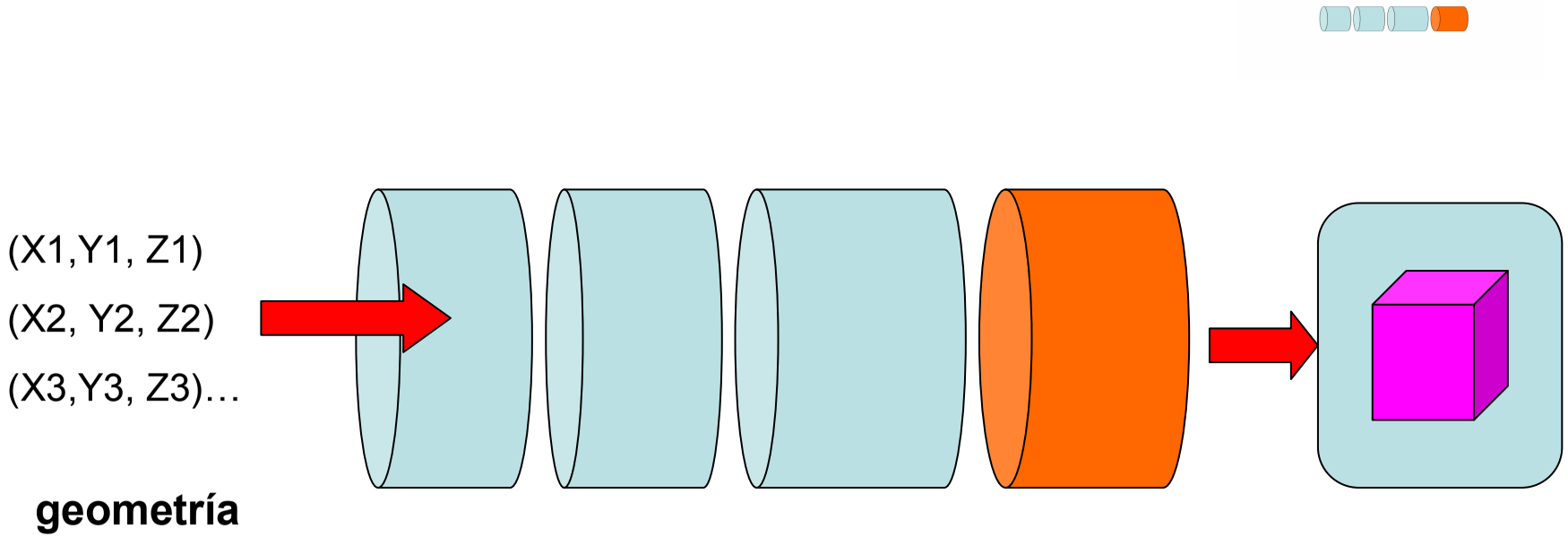
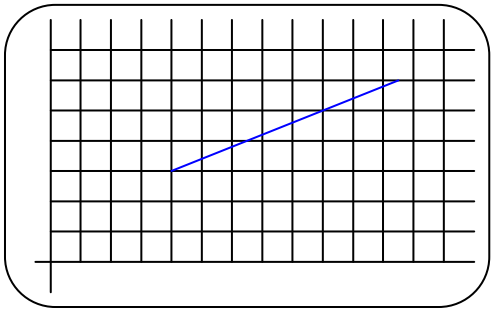
Primitivas gráficas



La pantalla de visualización es 2D. Al final hay que “dibujar” la escena como si tuviéramos como soporte un papel. Utilizando las primitivas de dibujado

Sistema de coordenadas

- Los dispositivos de salida de matriz de puntos (raster), están compuestos por pixeles cada uno de ellos situado en una posición de coordenadas de valores **enteros**. Al sistema de coordenadas se le denomina "**Sistema de Coordenadas de Dispositivo**".

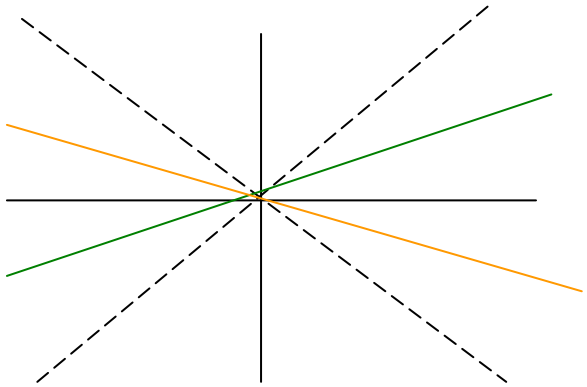


Así una recta no empieza en la posición (3.58,4.56). Las coordenadas son enteras y por tanto puede empezar en (3,4) o en (4,5)

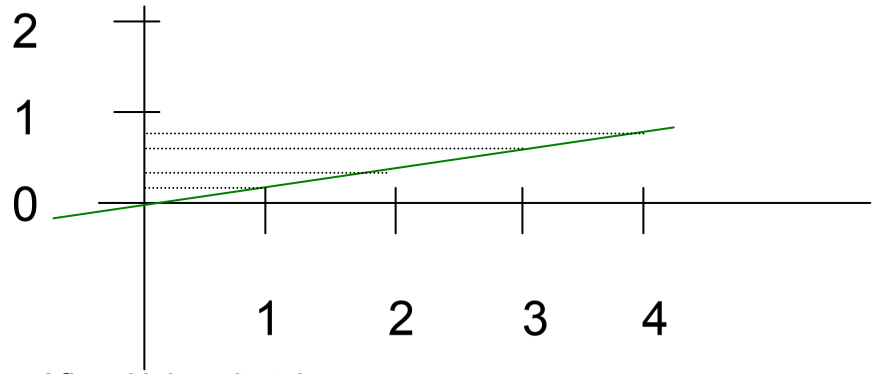
El pixel es la primitiva básica de dibujado. Sus propiedades son la posición y el color + transparencia. Es conceptualmente equivalente al punto

La línea como primitiva

- Supuestos previos:
  - El grosor de la línea será de 1 pixel
  - Distinguimos entre pendientes |m|>1 y |m|<1. Los casos m = 0, m = ∞, m = ±1 son particulares y se tratan aparte.

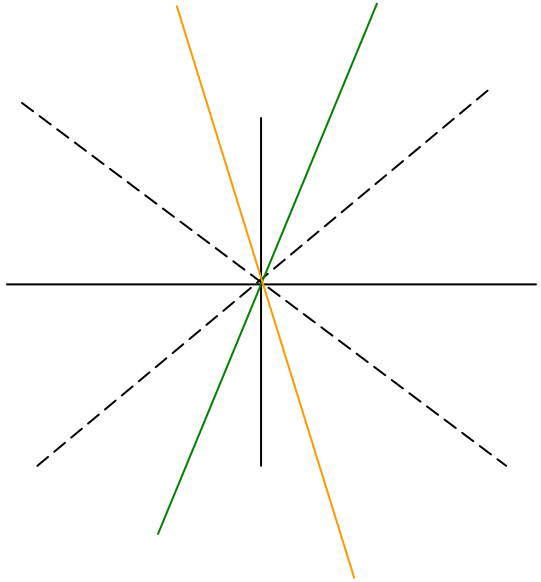


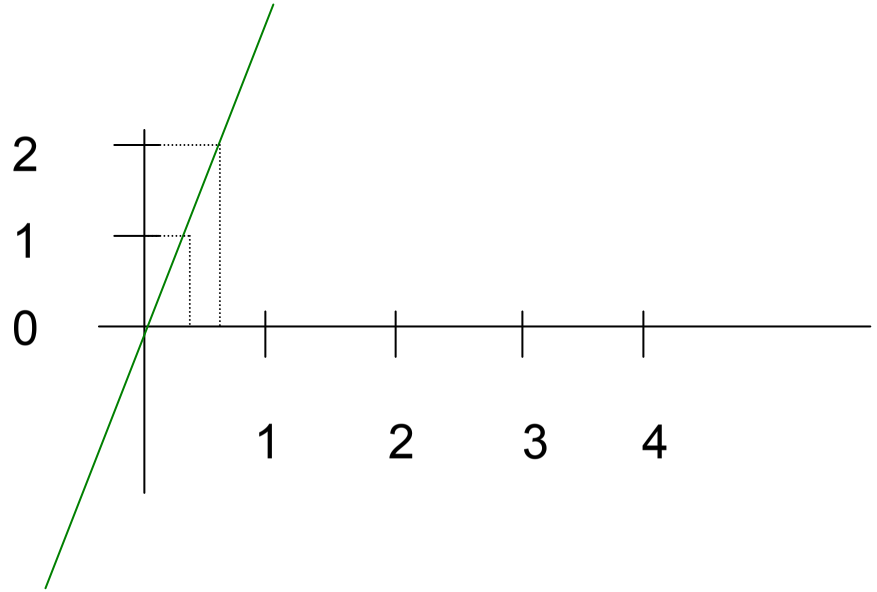
Para **|m<1|** la coordenada que tiene los valores uniformemente distribuídos es la coordenada **x**. Se le llama **coordenada de rastreo**



La línea como primitiva (2)

Para **|m>1|** la coordenada de rastreo es la coordenada **y**.



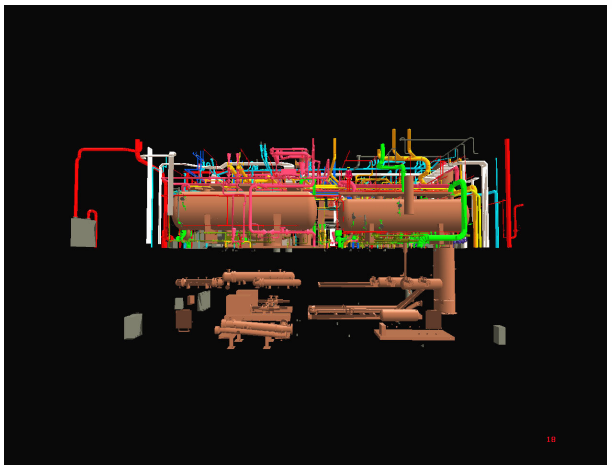


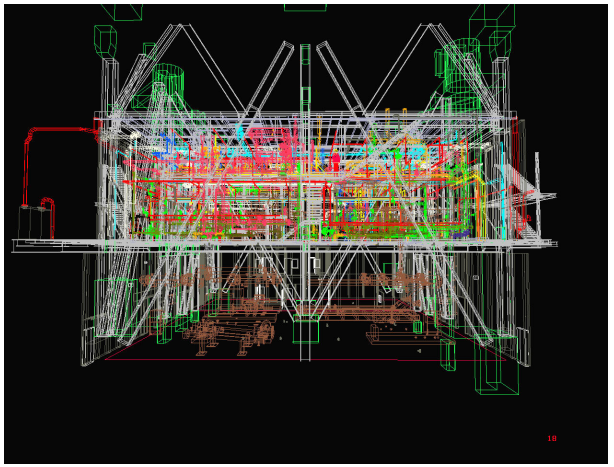
La línea como primitiva (3)

- **Restricciones para dibujar líneas**: Hay que pensar que **no se trata de dibujar una línea**. Una escena puede tener 100.000 polígonos, es decir, sobre **250.000 rectas**. Y deben poder ser dibujados a 24 escenas por segundo

- Vamos a ver tres algoritmos, cada uno de ellos es más eficiente que el anterior:

  - Fuerza Bruta
  - DDA
  - Bresenham o punto medio





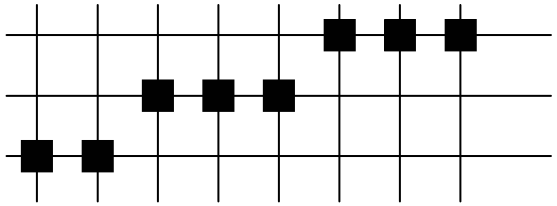
La línea como primitiva (4)

- **Algoritmo de fuerza bruta**
- **Supongamos |m|<1 (pensar como seria con |m|>1)**
- **$y = mx + b$**

X_i	$y_i = mx_i + b$
0	b
1	m+b
2	2m+b
3	3m+b

- **Se manda dibujar el punto (X_i, $\lfloor y_i + 0.5 \rfloor$)**

VENTAJAS:
Fácil de implementar
Simétrico (da igual empezar a dibujar por la x más pequeña que por la más grande)
DESVENTAJAS
La coordenada y se calcula en coma flotante (multiplicación + dos sumas)



La línea como primitiva (5)

- **Algoritmo DDA (Analizador Diferencial Digital)**
- **Supuestos previos:**
- Supongamos |m|<1. Es decir, usamos x como coordenada de barrido
- Las líneas se procesan de izquierda a derecha
- Dado $y_i = mx_i + b$ **y sabiendo que** $x_{i+1} = x_i + 1$

- $y_{i+1} = mx_{i+1} + b = m(x_i + 1) + b = mx_i + b + m = y_i + m$

- Es decir: $y_{i+1} = y_i + m$ **podemos calcular cada y en función del valor de la anterior**
- **Luego pintamos como antes el pixel** $(X_i, \lfloor y_i + 0.5 \rfloor)$

- **VENTAJAS RESPECTO DEL METODO ANTERIOR**
- **Ahorramos una multiplicación**
- **Cuestión: ¿qué expresión deberíamos aplicar para rectas con |m| > 1?**

La línea como primitiva (6)

- **Algoritmo DDA (Analizador Diferencial Digital)**

- **DESVENTAJAS:**
- **El método de dibujado no es simétrico como el anterior.**

  Ya que el cálculo de la coordenada dependiente se basa en el valor de la anterior calculada, no da lo mismo por donde se empiece a calcular los puntos de la recta.

```
#include <stdlib.h>
#include <math.h>

inline int round (const float a)  { return int (a + 0.5); }

void lineDDA (int x0, int y0, int xEnd, int yEnd)
{
   int dx = xEnd - x0,  dy = yEnd - y0,  steps,  k;
   float xIncrement, yIncrement, x = x0, y = y0;

   if (fabs (dx) > fabs (dy))
      steps = fabs (dx);
   else
      steps = fabs (dy);
   xIncrement = float (dx) / float (steps);
   yIncrement = float (dy) / float (steps);

   setPixel (round (x), round (y));
   for (k = 0; k < steps; k++) {
      x += xIncrement;
      y += yIncrement;
      setPixel (round (x), round (y));
   }
}
```

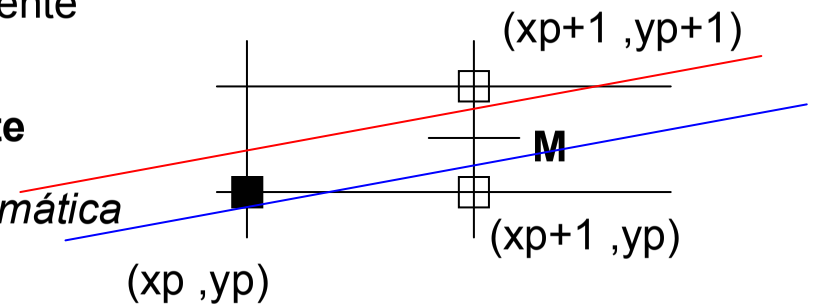
(*Hearn, Baker*)

La línea como primitiva (7)

- **Algoritmo de Bresenham (o del punto medio)**
- **Supuestos previos:**
  - Suponemos **0 < m <1**
  - Grosor de la línea 1 pixel. Elijo un trazado de izquierda a derecha ($x_0 < x_1$)
- **¿Cómo funciona?**
  - En cada momento, tras dibujar el pixel situado en (x_p ,y_p), tenemos dos posibilidades, ya que la recta es creciente

    (x_p+1 ,y_p) Elegimos el pixel **Este**

    (x_p+1 ,y_p+1) Elegimos el pixel **Noreste**



El algoritmo de Bresenham

elige el pixel que más cerca (distancia euclídea) está de la verdadera línea

- **Si M está por encima de la recta matemática, elijo el pixel E**
- **Si M está por debajo de la recta matemática, elijo el pixel NE**

La línea como primitiva (8)

Los casos particulares m=0, m=1, m=∞, son asumidos por el algoritmo aunque es muy ineficiente y en la practica se diseñan algoritmos aparte.

Como se ve, se utiliza únicamente aritmética de enteros para el cálculo del parámetro **d**

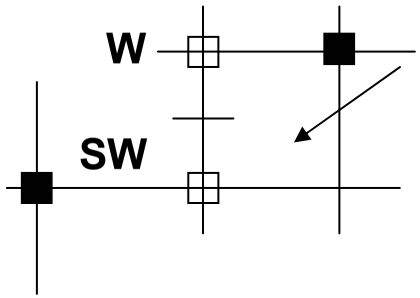
(*Foley, Van Dam*)

```
void MidpointLine (int x0, int y0, int x1, int y1, int value)
{
   int dx = x1 - x0;
   int dy = y1 - y0;
   int d = 2 * dy - dx;              /* Initial value of d */
   int incrE = 2 * dy;               /* Increment used for move to E */
   int incrNE = 2 * (dy - dx);       /* Increment used for move to NE */
   int x = x0;
   int y = y0;
   WritePixel (x, y, value);         /* The start pixel */

   while (x < x1) {
      if (d <= 0) {                  /* Choose E */
         d += incrE;
         x++;
      } else {                       /* Choose NE */
         d += incrNE;
         x++;
         y++;
      }
      WritePixel (x, y, value);      /* The selected pixel closest to the line */
   } /* while */

} /* MidpointLine */
```

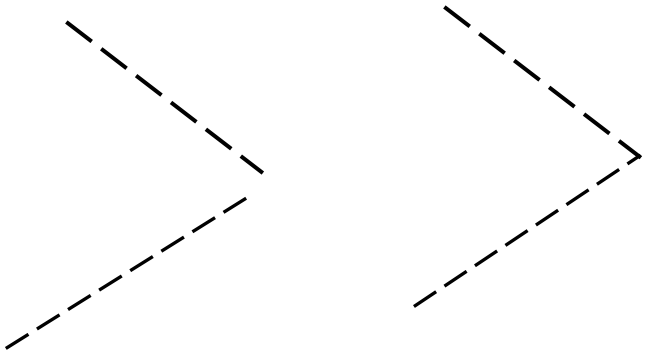
Fig. 3.8 The midpoint line scan-conversion algorithm.

La línea como primitiva (9)

- **Otras consideraciones acerca de Bresenham**
- No es un algoritmo simétrico. Hay que programar un algoritmo diferente para dibujar de derecha a izquierda



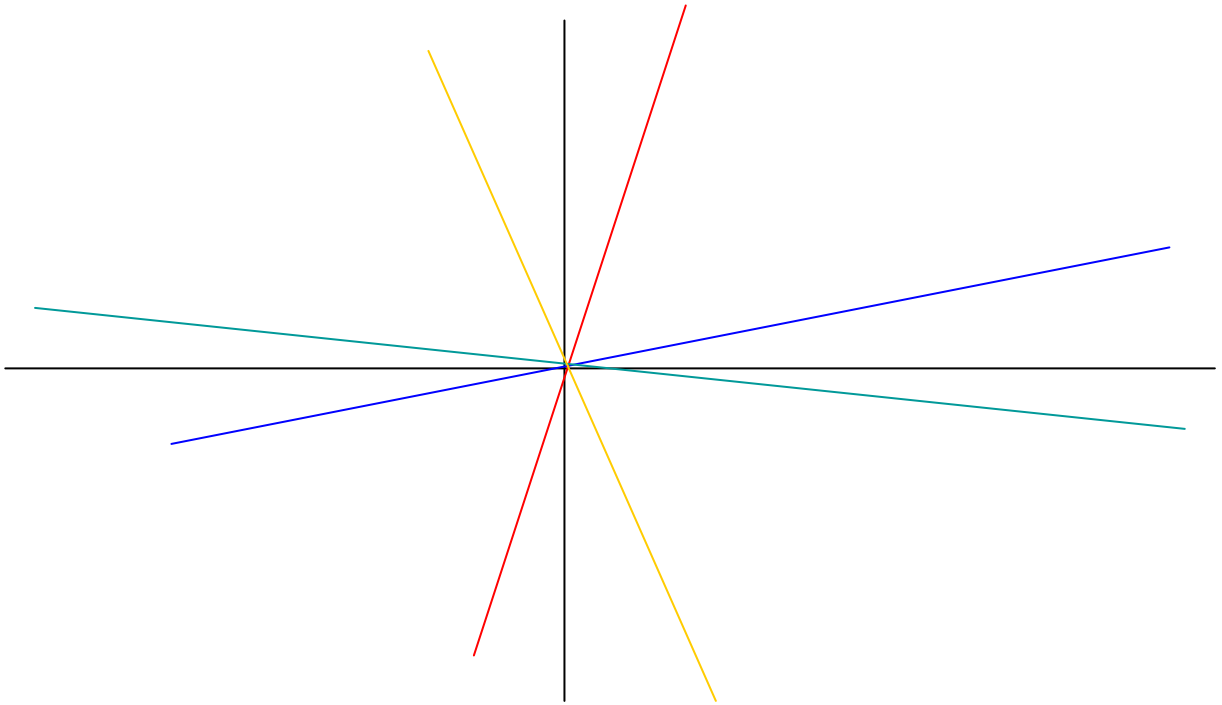
Si d=0, contrariamente al algoritmo visto, hay que elegir, no el pixel de al lado, sino el de abajo



La generación de rectas en un único sentido muchas veces no es posible

La línea como primitiva (10)

- Utilización de diferentes algoritmos para diferentes pendientes
- 0 < m <1 (elección entre E y NE)
- 1 < m < ∞ (elección entre N y NE)
- -1 < m < 0 (elección entre E y SE)
- -∞ < m <-1 (elección entre S y SE)



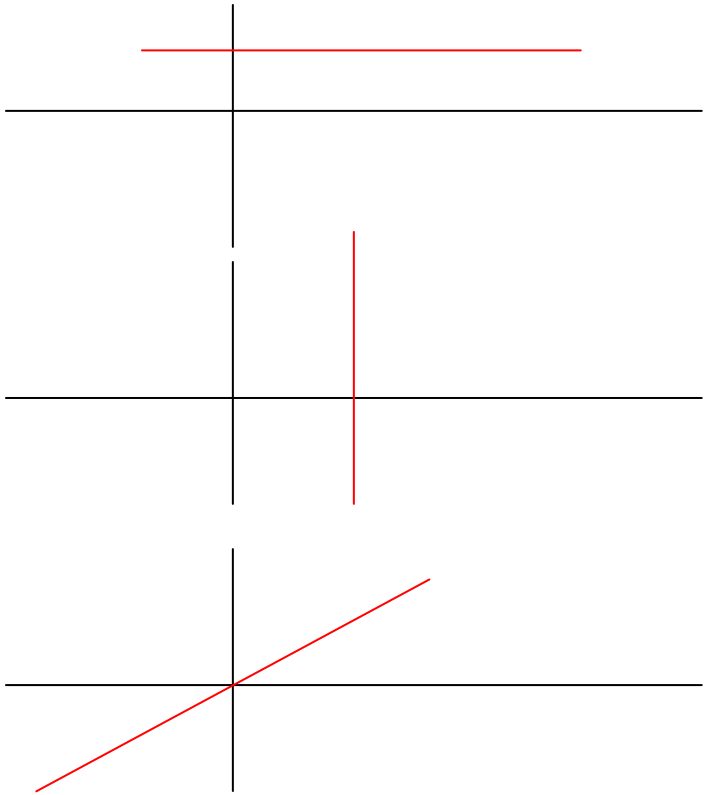
La línea como primitiva (11)

- **Casos especiales (Bresenham los acepta, pero es ineficiente)**

- **m = 0** **for (xi = x0; xi < x1; xi++)**
- **writepixel(xi,y)**

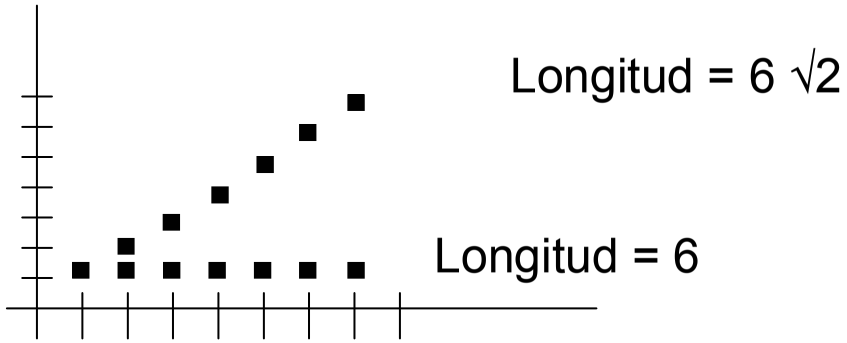
- **m = ∞** **for (yi = y0; yi < y1; yi++)**
- **writepixel(x,yi)**

- **m = 1** **for (xi = x0; xi < x1; xi++)**
- **writepixel(xi,xi)**



La línea como primitiva (12)

- Hay variaciones en la intensidad de la línea dibujada en función de la pendiente de la misma (**I = I (m)**). Esto es consecuencia del uso de un único pixel para la representación de la línea.



Solución: Dar escalas de grises en función de la m

Utilizar un grosor mayor que uno y usar técnicas de antialiasing

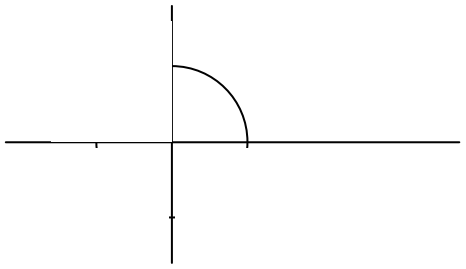
Circunferencia como primitiva

- **Consideración general:** Nos fijaremos en las circunferencias centradas en el origen de coordenadas (x_c ,y_c). Para hacer una circunferencia centrada en otro punto, aplicaremos un desplazamiento a esta primitiva $x = x' + x_c$, $y = y' + y_c$

- **Algoritmo de Fuerza Bruta**
- **Ecuación de circunferencia en el origen: $y = \pm \sqrt{(R^2 - x^2)}$ para R >=1**
- **Si consideramos como coordenada de rastreo arbitrariamente la x**

- **x** **$y = \sqrt{(R^2 - x^2)}$**
- **0** **R**
- **1** **$\lfloor \sqrt{(R^2 - 1)} \rfloor$** (pintando hacia el interior)
- **...**

Esto pintaría un cuarto de circunferencia. Por simetría se pinta el resto



Circunferencia como primitiva (2)

Desventajas

Conforme la tangente tiende a infinito, los valores de **y** se van separando. Esto podría solucionarse cambiando la coord. de barrido cuando la tangente sea >1

El cálculo es en punto flotante

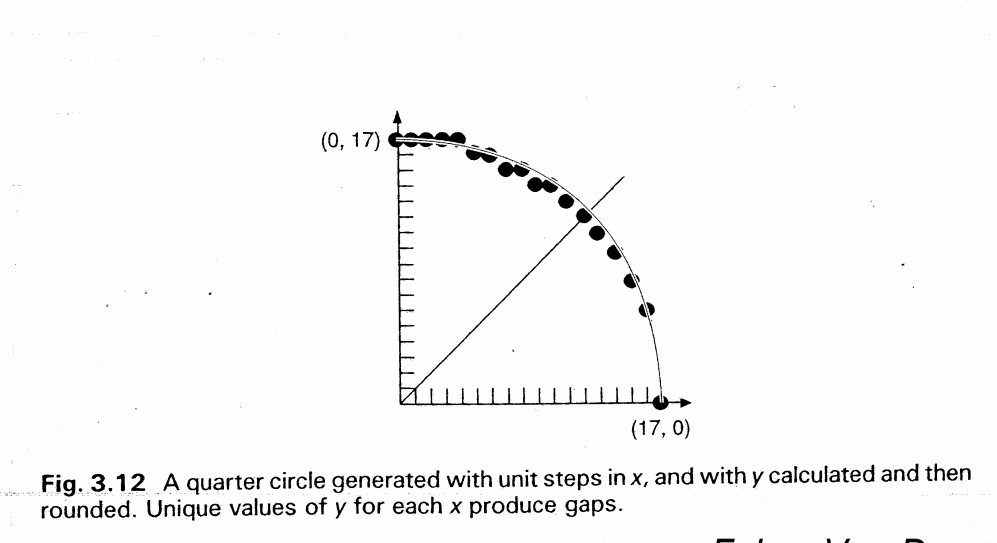


Fig. 3.12 A quarter circle generated with unit steps in *x*, and with *y* calculated and then rounded. Unique values of *y* for each *x* produce gaps.

Foley, Van Dam

Circunferencia como primitiva (3)

Fuerza Bruta en coord. Polares

Para evitar la separación no homogénea de los puntos, podemos usar coordenadas polares

$x = R \cos \theta + x_c \qquad 0 \leq \theta \leq \pi / 2$

$y = R \sin \theta + y_c$

- Dándole a θ un paso constante, obtenemos puntos de igual separación en la circunferencia.
- **Ventaja**: Puede usarse una **simetría de octante**
- **Desventaja**: Cálculos en punto flotante

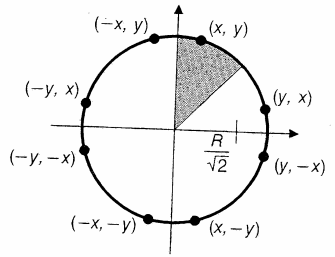
Circunferencia como primitiva (4)

- **Algoritmo de Bresenham o punto medio para circunferencias**

- Igual que el anterior, aprovecha la simetría de octante. Así sólo calcula 1/8 del total de los puntos. Para ello se usa el procedimi ***CirclePoints***

Procedimiento para dibujar la circunferencia por simetría

```
procedure CirclePoints(x, y, value:integer);
begin
   SetPixel(x, y, value);
   SetPixel(y, x, value);
   SetPixel(y, -x, value);
   SetPixel(x, -y, value);
   SetPixel(-x, -y, value);
   SetPixel(-y, -x, value);
   SetPixel(-y, x, value);
   SetPixel(-x, y, value);
end;
```



(Foley, Van Dam)

Supuestos de partida:

Circunferencia centrada en (0, 0)

Dibujamos un octante, es decir desde x = 0 hasta x = y = R / √2 siendo la x la coordenada de rastreo (podría ser la y tambien)

Circunferencia como primitiva (5)

- Comenzamos en el pixel (0,R). Esto tiene importancia solo para la expresión de la **d** inicial
- Dado un pixel coloreado de la circunferencia, tenemos dos opciones, elegir el pixel Este **E** o el Sureste **SE**

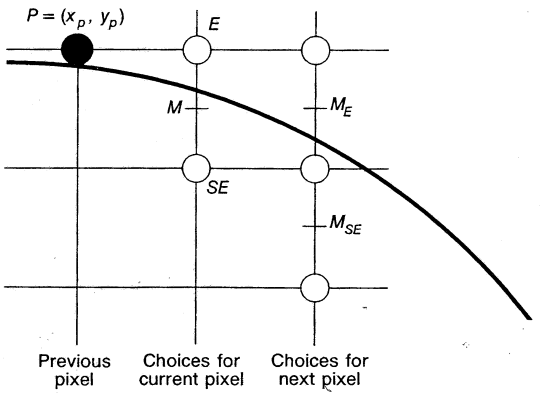


Fig. 3.14 The pixel grid for the midpoint circle algorithm showing *M* and the pixels *E* and *SE* to choose between.

(Foley, Van Dam)

Circunferencia como primitiva (6)

Versión 1 (poco eficiente)

La variable de decisión **d** inicialmente tiene un valor no entero por lo que las operaciones son en coma flotante

El cálculo de los incrementos de E y SE son una función de punto, ya que dependen del valor del punto anterior

Algoritmo de Bresenham o del Punto medio para Circunferencias

```
procedure MidpointCircle(radio, value: integer);
{Asumimos que el centro esta en el (0,0)}
var
x, y : integer;  d: real;
begin
x:=0;
y:=radio;
d:= 5/4 - radio;
CirclePoints(x, y, value);

   while y > x do
      Begin
         if d < 0 then
             begin
             d:= d + 2 * x + 3;
             x:= x+1;
             end;
         else
             begin
             d:= d +2 *(x - y) +5;
             x := x+1;
             y:= y - 1;
            end;
      CirclePoints(x, y, value)
      end {while}
end;
```

(Hearn, Baker)

Circunferencia como primitiva (7)

- Versión 2
- Convertimos el valor inicial de **d** en un entero **h.** Esto no altera la expresión de los incrementos
- h = d -1/4

Al ser **h** un entero, usa ahora aritmética de enteros. Sin embargo los cálculos de los incrementos, aun siguen siendo funciones de punto.

```
procedure MidpointCircle2(radio, value: integer);
{Asume el centro en (0,0). Usa aritmética de enteros)
var
   x, y, d: integer;
begin
   x:=0;
   y:= radio;
   d:= 1 - radio;
   CirclePoints(x, y, value);
   while y > x do
     begin
     if d < 0 then {select E}
        begin
        D: = d + 2*x +3;
        X:= x+1;
        end
      else
        begin   {Selecciona SE}
        d:= d+2*(x-y) + 5;
        x:= x+1;
        y:= y-1;
        end;
     CirclePoints(x, y, value);
    end   {while}
end;
```

Circunferencia como primitiva (8)

Versión 3 (definitiva)

Los incrementos se calculan en función de los valores anteriores. Por lo tanto aquí si que hay que calcular los valores iniciales

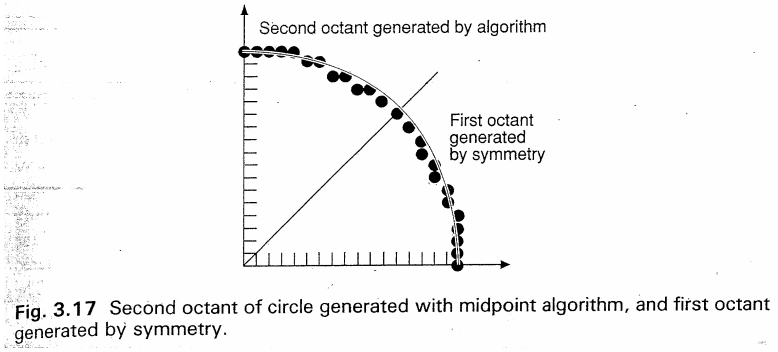


Fig. 3.17 Second octant of circle generated with midpoint algorithm, and first octant generated by symmetry.

```
void MidpointCircle (int radius, int value)
/* This procedure uses second-order partial differences to compute increments */
/* in the decision variable. Assumes center of circle is at origin */
{
    int x = 0;
    int y = radius;
    int d = 1 − radius;
    int deltaE = 3;
    int deltaSE = −2 * radius + 5;
    CirclePoints (x, y, value);

    while (y > x) {
        if (d < 0) {                /* Select E */
            d += deltaE;
            deltaE += 2;
            deltaSE += 2;
        } else {
            d += deltaSE;           /* Select SE */
            deltaE += 2;
            deltaSE += 4;
            y−−;
        }
        x++;
        CirclePoints (x, y, value);
    }  /* while */
}  /* MidpointCircle */
```

. 3.18 Midpoint circle scan-conversion algorithm using second-order differences.

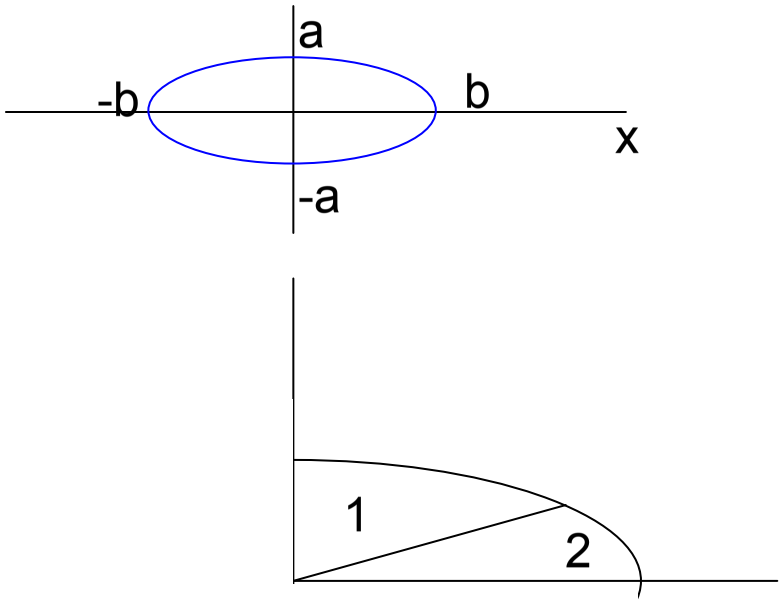
(Foley, Van Dam)

Elipse como primitiva

- **Algoritmo del punto medio para elipses con eje mayor en abcisas**

  Consideración general
- Suponemos que la elipse está centrada en el (0,0) y con el eje mayor en el eje de abcisas.

$$F(x,y) = b^2 x^2 + a^2 y^2 - a^2 b^2 = 0$$



- Igual que en la circunf. Comenzamos a dibujar por el pixel de posición (0,a)

La simetría de esta figura es claramente de tipo **cuadrante**. Sin embargo, para dibujar el cuadrante debemos dibujar el proceso en

dos partes

Elipse como primitiva (2)

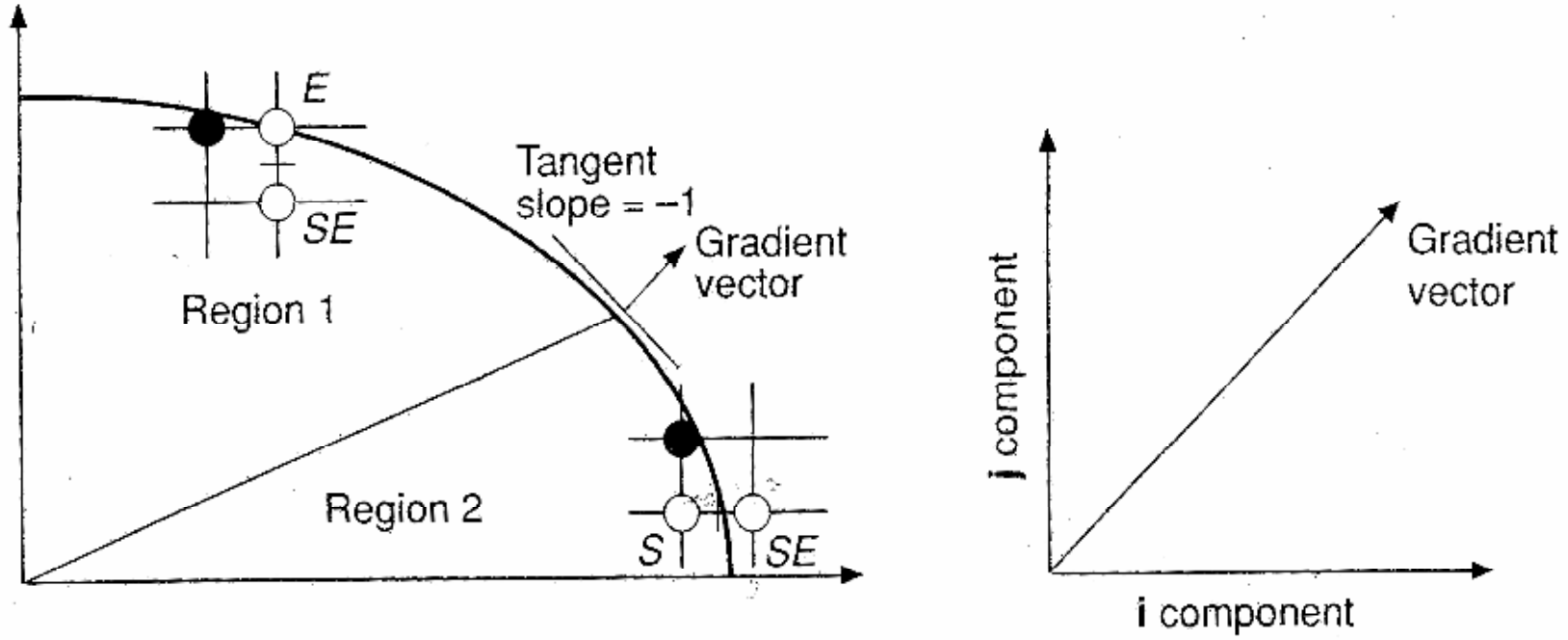
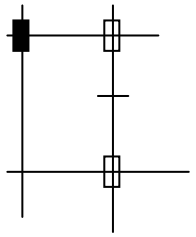


Fig. 3.20 Two regions of the ellipse defined by the 45° tangent.

(Foley, Van Dam)

Elipse como primitiva (3)

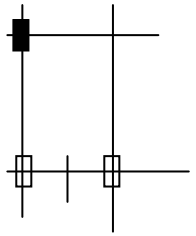
En la región 1 la elección de pixel es:



E

SE

En la región 2 la elección de pixel es:



S **SE**

¿Cuál es el criterio de cambio de un modo de pintar a otro?

El **gradiente** de la curva. $\nabla F = \partial F / \partial x\ \mathbf{i} + \partial F / \partial y\ \mathbf{j} = 2b^2 x\ \mathbf{i} + 2a^2 y\ \mathbf{j}$

Cuando **la pendiente del gradiente sea 1**se debe producir el cambio de orientación del dibujo.

$a^2 y / b^2 x = 1$ Es decir, cuando las dos componentes del gradiente sean de igual magnitud

Elipse como primitiva (4)

- **A tener en cuenta**
- En cada iteración del algoritmo debo analizar si he pasado de una región a otra

- En el momento de la transición, la variable de decisión **d**, debe de ser inicializada a partir de los valores del último punto de la región 1.
- Así, si el último punto es (x_p, y_p), entonces $d'_0 = F(x_p+1/2, y_p-1)$

- El algoritmo para cuando y =0

- Los incrementos de **d** pueden ser calculados en diferencias de segundo orden como en la circunferencia. Se puede incluso usar esta información para el calculo del valor del gradiente también (Da Silva 1989)

Elipse como primitiva (5)

Condición $a^2y = b^2x$ en el punto de evaluación

```
void MidpointEllipse (int a, int b, int value)
/* Assumes center of ellipse is at the origin. Note that overflow may occur */
/* for 16-bit integers because of the squares. */
{
    double d2;

    int x = 0;
    int y = b;
    double d1 = b^2 - (a^2 b) + (0.25 a^2);
    EllipsePoints (x, y, value);                    /* The 4-way symmetrical WritePixel */

    /* Test gradient if still in region 1 */
    while ( a^2 (y - 0.5) > b^2 (x + 1) ) {         /* Region 1 */
        if (d1 < 0)                                 /* Select E */
            d1 += b^2 (2x + 3);
        else {                                      /* Select SE */
            d1 += b^2 (2x + 3) + a^2 (-2y + 2);
            y--;
        }
        x++;
        EllipsePoints (x, y, value);
    } /* Region 1 */

    d2 = b^2 (x + 0.5)^2 + a^2 (y - 1)^2 - a^2 b^2;
    while (y > 0) {                                 /* Region 2 */
        if (d2 < 0) {                               /* Select SE */
            d2 += b^2 (2x + 2) + a^2 (-2y + 3);
            x++;
        } else
            d2 += a^2 (-2y + 3);                    /* Select S */
        y--;
        EllipsePoints (x, y, value);
    } /* Region 2 */
} /* MidpointEllipse */
```

Fig. 3.21 Pseudocode for midpoint ellipse scan-conversion algorithm.

(Foley, Van Dam)

Problemas con el dibujado

- Si elegimos trazar con un único pixel la primitiva, hay un efecto escalera (jagging) en las primitivas, debido a que pintamos en un dispositivo consistente en una matriz de puntos (raster)
- Los algoritmos de punto medio, pueden generar figuras no exactas en cuanto a sus dimensiones

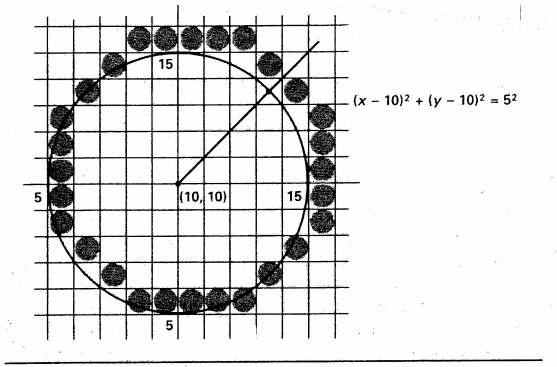


Figura 3-33
Trayectoria circular y trazo del algoritmo de punto medio para circunferencias aplicado a una circunferencia con radio 5 en las coordenadas de pantalla.

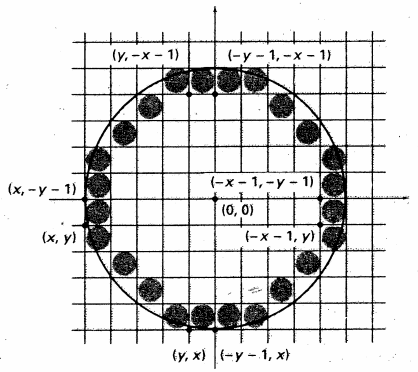


Figura 3-34
Modificación del trazo de la circunferencia de la figura 3-33 para conservar el diámetro específico de 10.

(Hearn, Baker)

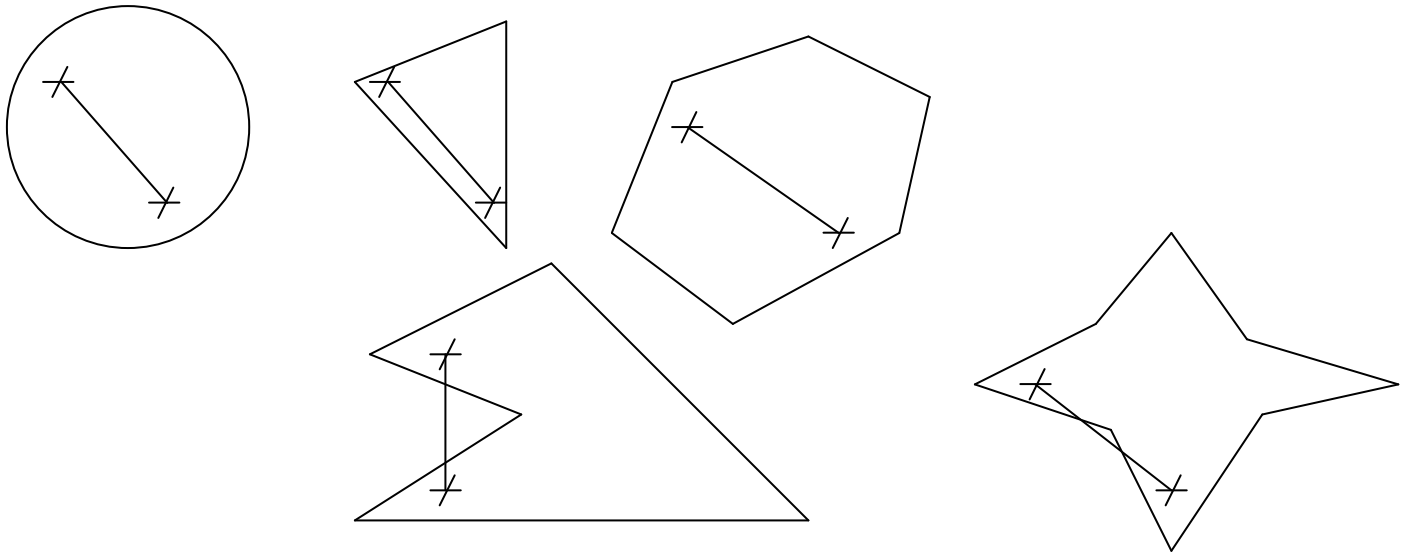
Rellenado de polígonos

Problema: Rellenar de un color el interior de una figura cerrada

Es un problema general, ya que los modelos suelen estar formados por mallas de polígonos convexos (en general triángulos o cuadriláteros)

Def. Polígono Convexo: Es aquel en el que la línea recta que une dos puntos cualesquiera del interior del polígono permanece toda ella en el interior del polígono.

Polígono Cóncavo: La definición complementaria de la anterior

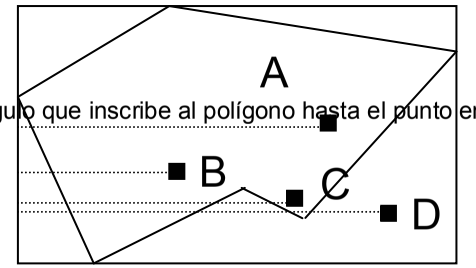


Rellenado de polígonos

- **Alg. Fuerza Bruta**
- Sea un polígono P y sea el rectángulo (xo,y0), (x1, y1) el mínimo rectángulo en el que está inscrito el polígono.

```
for(y=y0; y<= y1; y++)
   for(x=x0; x<=x1; x++)
       if((x,y) está dentro del polígono)
             Pintar(x,y);
```

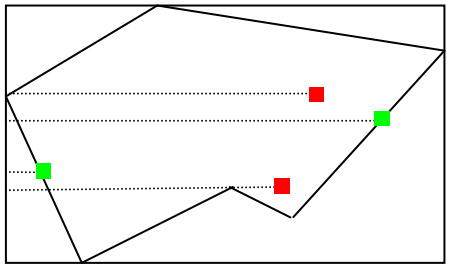
- ¿Cómo saber si un pixel es interior?
- Contando las intersecciones de las aristas de la recta horizontal que va desde el lado izquierdo del rectángulo que inscribe al polígono hasta el punto en cuestión.



Rellenado de polígonos

- **Inconvenientes:**
- El análisis es para cada pixel interior, con lo que es muy ineficiente
- El criterio de pertenencia al interior es muy simple y se desbarata en dos casos:
  - Cuando la línea de barrido intersecta con un vértice del polígono
  - Cuando el pixel a pintar está exactamente sobre una arista del polígono

Con un criterio como este (pinto solo si el número de intersecciones es impar), pintaríamos sólo uno de los dos píxeles del mismo color



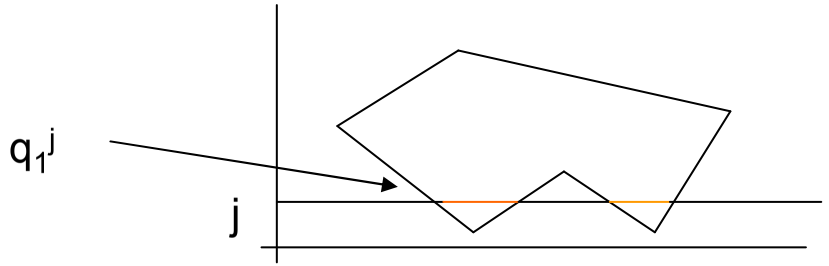
Rellenado por línea de rastreo

- **Esquema del algoritmo**
- 1. Para cada línea de rastreo (en el eje y) determinar cuáles son las intersecciones con las aristas del polígono y ordenarlas de menor a mayor
- 2. Aplicar el dibujado por tramos entre estos puntos de intersección utilizando un criterio para determinar qué tramos se pintan y cuáles no.

- **Mejoras respecto la fuerza bruta**
- - No se consulta si cada pixel pertenece o no al interior de la figura. Se aprovecha la **coherencia de línea**, es decir, se **pinta por tramos.**
- - Definimos un **criterio de paridad** con los casos particulares bien definidos.

Rellenado por línea de rastreo (1)

- Acerca del punto 2 del algoritmo general. Coherencia de línea
- Para cada línea de rastreo **j** se calculan las intersecciones de la línea de rastreo con las aristas del polígono $\{q_1^j, q_2^j q_3^j \ldots.\}$
- Se ordenan los puntos de intersección en orden creciente de coordenada x
- Estos puntos dividen a la línea de rastreo en tramos, que se pintarán o no



Rellenado por línea de rastreo (2)

- **Regla de paridad**: Decide el tramo que se pinta
  - Regla general: Inicialmente la paridad es par.

Cada intersección con una arista, cambia la paridad

Se pintan los tramos de paridad IMPAR

- Casos particulares

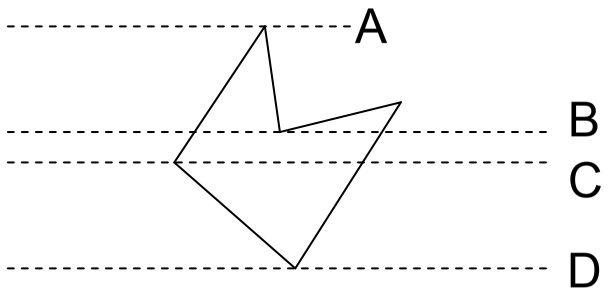
Efecto de los vértices en la regla de paridad. Si la línea atraviesa un vértice, contaremos cambio de paridad por cada arista de la cual el vértice es un **y mínimo**

A no cambia la paridad

B no cambia la paridad

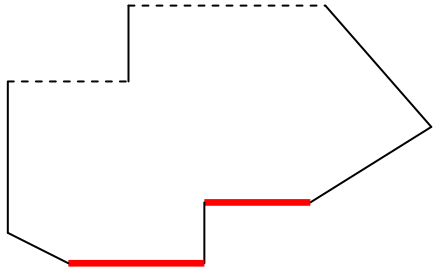
C cambia la paridad

D no cambia la paridad

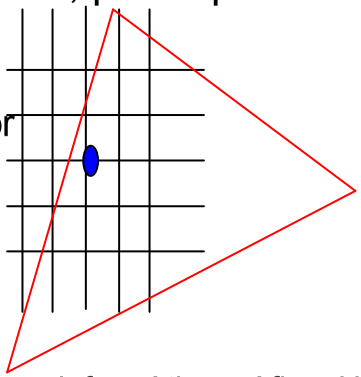


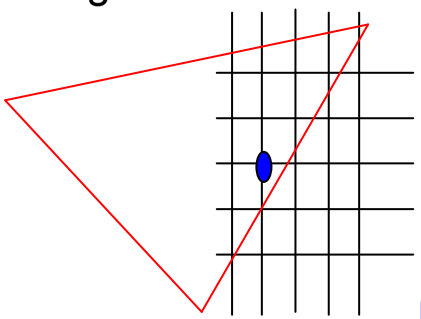
Rellenado por línea de rastreo (3)

- *Aristas horizontales*: No contribuyen a la regla anterior ya que no tienen ni y máximo ni y mínimo. Con este criterio , dado un polígono, se dibujarán las aristas horizontales inferiores pero no las superiores



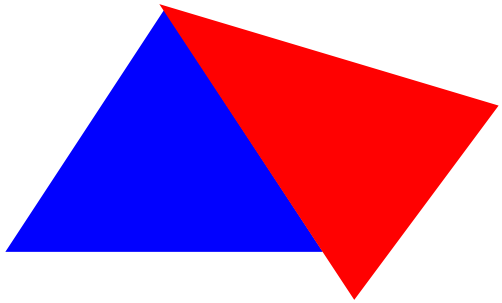
Dibujado de los pixeles interiores únicamente. Si el polígono tiene dibujadas fronteras, entonces se colorea el interior de las fronteras. Sino, se colorea entre los enteros inmediatos a éstas, por la parte interior de la figura.

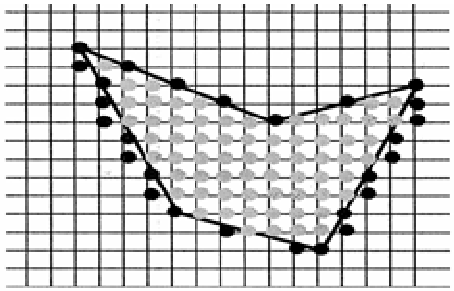




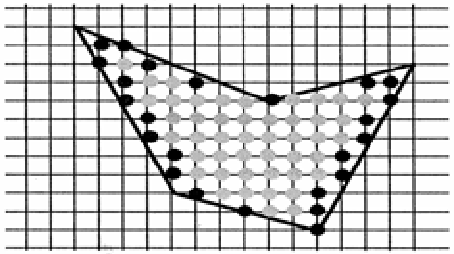
Rellenado por línea de rastreo (4)

- El elegir los píxeles interiores evita que los colores se mezclen en las fronteras de dos polígonos que la comparten



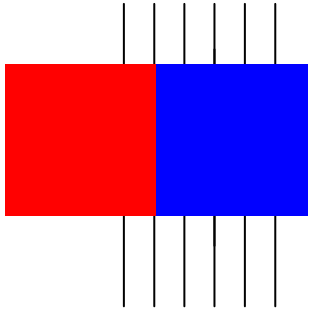


Some pixels lie outside the polygon.



Edge extrema only choosing inside edges. Care with abutting polygons.

En caso de que la intersección de la línea de barrido y la arista sea en un valor entero, se pinta el pixel si la arista es izquierda y no se pinta si es derecha. Esto evita que aristas compartidas se pinten dos veces



Rellenado por línea de rastreo (5)

Debido a sus restricciones, para cuñas, el criterio de paridad ofrece resultados solo aceptables

No se pinta porque la intersección es en un valor entero y es arista derecha

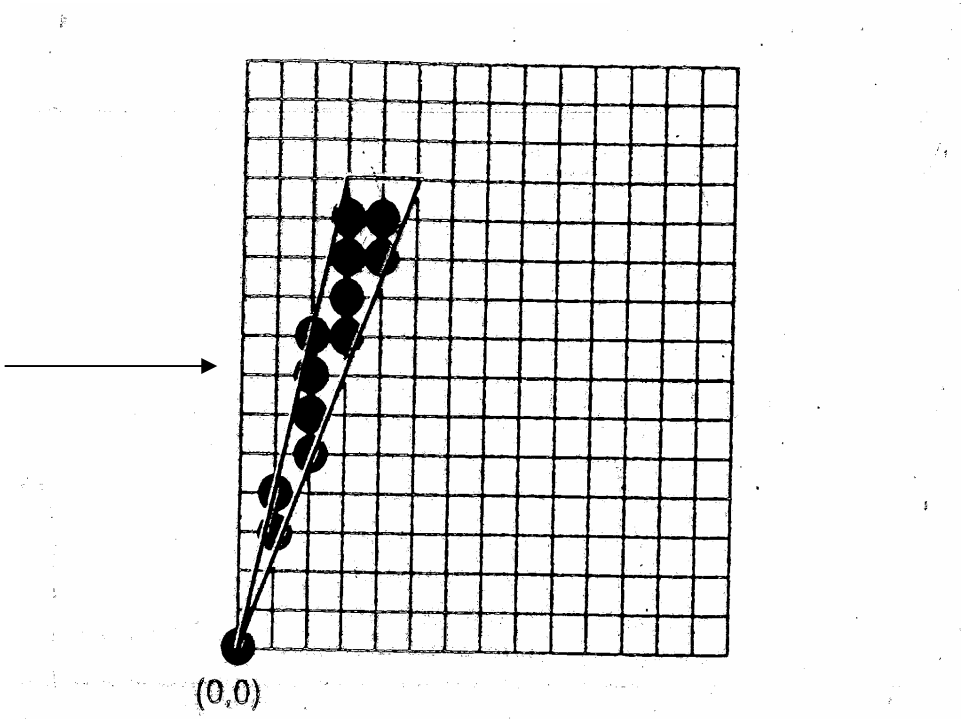


Fig. 3.25 Scan converting slivers of polygons.

(Foley, Van Dam)

Rellenado por línea de rastreo (6)

Acerca del punto 1 del alg. general

Podemos diseñar algoritmos para **encontrar los puntos interiores x de intersección de la línea de rastreo con una arista**. Para ello estos algoritmos deben diseñarse especialmente para aristas izquierdas y otros para derechas. Los casos de aristas horizontales son cubiertos por el criterio de paridad. Los de aristas verticales son casos especiales donde la intersección con la arista es siempre el mismo punto de coordenada x.

El algoritmo se basa en que

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

```
procedure LeftEdgeScan (xmin, ymin, xmax, ymax, value : integer);
var x, y, numerator, denominator, increment : integer;
begin
    x := xmin;
    numerator := xmax - xmin;
    denominator := ymax - ymin;
    increment := denominator;
    for y := ymin to ymax do
      begin
        WritePixel (x, y, value);
        increment := increment + numerator;
        if increment > denominator then
          {Overflow, so round up to next pixel and decrement the increment.}
          begin
            x := x + 1;
            increment := increment - denominator
          end;
      end
end;
```

Figure 3.26 Scan converting left edge of a polygon.

(Foley, Van Dam)

Rellenado por línea de rastreo (7)

- Ejemplo de traza del algoritmo LeftScanEdge()
- Supongamos una pendiente de una arista m = 5/2 donde x0,y0 = 0,1 x1,y1 = 6,16

Línea rastreo	numerator	denominator	increment	pixel	
1	6		15	15	0,1
2				21	1,1
		6			
3		12	1,3		
4		18	2,4		
		3			
5		9	2,5		
6		15	2,6		
etc					

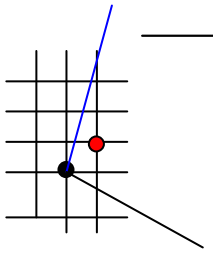
En este caso la intersección es un entero

Rellenado por línea de rastreo (8)

Dos diferencias clave entre el algoritmo para aristas izquierdas y el de aristas derechas

Necesariamente si es una arista izquierda, como debo elegir un pixel interno a la arista, el pixel inmediatamente superior al comienzo (que es entero) debe ser x+1

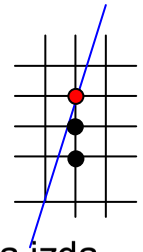
Arista izda.



```
procedure LeftEdgeScan (xmin, ymin, xmax, ymax, value : integer);
var x, y, numerator, denominator, increment : integer;
begin
    x := xmin;
    numerator := xmax - xmin;
    denominator := ymax - ymin;
    increment := denominator;
    for y := ymin to ymax do
      begin
        WritePixel (x, y, value);
        increment := increment + numerator;
        if increment > denominator then
          {Overflow, so round up to next pixel and decrement the increment.}
          begin
            x := x + 1;
            increment := increment - denominator
          end;
      end
end;
```

Figure 3.26 Scan converting left edge of a polygon.

Es un menor estricto porque cuando increment = denominator estoy en una intersección de valor entero, y debe pintarse



Arista izda.

Rellenado por línea de rastreo (9)

- **Algoritmo explícito**
- 1. Preparamos una estructura de datos de tipo “lista de listas” llamada “Lista de aristas (ET)”. En ella cada elemento corresponde a un valor y de la coordenada de rastreo y contiene la lista de las aristas que comienzan en dicho valor de y. Para cada arista se guardan tres datos:
    - El valor de la coordenada y máximo (donde acaba la arista)
    - La coordenada x de intersección con la línea de rastreo
    - El valor 1/m de la arista
- *Esta lista de arista esta ordenada por valor creciente de x*

- 2. Una vez creada la ET el proceso sigue de esta manera:

Rellenado por línea de rastreo (10)

- a) Sea ***y*** el valor de la coordenada y más pequeño que tiene una entrada en ET
- B) Crea la lista de aristas activa (AET) e inicialízala a vacía
- C) Repetir hasta que AET y ET estén vacías:
  - C1) Mover de ET a AET aquellas aristas cuya ***y*** mínima sea ***y*** y ordenar el AET sobre los valores de ***x***
  - C2) Rellenar los tramos de arista a arista SIGUIENDO LAS REGLAS DE PARIDAD VISTAS
  - C3) Eliminar de AET aquellas entradas para las cuales ***y*** = ***y*** maxima
  - C4) Incrementar ***y*** en uno (pasamos a la siguiente línea de rastreo)
  - C5) Actualizar ***x*** para la nueva línea

- El apartado C5 puede hacerse calculando la nueva ***x*** en el momento o bien, si hemos usado los algoritmos tipo *LeftscanEdge* previamente para cada arista y hemos almacenado sus valores en una lista, podemos reemplazar esos valores directamente.

Rellenado por línea de rastreo (11)

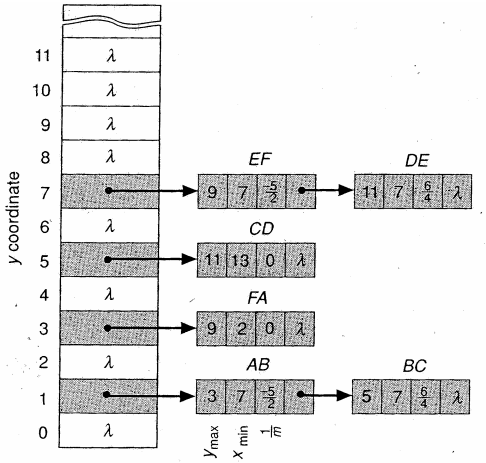


Fig. 3.27 Bucket-sorted edge table for polygon of Fig. 3.22.

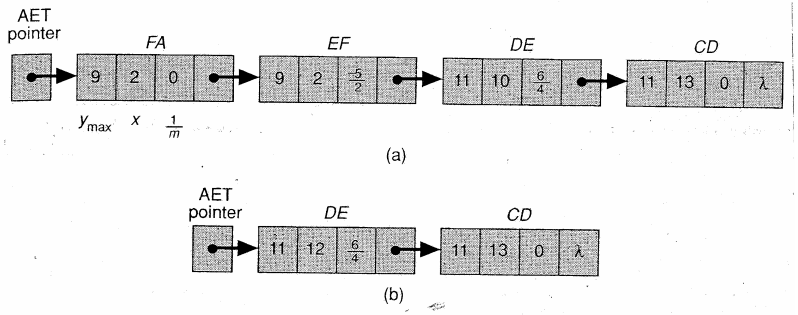


Fig. 3.28 Active-edge table for polygon of Fig. 3.22. (a) Scan line 9. (b) Scan line 10. (Note *DE*'s *x* coordinate in (b) has been rounded up for that left edge.)

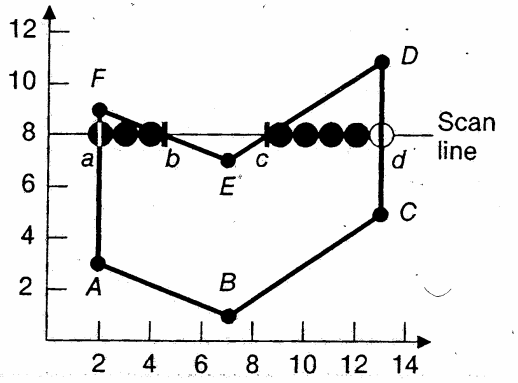


Fig. 3.22 Polygon and scan line 8.

C1) Mover de ET a AET aquellas aristas cuya ***y*** mínima sea ***y*** y ordenar el AET sobre los valores de ***x***
C2) Rellenar los tramos de arista a arista SIGUIENDO LAS REGLAS DE PARIDAD VISTAS
C3) Eliminar de AET aquellas entradas para las cuales ***y*** = ***y*** maxima
C4) Incrementar ***y*** en uno (pasamos a la siguiente línea de rastreo)
C5) Actualizar ***x*** para la nueva línea

Rellenado por línea de rastreo (12)

Para otras figuras geométricas, como circunferencias, triángulos o cuadrados, su relleno se hace por tramos únicos. Se usa únicamente una tabla de tramos ordenados por línea de barrido

Hay que modificar el alg. del punto medio para dibujar puntos interiores a la circunferencia

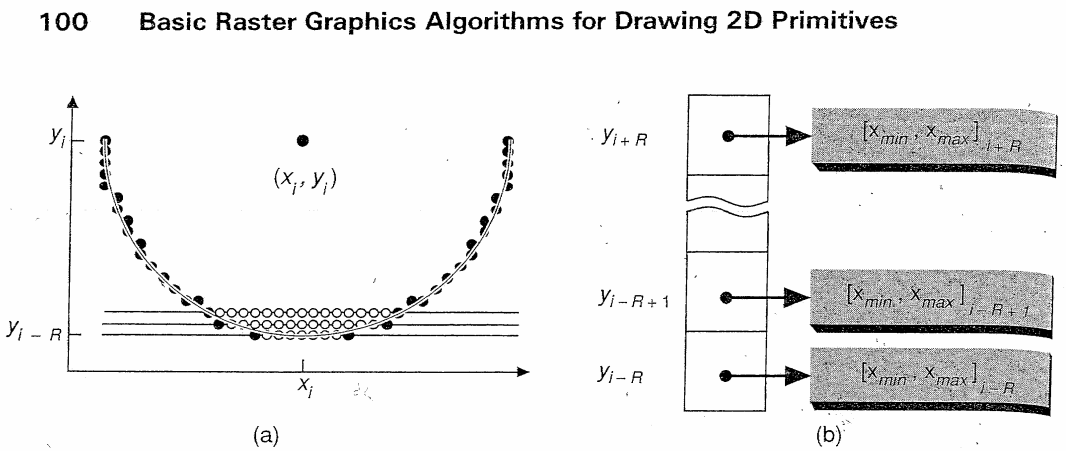
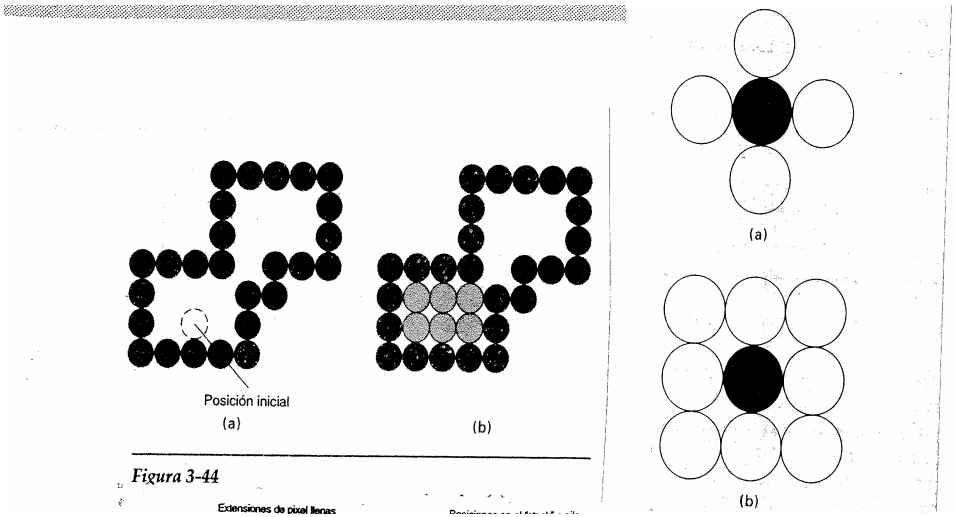


Fig. 3.29 Filling a circle with spans. (a) Three spans. (b) Span table. Each span is stored with its extrema.

Algoritmo de la semilla (1)

La idea consiste en suministrar un pixel interior del polígono (semilla) y a partir de él , colorear el interior pintando los píxeles vecinos hasta llegar a la frontera.



Dos condiciones críticas:

1ª: Los pixeles de la frontera deben estar pintados de un color diferente al fondo y al color que queremos rellenar y la figura debe ser cerrada

2ª: Dependiendo de la forma geométrica del polígono, debe elegirse un relleno de los vecinos de tipo 4 (fig (a)) o de tipo 8 (fig (b))

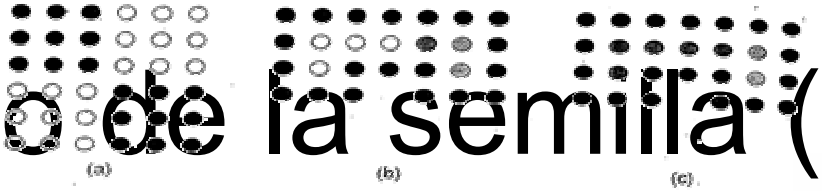
Algoritmo de la semilla (2)

- **Primera aproximación**
- Dos algoritmos de tipo recursivo.
- El algoritmo *FloodFill4* se basa en que el color actual del fondo que queremos pintar es diferente del de la frontera y del color nuevo de relleno. Usa 4 llamadas a los 4 vecinos de la fig (a) y busca pixeles con el color actual de fondo para colorearlos con el de relleno

- El algoritmo *BoundaryFill4* no tiene en cuenta el color actual del relleno. Irá coloreando los vecinos que no sean del color de la frontera y no hayan sido rellenados previamente. Sólo importa el color de la frontera.

- **Gran Desventaja:**
- Hay un gran número de llamadas recursivas (una por pixel) que pueden desbordar la pila del programa de rellenado.

Algoritmo de la semilla (3)




```
procedure FloodFill4 (
   x, y : integer;          {Starting point in region}
   oldValue,                {Value that defines interior}
   newValue : color);       {Replacement value, must differ from oldValue}
begin
   if ReadPixel (x, y) = oldValue then
      begin
         WritePixel (x, y, newValue);
         FloodFill4 (x, y − 1, oldValue, newValue);
         FloodFill4 (x, y + 1, oldValue, newValue);
         FloodFill4 (x − 1, y, oldValue, newValue);
         FloodFill4 (x + 1, y, oldValue, newValue)
      end
end; {FloodFill4.}

procedure BoundaryFill4 (
   x, y : integer;          {Starting point in region}
   boundaryValue,           {Value that defines boundary}
   newValue : color);       {Replacement value}
var
   c : color;               {Temporary variable}
begin
   c := readPixel(x, y);
   if c <> boundaryValue and      {Not yet at the boundary . . .}
         c <> newValue      then  {Nor have we been here before . . .}
      begin
         WritePixel (x, y, newValue);
         BoundaryFill4 (x, y − 1, boundaryValue, newValue);
         three other cases
      end
end; {BoundaryFill4}
```

Fig. 19.55 The flood-fill and boundary-fill algorithms.

Foley, Van Dam

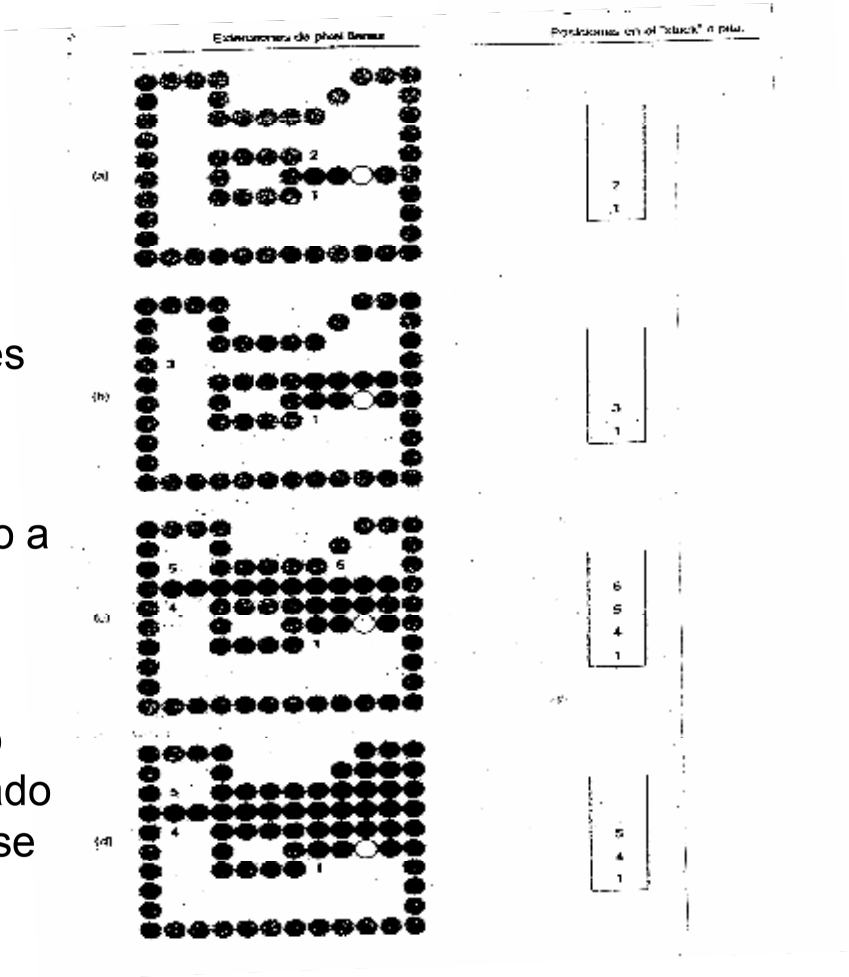
Algoritmo de la semilla (4)

Segunda aproximación:

Sustituimos las llamadas recursivas por una estructura de datos de tipo pila y realizamos rellenos por líneas

A partir de la semilla se pinta la línea de pixeles a la que pertenece, por la izda y la dcha de la semilla, hasta toparse con la frontera.

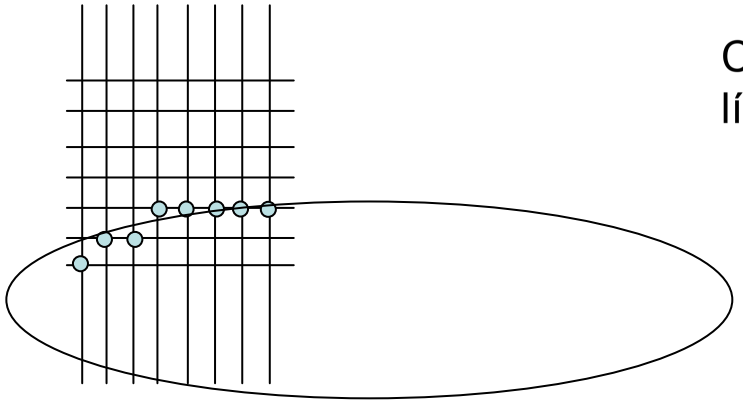
Conforme se pinta la línea, se van consultando a su vez los pixeles superior e inferior y se van tomando como nuevas semillas los pixeles adyacentes a la frontera. Aunque la línea de pintar actual se acabe, seguiremos rastreando las semilla superior e inferior si no hemos topado en éstas con la frontera. Las nuevas semillas se apilan en la pila de semillas.



Hearn, Baker

Algoritmo de la semilla (5)

- Casos Particulares
- El hecho de comprobar cada pixel , por arriba y por abajo si es del color de la frontera es una penalización de este algoritmo.
- Si conocemos a priori la forma de la figura a colorear (por ejemplo una malla de triángulos o cuadrados, o bien círculos, se pueden diseñar algoritmos “ad Hoc” para estas figuras con la misma filosofía que el anterior pero que nos eviten consultar tantas veces los pixeles de las líneas superior e inferior.
- Ej círculo/elipse



Cuando choque con la frontera , la línea superior elijo -10 pixel

Antialising

El aliasing es un fenómeno asociado al hecho de la discretización de una señal continua.

En general, el muestreo de una señal continua produce una pérdida de información de la señal. Esta información perdida se puede recuperar por métodos de reconstrucción (la interpolación entre dos muestras es el más sencillo de ellos).

Sin embargo si la frecuencia del muestreo está por debajo de la **frecuencia de Nyquist** de la señal, la pérdida no es recuperable

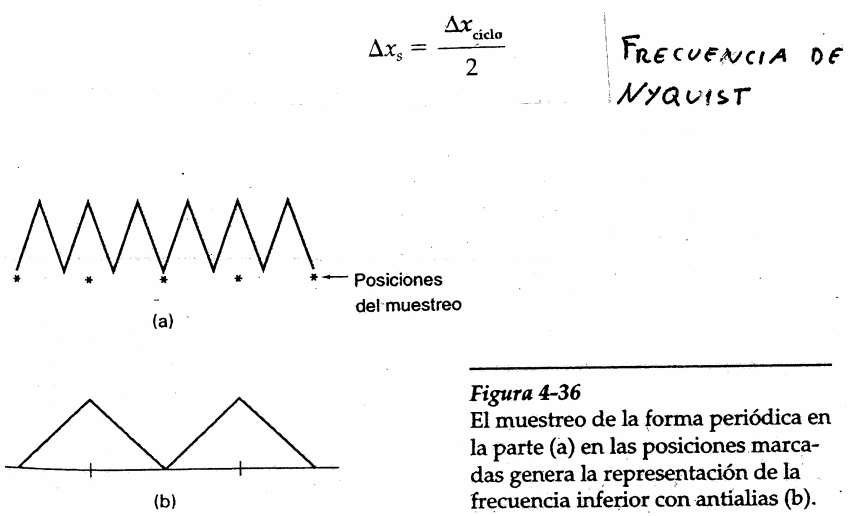


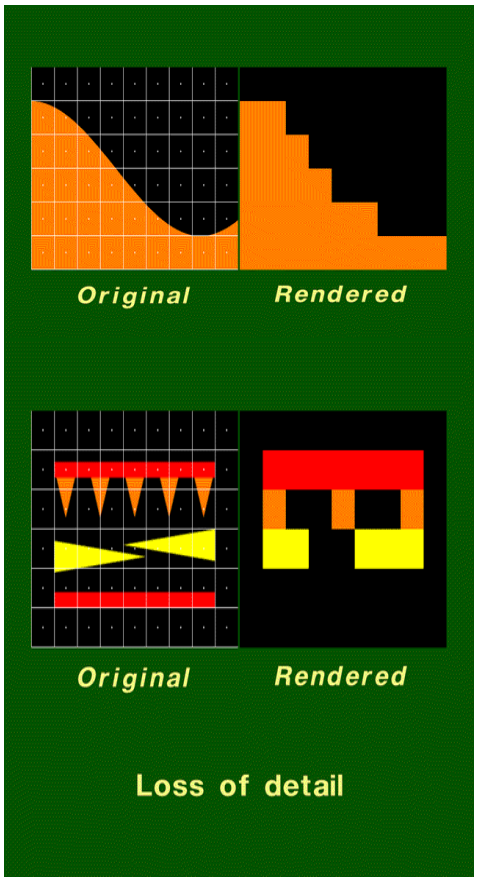
Figura 4-36
El muestreo de la forma periódica en la parte (a) en las posiciones marcadas genera la representación de la frecuencia inferior con antialias (b).

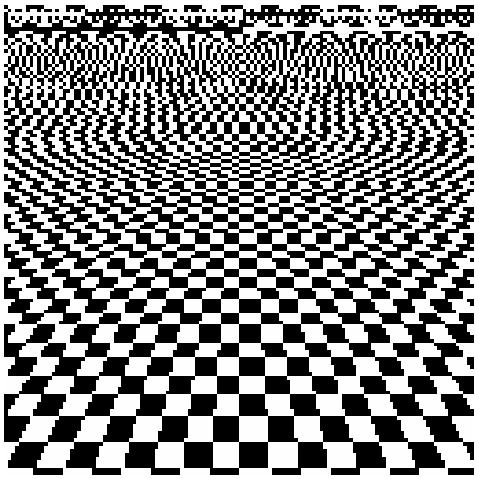
Hearn, Baker

Una imagen puede considerarse un caso extremo de señal periódica donde la frecuencia es infinita. En nuestro caso, desgraciadamente el fenómeno se da siempre, por el hecho de discretizar la imagen para dibujarla en un dispositivo raster.

Antialising (2)

En una imagen el efecto de aliasing se traduce en un **escalonamiento** (jagging) de la figura. También pueden producirse **artefactos visuales** como por ejemplo los patrones de Moiré





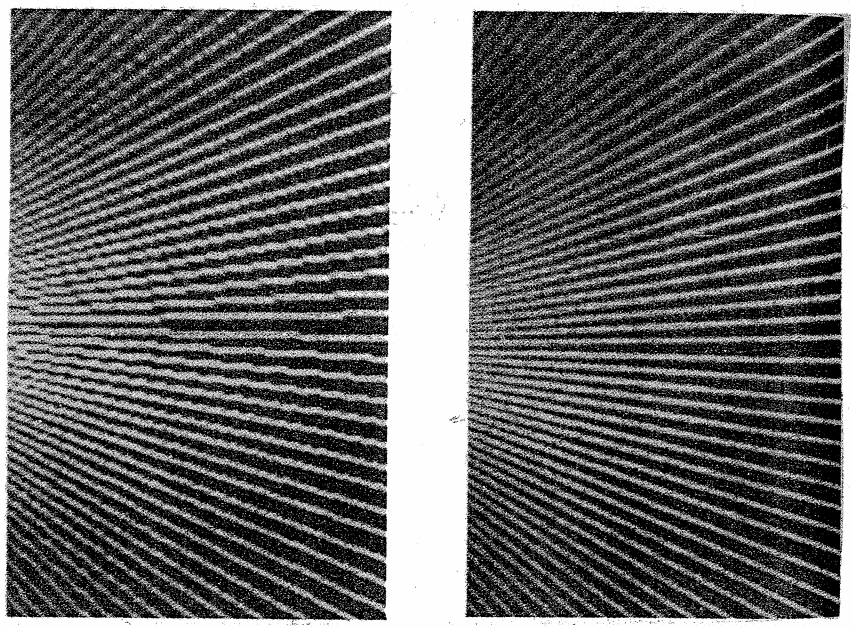
Siggraph, wikipedia

Un aumento de resolución del dibujado **no elimina** estos artefactos sino que los mitiga a una escala concreta (a un gran costo computacional)

Aliasing (3)

- **Antialiasing**: Cualquier técnica que reduzca el efecto visual del aliasing en la imagen (sin cambiar de resolución).

- Para ello se modifica el color de los pixeles elegidos para trazar las primitivas y también los que están en su alrededor. Estas técnicas de antialising para imágenes están **basadas en el comportamiento fisiológico del ojo humano** que tiende a fundir los colores próximos espacialmente si éstos son parecidos (¿servirán estas técnicas para ballenas?)



Hearn, Baker

Aliasing (4)

- Por lo tanto, las primitivas trazadas con antialiasing no van a colorear un único pixel . Nuestras primitivas ahora van a tener un grosor sobre la malla de dibujado y van a ocupar total o parcialmente el área de los pixeles involucrados.

- La idea de los algoritmos de antialiasing consiste básicamente en estimar el área parcial ocupada por la primitiva para un pixel concreto y determinar así su color

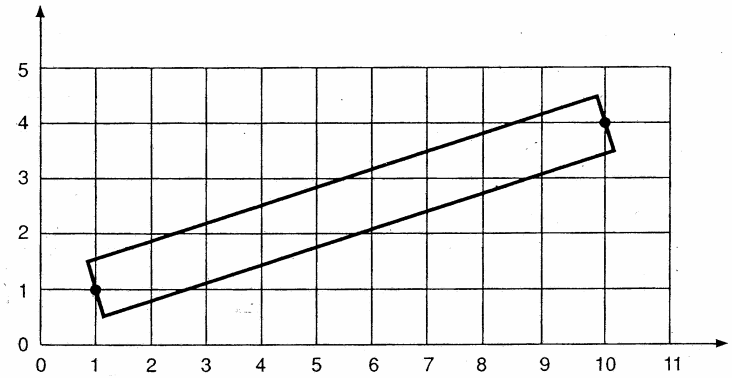


Figure 3.55 Line of nonzero width from point (1,1) to point (10,4).

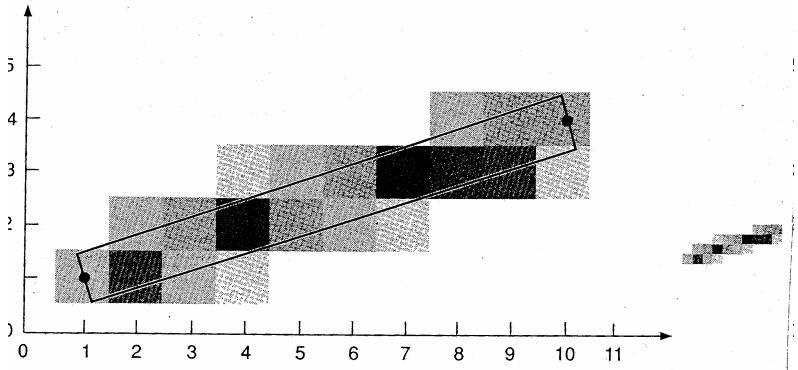


Fig. 3.56 Intensity proportional to area covered.

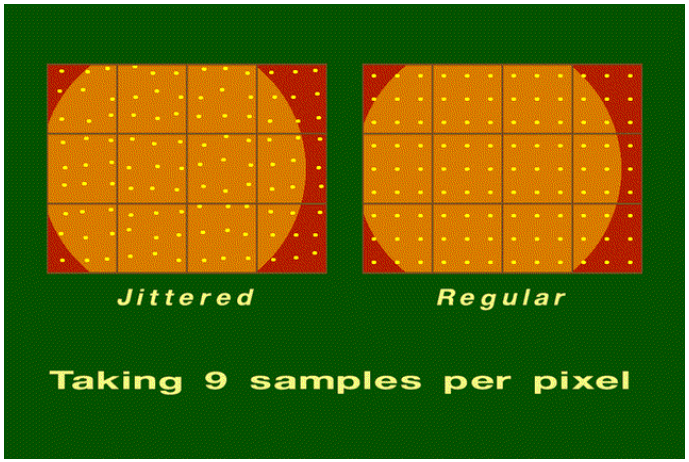
Foley, Van Dam

Aliasing (5)

¿cómo calcular el área que una primitiva ocupa en un pixel?

La técnica más extendida es la del **Postfiltrado**, tambien llamada **sobremuestreo** o “supersampling”.

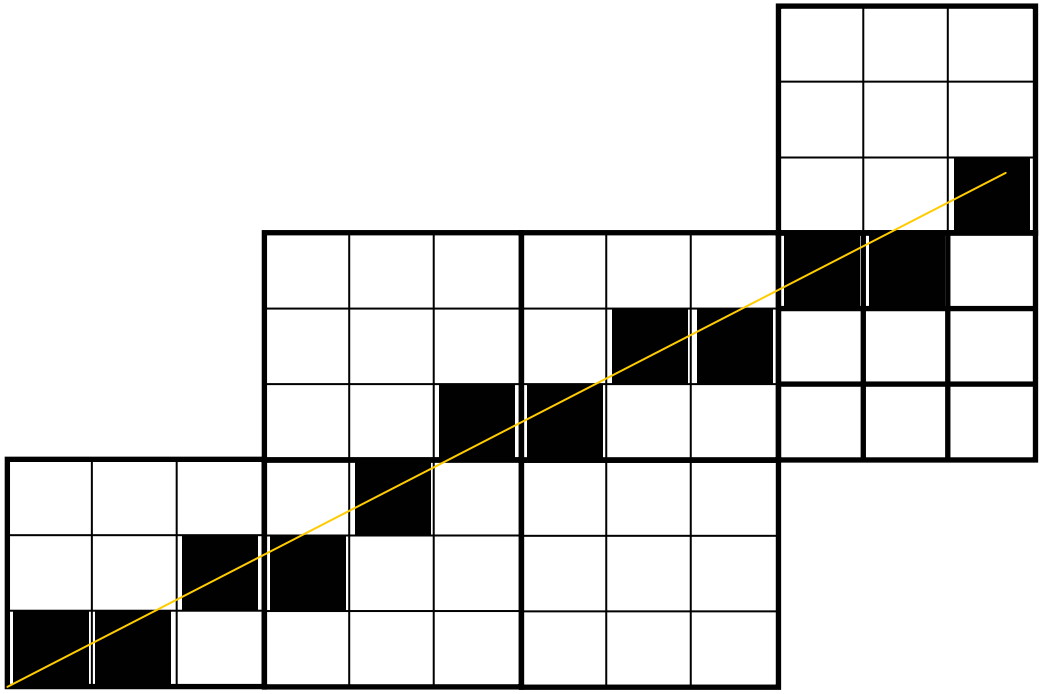
La idea consiste en dibujar sin antialiasing la figura a una resolución superior a la de la imagen que queremos conseguir con antialiasing. Por ejemplo si la resolución de la imagen a tratar es 512x512, se hace un renderizado tres veces más fino en *x* e *y* a una resolución de 1536 x 1536. De esta manera, cada pixel de la imagen a tratar tiene un sobremuestreo de nueve subpixeles



ACM Siggraph

Aliasing (6)

- Así un pixel de la imagen a tratar puede estar cubierto por 3, 2, 1, 0 píxeles del sobremuestreo



Sobremuestreo sin peso:

•La intensidad de un pixel decrece con la distancia del centro del pixel a la línea matemática

•Un pixel no se colorea si no intersecta a la recta matemática

•Areas iguales cubiertas dan el mismo color de lixel

Aliasing (7)

El sobremuestreo con peso, en cambio , usa filtros basados también en la distancia del centro del pixel a la primitiva para establecer la importancia del área cubierta.

Se pueden usar filtros con pesos discretos o bien más sofisticados como los filtros cónicos

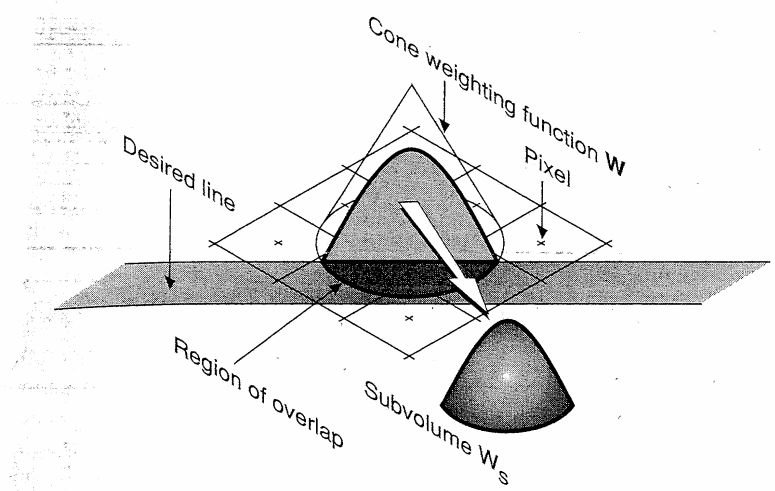
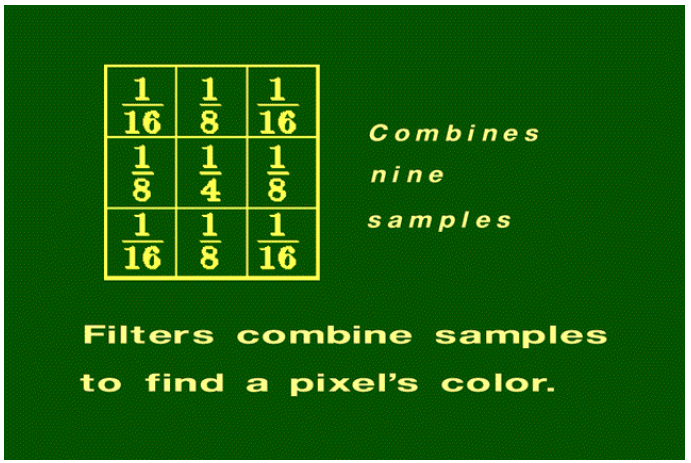


Fig. 3.58 Cone filter for circular pixel with diameter of two grid units.



Aliasing (8)

- Otra técnica consiste en pensar las rectas en este caso como figuras con una anchura geométrica (de un pixel generalmente) y calcular analíticamente la influencia que esta figura con anchura real tiene en cada pixel, generalmente usando el parámetro distancia al centro del pixel. Estas se llaman técnicas de **Prefiltrado.**
- Una de ellas como ejemplo es el Algoritmo de **Gupta – Sproull** para el trazado de rectas.

- **Algoritmo de antialiasing para rectas de Gupta-Sproull**
- La idea se basa en utilizar el algoritmo de Bresenham para calcular también la distancia a los pixeles contiguos y determinar así a través de un filtro basado en la distancia, el color del pixel. Este filtro lo llamaremos *Filter(D)* donde **D** es la distancia entre el centro del pixel y el centro de la recta que queremos trazar.

- Buscamos D = D(d) . Donde 0 <= D <= 2

Aliasing (9)

Tomemos el algoritmo de Bresenham y consideremos ahora los **pixeles superior e inferior** al elegido por el algoritmo

$D = v \cos \Phi = v\, dx / (dx^2 + dy^2)^{1/2}$

Donde $dx = x1 - x0$ $dy = y1 - y0$

$v = y_{recta} - y_{pixel}$ (variable con signo)

Se trata de calcular la variable *v* de manera incremental usando el parámetro **d** del algoritmo de Bresenham

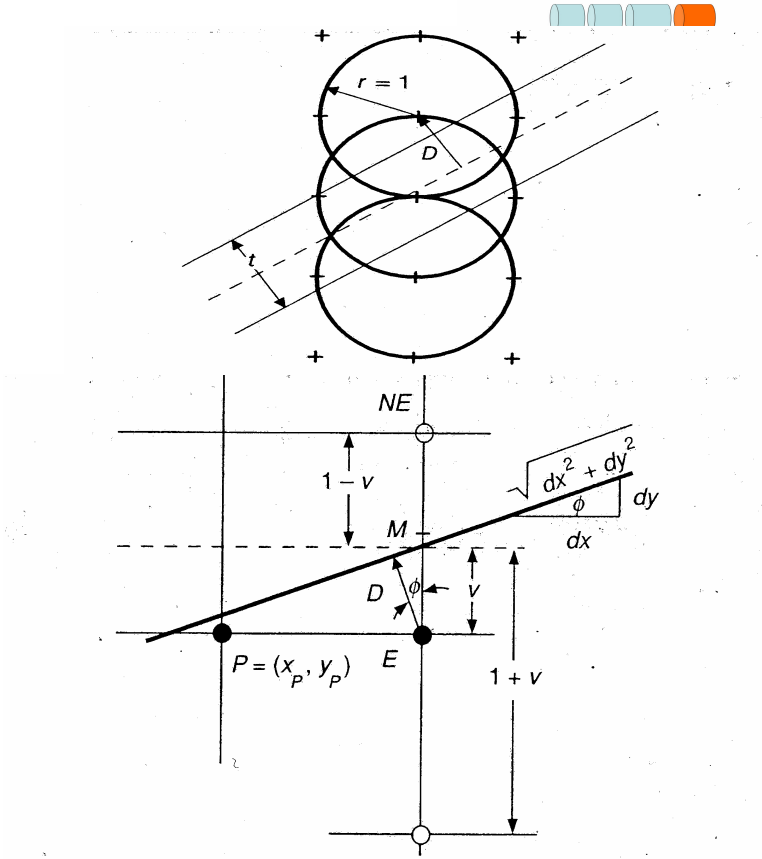


Fig. 3.61 Calculating distances to line in midpoint algorithm.

Aliasing (10)

```
procedure AntiAliasedLineMidpoint (x1, y1, x2, y2 : integer);
{This algorithm uses Gupta-Sproull's table of intensity as a function of area coverage
for a circular support in the IntensifyPixel function. Note that overflow may occur in
the computation of the denominator for 16-bit integers, because of the squares.}
var
    dx, dy, incrE, incrNE, d, x, y, two_v_dx : integer;
    invDenom, two_dx_invDenom : real;

    procedure IntensifyPixel (x, y : integer; distance : real);
    var
      intensity : real;
    begin
      intensity := Filter(Round(Abs(distance)));
      {Table lookup done on an integer index; thickness 1}
      WritePixel (x, y, intensity)
    end;

begin
    dx := x2 - x1;
    dy := y2 - y1;
    d := 2 * dy - dx;                          {Initial value d_start as before}
    incrE := 2 * dy;                           {Increment used for move to E}
    incrNE := 2 * (dy - dx);                   {Increment used for move to NE}
    two_v_dx := 0;                             {Numerator; v = 0 for start pixel}
    invDenom := 1/(2 * Sqrt (dx * dx + dy * dy));  {Precomputed inverse denominator}
    two_dx_invDenom := 2 * dx * invDenom;      {Precomputed constant}
    x := x1;
    y := y1;
    IntensifyPixel (x, y, 0);                  {Start pixel}
    IntensifyPixel (x, y + 1, two_dx_invDenom);    {Neighbor}
    IntensifyPixel (x, y - 1, two_dx_invDenom);    {Neighbor}
    while x < x2 do
      begin
        if d < 0 then
          begin                                {Choose E}
            two_v_dx := d+dx;
            d := d + incrE;
            x := x + 1
          end
        else
          begin                                {Choose NE}
            two_v_dx := d - dx;
            d := d + incrNE;
            x := x + 1;
            y := y + 1
          end;
        {Now set chosen pixel and its neighbors}
        IntensifyPixel (x, y, two_v_dx * invDenom);
        IntensifyPixel (x, y + 1, two_dx_invDenom - two_v_dx * invDenom);
        IntensifyPixel (x, y - 1, two_dx_invDenom + two_v_dx * invDenom)
      end {while}
end;
```

Fig. 3.62 Gupta–Sproull algorithm for antialiased scan conversion of lines.

```
void IntensifyPixel (int x, int y, double distance)
{
    double intensity = Filter (Round (fabs (distance)));
    /* Table lookup done on an integer index; thickness 1 */
    WritePixel (x, y, intensity);
} /* IntensifyPixel */
```

Fig. 3.62 Gupta–Sproull algorithm for antialiased scan conversion of lines.

Foley, Van Dam

Ponderación	Paso	Distancia	Ponderación	Paso	Distancia
0-1/8	0 (blanco)	1.40-2.00	1/8-2/8	1	1.20-1.40
2/8-3/8	2	1.08-1.20	3/8-4/8	3	1.00-1.08
4/8-5/8	4	0.92-1.00	5/8-6/8	5	0.80-0.92
6/8-7/8	6	0.60-0.80	7/8-1	7 (negro)	0.00-0.60

Tabla de Gupta-Sproull para 8 valores

Se puede evitar el cálculo tabulando los posibles resultados

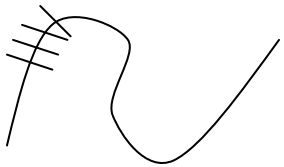
Ej Valores enteros de x tal que $x / k \in [1.20, 1.40]$

Curvas paramétricas

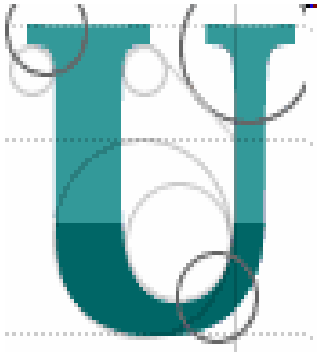
- **Problema**: aproximación de curvas suaves de formas no regulares. (Estas curvas son muy útiles en diseño).

- **Primera aproximación**: Discretizando en polilíneas.

  Desventajas de esta aproximación.
- Se necesitan muchos vértices para aproximar una curva suave
- No permite la modificación de la forma de la curva, ta que no es suave sino que se produce una deformación



- **Segunda aproximación**: Usando cónicas
- Desventajas: A veces es difícil calcular las cónicas
- que aproximen a una forma determinada.
- Es imposible la modificación de la curva. Hay que volver a
- calcular de nuevo.



Curvas paramétricas (2)

- **Elección correcta**: Polinomios cúbicos (grado 3) $P(x) = ax^3 + bx^2 + cx + d$
- ¿Por qué no de mayor grado?. Los polinomios de mayor grado hacen aumentar los parámetros de control y también tienen más máximos y mínimos locales, con lo que es difícil de manejar.

- ¿qué **forma de representación** elegimos?

- *Explicita* (en función de una variable independiente) x ; $y = f(x)$; $z = g(x)$
- Inconvenientes: Es necesario utilizar varias expresiones matemáticas para describir partes de una misma figura geométrica: $x = x$; $y = \pm \sqrt{(R^2 - x^2)}$

- *Implícita* (de la forma $F(x,y,z) = 0$).
- Inconvenientes: Pueden dar más soluciones de las que queremos, por los que hay que añadir restricciones $F(x,y) = x^2 + y^2 = 1$ $x \geq 0$ (definición semicircunferencia).

Curvas paramétricas (2)

- **Forma adecuada**: Ecuaciones paramétricas x= x(t); y = y(t); z = z(t)
- Ejemplo: x = R cos(2*π*t); y = R sin(2*π*t);

- Nuestras curvas estarán compuestas a **trozos** por curvas paramétricas de tipo cúbico. La curva total es la unión de varias curvas paramétricas.

- Cada curva paramétrica se define por:
- Q(t) = [x(t), y(t), z(t)}
- $x(t) = a_x t^3 + b_x t^2 + c_x t + d_x$
- $y(t) = a_y t^3 + b_y t^2 + c_y t + d_y$ donde $0 \leq t \leq 1$ **(Ecuaciones I)**
- $z(t) = a_z t^3 + b_z t^2 + c_z t + d_z$

- **Propiedades** de las curvas paramétricas:
- 4 condiciones de contorno para determinar los cuatro coeficientes
- Pueden no estat en un mismo plano. (Las cónicas están en un mismo plano porque necesitan 3 puntos para definirse).

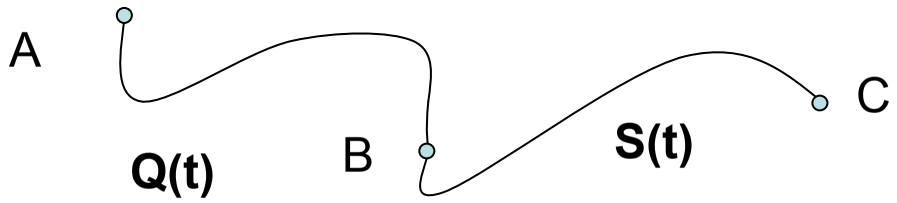
Curvas paramétricas (3)

- Notación matricial
-
- $T = [\, t^3 \quad t^2 \quad t \quad 1 \,]$ $\qquad C = \begin{pmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{pmatrix}$
-

$$\mathbf{Q(t) = T\,C}$$

Continuidad en los puntos de unión

La construción de una curva suave se basas en la adición de tramos de curvas paramétricas cúbicas En el punto de unión B se deben cumplir algunas características



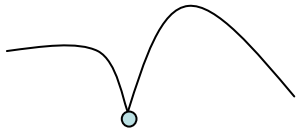
Curvas paramétricas (4)

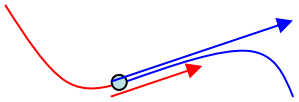
- Se calcula la derivada de las curvas paramétricas en dicho punto

- $d(Q(t)) / dt = Q'(t) = dT/dt \; C = [\, 3t^2 \quad 2t \quad 1 \quad 0 \,] \; C =$
- $= [\; 3a_x t^2 + 2 b_x t + c_x \;\; 3a_y t^2 + 2 b_y t + c_y \;\; 3a_z t^2 + 2 b_z t + c_z \,]$

- Se analizan las tangentes en el punto B. $Q'(t_B)$ y $S'(t_B)$
- **Tipos de continuidad**

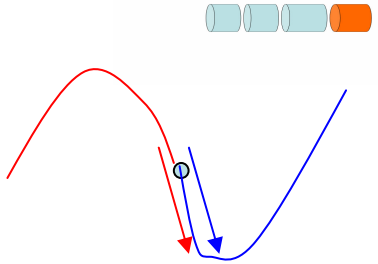
- *Continuidad Geométrica cero (Continuidad G^0)*
- *Continuidad cero (Continuidad C^0)*
- Las dos curvas se unen brúscamente





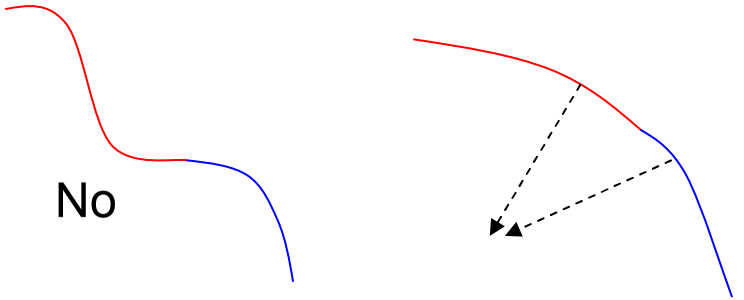
- *Continuidad Geométrica Uno(Continuidad G^1)*
- En el punto de unión las tangentes tienen la misma pendiente pero diferente módulo

Curvas paramétricas (5)



- Continuidad C^1
- Las dos curvas tienen tangentes de igual dirección y módulo

- Continuidad C^2
- La segunda derivada tiene el mismo valor y dirección en el punto B. Esto implica que conserva la curvatura de la curva.



- **Para el modelado geométrico es suficiente la continuidad G^1**

Curvas paramétricas (6)

- Vamos a **desarrollar una formulación** adecuada de las curvas **para poderlas diseñar y modificar gráficamente y de manera interactiva sencilla**
- Para solucionar el sistema de ecuaciones x(t) y(t) z(t) de un tramo, necesitamos conocer cuatro condiciones de contorno para calcular los coeficientes. Estas condiciones son los puntos de inicio y final del tramo y sus tangentes en dichos puntos, que nos servirán para imponer la G^1 entre tramos.

- Construyamos una notación donde quede de manifiesto esta información.
- **Q(t) = T C = T MG donde C = MG**
- **M** es de dim 4x4 y se denomina **Matriz Base**
- G es un vector de cuatro componentes (cada una de ellas a su vez, de tres dimensiones) y se llama **vector geométrico**
-
-
- $$Q(t) = [x(t)\ \ y(t)\ \ z(t9] = [t^3\ \ t^2\ \ t\ \ 1] \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{pmatrix} \begin{pmatrix} G_1 \\ G_2 \\ G_3 \\ G_4 \end{pmatrix}$$
-

Curvas paramétricas (7)

- $x(t) = (t^3 m_{11} + t^2 m_{21} + t\ m_{31} + m_{41})\ g_{1x} + (t^3 m_{12} + t^2 m_{22} + t\ m_{32} + m_{42})\ g_{2x} +$
- $(t^3 m_{31} + t^2 m_{32} + t\ m_{33} + m_{34})\ g_{3x} + (t^3 m_{41} + t^2 m_{42} + t\ m_{43} + m_{44})\ g_{4x}$

- Vemos que x(t) es una combinación lineal de los elementos de la matriz **G**.
- Estas funciones peso se llaman **Funciones de Mezcla** (blending functions).
- Las notaremos como **B = TM**

- **Q(t) = B G**

- Mientras que **B recoge las características generales** de una clase de curvas, la matriz **G** (vector geométrico) **recoge las condiciones de contorno de la curva particular**

Curvas paramétricas (8)

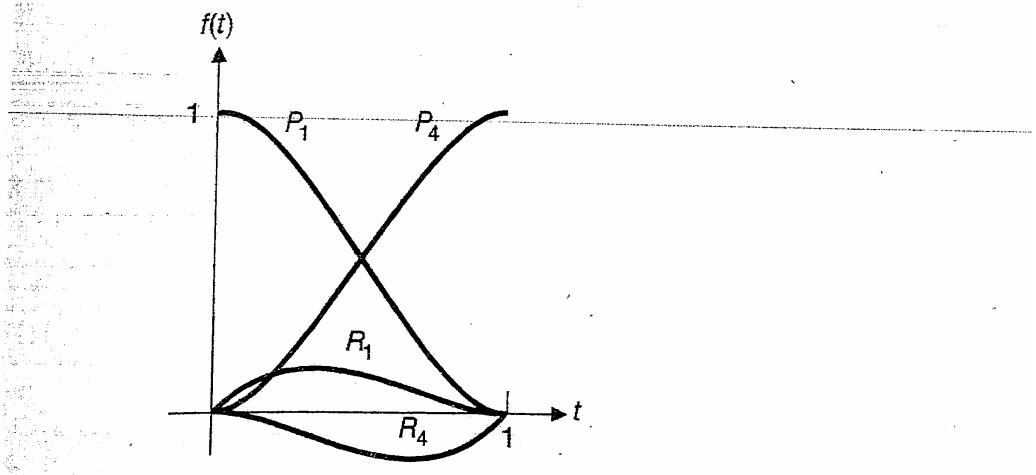
- **Curvas de Hermite**
- **Condiciones de contorno:** Puntos inicial y final P_1 P_4 y tangentes a esos puntos R_1 R_4

- Si aplico las condiciones de contorno a las expresiones matriciales:

- **$x(t=0) = P_{1x}$**
- **$x(t=1) = P_{4x}$** **(ver desarrollo)**
- **$x'(t=0) = R_{1x}$** **===>** **$M_H = \begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$**
- **$x'(t=1) = R_{4x}$**
-

- **$B_H = T\ M_H = [\ (2t^3 - 3t^2 + 1) \quad (-2t^3 + 3t^2) \quad (t^3 - 2t^2 + t) \quad (t^3 - t^2)\ ]$**

- $Q(t) = M_H B_H = (2t^3 - 3t^2 + 1) P_1 + (-2t^3 + 3t^2) P_4 + (t^3 - 2t^2 + t) R_1 + (t^3 - t^2) R_4$



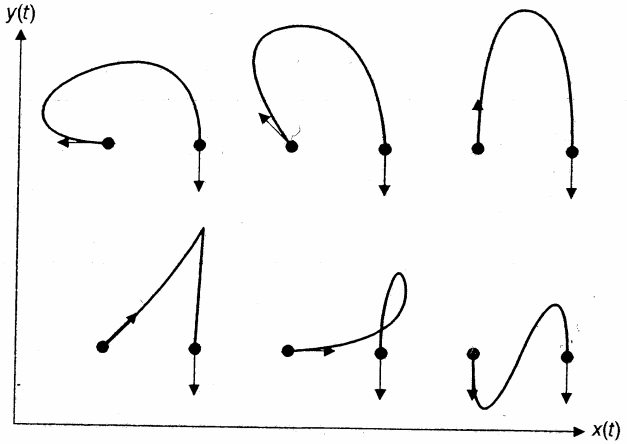
12 The Hermite blending functions, labeled by the elements of the geometry hat they weight.

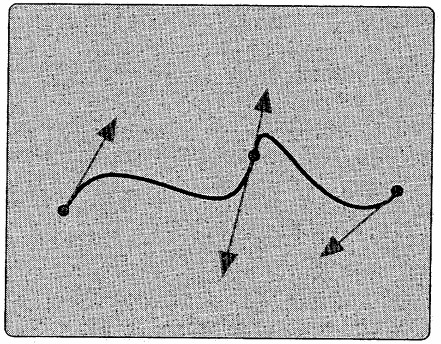
Notar que la influencia de cada polinomio de Hermite componente de B_H tiene una **influencia global** sobre la curva. Aunque hay zonas de la misma donde tiene más peso.

Para rotar o escalar una curva basta con aplicar dicha transformación al vector de geometría G_H. Esto es lo mismo que decir que la formulación matemática matricial que hemos desarrollado es invariante frente a rotaciones, escalados,…

Curvas paramétricas (9)

- **Representación gráfica:**
- El control gráfico de las curvas de Hermite viene dado por los puntos $\mathbf{P_1}$ $\mathbf{P_4}$ y por los vectores tangentes a esos puntos $\mathbf{R_1}$ $\mathbf{R_4}$
- No es un control demasiado intuitivo ya que estamos controlando magnitudes vectoriales
- ($\mathbf{R_1}$ y $\mathbf{R_4}$)
- En el punto de unión de las curvas, las tangentes deben de ser colineares para que se cumpla la condición G^1.





Curvas paramétricas (10)

- Para imponer a dos curvas de Hermite la continuidad en el punto de unión, se debe cumplir en los vectores geométricos:

$$\begin{pmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{pmatrix} \quad y \quad \begin{pmatrix} P_4 \\ P_7 \\ K\,R_4 \\ R_7 \end{pmatrix} \quad \text{con } k>0$$

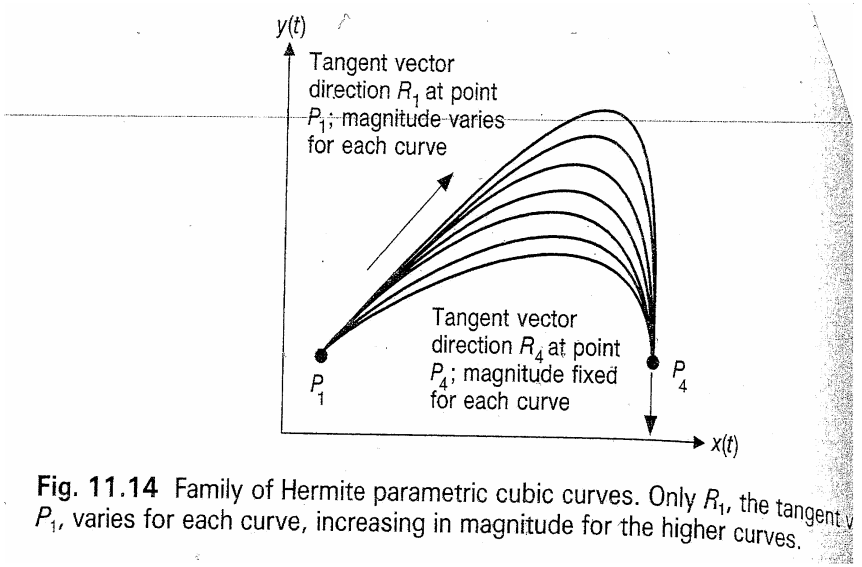
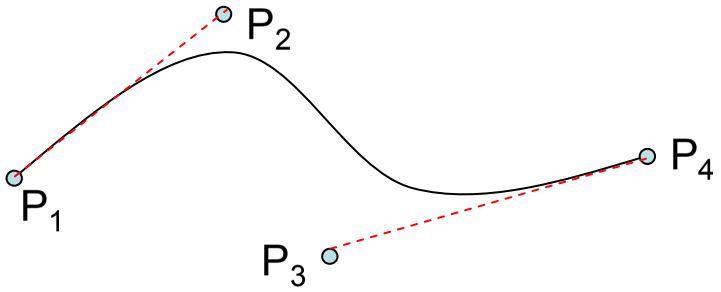
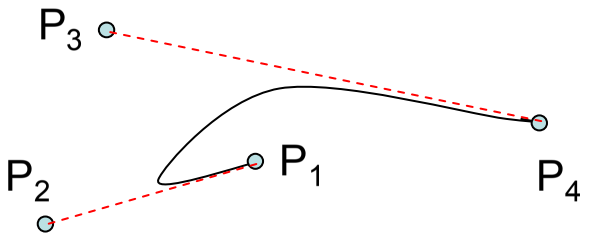


Fig. 11.14 Family of Hermite parametric cubic curves. Only R_1, the tangent v P_1, varies for each curve, increasing in magnitude for the higher curves.

Curvas paramétricas (11)

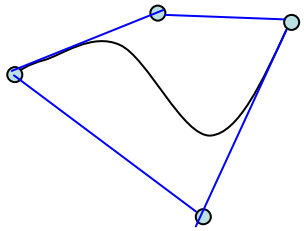
- **Curvas de Bèzier**
- Las curvas de Bèzier se especifican mediante un vector geométrico que contiene los puntos inicial y final de la curva (P_1 y P_4) y otros dos punto intermedios (P_2 y P_3) que junto con los anteriores determinan las tangentes en los puntos inicial y final.





Donde los puntos 2 y 3 no están sobre la curva.

La curva siempre cae en la envolvente convexa de los puntos de control



Curvas paramétricas (12)

- Por tanto: $G_B = \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ \end{pmatrix}$
- Se puede establecer una relación entre los puntos de control de Bèzier y los vectores R_i de Hermite
- $R_1 = 3(P_2 - P_1)$ $R_4 = 3(P_4 - P_3)$ donde el factor 3 viene de imponer $d^2 Q(t)/dt^2 = 0$ en los puntos (0,0) (1,0), (2,0), (3,0)
- Poniendo estas ecuaciones de forma matricial

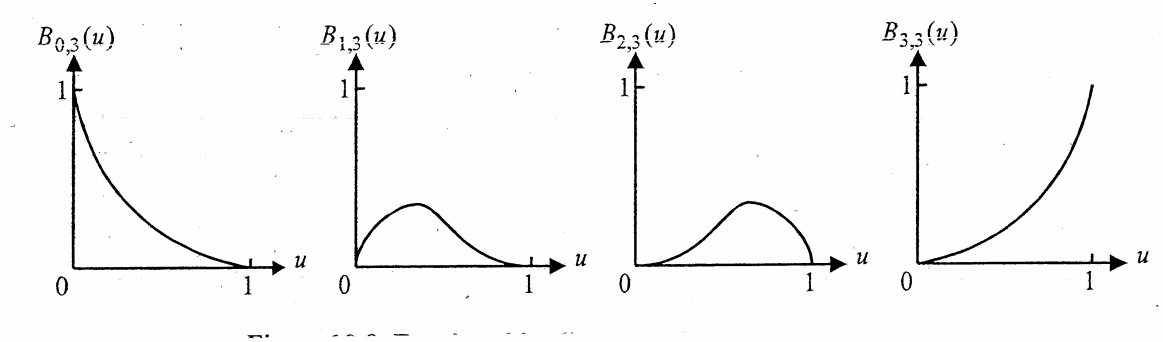
$$G_H = \begin{pmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{pmatrix} \begin{pmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{pmatrix} = M_{HB}\, G_B$$

Curvas paramétricas (13)

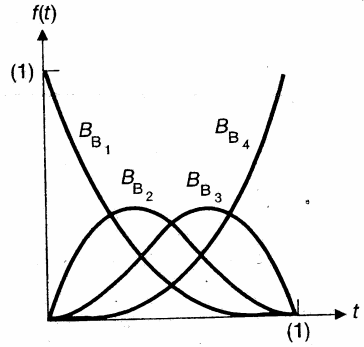
- De la expresión matricial anterior podemos obtener la matriz de blending de Bèzier
- $Q(t) = T\ M_H\ G_H = T\ M_H\ (M_{HB}\ G_B) = T\ (M_H\ M_{HB}\ )\ G_B = T\ M_B\ G_B$
- $M_B = M_H\ M_{HB} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$
- **$Q(t) = T\ M_B\ G_B = (-t^3 + 3t^2 - 3t + 1)P_1 + (3t^3 - 6t^2 + 3t\ )\ P_2 + (-3t^3 + 3t^2\ )P_3 + t^3\ P_4$**

Polinomios de Bernstein

Curvas paramétricas (14)



Polinomios de Bernstein para curvas de bèzier de grado 3 (n=3)



La influencia de cada polinomio es total respecto del párámetro t aunque tiene un rango de valores donde es más notable.

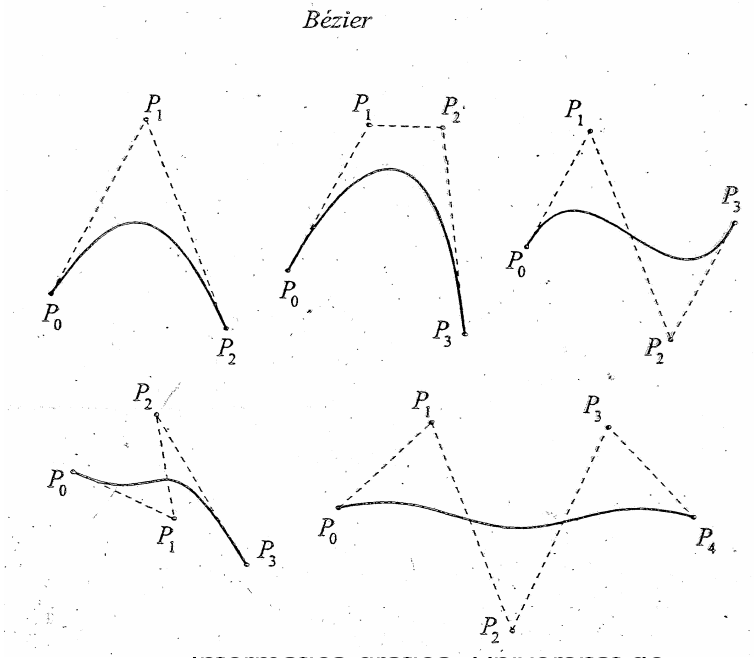
Curvas paramétricas (15)

- Los polinomios de Bernstein pueden definirse en la práctica desde n = 3.
- Así hay curvas con tres, cuatro, cinco etc puntos de control.
- En general, su expresión es:

$$x(u) = \sum_{k=0}^{n} x_k B_{k,n}(u) \quad (10.9)$$

$$y(u) = \sum_{k=0}^{n} y_k B_{k,n}(u) \quad (10.10)$$

$$z(u) = \sum_{k=0}^{n} z_k B_{k,n}(u) \quad (10.11)$$

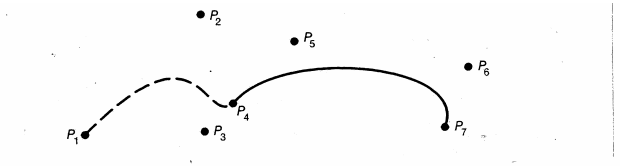


Curvas paramétricas (15)

Propiedades de las curvas de Bèzier

- La continuidad G^1 para las curvas de Bèzier implica que $dx^q(t)/dt \Big|_{t=1} = dx^r(t)/dt \Big|_{t=0}$

  Esto significa que $P_4 - P_3 = P_5 - P_4$. Es decir son colineales

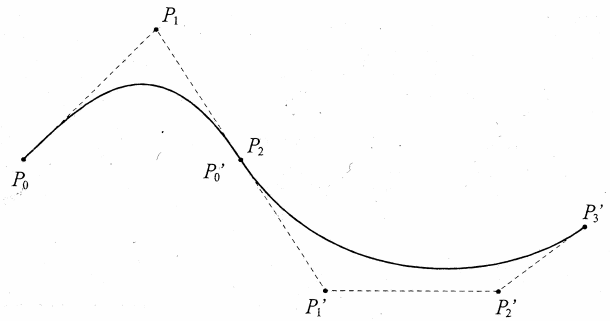


•La suma de los polinomios de Bernstein para cualquier valor de t es 1, no siendo ninguno negativo. Esto justifica que la curva quede encerrada en la envolvente convexa de los puntos de control

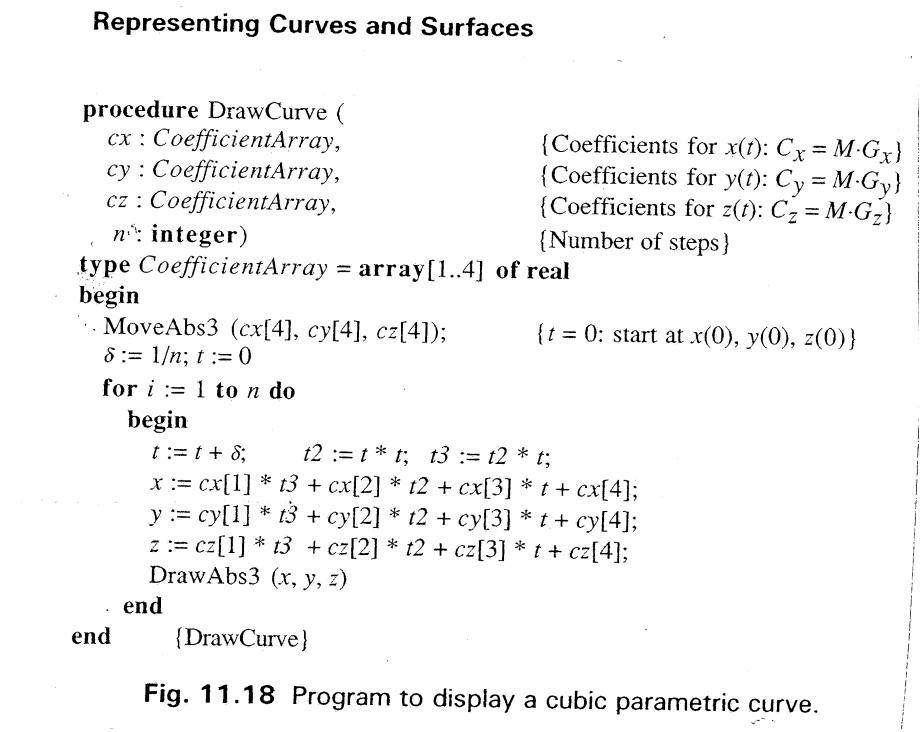
•Si $P_1 = P_4$ entonces la curva es cerrada

•El control de la curva es global. La modificación de uno de los puntos de control afecta a la forma de toda la curva.

•Las curvas se comportan suavemente a los cambios de valor de los puntos de control



Algoritmos de diseño de curvas(1)



Representing Curves and Surfaces

```
procedure DrawCurve (
   cx : CoefficientArray,                {Coefficients for x(t): C_X = M·G_X}
   cy : CoefficientArray,                {Coefficients for y(t): C_Y = M·G_Y}
   cz : CoefficientArray,                {Coefficients for z(t): C_Z = M·G_Z}
   n : integer)                          {Number of steps}
type CoefficientArray = array[1..4] of real
begin
   MoveAbs3 (cx[4], cy[4], cz[4]);       {t = 0: start at x(0), y(0), z(0)}
   δ := 1/n; t := 0
   for i := 1 to n do
      begin
         t := t + δ;      t2 := t * t;   t3 := t2 * t;
         x := cx[1] * t3 + cx[2] * t2 + cx[3] * t + cx[4];
         y := cy[1] * t3 + cy[2] * t2 + cy[3] * t + cy[4];
         z := cz[1] * t3 + cz[2] * t2 + cz[3] * t + cz[4];
         DrawAbs3 (x, y, z)
      end
end      {DrawCurve}
```

Fig. 11.18 Program to display a cubic parametric curve.

Algoritmo de fuerza bruta

Este algoritmo dibujaría un curva paramétrica a partir de las **(ECUACIONES I)**

Con este algoritmo no hay posibilidad de modificar interactivamente a la curva, ya que hay que redefinir cada uno de los coeficientes de las ecuaciones

No es de ninguna utilidad para diseño interactivo de curvas.

Un algoritmo de fuerza bruta iría calculando valores de x(t) y(t) z(t) para valores de $t \in [0,1]$ con un **paso** $\delta = 1/n$. $\delta 0 = 0$; $\delta 1 = 1/n$; $\delta 2 = 2/n$; ….; $\delta n = 1$
Cada vez que el vector geométrico varia, hay que recalcular los puntos de la curva y redibujarla.

Algoritmos de diseño de curvas(2)

Basándose en la idea de evaluar la curva en tramos homogéneos δ, se implementa el diseño de las curvas de Bèzier

```
glColor3f(1.0, 0.0, 0.0);
glFlush();
t=0;
while(t<=1){
  pxold=px;
  pyold=py;
  pzold=pz;
  uno_t_2= (1-t)*(1-t);
  t_2 = t*t;

  //se podria crear un procedimiento mas eficiente basado en diferencias hacia delante
  //ver Foley, para hacer los calculos siguientes
  px = uno_t_2*(1-t)*G[0][0]+3*t*uno_t_2*G[1][0]+ 3*t_2*(1-t)*G[2][0]+t_2*t*G[3][0];
  py = uno_t_2*(1-t)*G[0][1]+3*t*uno_t_2*G[1][1]+ 3*t_2*(1-t)*G[2][1]+t_2*t*G[3][1];
  pz = uno_t_2*(1-t)*G[0][2]+3*t*uno_t_2*G[1][2]+ 3*t_2*(1-t)*G[2][2]+t_2*t*G[3][2];

  if(primero==0){
    glBegin(GL_POINTS);
            glVertex3f(px,py,pz);
    glEnd();
            primero++;
  }
  else{
            glBegin(GL_LINES);
            glVertex3f(pxold,pyold,pzold);
            glVertex3f(px,py,pz);
    glEnd();
  }
  t+=1.0/n;
 } glFlush();
}
```