Rasterización (Scan-Conversion)

Aquí trataremos el problema de representar un objeto mediante un conjunto discreto de puntos. Los objetos continuos del modelo (curvas y superficies) se representan en un monitor u otro dispositivo raster mediante un conjunto discreto de pixeles. Comenzaremos analizando el caso más simple, el de una curva. Al proyectar una curva cualquiera sobre el plano de la imagen obtenemos una curva plana. Estudiaremos como transformarla en una sucesión adecuada de puntos.

Una curva plana es un conjunto continuo e infinito de pares de números reales $\{x(t), y(t)\}$[1], ambas coordenadas son funciones continuas de un parámetro real t que varía entre t_0 y t_1 ($t \in [t_0, t_1]$) si la línea es acotada, en caso contrario alguna o ambas cotas pueden ser infinitas.

Los dispositivos vectoriales pueden asumir que el parámetro t representa el tiempo y mover dos motores independientes para trazar la línea con un marcador. Como ejemplo de esto tenemos los pantógrafos, que mueven un soplete en x e y para cortar chapas y los *plotters* de corte para autoadhesivos utilizados en carteles y gigantografías. En la práctica, con motores "paso a paso" (*stepper*) se aproximan las curvas de acuerdo a la precisión de los motores. Pero aún así, al ir de una posición discreta a la siguiente, el marcador sigue marcando o cortando, realiza un trazo continuo.

La mayoría de los equipos actuales de impresión de tintas (*plotters* e impresoras) utilizan el método digital o *raster* que consiste en inyectar tinta en puntos aislados del papel, que se recorre línea por línea (*scan-lines*). Lo mismo sucede en los monitores, que constan de una matriz de puntos brillantes.

Las curvas *raster* son una sucesión finita de puntos **contiguos** de coordenadas enteras y de modo que no haya espacio entre dos puntos sucesivos de la línea: $max(|\Delta x|, |\Delta y|)=1$. Una sucesión de puntos contiguos suele denominarse "*water tight*", a prueba de agua o impermeable.

En todo lo que sigue, vamos a suponer que la curva continua no tiene grandes variaciones en una unidad, es decir que entre un píxel y el siguiente la curva original o real es prácticamente recta, no zigzaguea entre pixeles contiguos. En caso contrario, cualquier proceso daría un mal resultado pues no podemos "muestrear" la curva con una precisión del orden de la distancia entre píxeles, que es la medida que usamos como unidad.

Si a cada punto de la curva lo asociamos al punto mas cercano de coordenadas enteras, nos queda el conjunto $\{int(x(t)+0.5), int(y(t)+0.5)\}$. Éste es un conjunto contiguo, discreto y finito de pares, que representa muy bien a la curva. Deberíamos crear ese conjunto, pero no se puede definir un algoritmo que recorra el parámetro t como una variable real continua (con una variable continua no se puede decir "para cada valor" en un algoritmo).

La solución más simple consiste en hacer saltos discretos del parámetro, se divide el intervalo en muchas partes y se aproxima la curva mediante una poligonal de segmentos rectos. La aproximación rectilínea es tanto peor cuanto mayor sea la distancia entre los puntos y la curvatura de la línea. La curvatura será en general variable y por eso suelen hacerse poligonales de muchos segmentos. Aún así es el método más sencillo y generalmente empleado como una primera aproximación imprecisa.

La solución más refinada y precisa consiste en hacer saltos de t que provoquen un salto entero máximo de una unidad en x o en y. Para calcular el incremento de t que hará que el incremento de x sea uno, es necesario conocer la derivada $x'=dx/dt$, de modo que si queremos $|dx|=1 \Rightarrow |dt|=1/|x'|$. Del mismo modo, para encontrar el incremento de t que provoque un salto unitario de y se necesita $y' = dy/dt$. Se calcula el salto de t que haga que el máximo, de entre ambos saltos de coordenada, x o y, sea uno.

---

[1] Repasar la definición matemática de función ($R \rightarrow R$) continua en un punto x_0: $\forall \varepsilon>0, \exists \delta>0 / 0<|x-x_0|<\delta \Rightarrow |f(x)-f(x_0)|<\delta$. Pensemos en el volumen del equipo de música y el giro del dial: si quiero subir o bajar el volumen menos que ε, lo logro girando el dial menos que δ. Que eso no suceda implica que hay un ε sin su δ: por mas poquito que gire el dial el volumen pega un salto (discontinuo) y no puedo acotarlo a ε, si el potenciómetro de volumen es defectuoso puede pasar eso en algún "punto" y decimos que la función volumen=*f*(ángulo) es discontinua en ese punto.

Esa definición para funciones reales de variable real, no tiene sentido para funciones discretas o de variables discretas. La idea es que la función varía poco cuando la variable varía poco. Para valores discretos, si la "cercanía" y "proximidad" están bien definidas, se puede usar el término "contigüidad" para definir dos valores próximos o juntos. Si la variable discreta se puede poner en correspondencia con los números enteros la contigüidad se da cuando la diferencia es uno.

$dt = \min(1/|x'|,1/|y'|) = 1/\max(|x'|,|y'|)$

Esto es: cuando la curva es de tendencia horizontal ($x' > y'$, x varía mas que y para un mismo dt) el algoritmo se mueve de a un paso en x y si es de tendencia vertical, se mueve de a un paso en y.

Este es el mecanismo (DDA o *Digital Differential Analizer*) empleado en la mayoría de los casos en los que es factible. El nombre proviene de los antiguos aparatos diferenciadores mecánicos.

Supongamos (o hagamos que) $t_0<t_1$. El algoritmo calcula las derivadas en t_0 y pinta o marca un punto en $\{x_0=int(x(t_0)+.5), y_0=int(y(t_0)+.5)\}$. Luego, con la variable de mayor derivada, digamos x, calcula si debe avanzar o retroceder una unidad, esto lo dictamina el signo de x', dando $dx=\pm1$. Con $dt=1/|x'|$ calcula todo de nuevo en $t+dt$, hasta llegar a t_1:

```
int redondea(float a) {return int(a+.5);}
void intercambia (float &a, float &b) {float tmp=a; a=b; b=tmp;}
void evalua_curva(float t, float &x, float &y, float &dx, float &dy){
  // de acuerdo a la curva en cuestión evalúa x e y en t y las derivadas
  // ej: elipse de semiejes 100 y 200 centrada en 500,300
  x=100*cos(t)+500; y=200*sin(t)+300; dx=-100*sin(t); dy=200*cos(t);
}
void curvaDDA(float t0, float t1) {
  if (t1<t0) intercambia(t0,t1);
  float t=t0,x,y,dx,dy;
  do{
    evalua_curva(t,x,y,dx,dy); pinta(redondea(x),redondea(y));
    t+=1/max(fabs(dx), fabs(dy));
  } while(t≤t1);
}
```

El algoritmo DDA se utiliza en la mayoría de los casos en los que la curva viene dada en forma de polinomio (por ejemplo NURBS) o es fácilmente diferenciable. En el resto de los casos (ej: curvas fractales o definidas como límite de subdivisiones) no hay más remedio que calcular una sucesión apropiada de (muchos) puntos.

Otro algoritmo pensable consiste en pintar el primer y ultimo punto de la curva, si el intervalo en x o y es mayor que la unidad, entonces se pinta el punto de parámetro intermedio ($t=(t_0+t_1)/2$) y se continúa de manera recursiva en cada mitad. El problema del esquema de divisiones binarias es que puede que haya curva entre dos puntos a distancia uno (o cero); basta considerar una circunferencia completa para ver el problema. Si se verifica la distancia al punto medio, tampoco hay garantías (ej: un 8).

Rasterización de Segmentos Rectilíneos

Los métodos antes planteados se simplifican si las derivadas x' e y' son constantes, es decir: para segmentos rectilíneos. Analizaremos el algoritmo DDA y una variante más eficiente de Bresenham.

Algoritmo DDA

En este caso, una de las dos coordenadas hará las veces de parámetro: se calcula la coordenada menos variable en función de la otra. El algoritmo para trazar un segmento rectilíneo entre $\{x_1,y_1\}$ y $\{x_2,y_2\}$ queda así:

```
void linea_DDA(float x1, float y1, float x2, float y2) {
    float dx=x2–x1, dy=y2–y1, m;
    if (redondea(dx)==0 && redondea(dy)==0) {pinta(redondea(x1), redondea(y1)); return;}
    if (fabs(dx)>=fabs(dy)){ // horizontal
        if (dx<0) {intercambia(x1,x2); intercambia(y1,y2); dx=–dx; dy=–dy;} // x debe avanzar
        for (m=dy/dx, y=y1, x=x1; x<= x2; x++, y+=m) pinta(redondea(x), redondea(y));
    }
    else { // vertical
        if (dy<0) {intercambia(x1,x2); intercambia(y1,y2); dx=–dx; dy=–dy;} // y debe avanzar
        for (m=dx/dy, x=x1, y=y1; y<=y2; y++, x+=m) pinta(redondea(x), redondea(y));
    }
}
```

Este algoritmo (y sus variantes) se encuentra en web como "*DDA line algorithm*".

Lo interesante es que va sumando diferenciales fijos. Tiene una sola división (por dx) al inicializar el lazo, sin peligro de división por cero y no hay ninguna división ni multiplicación en el interior. Esas características lo hacen muy eficiente para la mayoría de las situaciones.

Una obvia mejora consiste en evitar uno de los redondeos en el lazo, el de la variable que se incrementa de a uno, pues la parte fraccional será siempre igual. Por ejemplo para x:

```
for (m=dy/dx, y=y1, x=redondea(x1); x<=redondea(x2); x++, y+=m) pinta(x, redondea(y));
```

El otro redondeo (en y) es inevitable pues m es una variable real.

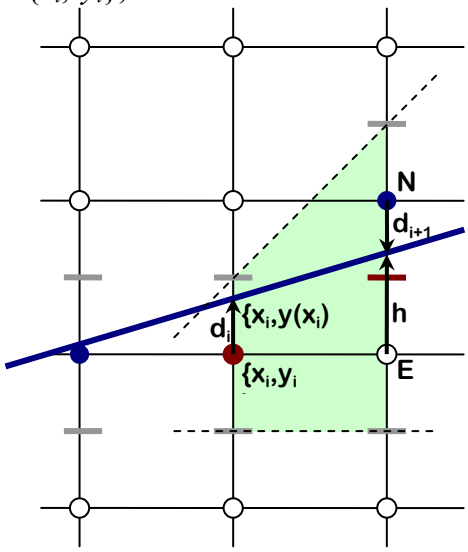
Algoritmo de Bresenham o de Punto Medio

Es el algoritmo DDA pero con una mejora que consiste en utilizar solo aritmética de enteros. Para deshacernos de las divisiones hay que multiplicar todo por 2 (para el redondeo) y por el denominador de la pendiente (dx o dy). Luego hay que procurar que todos los valores sean enteros.

En la figura los píxeles están representados como pequeños círculos centrados en sus coordenadas.

Consideremos el caso de referencia: $0 \le dy \le dx$. Pintamos los píxeles de a uno, incrementando x. Acabamos de pintar el i–ésimo, píxel de coordenadas $\{x_i, y_i\}$ enteras.

Con la pendiente entre cero y uno y dado que se pintó el píxel $\{x_i, y_i\}$, la zona sombreada es el conjunto de lugares donde puede estar la línea.



El .5 que usamos para redondear la variable y, equivale gráficamente a preguntar si la línea pasa por encima o por debajo del punto medio entre dos píxeles.

Usaremos la variable real d_i para indicar el desplazamiento vertical de la recta respecto al píxel recién pintado, h será el desplazamiento siguiente respecto al mismo valor de y. Si $h > .5$ pintaremos el píxel NE (noreste) y si no pintaremos el píxel E. Dado que x aumenta en una unidad: $h = d_i + dy/dx$.

La variable d_{i+1} es la diferencia entre el valor de y (real) de la recta y el valor de y (entero) en el píxel efectivamente pintado, se representó así porque está pintado el píxel NE, si se hubiese pintado el píxel E, d_{i+1} tendría el mismo valor que h, pero en NE, el valor es $h - 1$:

NE: $y_{i+1} = y_i + 1 \Rightarrow d_{i+1} = (y(x_{i+1}) - y_{i+1}) = (y(x_{i+1}) - (y_i + 1)) = (y(x_{i+1}) - y_i) - 1 = h - 1$.

El test fundamental consiste en averiguar si h es o no mayor que .5 y, dependiendo del resultado, las variables se actualizan de un modo o de otro.

$$¿h = d_i + dy/dx > .5? \begin{cases} \text{Si: pintar NE y hacer:} & d_{i+1} = h - 1 = d_i + dy/dx - 1 \\ \text{No: pintar E y hacer:} & d_{i+1} = h = d_i + dy/dx \end{cases}$$

Para deshacernos del 0.5 (1/2) y de las divisiones por dx, tanto en la consulta como en la actualización de datos, multiplicamos todo por 2 dx y reagrupamos:

$$¿2\,dx\,d_i + 2\,dy - dx > 0? \begin{cases} \text{Si: pintar NE y hacer:} & 2\,dx\,d_{i+1} = 2\,dx\,d_i + 2\,dy - 2\,dx \\ \text{No: pintar E y hacer:} & 2\,dx\,d_{i+1} = 2\,dx\,d_i + 2\,dy \end{cases}$$

Ya no hay divisiones, pero las variables dy, dx y d aun son reales. Las variables serán:

$D_i = 2\,dx\,d_i + 2\,dy - dx$,	la variable de decisión, que se compara con 0, y se incrementa con
$E = 2\,dy$,	y así queda si pintamos el píxel E, pero se le resta
$NE = 2\,dx$,	cuando pintamos el píxel NE.

Se suele hacer la simplificación que consiste en redondear las posiciones de los puntos inicial y final, y considerar nulo el primer error: $d_0 = 0 \Rightarrow D_0 = 2\,dy - dx$, luego D_0 es entero y se incrementa en cantidades enteras. Las variables y las operaciones son ahora todas enteras.

El error del redondeo inicial genera una desviación de la línea que será tanto menor cuanto mas larga sea la línea. No se pierde mucha eficiencia porque el redondeo se realiza una sola vez, al principio, y no dentro del lazo.

Desarrollamos solo el caso de referencia ($|dx| \geq |dy|$ y x crece), el resto se obtiene por simetrías:

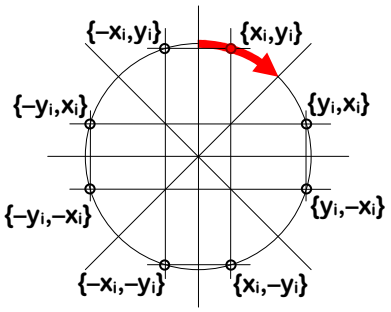
```
void linea_Bresenham(int x1, int y1, int x2, int y2) {
    int dx=x2-x1, dy=y2-y1, x,y,NE,E,D,xstep,ystep;
    if (!(dx|dy)) {pinta(x1,y1); return;}
    if (abs(dx)>=abs(dy)) {
        if (dx>=0) {
            x=x1; y=y1; NE=dx<<1; // (<<1 = *2, shift de un bit)
            ystep=1; if (dy<0) {ystep=-1; dy=-dy;}
            E=dy<<1; D=E-dx;
            while (x<x2) {
                pinta(x,y);
                if (D>0) {y+=ystep; D-=NE;}
                x++; D+=E;
            }
        }
    }
    pinta(x, y); // punto final
}
void linea_Bresenham(float x1, float y1, float x2, float y2) {
    linea_Bresenham(redondea(x1), redondea(x2), redondea(y1), redondea(y2));
}
```

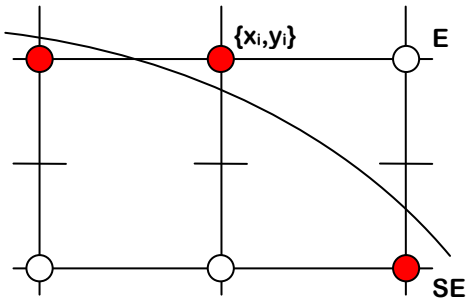
Como puede verse, todas las operaciones sensibles se realizan en enteros y solo hay multiplicaciones por 2 (corrimiento de un bit hacia la izquierda) fuera del lazo.

Este es el algoritmo que emplean las placas gráficas para representar los segmentos rectilíneos.

Rasterización de Circunferencias

Como primera observación: el círculo tiene simetría central, que se aprovecha utilizando el $\{0,0\}$ como centro y rasterizando solamente un octavo del círculo. ¿Por que un octavo, y no una porción menor? Porque las simetrías que produce son sólo cambios de signo e intercambio de coordenadas, que transforman enteros en enteros; como puede verse en la figura siguiente.





Se calcula el tramo que va de 90º a 45º, representado en el dibujo mediante un punto genérico de coordenadas $\{x_i, y_i\}$. Los puntos simétricos se pintan con el mismo par de enteros intercambiados o cambiados de signo. Al pintar los puntos hay que trasladarlos del origen al centro real.

Iremos directamente al método del punto medio con el algoritmo de Bresenham. La curva es de tendencia horizontal, con x creciente e y decreciente; haremos que x aumente siempre una unidad. Hay que averiguar si el próximo punto medio está dentro o fuera de la circunferencia. Si el punto medio está dentro pintamos el píxel E y si esta fuera pintamos el SE, como se muestra en la figura.

Un punto cualquiera $\{x, y\}$ está fuera del círculo si $x^2 + y^2 > r^2$. Habrá que analizar si el próximo punto medio (en $x_i + 1$ e $y_i - .5$, dado que x crece e y decrece) está o no dentro de la circunferencia:

$$¿(x_i + 1)^2 + (y_i - .5)^2 - r^2 > 0? \begin{cases} \text{Si: pintar SE} \\ \\ \text{No: pintar E} \end{cases}$$

La variable de decisión es:

$d_i = (x_i + 1)^2 + (y_i - 1/2)^2 - r^2 = x_i^2 + 2\,x_i + 1 + y_i^2 - y_i + 1/4 - r^2$

Si resulta pintado E, el nuevo valor será

$d_{i+1} = (x_i + 2)^2 + (y_i - 1/2)^2 - r^2 = x_i^2 + 4\,x_i + 4 + y_i^2 - y_i + 1/4 - r^2 = d_i + 2\,x_i + 3$

Si, en cambio resulta pintado SE, será:

$d_{i+1} = (x_i + 2)^2 + (y_i - 3/2)^2 - r^2 = x_i^2 + 4\,x_i + 4 + y_i^2 - 3\,y_i + 9/4 - r^2 = d_i + 2\,x_i - 2\,y_i + 5$

Podemos resumirlo fijando los incrementos:

$E_i = 2\,x_i + 3$

$SE_i = 2\,(x_i - y_i) + 5$

Asumiremos además, para simplificar las cosas, que el radio y las coordenadas del centro ya están redondeados o son números enteros. Así, partimos del punto de coordenadas $\{0, r\}$ y el primer d_i será:

$d_0 = 0^2 + 2 \times 0 + 1 + r^2 - r + 1/4 - r^2 = 5/4 - r$

Para evitar el valor fraccional, cambiamos la variable d por una variable h:

$h = d - 1/4 \qquad \Rightarrow h_0 = d_0 - 1/4 = 1 - r;$

Dado que $d > 0 \Leftrightarrow h > -1/4$, la comparación se puede hacer con $-1/4$, pero dado que el primer valor y los incrementos son enteros, preguntar si es mayor que $-1/4$ es lo mismo que preguntar si es mayor o igual que cero.

Este algoritmo ya es bastante eficiente, pero puede mejorarse. Vemos que la variable de decisión depende cuadráticamente de x e y, mientras que los incrementos son lineales, esto es porque se trata de una curva de segundo grado. El incremento de los incrementos (diferencias de segundo orden) es constante y con ello podemos evitar las multiplicaciones por dos.

Cuando pasamos hacia el píxel E, los nuevos incrementos serán:

$E_{i+1} = 2\,(x_i + 1) + 3 = 2\,x_i + 3 + 2 = E_i + 2 \quad \Rightarrow \quad EE = 2$

$SE_{i+1} = 2\,(x_i + 1 - y_i) + 5 = 2\,(x_i - y_i) + 5 + 2 = SE_i + 2 \quad \Rightarrow \quad ESE = 2$

En cambio, si pasamos a SE, serán:

$E_{i+1} = 2\,(x_i + 1) + 3 = 2\,x_i + 3 + 2 = E_i + 2$ (no cambia nada) $\quad \Rightarrow \quad SEE = 2$

$SE_{i+1} = 2\,(x_i + 1 - (y_i - 1)) + 5 = 2\,(x_i - y_i) + 5 + 4 = SE_i + 4 \quad \Rightarrow \quad SESE = 4$

Para el punto de partida: $\{0, r\}$, los valores iniciales son:

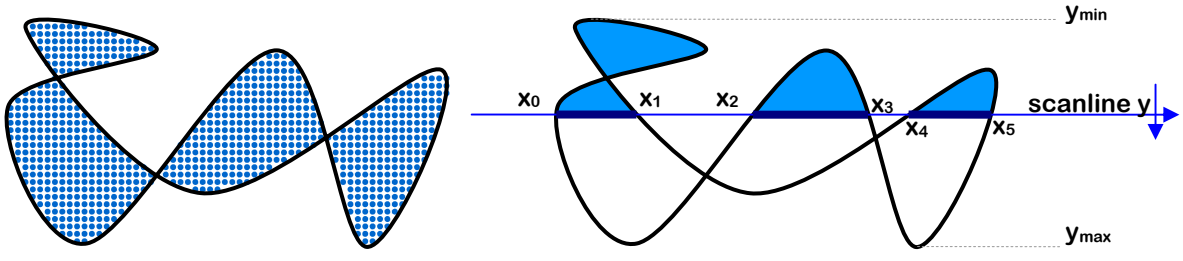
$E_0 = 2 \times 0 + 3 = 3$

$SE_0 = 2\,(0 - r) + 5 = 5 - 2\,r$

Estos cambios complican un poco el código pero lo hacen más eficiente.

Rasterización de Figuras Cerradas

Rasterizar una figura cerrada consiste en pintar los píxeles que queden en el interior. El obvio problema es la determinación eficiente del conjunto de píxeles interiores.

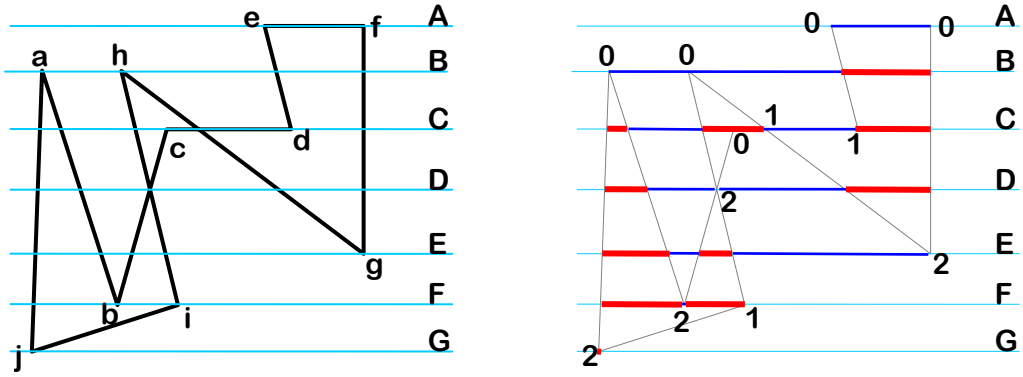


Un algoritmo razonable consiste en rasterizar primero la frontera, manteniendo para cada valor de y una secuencia ordenada con las coordenadas x de los píxeles pintados: $X(y) = \{x_0, x_1, x_2, x_3, \ldots, x_{n-1}\}$. Luego, para cada y entre $y_{min}+1$ e $y_{max}-1$, se recorre la lista $X(y)$, pintando los segmentos impares:

```
y=ymin+1; j=1; while(y<ymax) {
  i=0; while(i<n-1) {
    x=X[j][i]+1; while(x<X[i+1]){pinta(x,y); x++;} i+=2;
  }
  y++; j++;
}
```

El recorrido por sucesivas líneas o *scan−lines* asemeja el barrido horizontal de los viejos monitores y televisores tipo CRT o tubo de rayos catódicos. Ya lo veremos con detalle, pero el *buffer* de memoria de la imagen se almacena del mismo modo: la matriz de píxeles se organiza como una sucesión de líneas horizontales. Con un algoritmo que barre *scan−lines* se optimiza la memoria *cache* del sistema.

Este método permite pintar figuras cóncavas y auto interceptadas, pero no está exento de algunos problemas que pueden verse analizando la siguiente figura.

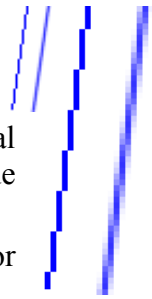


Para solucionar el problema de los tramos horizontales, simplemente no se agregan a la lista. Al rasterizar la frontera, los pasos hacia el Este o hacia el Oeste no se agregan a la lista X.

Los vértices sólo se cuentan cuando corresponden al y mínimo (estricto) del segmento rasterizado y además, se permite repetir valores de x. Es decir que estarán duplicados en la lista los puntos como b, j y g, que son mínimo de dos segmentos; puntos como el i y el d se cuentan solo una vez y puntos como a, c, e, f y el h no están incluidos. Los puntos de auto intersección no reciben tratamiento especial, la intersección $b - c$ / $h - i$ está duplicada; en cambio $c - d$ no cuenta, de modo que su intersección con el segmento $g - h$ está una sola vez. Un x duplicado puede significar entrada/salida o salida/entrada.

Cualquier curva suave puede ser tratada como una poligonal de segmentos de un píxel de longitud.

Antialiasing



En la teoría del muestreo, al tomar una secuencia discreta de datos de una señal continua, se denomina "*aliasing*" a la falsa interpretación de una frecuencia por otra. Aquí, en Computación Gráfica, es el efecto de serruchado de las imágenes rasterizadas.

Una línea es un objeto unidimensional, sin espesor; pero para visualizarlas les damos al menos un píxel de espesor. En la asignación de los píxeles a pintar se comete un error de aproximación que es más notable cuando la línea es casi horizontal o casi vertical.

Hay muchas técnicas para atenuar el defecto visual. En general se basan en mezclar el color de la línea con el del fondo.

Hay una técnica muy simple basada en el factor de cubrimiento del píxel (*antialiasing by area averaging*) se conoce como algoritmo de línea de Xiaolin Wu: En el algoritmo de Bresenham, el desplazamiento d_i nos brinda un indicador de la proporción de línea que pasa por el píxel. Suponiendo que $dx > dy > 0$, se pintan E y NE, pero con una mezcla proporcional entre color de línea y el fondo. Otra técnica consiste en pintar mas de una vez la línea, primero centrada y con el color asignado y luego a los lados con el color atenuado o mezclado con el color de fondo.

El problema se presenta también con los bordes de las figuras o aun con las imágenes (texturas). En general, el *antialiasing* está mal o pobremente implementado y resulta muy complicado hacerlo en forma aceptable. Si bien toda la teoría es de los años 80 y 90, los fabricantes de placas aun compiten con mejores y más eficientes técnicas de *antialiasing*. Un evidente problema es que se mezcla el color con el fondo y el fondo puede cambiar a medida que se van graficando otros objetos.

En general conviene suavizar la imagen final. Los mejores resultados se obtienen rasterizando en una imagen mayor (2, 4 u 8 veces mayor) y luego reduciendo la imagen para mostrar. Esta técnica se denomina *supersampling* y *full scene antialiasing*.