Líneas y Polígonos

Gráficas por Computadora (CC2018) - 2026

Líneas y Polígonos

Semestre 02, 2026

Objetivos de la clase

- Entender por qué dibujar una recta en píxeles es un problema de aproximación.
- Comparar tres enfoques: la ecuación de la recta, DDA y Bresenham.
- Comprender por qué el punto flotante es costoso y cómo Bresenham lo evita con enteros.
- Seguir el algoritmo de Bresenham paso a paso en un ejemplo completo.
- Aplicar el trazado de líneas para dibujar y rellenar polígonos.

El reto de dibujar una línea

En matemáticas una recta es continua e infinita: pasa por infinitos puntos.

La pantalla, en cambio, es una malla finita de píxeles: solo podemos encender casillas.

El problema es de aproximación: no podemos dibujar la recta ideal, así que debemos elegir el conjunto de píxeles que mejor la represente.

A convertir una figura geométrica (recta, polígono, círculo) en los píxeles que la representan se le llama rasterizar. Un buen algoritmo de rasterización debe ser:

- Preciso: los píxeles no se separan de la recta real.
- Eficiente: pocas operaciones por píxel, porque se ejecuta millones de veces.

Tres enfoques

Veremos tres algoritmos, de lo simple y caro a lo rápido y exacto:

1. Ecuación de la recta: directa, pero una multiplicación flotante por píxel.
2. DDA: incremental, cambia la multiplicación por una suma (aún flotante).
3. Bresenham: solo enteros, sin redondeo. El estándar actual.

Enfoque 1: ecuación de la recta

Recorremos cada `x` y calculamos su `y` con la ecuación `y = m·x + b`:

```
def linea_ecuacion(x0, y0, x1, y1):
    m = (y1 - y0) / (x1 - x0)   # pendiente
    b = y0 - m * x0             # intercepto
    for x in range(x0, x1 + 1):
        y = m * x + b           # 1 mult + 1 suma por pixel
        plot(x, round(y))       # redondeo al pixel mas cercano
```

- Cada píxel cuesta una multiplicación de punto flotante más un redondeo.
- Si la recta es más vertical que horizontal (pendiente grande), avanzar de uno en uno en `x` deja huecos entre píxeles.

Enfoque 2: DDA (Digital Differential Analyzer)

DDA es un algoritmo incremental de rasterización. En lugar de recalcular `y = m·x + b` desde cero en cada píxel, trabaja con las diferencias (los incrementos) entre píxeles consecutivos; de ahí el nombre Differential Analyzer.

La observación clave: al avanzar 1 en `x`, la `y` aumenta exactamente `m`. Entonces basta sumar `m` en cada paso en vez de multiplicar. Además, DDA elige como eje dominante el de mayor variación para no dejar huecos en líneas empinadas.

```
def dda(x0, y0, x1, y1):
    dx = x1 - x0
    dy = y1 - y0
    pasos = max(abs(dx), abs(dy))   # eje dominante
    x_inc = dx / pasos              # avance de x por paso
    y_inc = dy / pasos              # avance de y por paso
    x, y = x0, y0
    for _ in range(pasos):
        plot(round(x), round(y))
        x += x_inc                  # solo una suma
        y += y_inc                  # solo una suma
```

Ventajas y límites:

- Más rápido que la ecuación: cambia la multiplicación por una suma.
- No deja huecos: pinta un píxel por cada paso del eje dominante.
- Pero sigue usando punto flotante (`x_inc`, `y_inc` son fracciones) y el redondeo repetido acumula error y es lento en hardware limitado.

Enfoque 3: algoritmo de Bresenham

Publicado por Jack E. Bresenham en 1962 mientras trabajaba en IBM, originalmente para controlar un plotter. Es un algoritmo de rasterización de líneas

que usa únicamente aritmética entera, y hoy sigue siendo el estándar en GPUs y librerías gráficas. La misma idea sirve para rasterizar círculos (variante del mismo autor).

La idea: un término de error

- Recorremos el eje dominante un píxel a la vez.
- En cada columna la recta real cae entre dos píxeles candidatos: el de la misma fila y el de la fila siguiente.
- Un término de error entero acumula qué tan lejos va la recta del píxel elegido.
- Cuando el error cruza un umbral, el otro píxel está más cerca: avanzamos en el eje menor.

Todo se resuelve con sumas, restas y comparaciones de enteros, sin punto flotante ni redondeo.

Inicialización

Se calcula una sola vez, antes del ciclo:

```
dx  = abs(x1 - x0)            # distancia horizontal
dy  = -abs(y1 - y0)          # distancia vertical (negativa)
sx  = 1 si x0 < x1, si no -1 # sentido de avance en x
sy  = 1 si y0 < y1, si no -1 # sentido de avance en y
err = dx + dy                # termino de error inicial
```

`dy` se guarda negativa para simplificar las comparaciones, y el signo de `err` decidirá cada avance.

El ciclo

```
while True:
    plot(x0, y0)                    # pinta el pixel actual
    if x0 == x1 and y0 == y1:
        break                       # llegamos al destino
    e2 = 2 * err                    # el doble, para comparar sin fracciones
    if e2 >= dy:                    # conviene avanzar en x
        err += dy
        x0 += sx
    if e2 <= dx:                    # conviene avanzar en y
        err += dx
        y0 += sy
```

`e2 = 2·err` permite comparar contra `dx` y `dy` usando solo enteros. Ambas condiciones pueden cumplirse en el mismo paso: eso es un avance diagonal.

Ejemplo: recta de (2, 3) a (8, 5)

Fórmulas generales:

```
dx  = |x1 - x0|
dy  = -|y1 - y0|
sx  = +1 si x0 < x1, si no -1
sy  = +1 si y0 < y1, si no -1
err = dx + dy
```

Sustituyendo los valores:

```
dx  = |8 - 2| = 6
dy  = -|5 - 3| = -2
sx  = +1,  sy = +1
err = 6 + (-2) = 4
```

Como `dx > |dy|`, X es el eje dominante: avanzamos en X en cada paso y solo a veces subimos en Y.

Los primeros pasos

En cada paso: pintamos el píxel actual, calculamos `e2 = 2·err` y decidimos si avanzamos en X (si `e2 >= dy`) y/o en Y (si `e2 <= dx`), ajustando `err`. Recuerda que `dx = 6` y `dy = -2`.

Paso 1 — en (2, 3), err = 4. `e2 = 8`:

- ¿Avanzar en X? `e2 >= dy` → 8 >= -2 → sí. x pasa a 3 y `err = 4 + (-2) = 2`.
- ¿Avanzar en Y? `e2 <= dx` → 8 <= 6 → no. y se queda en 3.
- Resultado: (3, 3), err = 2.

Paso 2 — en (3, 3), err = 2. `e2 = 4`:

- ¿Avanzar en X? 4 >= -2 → sí. x pasa a 4 y `err = 2 + (-2) = 0`.
- ¿Avanzar en Y? 4 <= 6 → sí. y pasa a 4 y `err = 0 + 6 = 6`.
- Resultado: (4, 4), err = 6. Avance diagonal.

Paso 3 — en (4, 4), err = 6. `e2 = 12`:

- ¿Avanzar en X? 12 >= -2 → sí. x pasa a 5 y `err = 6 + (-2) = 4`.
- ¿Avanzar en Y? 12 <= 6 → no. y se queda en 4.
- Resultado: (5, 4), err = 4.

Traza completa

Los valores de `err` y `e2` son los que hay al pintar cada píxel.

Paso	(x, y)	err	e2	Avance
1	(2, 3)	4	8	X
2	(3, 3)	2	4	X e Y
3	(4, 4)	6	12	X
4	(5, 4)	4	8	X
5	(6, 4)	2	4	X e Y
6	(7, 5)	6	12	X
7	(8, 5)	—	—	fin

La línea rasterizada

```
y=5  ·  ·  ·  ·  ·  ●  ●
y=4  ·  ·  ●  ●  ●  ·  ·
y=3  ●  ●  ·  ·  ·  ·  ·
     ──────────────────────
x=   2  3  4  5  6  7  8
```

7 píxeles para un `dx` de 6, todo con aritmética entera.

DDA frente a Bresenham

- DDA: incrementos de punto flotante y redondeo por píxel. Simple de entender, pero más lento y acumula error de redondeo.
- Bresenham: solo enteros, sin redondeo. Más rápido en hardware y sin error acumulado; es el que usan las librerías reales.

Polígonos

Un polígono es una figura cerrada formada por una secuencia de líneas.

Tiene dos componentes principales:

- Vértices: los puntos que definen la figura.
- Aristas: los segmentos de línea entre vértices consecutivos.

Polígono con sus vértices y aristas

Dibujar un polígono

- Recibimos un arreglo de puntos: `[(x0, y0), (x1, y1), ..., (xn, yn)]`.
- Para cada par de puntos consecutivos dibujamos una línea (con Bresenham).

- Conectamos el último punto con el primero para cerrar la figura.

Rellenar un polígono

Dibujar el contorno solo enciende el borde. Para pintar el interior hay que decidir qué píxeles quedan dentro de la figura. Existen dos familias clásicas de algoritmos.

Flood fill

Parte de un píxel semilla dentro de la figura y contagia el color de relleno a los vecinos que todavía tienen el color de fondo, hasta chocar con el borde. Es exactamente el balde de pintura de programas como Paint: trabaja sobre los píxeles ya pintados, no sobre la geometría, y necesita una semilla interior.

Cómo funciona:

1. Elegir una semilla interior y recordar el color de fondo (viejo) y el de relleno (nuevo).
2. Pintar el píxel actual del color nuevo.
3. Mirar sus vecinos (4-conectado: arriba, abajo, izquierda, derecha; 8-conectado incluye diagonales).
4. A cada vecino que aún tenga el color viejo, aplicarle lo mismo. El borde detiene la expansión.

```
def flood_fill(x, y, viejo, nuevo):
    if color(x, y) != viejo:
        return
    pila = [(x, y)]
    while pila:
        px, py = pila.pop()
        if color(px, py) != viejo:
```

```
            continue
        pintar(px, py, nuevo)
        pila += [(px+1, py), (px-1, py),
                 (px, py+1), (px, py-1)]
```

Se usa una pila explícita en lugar de recursión para no desbordar la memoria en figuras grandes. Cada píxel se pinta una sola vez.

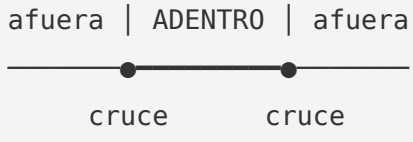
Scanline fill

Rellena la figura fila por fila. Cada fila horizontal es una scanline: para ella se calcula por dónde entra y sale del polígono y se pinta el tramo interior. Trabaja sobre la geometría (vértices y aristas), no sobre píxeles ya pintados, y no necesita semilla.

Cómo funciona:

1. Para cada fila, hallar dónde cruza las aristas del polígono.
2. Ordenar esos cruces por su coordenada x.
3. Pintar el tramo entre el 1.º y 2.º cruce, el 3.º y 4.º, etc.

La razón es la regla par-impar: cada vez que la fila cruza una arista, alterna entre afuera y adentro, por eso se pinta entre pares de cruces.




```
def scanline_fill(poligono):
    for y in range(y_min, y_max):
        cruces = []
        for a, b in aristas(poligono):
            if cruza(a, b, y):
                cruces.append(x_corte(a, b, y))
```

```
        cruces.sort()
        for i in range(0, len(cruces), 2):
            pintar_tramo(cruces[i], cruces[i+1], y)
```

Los vértices y las aristas horizontales requieren cuidado extra para no contar un cruce de más.

Flood fill frente a scanline

- Flood fill: trabaja sobre píxeles y necesita una semilla. Rellena cualquier forma, pero gasta memoria y visita todos los píxeles: costoso en figuras grandes.
- Scanline: trabaja sobre la geometría, sin semilla. Más eficiente y es el estándar en rasterizadores y GPUs, aunque más complejo por los casos de vértices.