágenes vienen definidas por la intensidad de sus pixels



Primitivas 2D

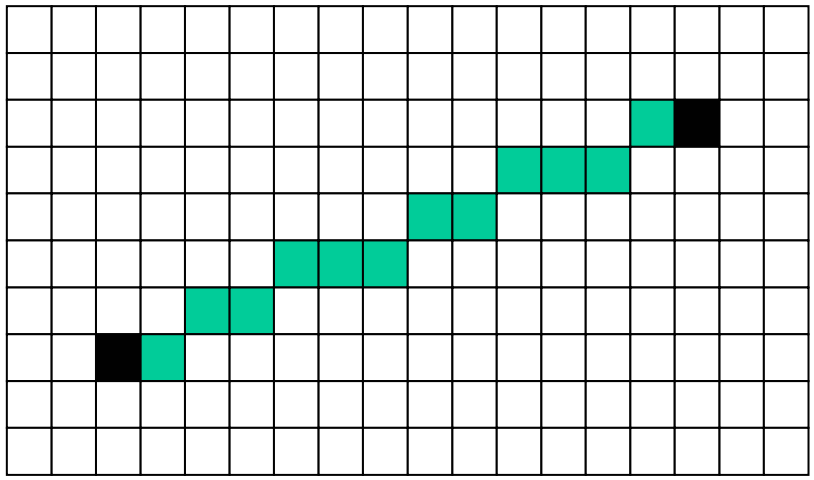
- Los objetos presentes en la imagen se componen de primitivas simples (líneas, puntos)
- El sistema gráfico dibuja estas primitivas transformándolos en pixels → Rasterización



- Los métodos de conversión deben ser lo más eficientes posible
- La primitiva "Punto" es la más sencilla:
  - se coloca la intensidad deseada en la celda de memoria del frame buffer correspondiente
  - Cuando el haz de electrones pase por esa línea horizontal (scan-line), emitirá al pasar por esa posición

Dibujo de líneas rectas

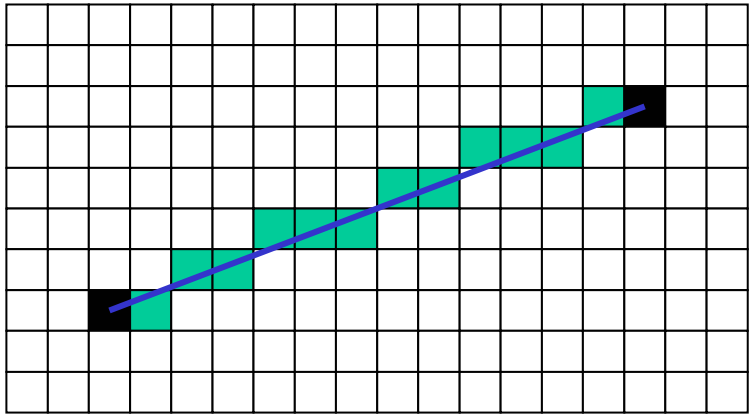
- Para dibujar líneas rectas, habrá que calcular las posiciones intermedias entre los dos extremos
- Este problema no existía en las pantallas vectoriales o plotters
- Sin embargo, las posiciones de los pixels son valores enteros, y los puntos obtenidos de la ecuación son reales → existe un error (aliasing)
- A menor resolución, mayor es el efecto



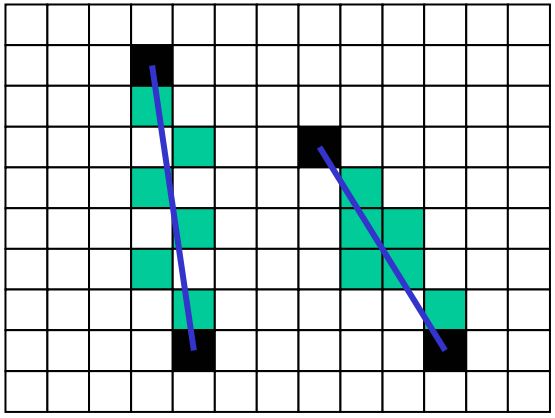
- Es necesario disponer de métodos para convertir primitivas en pixels de la forma más eficiente posible

Consideraciones para el dibujo de rectas

- Hay que calcular las coordenadas de los pixels que estén lo más cerca posible de una línea recta ideal, infinitamente delgada, superpuesta sobre la matriz de pixels.
- Las consideraciones que un buen algoritmo debe cumplir son:
  - la secuencia de pixels debe ser lo más recta posible
  - las líneas deben dibujarse con el mismo grosor e intensidad independientemente de su inclinación
  - las líneas deben dibujarse lo más rápido posible



correcto

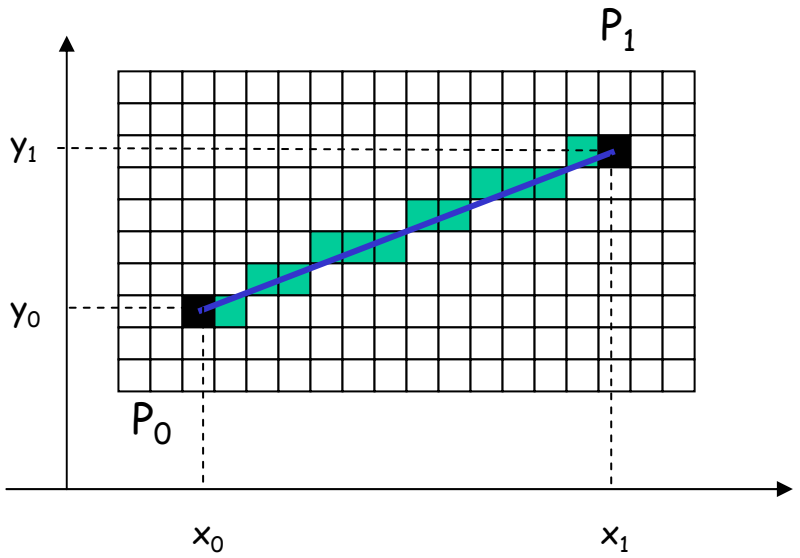


incorrecto

El algoritmo más sencillo

La ecuación de una recta es $y = mx + b$

- m es la pendiente
- b es el corte con el eje y



Calcular $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$

Calcular $b = y_0 - mx_0$

```
Para x=x0 hasta x=x1
    y = mx + b
    Pintar Pixel (x, round(y))
```

- No es muy eficiente
- Cada paso requiere una multiplicación flotante, una suma y un redondeo

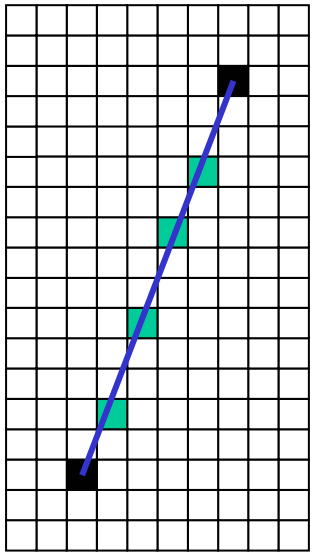
Algoritmo Básico Incremental (DDA)

- Podemos eliminar la multiplicación de la siguiente manera:

  Sabemos que $y_i = mx_i + b$

  Entonces $y_{i+1} = mx_{i+1} + b = \ldots \Rightarrow y_{i+1} = y_i + m\,\Delta x$

- Como $\Delta x=1$, llegamos a la fórmula final $\boxed{y_{i+1} = y_i + m}$
- Cada pixel se calcula en función del anterior
- No hace falta calcular b



Si m>1 falla pues quedan huecos

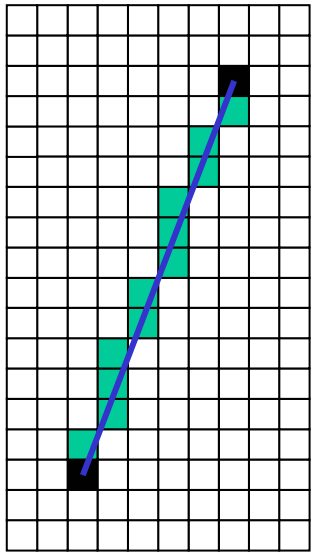
Solución: intercambiamos las variables x e y

Sabemos que $x_i = (1/m)(y_i - b)$. Entonces:

$x_{i+1} = (1/m)y_{i+1} - b/m = \ldots \Rightarrow x_{i+1} = x_i + \Delta y/m$

Como $\Delta y=1$, llegamos a la fórmula final

$$\boxed{x_{i+1} = x_i + 1/m}$$



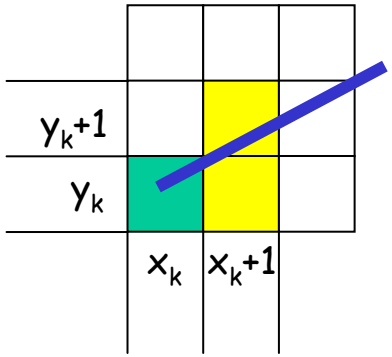
Algoritmo Básico Incremental (DDA)

```
Funcion Linea_DDA (int x0, y0, x1, y1)
dx = x1 – x0
dy = y1 – y0
Si abs(dx) > abs(dy) entonces steps = abs(dx)
        Si no steps = abs(dy)
xinc = dx / steps
yinc = dy / steps
x = x0
y = y0
Pintar Pixel (round(x), round(y))
Para k=1 hasta k=steps
        x = x + xinc
        y = y + yinc
        Pintar Pixel (round(x), round(y))
```

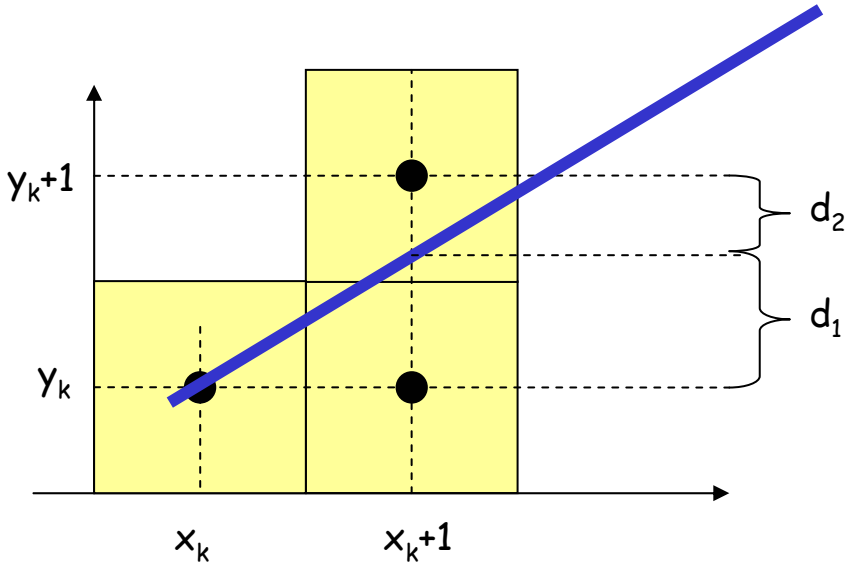
- Inconvenientes:
  - Existen errores de acumulación
  - El redondeo es muy lento

Algoritmo de Bresenham

- Sólo usa aritmética entera
- Supongamos el caso $0 < m < 1$ → hay que decidir qué pixel dibujamos a continuación, y ¡sólo hay dos candidatos!



- El algoritmo debe decidir cuál de los dos pintar
- Partamos del pixel (x_k, y_k), y hay que decidir entre el pixel (x_{k+1}, y_k) o (x_{k+1}, y_{k+1})
- Para ello calculemos la distancia vertical entre el centro de cada pixel y la línea real



$$y = m\,(x_k + 1) + b$$

$$d_2 = (y_k + 1 - y) = yk + 1 - m\,(x_k + 1) - b$$

$$d_1 = y - y_k = m\,(x_k + 1) + b - y_k$$

Algoritmo de Bresenham

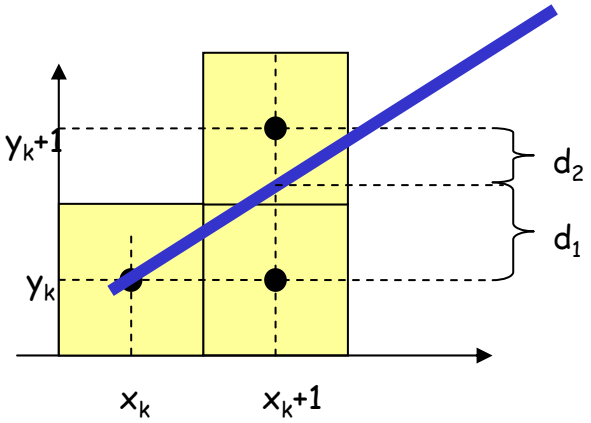
- La diferencia entre ambas constantes nos ayudará a decidir qué pixel pintar

  $d_1 - d_2 = 2m\,(x_k + 1) - 2y_k + 2b - 1$

- Multiplicando por Δx eliminamos el parámetro m, que no es entero

  $p_k = \Delta x\,(d_1 - d_2) = 2\,\Delta y\,x_k - 2\,\Delta x\,y_k + C$, donde $C = 2\,\Delta y + \Delta x\,(2b - 1)$

- Como $\Delta x > 0$, el signo de p_k coincide con el de la diferencia $(d_1 - d_2)$, y por tanto:
- Si $p_k > 0 \rightarrow d_1 > d_2 \rightarrow$ hay que pintar el pixel (x_k+1, y_k+1)
- Si $p_k < 0 \rightarrow d_1 < d_2 \rightarrow$ hay que pintar el pixel (x_k+1, y_k)



- La gran ventaja es que puede calcularse p_{k+1} a partir del anterior p_k, utilizando solamente ¡aritmética entera!

  $p_{k+1} = \ldots = p_k + 2\,\Delta y - 2\,\Delta x\,\underbrace{(y_{k+1} - y_k)}$

  0 ó 1 dependiendo del signo de p_k

Algoritmo de Bresenham

```
Funcion Bresenham (int x0, y0, x1, y1)
// sólo para el caso 0 < m < 1, siendo x0 < x1

Pintar Pixel (x0, y0)

Calcular las constantes A=2Δy, B=2Δy-2Δx

Obtener el valor para p0 = 2Δy-Δx

Para cada xk sobre la línea
        si pk < 0
                Pintar Pixel (xk+1, yk)
                pk+1 = pk + A
        si pk > 0
                Pintar Pixel (xk+1, yk+1)
                pk+1 = pk + B
```

- Si m > 1, intercambiamos las variables x e y
- Si m < 0, el cambio es similar

Ejemplo

$P_0 = (20, 10)$

$P_1 = (30, 18)$

$\Delta x = 10$

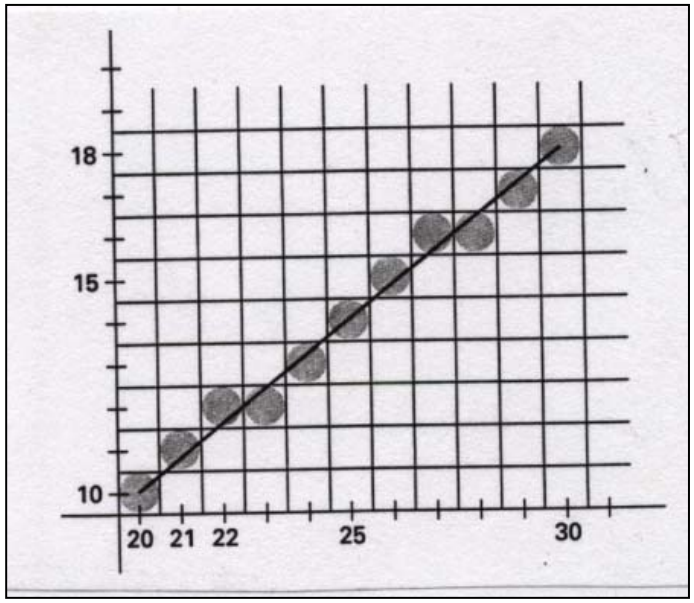
$\Delta y = 8$

$p_0 = 6$

$2\Delta y = 16$

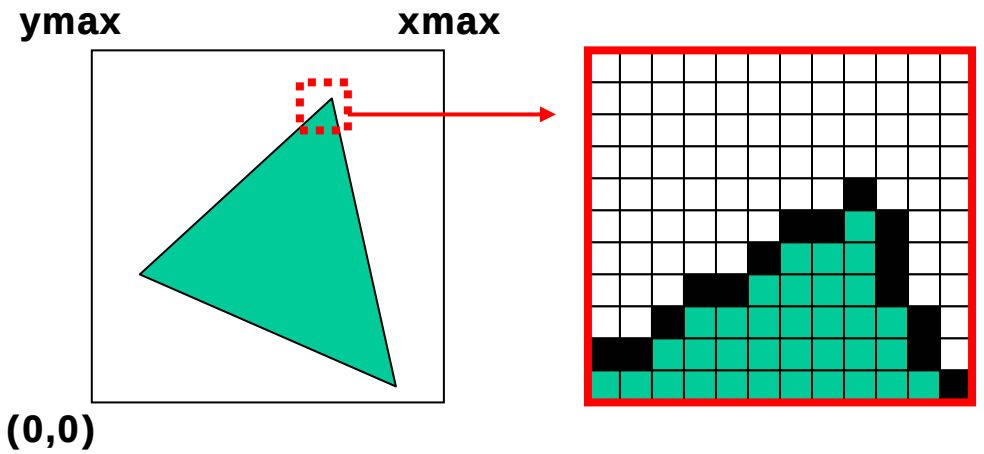
$2\Delta y - 2\Delta x = -4$

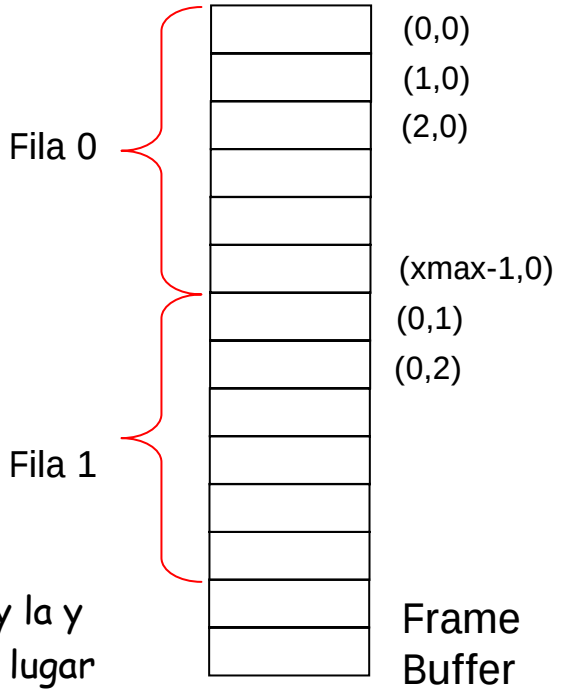
k	p_k	(x_{k+1}, y_{k+1})	k	p_k	$(x_{k+1}, y_{k+}$
0	6	(21, 11)	5	6	(26, 15)
1	2	(22, 12)	6	2	(27, 16)
2	−2	(23, 12)	7	−2	(28, 16)
3	14	(24, 13)	8	14	(29, 17)
4	10	(25, 14)	9	10	(30, 18)



Direccionando el Frame Buffer

- Para acceder al pixel (0,0) → I(0,0) = FB[0]
- Para acceder al pixel (x,y) → I(x,y) = FB[0] + y (xmax + 1) + x
- Para acceder al pixel (x+1,y) → I(x+1,y) = I(x,y) + 1
- Para acceder al pixel (x+1,y+1) → I(x+1, y+1) = I(x,y) + xmax + 1





- Si el punto (0,0) fuera la esquina superior izquierda, y la y creciera hacia abajo, habría que usar I(x, ymax-y) en lugar de I(x,y)

Problemas con la intensidad

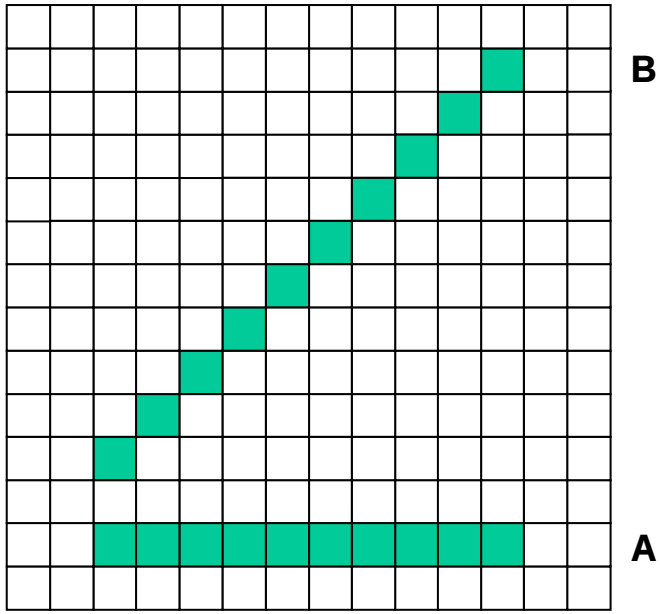
- El grosor de la línea depende de la pendiente → $\|B\| = \sqrt{2}\,\|A\|$

- Sin embargo, ambas usan el mismo número de pixels

- Si I es la intensidad de cada pixel, la intensidad por unidad de longitud de A es I, pero la de B es I/V2 → el ojo lo nota

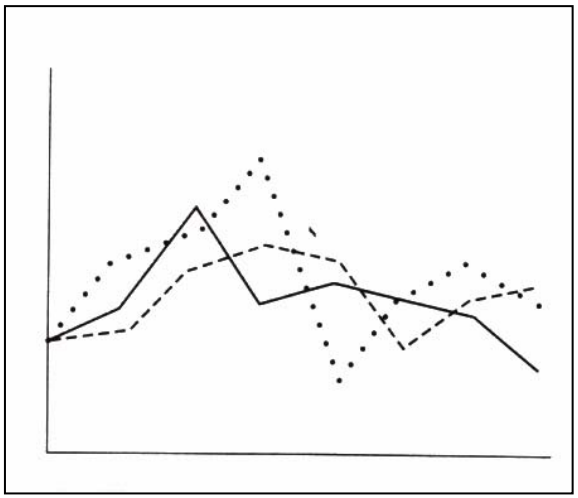
- Solución:

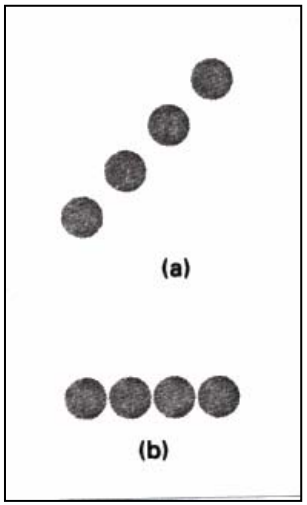
- Que la intensidad de los pixels dependa de la pendiente



Tipos de línea

- Existen varios tipos: continua, discontinua, con puntos
- Los procedimientos para dibujar estas líneas van mostrando secciones contiguas de pixels, y luego se van saltando otros
- ¿Cómo se puede implementar esto?
- Las secciones de pixels se especifican mediante una máscara
- Ejemplo: 1111000 → se pintan 4 pixels y se saltan 3

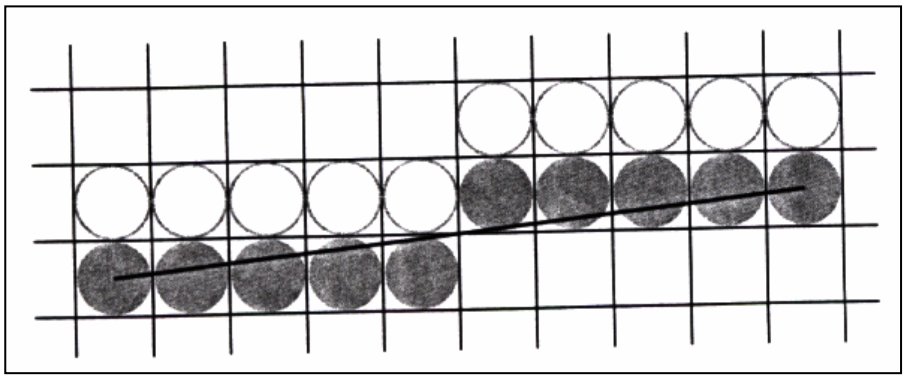


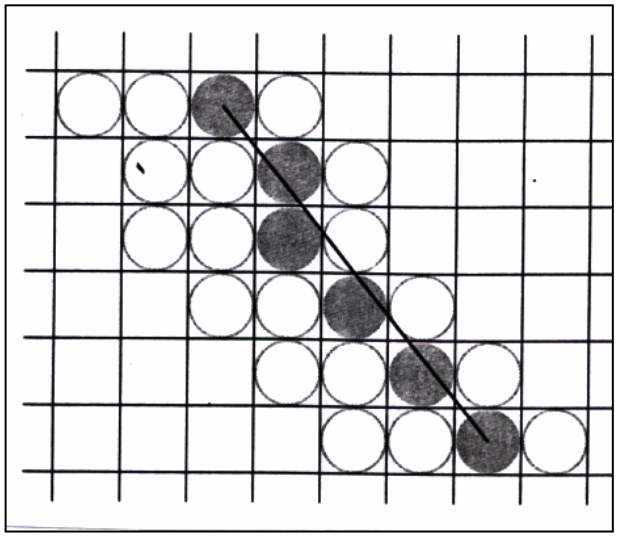


- Al fijar el número de pixels, las longitudes son diferentes según la dirección
- Solución:
- ajustar el número de pixels dependiendo de la pendiente
- Otra forma para dibujar líneas discontinuas sería tratar cada tramo como una línea individual

Grosor de línea

- ¿Cómo podemos pintar líneas de grosor mayor que 1?
- Solución: si la pendiente es menor que 1, para cada posición de x pintamos una sección vertical de pixels, tantos como ancho de línea queramos, por igual a cada lado
- Si la pendiente es mayor que 1, se usan secciones horizontales

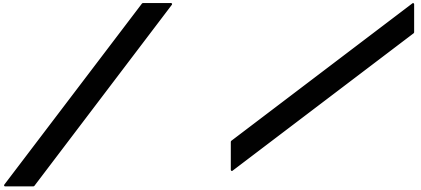




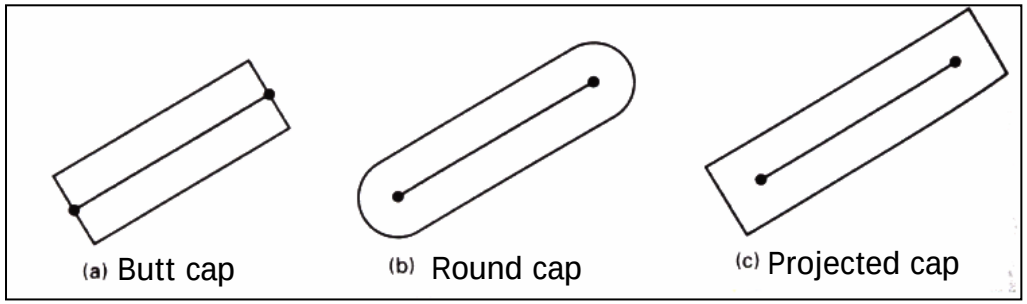
- Hay que tener en cuenta que el ancho de las líneas horizontales y verticales será V2 veces más grueso que las diagonales

Problemas en las terminaciones

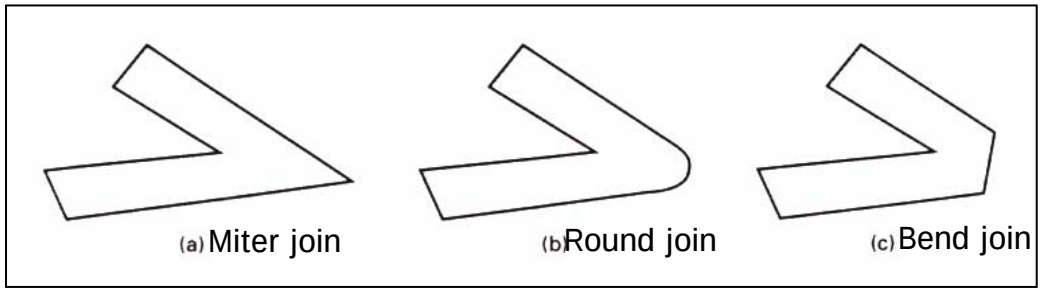
- Existe un problema en los bordes finales de las líneas: son siempre horizontales o verticales!



- Tres soluciones diferentes:



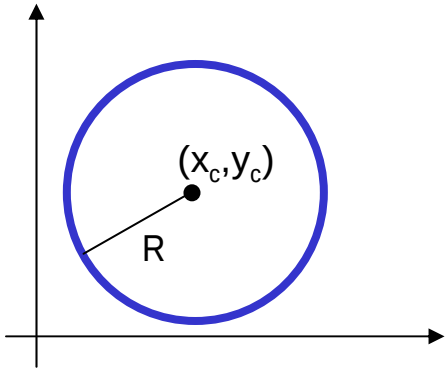
- Otra forma para dibujar líneas gruesas es pintar el rectángulo relleno
- También aparecen problemas al conectar líneas → aparecen huecos en las uniones!
- Tres soluciones diferentes:



Dibujo de circunferencias

- La ecuación de un círculo de radio R centrado en (x_0, y_0) es

  $(x-x_c)^2 + (y-y_c)^2 = R^2$

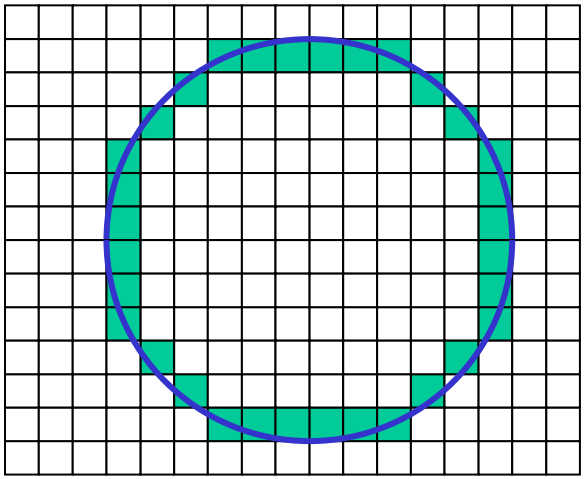


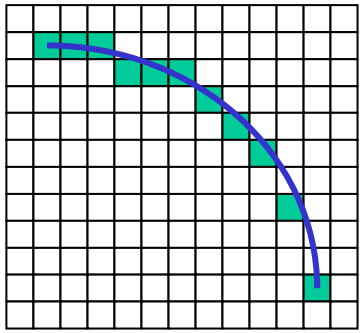
Algoritmo de fuerza bruta:

```
Para x=xc-R hasta x=xc+R

    Calcular    y = y0 ± sqrt(R^2 - (x - x0)^2)

    Pintar Pixel (x, round(y))
```





- No es nada eficiente
- Cada paso requiere una raíz cuadrada
- El espaciado entre pixels no es uniforme

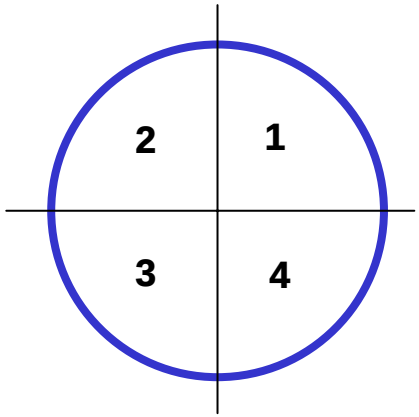
Otra forma

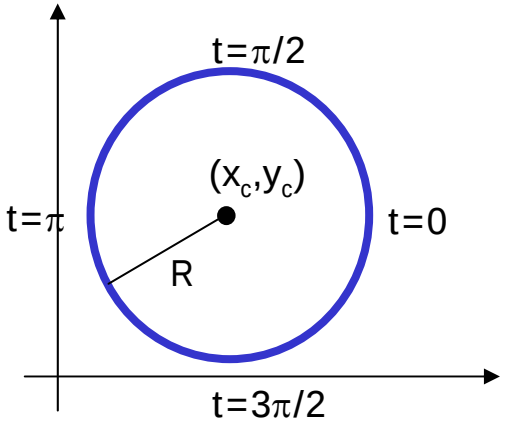
- ¿Cómo solucionar lo de los agujeritos?
- Pasando a coordenadas polares

$$x = x_c + R \cos t$$

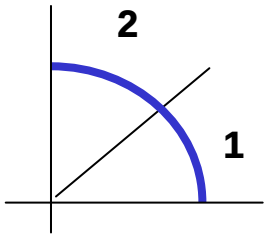
$$y = y_c + R \sin t$$

- El valor del incremento del ángulo t debe ser lo suficientemente pequeño para evitar los huecos
- Podemos reducir cálculo aplicando simetrías:





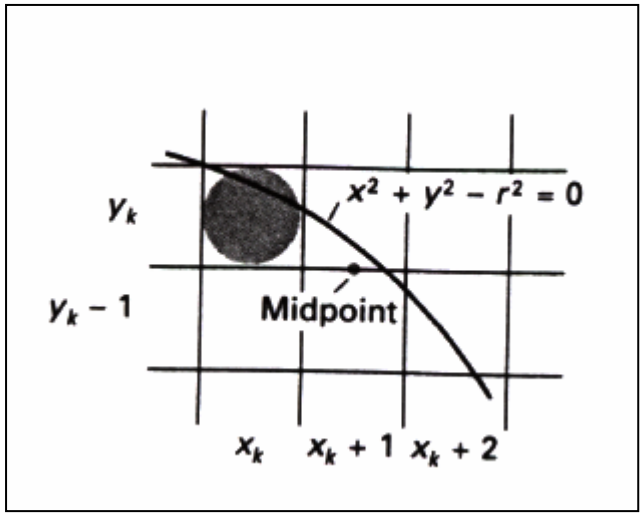
- Incluso el primer octante es simétrico al segundo a través de la diagonal



- Conclusión: dibujando sólo el segundo octante, desde x=0 hasta x=y podemos pintar todo el círculo
- Problema: se necesitan raíces cuadradas y funciones trigonométricas → demasiado costoso

Algoritmo del Punto Medio

- Hay que determinar el pixel más cercano a la circunferencia
- Consideremos el centro del círculo en (0,0)
- Comenzaremos en el punto (0,R) e iremos desde x=0 hasta x=y, donde la pendiente va de 0 a -1
- Sólo pintaremos el primer octante
- Sea la función:



$$f(x,y) = x^2 + y^2 - R^2 \begin{cases} < 0 \rightarrow (x,y) \text{ está dentro} \\ = 0 \rightarrow (x,y) \text{ está sobre la circunferencia} \\ > 0 \rightarrow (x,y) \text{ está fuera} \end{cases}$$

- Este test lo ejecutaremos en los puntos medios entre los pixels que hay que decidir

Algoritmo del Punto Medio

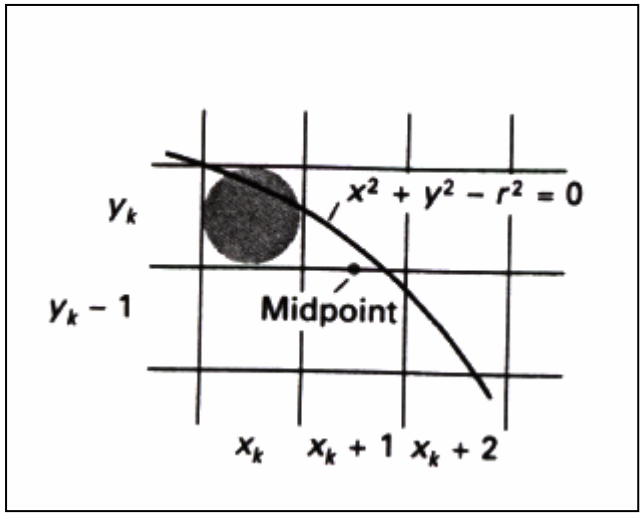
- Supongamos ya dibujado el pixel (x_k, y_k)

$p_k = f(x_k + 1, y_k - \frac{1}{2}) = (x_k + 1)^2 + (y_k - \frac{1}{2})^2 - R^2$

$p_{k+1} = f(x_{k+1} + 1, y_{k+1} - \frac{1}{2}) = (x_{k+1} + 1)^2 + (y_{k+1} - \frac{1}{2})^2 - R^2$

$p_{k+1} = p_k + 2x_{k+1} + 1 + (y^2_{k+1} - y^2_k) - (y_{k+1} - y_k)$

0 ó 1 dependiendo del signo de p_k



- Por lo tanto:
- Si $p_k < 0 \rightarrow p_{k+1} = p_k + 2x_{k+1} + 1 \rightarrow$ hay que pintar el pixel (x_k+1, y_k)
- Si $p_k > 0 \rightarrow p_{k+1} = p_k + 2x_{k+1} - 2y_{k+1} + 1 \rightarrow$ hay que pintar el pixel (x_k+1, y_k-1)

- Empezamos en el punto (0, R). ¿Cuánto vale p_0?

$p_0 = f(1, R-1/2) = \ldots = 5/4 - R \rightarrow$ no es entero!

Sin embargo, da igual. Podemos redondearlo, porque todos los incrementos son enteros, y sólo queremos utilizar el signo de p_k, y no su valor

Algoritmo del Punto Medio

```
Funcion PuntoMedio (int xc, yc, float R)

Pintar Pixel (0, R)

Calcular p0 = 5/4 – R  // si R es entero, p0 = 1-r

Para cada xk
        si pk < 0
                Pintar Pixel (xk+1, yk)
                pk+1 = pk + 2xk + 3
        si pk > 0
                Pintar Pixel (xk+1, yk-1)
                pk+1 = pk + 2xk – 2yk + 5

Determinar por simetría los puntos de los otros 7 octantes

Pintar Pixel (x+xc, y+yc)
```

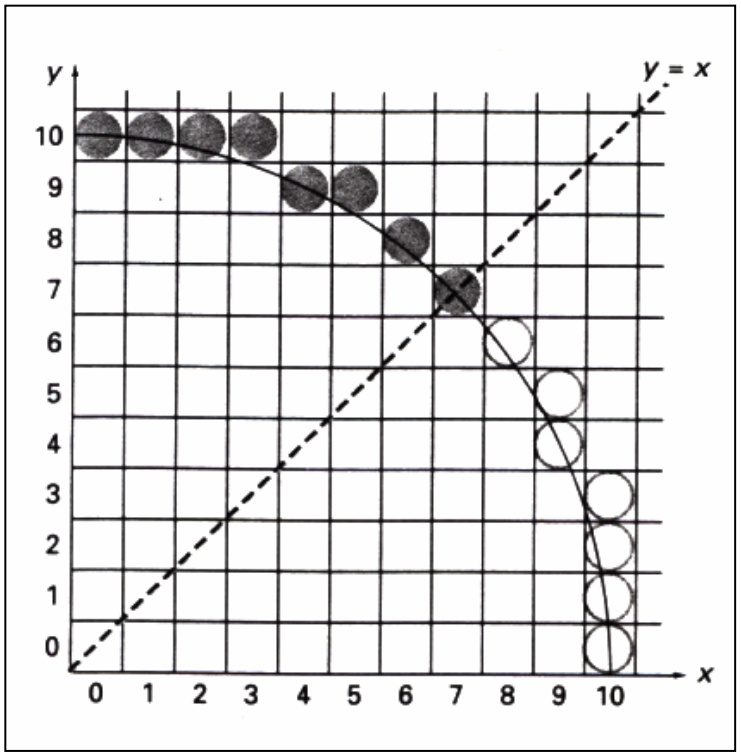
Ejemplo

R = 10

$p_0 = 9$

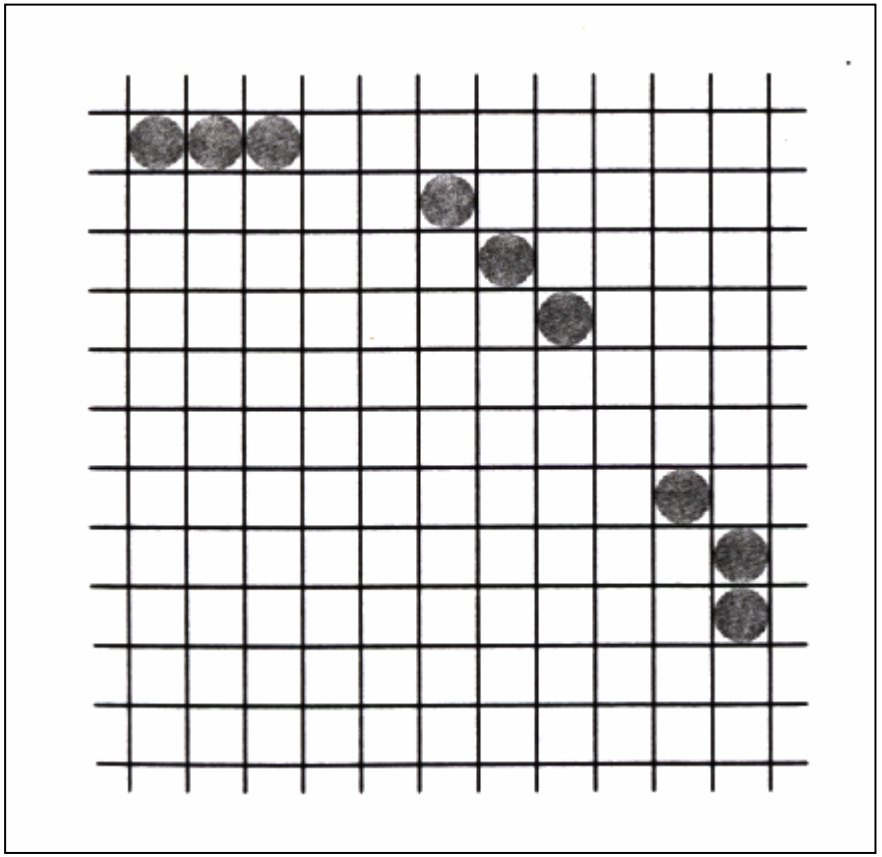
$(x_0, y_0) = (0, 10)$

k	p_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0	−9	(1, 10)	2	20
1	−6	(2, 10)	4	20
2	−1	(3, 10)	6	20
3	6	(4, 9)	8	18
4	−3	(5, 9)	10	18
5	8	(6, 8)	12	16
6	5	(7, 7)	14	14



Tipos de líneas

- Para dibujar líneas discontinuas usaremos máscaras como en las rectas
- Al copiar al resto de octantes hay que tener en cuenta la secuencia del interespaciado
- Las longitudes varían con la pendiente



Grosor de línea

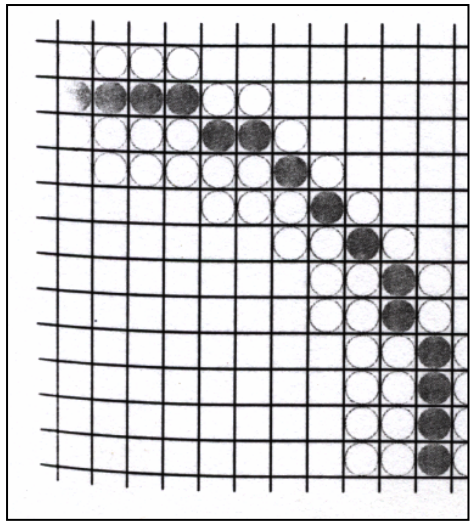
- Existen 3 métodos:

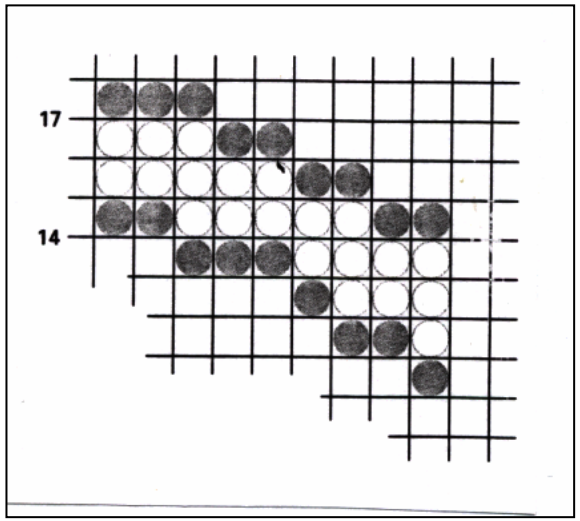
1. Pintando secciones horizontales o verticales según sea la pendiente mayor o menor que 1

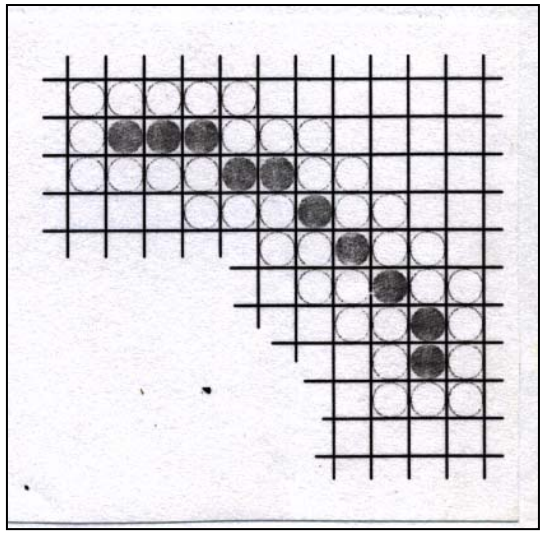
2. Rellenar el espacio entre dos curvas paralelas, separadas por una distancia igual al ancho que queremos

3. Usar una brocha e irla moviendo a lo largo de la curva

1	1	1
1	1	1
1	1	1

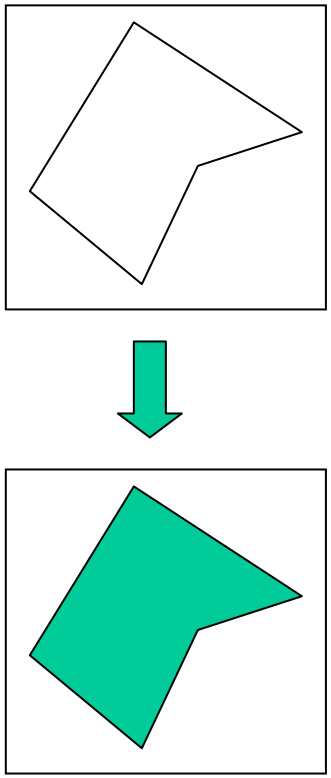


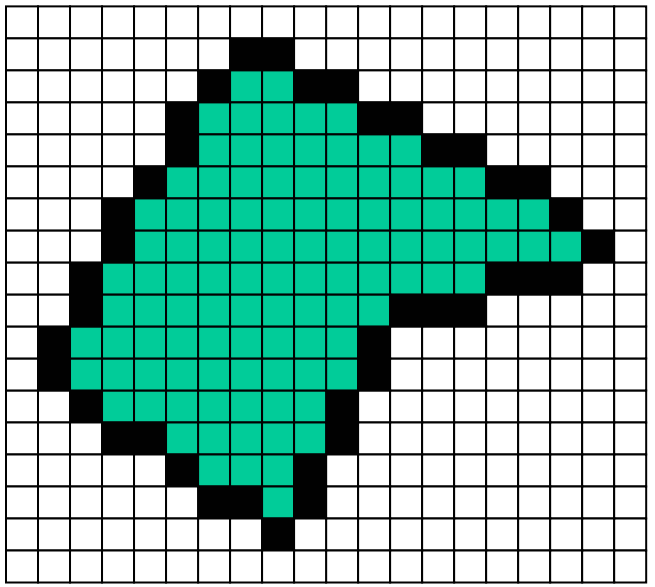




Relleno de primitivas

- Dada un área cerrada, hay que ser capaz de rellenar los pixels interiores con un color determinado

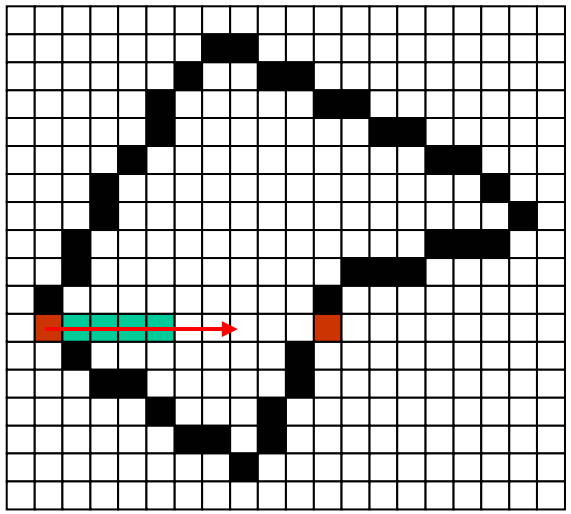




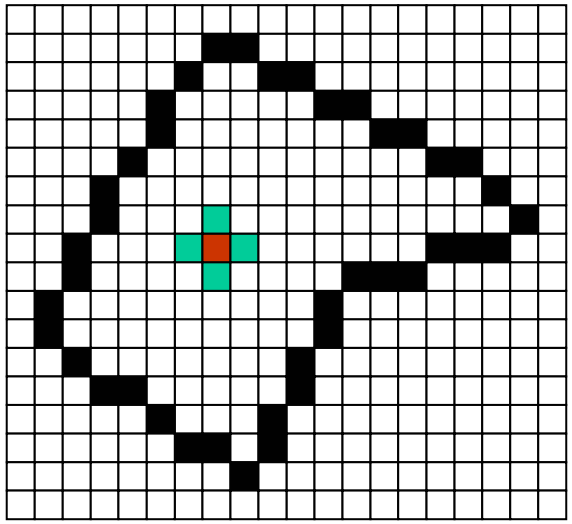
Relleno de primitivas

- Existe 2 categorías de métodos

1. Relleno por scan - line: fila a fila va trazando líneas de color entre aristas

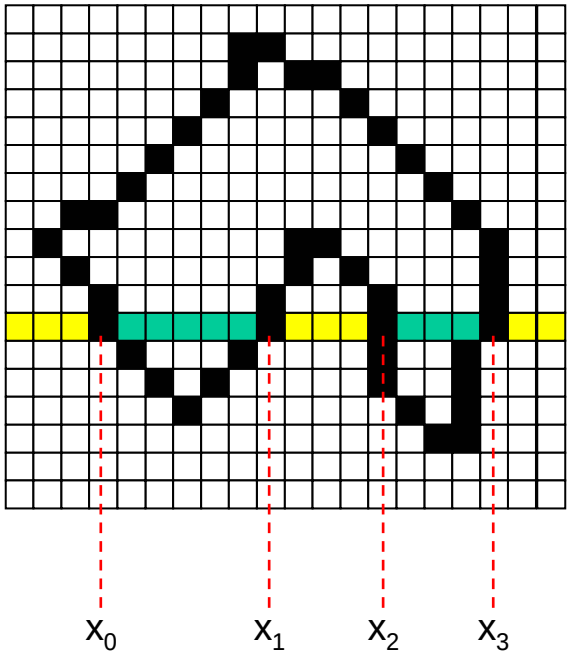


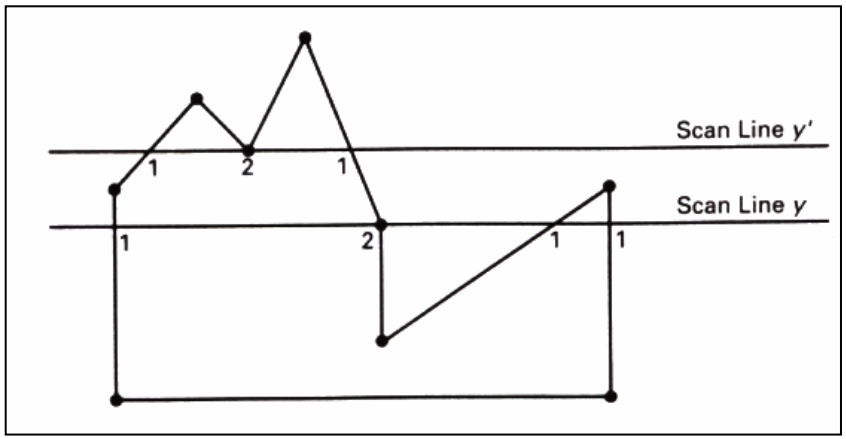
2. Relleno por inundación: a partir de un punto central, se va expandiendo recursivamente hasta alcanzar el borde del objeto



Relleno por Scan-Line

- Para cada scan-line que cruce el polígono se busca la intersecciónentre la línea de barrido y las aristas del polígono
- Dichas intersecciones se ordenan y se rellenan a pares
- El problema es existen problemas cuando se intersecta un vértice

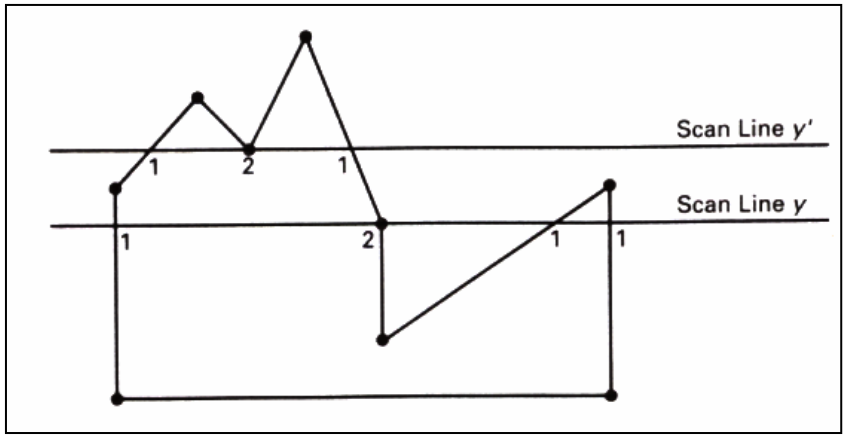




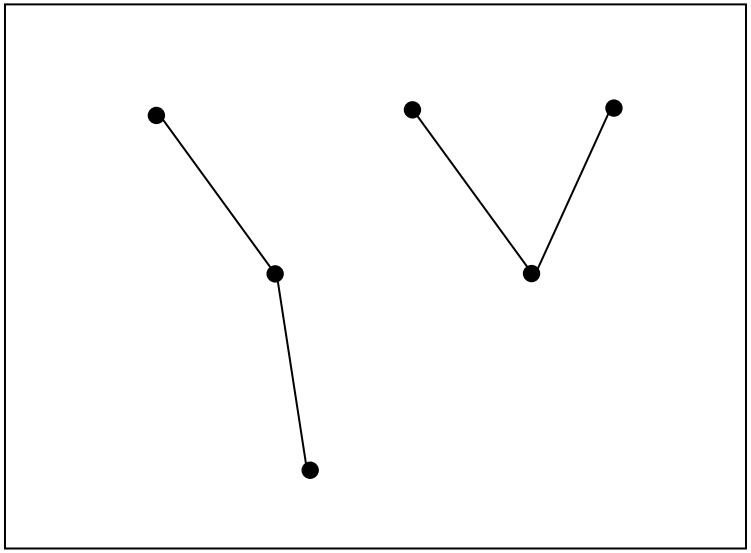
- En la scan-line *y* aparecerían 5 aristas intersectadas!
- ¿Cómo lo solucionamos?

Relleno por Scan-Line

- Solución: contarlo sólo una vez
- Pero entonces habría problemas en la scan-line *y'*



- Solución: contarlo sólo una vez
- Pero entonces habría problemas en la scan-line *y'*
- ¿Cómo distinguir entre ambos casos?
- La diferencia de la línea *y'* es que las aristas están al mismo lado de la scan-line
- ¿Cómo detectarlo?
- Mirando si los tres vértices en cuestión son monótonamente crecientes o decrecientes



Aceleración del Scan-Line

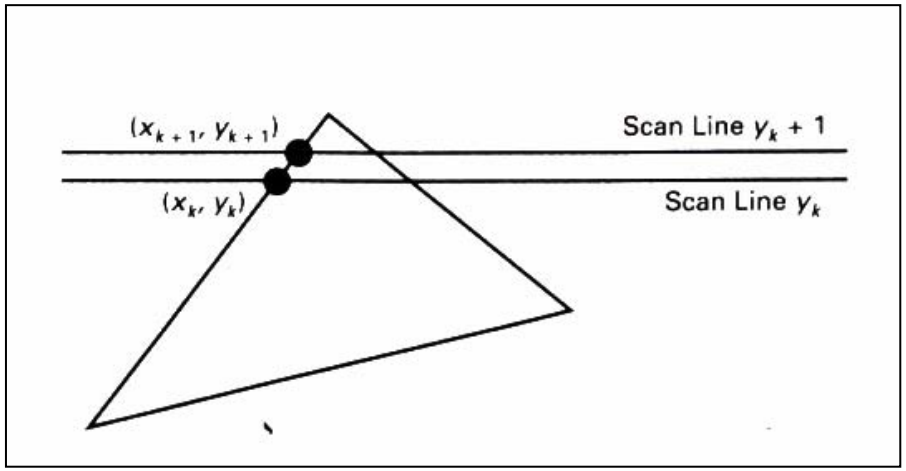
- En lugar de calcular para cada scan-line las intersecciones contodos las aristas del polígono, podemos ir aprovechando el cálculo en cada scan-line anterior

- La pendiente de la arista es

  $m = (y_{k+1} - y_k) / (x_{k+1} - x_k)$

- Como $\Delta y = 1$ entre cada scan-line:

  $x_{k+1} = x_k + 1/m$



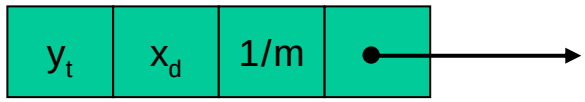
- Para acelerar aún más podemos pasar a aritmética entera:

  $x_{k+1} = x_k + \Delta x / \Delta y$

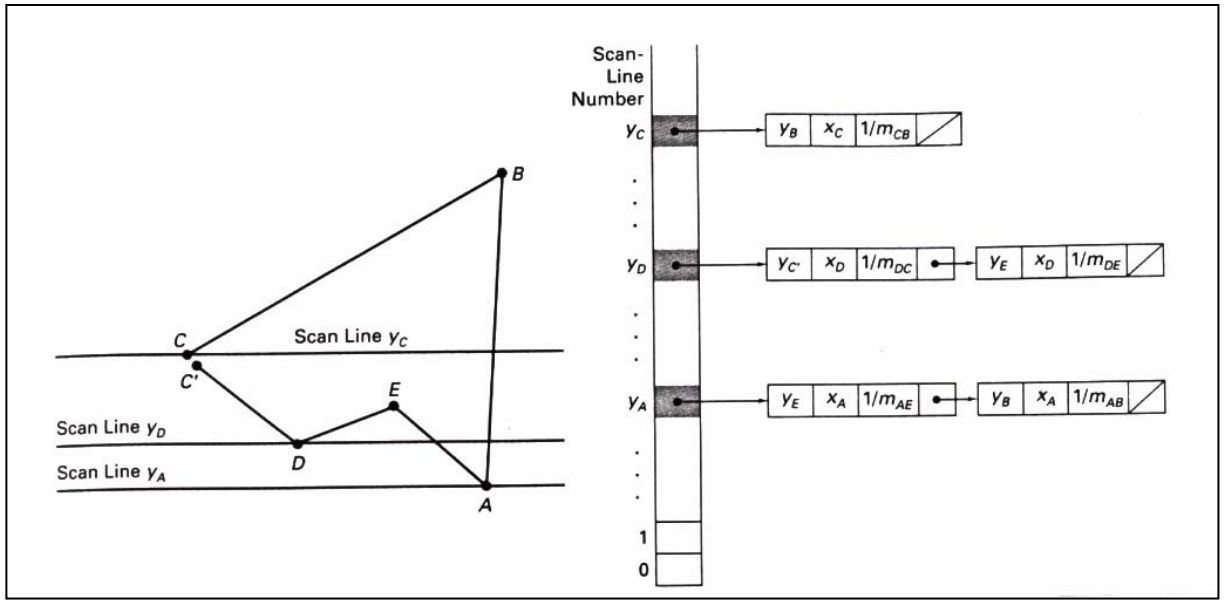
- Podemos ir incrementando el contador en Δx unidades en cada scan-line
- Cuando el contador supere Δy, restamos Δy y hacemos x++

Algoritmo optimizado para el relleno scan-line

- Primero hay que crear una tabla de bordes (TB), para todas las aristas del polígono (exceptuando las horizontales)
- Cada arista viene representada por cuatro valores
  - Coordenada y del punto más alto
  - Coordenada x del punto más bajo
  - Inversa de la pendiente
  - Puntero a otra arista en la misma scan-line

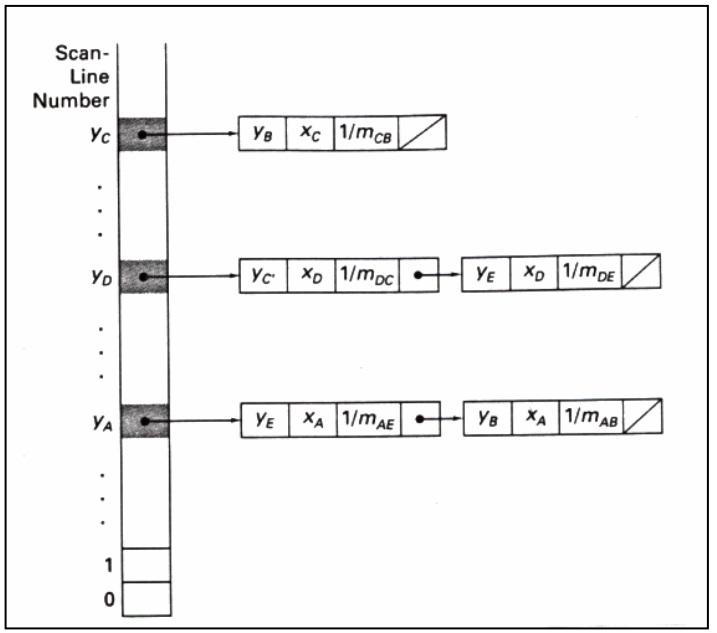


- Se crea un vector vacío, con tantas posiciones como filas tenga la pantalla, y se coloca cada arista en la posición de la scan-line del punto más bajo

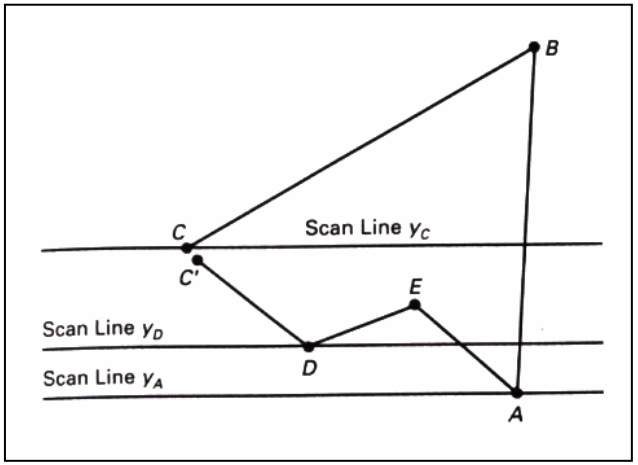


Algoritmo optimizado para el relleno scan-line

- Comenzamos desde abajo, y vamos creando una lista de bordes activos (LBA), que contendrán en cada iteración las aristas cruzadas por dicha scan-line

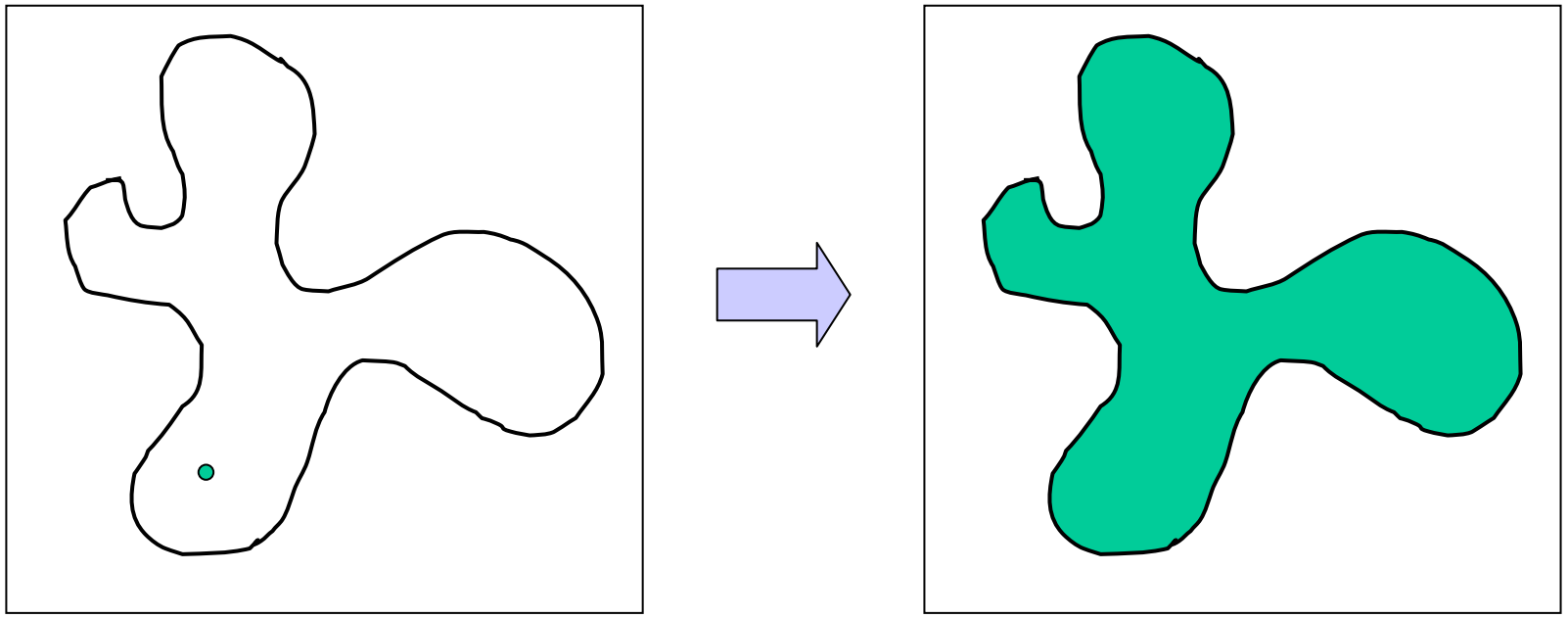



```
Funcion Scanline()
Inicializar LBA vacía y crear TB
Repetir hasta que LBA y TB vacías
   Mover de TB a LBA lados con y_min = y
   Ordenar LBA según x
   Rellenar usando pares de x de LBA
   Eliminar lados de LBA con y = y_max
   Incrementar y a la siguiente scan-line
   Actualizar las x en LBA
```



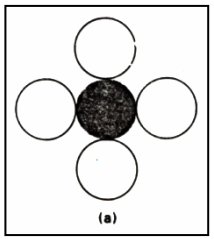
Relleno por inundación

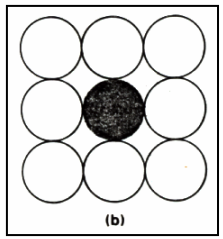
- Empieza en un punto interior y pinta hasta encontrar la frontera del objeto
- Partimos de un punto inicial (x,y), un color de relleno y un color de frontera
- El algoritmo va testeando los pixels vecinos a los ya pintados, viendo si son frontera o no
- No sólo sirven para polígonos, sino para cualquier área curva sobre una imagen → se usan en los programas de dibujo

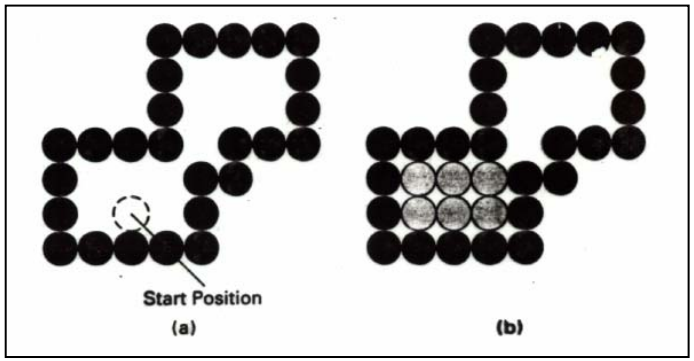


Algoritmo de relleno por inundación

- Hay dos formas de considerar los vecinos : 4 u 8
- Dependiendo de qué esquema elijamos, el relleno será diferente







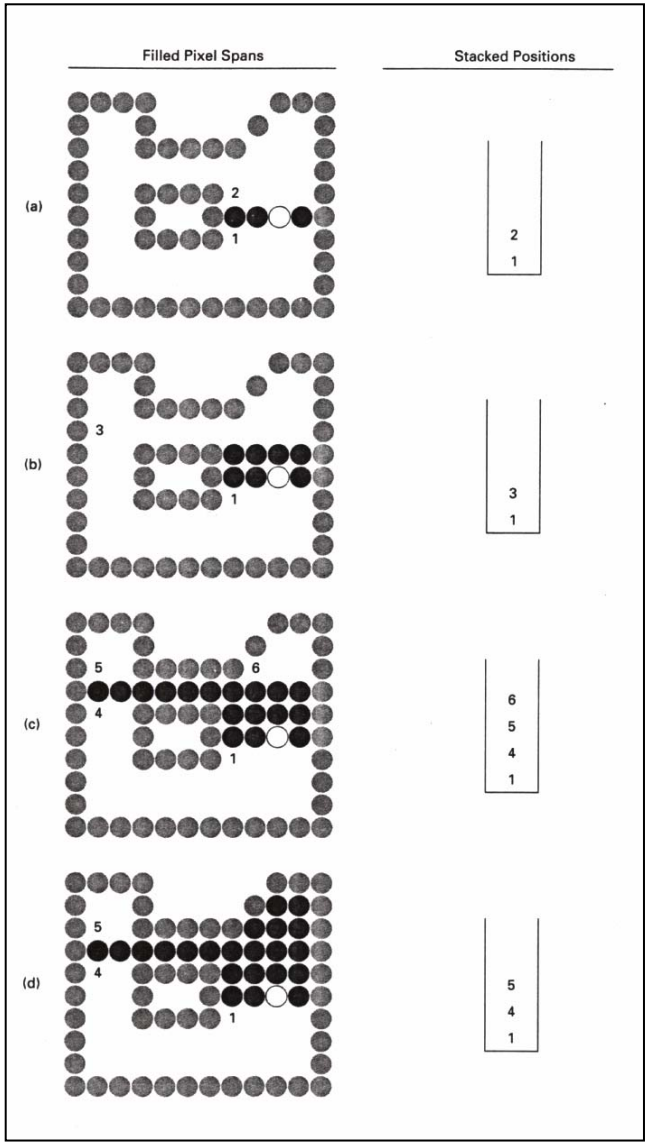
- El algoritmo se presta a un esquema recursivo muy simple

```
Funcion Inundación(x, y, col1, col2)
color = LeerPixel (x,y)
Si (color!=col1 && color!=col2) entonces
     PintaPixel (x,y,col1)
     Inundación (x+1, y, col1, col2);
     Inundación (x-1, y, col1, col2);
     Inundación (x, y+1, col1, col2);
     Inundación (x, y-1, col1, col2);
```

Algoritmo optimizado de relleno por inundación

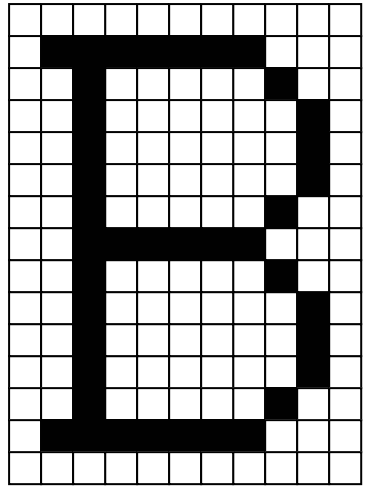
- El algoritmo anterior necesita mucha memoria, y si el área a rellenar es muy grande se desborda la pila
- ¿Cómo podemos ahorrar memoria para poder rellenar áreas de cualquier tamaño?

- La solución consiste en no explorar todos los vecinos de cada pixel, sino sólo a lo largo de un scan-line
- Rellenamos el span donde se encuentra el punto inicial
- Además, guardamos las posiciones iniciales de todos los spans de las líneas horizontales contiguas al scan-line



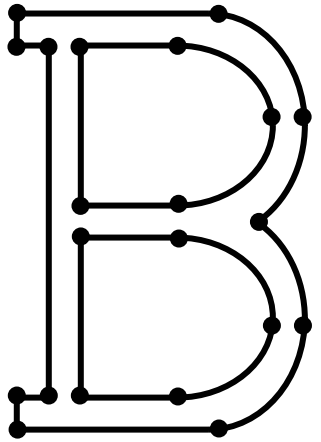
Generación de caracteres de texto

- Las letras y números pueden dibujarse en muchos estilos y tamaños
- Typeface: cada díseño diferente para una familia entera de caracteres (Courier, Helvetica, Arial)
- Existen dos formas de representación:



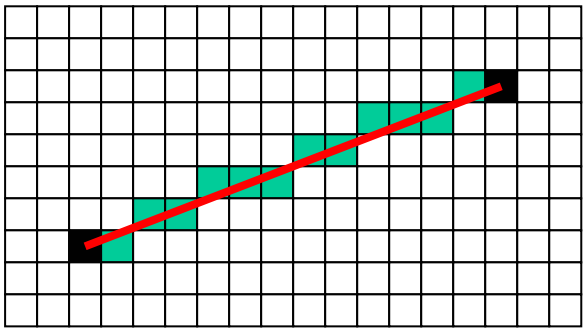
- **Bitmap Fonts** (usando una malla rectangular de pixels)
  - más simples de definir y dibujar
  - requieren mucho espacio de almacenamiento, porque cada variación en tamaña o formato requiere un nuevo bitmap

- **Outline Fonts** (usando una lista de segmentos rectos y curvos)
  - ahorran más memoria
  - los diferentes tamaños y formatos se crean fácilmente a partir de la forma original
  - lleva más tiempo procesarlas
  - pueden pintarse huecas o rellenas
  - Pueden pintarse en cualquier orientación



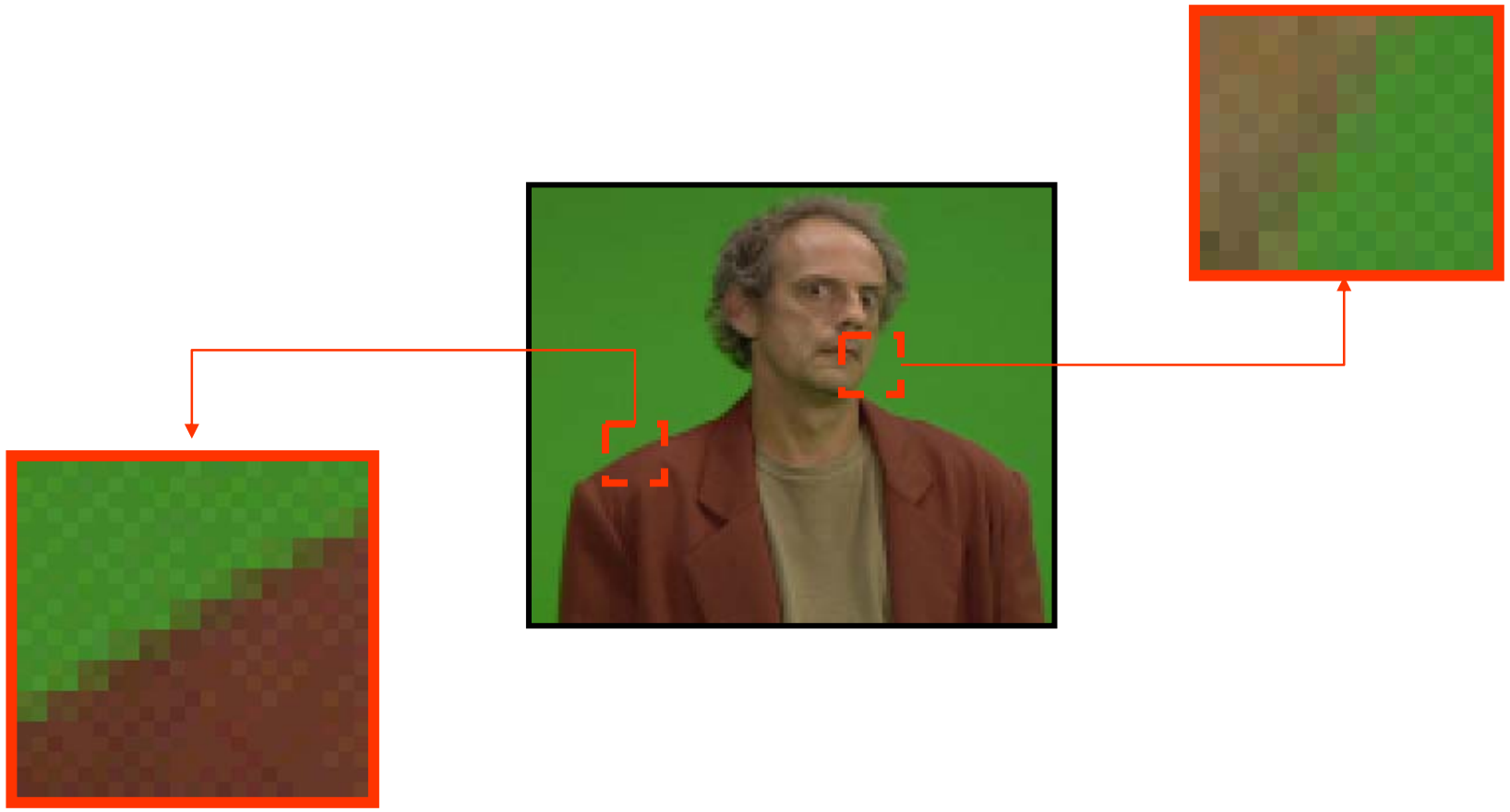
Técnicas anti-aliasing

- Las primitivas construidas con los algoritmos raster tienen una apariencia de "escalera", debido a la discretización en pixels



- Soluciones hardware:
  - mayor resolución de las pantallas → existe un límite para que el Frame Buffer mantenga su refresco a 30 Hz
  - pixels más pequeños: existe un límite en la precisión del haz de electrones

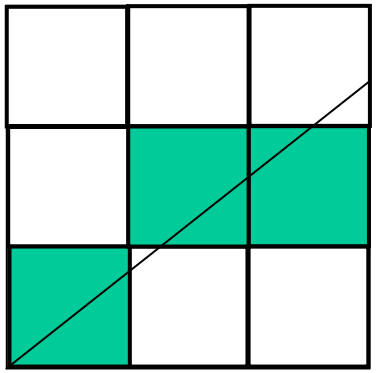
Imagen digitalizada



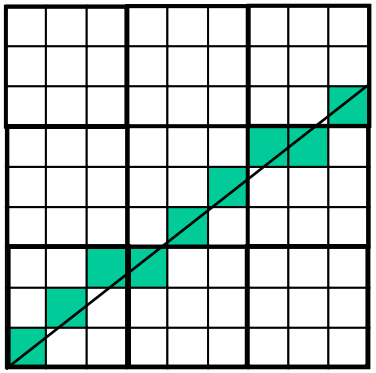
Super-sampling

- Consiste en incrementar virtualmente la malla de pixels, engañando a Bresenham
- Una vez calculada la recta para los subpixels, determinamos el color del pixel real
- Cada pixel representa entonces un área finita de la pantalla, y no un punto infinitesimal

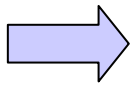
- Para dibujar una recta, contamos el número de subpixels que están sobre la línea
- La intensidad del pixel final será proporcional al contador anterior
- Para máscaras 3x3 tendremos 3 posibles intensidades

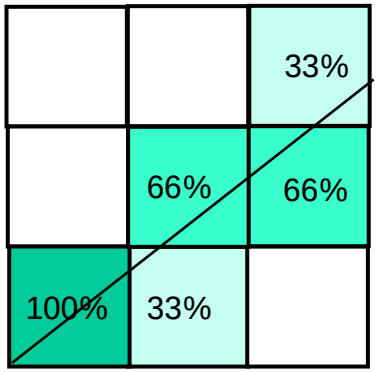


Bresenham en pixels



Bresenham en subpixels

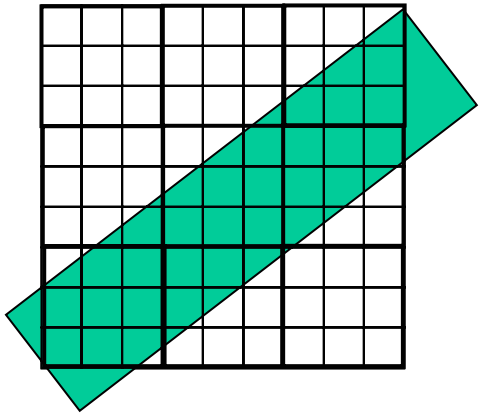




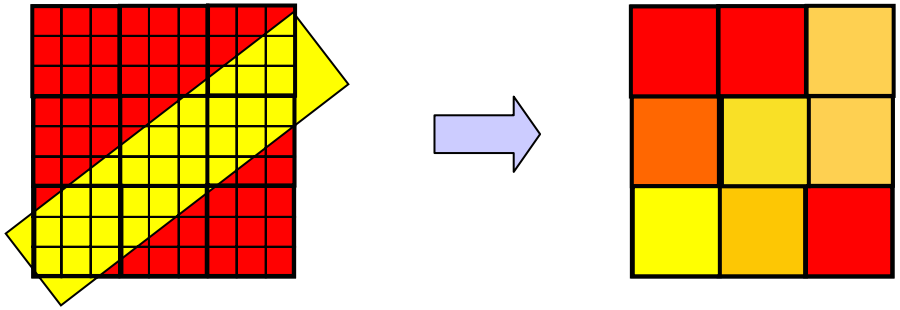
Apariencia final

Super-sampling

- Otra versión de super-sampling diferente consiste en considerar que las líneas tienen un grosor de 1 pixel → son en realidad un rectángulo
- Lo que hacemos entonces es contar los subpixels que caen dentro del rectángulo
- Para máscaras 3x3 tendríamos 9 intensidades diferentes
- Se requiere más cálculo que la versión anterior, pero el resultado es más exacto

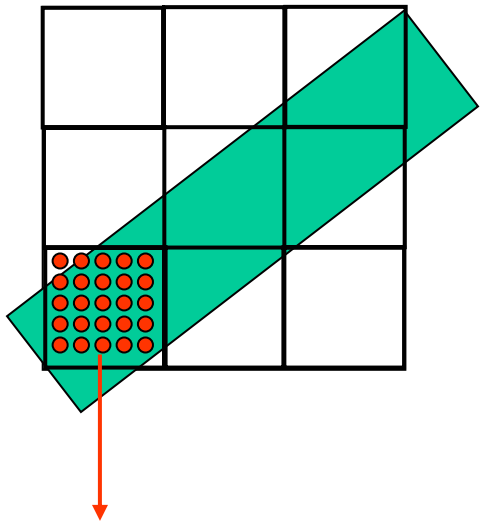


- NOTA: Si el fondo de la imagen tiene color, debemos promediar entre ambos colores para determinar el valor del pixel



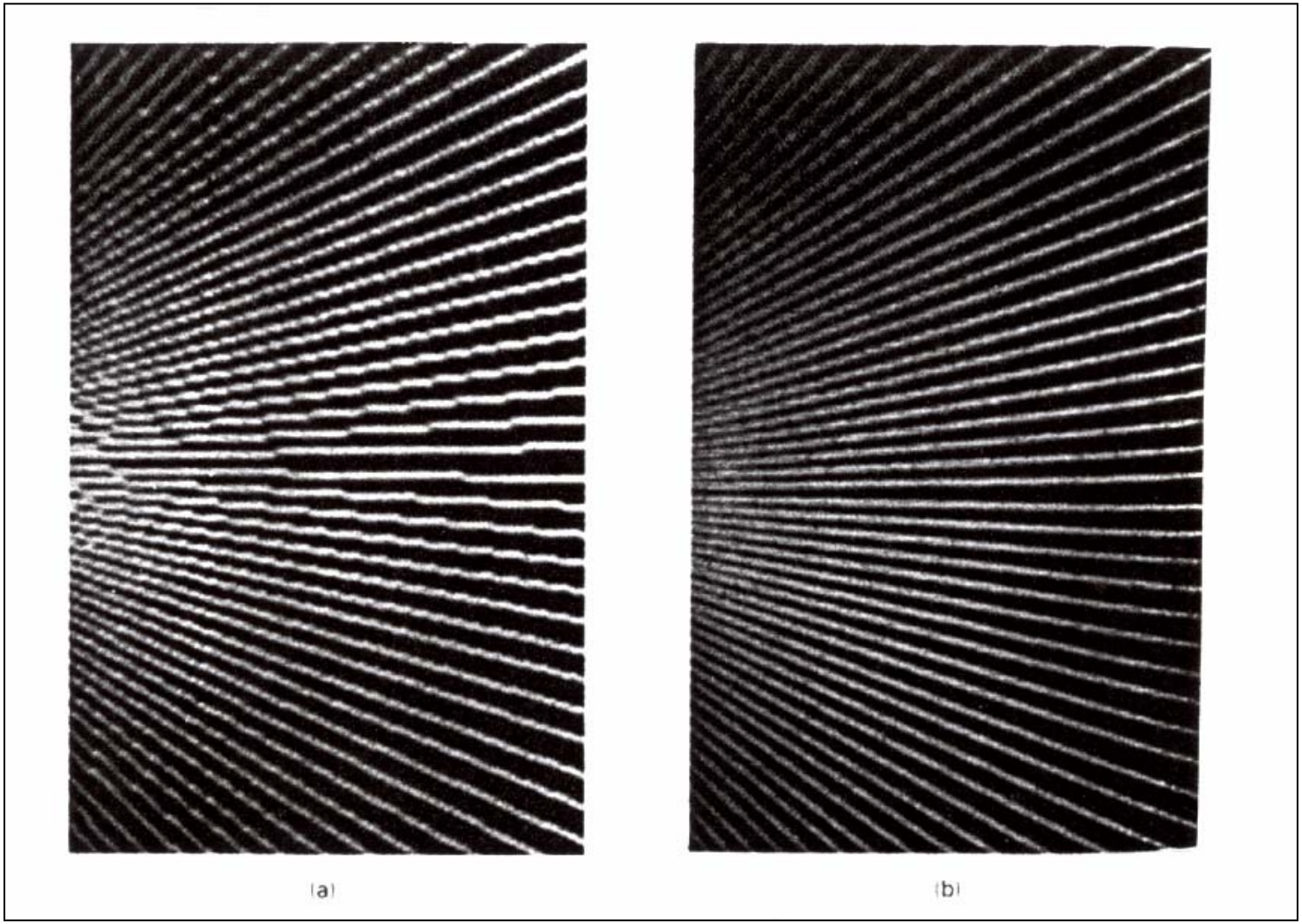
Area-sampling

- La intensidad del pixel viene dada por el área de intersección entre cada pixel y el objeto que se va a dibujar
- Para estimar el área sería muy costoso evaluar la integral
- Lo que hacemos es testear una malla de puntos interiores al pixel y calcular cuántos caen dentro del rectángulo → método de integración de Montecarlo
- Si el fondo también tiene color, promediamos entre ambos como antes



90% de pixels dentro → 90% de intensidad

Ejemplo de anti-aliasing de líneas

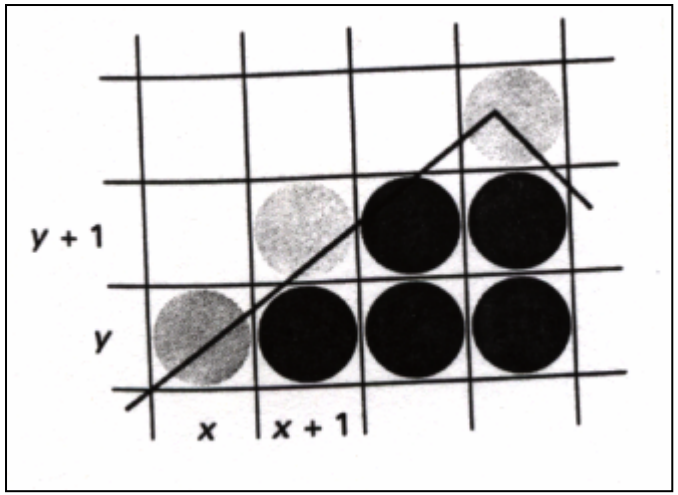


sin anti-aliasing

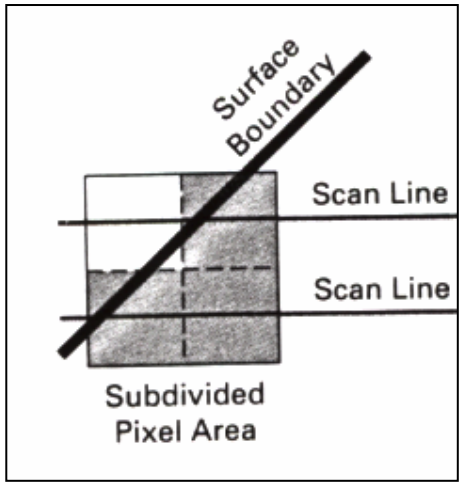
con anti-aliasing

Anti-aliasing de contornos

- Cuando el aliasing se produce en un contorno que separa dos zonas de color diferente (aristas de un polígono relleno) → aliasing de contornos
- La solución consiste en incorporar las técnicas anteriores a los algoritmos de scan-line
- Area-sampling: Según el área de polígono que caiga dentro de cada pixel de la frontera, determinamos el color final
- Super-sampling: Añadimos más scan-lines en la imagen virtual que le pasamos al algoritmo de relleno → decidimos el color de los pixels frontera en función de dónde acabe cada scan-line

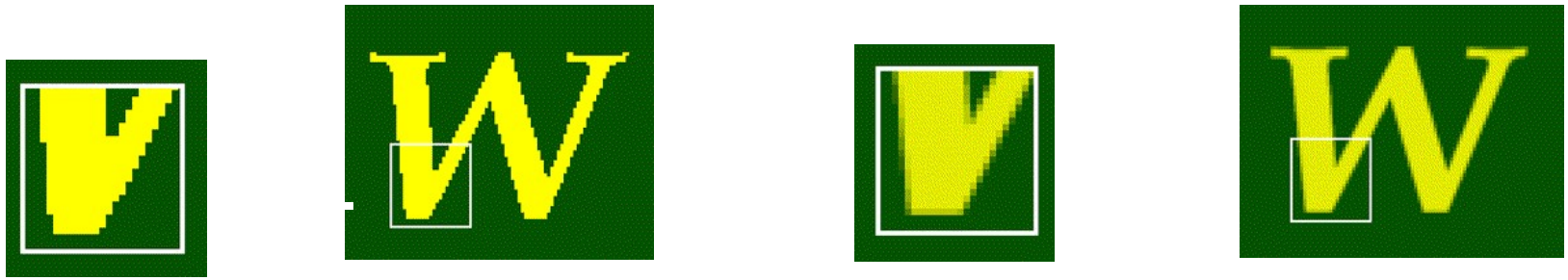


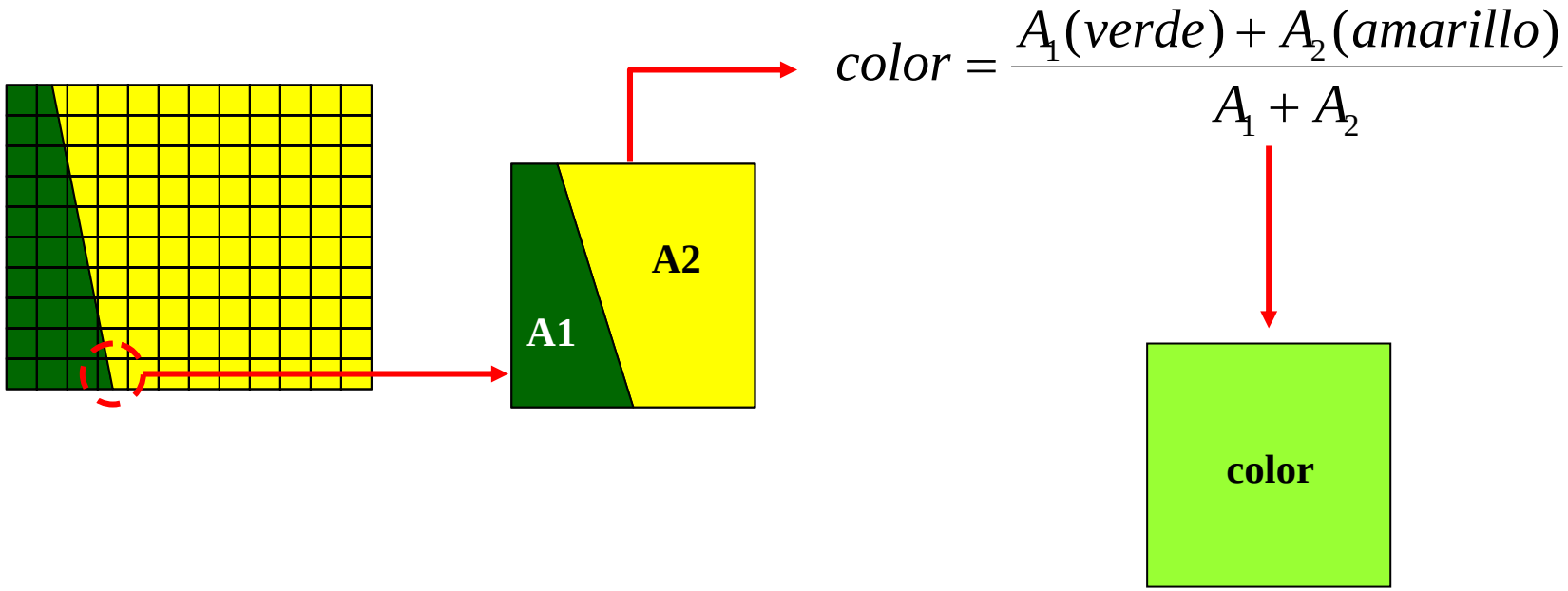
Área-sampling



Super-sampling

Ejemplo de anti-aliasing de contornos





$$color = \frac{A_1(verde) + A_2(amarillo)}{A_1 + A_2}$$