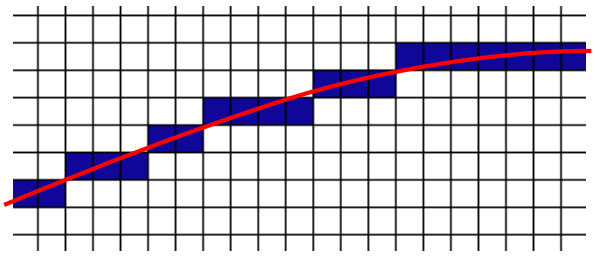
Rasterización (Scan-Conversion)



Aquí trataremos el problema de representar un objeto continuo, analítico y descripto en forma **vectorial**, mediante un conjunto discreto y finito de puntos. Los objetos modelados (curvas y superficies) son continuos, pero se representan mediante una imagen discreta que se transfiere a un monitor u otro dispositivo ***raster*** como un conjunto discreto de pixeles.

Comenzaremos por las curvas planas. En computación gráfica se representan generalmente en forma paramétrica, mediante dos funciones reales y continuas[1]: $\{x(t), y(t)\}$, de un parámetro real $t \in [t_0, t_1]$.

Los dispositivos vectoriales o analógicos pueden asumir que el parámetro *t* representa el tiempo y mover dos motores independientes para trazar la línea continua 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 dibujo o de corte sobre autoadhesivos, utilizados en carteles y gigantografías. En la práctica, con motores "paso a paso" (*stepper*) se aproximan las curvas por pequeños saltos finitos, de acuerdo a la precisión de los motores. Pero, aun así, al ir de una posición discreta a la siguiente, el marcador sigue marcando o cortando y realiza un trazo continuo. También existen imágenes vectoriales en algunos programas de creación y edición que manipulan y guardan una forma vectorizada de la geometría (plana) que se manipula. Así funcionan Corel Draw o Inkscape, por ejemplo, y así se definen también las fuentes *true-type*, mediante curvas polinómicas, todo esto permite ampliar la imagen o el texto y recalcular la imagen, para que no "se vean" pixeles grandes, pero están pixeladas.

La mayoría de los equipos actuales de inyección de tintas (*plotters* e impresoras) un método discreto o *raster* que consiste en eyectar microgotas de tinta en puntos aislados de un papel que se recorre línea por línea horizontal (***scan–lines***). Lo mismo sucede en los monitores, que constan de una matriz de puntos luminosos, con tres colores por píxel, que se refrescan o alteran barriendo líneas horizontales en orden. Una imagen grabada se manipula como una matriz rectangular de pixeles de color, pero la memoria que la almacena está organizada como una sucesión unidimensional de tres o cuatro valores (RGBA) por pixel, completando una línea horizontal a continuación de la anterior, igual que las *scan–lines*.

Al proyectar una curva 3D sobre el plano de la imagen obtenemos una curva plana. Estudiaremos cómo se define una sucesión de puntos que la represente en forma **adecuada**, entre otros requisitos pediremos que el error sea menor que medio pixel. Consideraremos que la escala es tal que la distancia entre dos pixeles horizontales o verticales es una unidad. En todo lo que sigue, vamos a suponer que la curva no tiene grandes variaciones en una unidad, es decir que entre un pixel y el siguiente la curva es prácticamente recta, no zigzaguea entre pixeles contiguos; en caso contrario, cualquier proceso daría un mal resultado, pues no podríamos "muestrear" esa curva con una precisión del orden de un píxel.

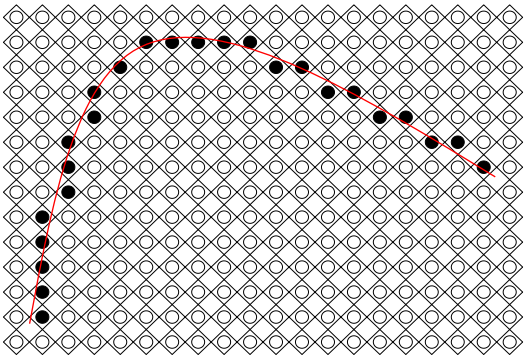
Las curvas *rasterizadas* se representan mediante una sucesión finita de puntos **contiguos** de coordenadas enteras y de modo que no haya espacio entre dos puntos sucesivos de la línea: $\boldsymbol{max(|\Delta x|, |\Delta y|)=1}$. Una sucesión de puntos contiguos suele denominarse "*water tight*",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

Si bien una curva es un objeto matemático unidimensional, se representa, como mínimo, con un pixel de ancho. Por lo tanto, no se pueden pintar todos los cuadraditos/pixeles que atraviesa la línea, porque veríamos una curva de ancho variable. Exigiremos que las curvas de **tendencia horizontal** o ***x–major*** o **(pendiente ≤ 1)** tengan un pixel de alto y las de tendencia vertical (*y–major*) uno de ancho.

Por simplicidad hablaremos de "pintar un pixel", pero en realidad el proceso de rasterización consiste en generar los fragmentos de primitiva que se corresponden con los pixeles de la imagen final y asignarle datos interpolados de los vértices. Un fragmento contiene varios datos asociados, además del color, y no necesariamente llegará a ser un pixel en la imagen. Diremos "pintar un pixel" en lugar del correcto "generar un fragmento en la posición correspondiente a un pixel y asignarle las variables".

---

[1] Recordar la definición de función $\mathbb{R} \rightarrow \mathbb{R}$ continua: $f(x)$ es continua en $x_0 \Leftrightarrow \forall\ \varepsilon > 0, \exists\ \delta > 0 / |x-x_0| < \delta \Rightarrow |f(x)-f(x_0)| < \varepsilon$. Para subir o bajar un poquito (menos que ε) la temperatura de la ducha, giro un poquito (menos que δ) la llave de agua fría. Si no puedo, si por más poquitito que gire la llave, me quemo o me congelo, es que el ε de temperatura no tiene un correspondiente δ de la llave; decimos que la función temperatura=*f*(ángulo) es discontinua en ese punto. Esa definición es para funciones reales de (una) variable real, pero no tiene sentido para funciones o variables discretas.



OpenGL, en particular, especifica así las condiciones que debe cumplir el resultado (aunque solo acepta segmentos de recta): Imagínese un diamante (cuadrado a 45º) alrededor de cada centro de pixel, con sus vértices en los puntos medios entre pixeles vecinos. Se genera el fragmento si y solo si la curva **sale** del diamante. Si pasa justo por un vértice, entre dos vecinos verticales se elige el inferior y si es entre dos horizontales se elige el izquierdo. La salida es porque exige que no se genere el fragmento final, de modo que para una secuencia de líneas ningún fragmento se duplique.

Analicemos cómo definir qué píxeles deben pintarse (que fragmentos deben generarse).

Si a **cada punto** de la curva lo asociamos al punto más cercano de coordenadas enteras, nos queda el conjunto $\{int(x(t)+0.5),\ int(y(t)+0.5)\}$ (o $\{round(x(t)),\ round(y(t))\}$). Éste es un conjunto contiguo, discreto y finito de pares, que representa muy bien a la curva. 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**) y entonces debemos encontrar una formulación finita.

Algoritmo diferencial o DDA: La solución consiste en hacer saltos finitos del parámetro t que generen fragmentos contiguos, el cambio máximo de x o y debe ser una unidad: Δt t.q. Max $(|\Delta x|,|\Delta y|) = 1$.

Podemos suponer que la curva es prácticamente recta en un pixel, entonces podemos aproximar incrementos mediante diferenciales (Aproximación de Taylor de primer orden). Si conocemos la derivada paramétrica $\dot{x} = dx/dt$ y queremos lograr que $|\Delta x| \leq 1 \Rightarrow$ hacemos un salto $|\Delta t| \leq |\Delta x|/|\dot{x}| = 1/|\dot{x}|$. Del mismo modo, para limitar el cambio de y se necesita $\dot{y} = dy/dt$. Se calcula, entonces, el salto de t que haga que el máximo cambio de coordenada, x o y, sea uno:

$dt = \min(1/|\dot{x}|,1/|\dot{y}|) = 1/\max(|\dot{x}|,|\dot{y}|)$

Esto es: Cuando la curva es de tendencia horizontal $\dot{x} > \dot{y}$; entonces, para un mismo dt x varía más que y y el algoritmo se mueve de a un paso en x. Si es de tendencia vertical, se mueve de a un paso en y.

Supongamos (o hagamos que) $t_0 < t_1$. El algoritmo calcula las derivadas en t_0 y pinta el primer punto, redondeando las coordenadas: $\{x_0=\text{int}(x(t_0)+.5),\ y_0=\text{int}(y(t_0)+.5)\}$. Luego, con la variable de mayor derivada, digamos x, calcula si debe avanzar o retroceder una unidad, esto lo dictamina el signo de $\dot{x}$, dando $dx=\pm1$. Con $dt=1/|\dot{x}|$, se calcula todo de nuevo en $t+dt$, y así hasta llegar a t_1:

```
void intercambia (float &a, float &b) {float tmp=a; a=b; b=tmp;} // rutina accesoria para intercambiar variables
void evalua_curva(float t, float &x, float &y, float &dx, float &dy){ // coordenadas y derivadas en t
    // 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 con t como ángulo:
    float c=cos(t), s=sin(t);
    x=100*c+500; y=200*s+300; dx=-100*s; dy=200*c;
}
void curvaDDA(float t0, float t1) {
    if (t1<t0) intercambia(t0,t1); // t0 debe ser menor o igual que t1
    float t=t0,x,y,dx,dy;
    do{
        evalua_curva(t,x,y,dx,dy); pinta(round(x),round(y));
        t+=1/max(fabs(dx), fabs(dy)); // debería garantizarse que no divida por cero!!!
    } while(t<t1);
}
```

El nombre (DDA, por *Digital Differential Analizer*) proviene de los antiguos aparatos mecánicos que medían las derivadas, haciendo pasar una ruedita por la curva.

Existen dos fuentes de problemas potenciales que pueden llevar a elegir otro método:

– No es sencillo garantizar que no haya división por cero, pero en general puede (y debe!!) hacerse.

– No siempre se puede saber si la curva es suficientemente suave (Taylor: error = $\max(\ddot{x},\ddot{y})/2$) por lo tanto también hay que garantizar que no haya dejado pixeles discontiguos.

Aun así, este es el algoritmo empleado cuando se conocen la ecuación y sus derivadas en forma paramétrica y la curva es suficientemente suave; eso comprende todos los casos prácticos y los que se utilizan en Computación Gráfica.

Rasterización de Segmentos Rectilíneos

Algoritmo DDA para segmentos rectos

En un segmento recto, el algoritmo DDA se simplifica mucho, porque las derivadas $\dot{x}$ e $\dot{y}$ son constantes. Se puede usar como parámetro a la coordenada que varíe más, la que define la tendencia.

Para trazar un segmento rectilíneo entre $\{x_1, y_1\}$ y $\{x_2, y_2\}$, el algoritmo queda así:

```
void linea_DDA(float x1, float y1, float x2, float y2) {
    float dx=x2–x1, dy=y2–y1;
    if (fabs(dx)>=fabs(dy)){ // horizontal (>=, ==> ambos pueden ser 0)
        int x=round(x1), y=round(y1), x2=round(x2);
        pinta(x,y); if (x2==x) return; // pinta uno y sale si dx ~0
        float m=dy/dx; // dx no es nulo pero puede ser muy chico
        if (dx>0) while((++x)<x2) {y+=m; pinta(x,round(y));}
        else while((--x)>x2) {y+=m; pinta(x,round(y));}
    }
    else { // vertical (|dy|>|dx| ==> dy no es nulo)
        int x=round(x1), y=round(y1), y2=round(y2); pinta(round(x),y);
        float m=dx/dy;
        if (dy>0) while((++y)<y2) {x+=m; pinta(round(x),y);}
        else while((--y)<y2) {x+=m; pinta(round(x),y);}
    }
}
```

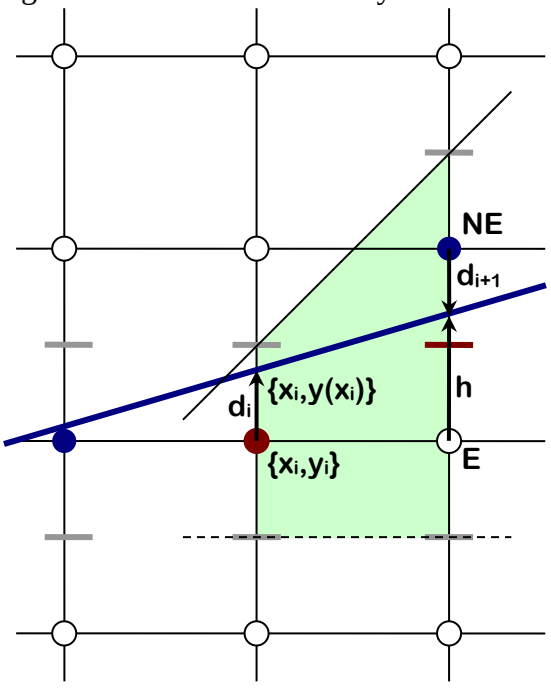
Este algoritmo (y sus variantes) se encuentra en web como "*DDA line algorithm*".

Lo interesante es que va sumando una pendiente fija. Tiene una sola división (por d*x* o d*y*) al inicializar el lazo, evitando fácilmente la división por cero y no hay ninguna división ni multiplicación en el interior del lazo. El redondeo de la variable que se incrementa se hace solo al principio; pues la parte fraccional será siempre igual; el otro redondeo es inevitable pues *m* es una variable real. Estas características lo hacen muy eficiente para la mayoría de las situaciones.

Algoritmo de Bresenham o de Punto Medio, para segmentos rectos

El algoritmo debido a Bresenham (1962) es el algoritmo DDA, pero con una mejora que consiste en utilizar aritmética de enteros. Se puede reformular el algoritmo anterior, pero para "ver" la geometría en vez del álgebra, partiremos desde cero con una interpretación geométrica de las variables y los saltos.

En la figura, los pixeles están representados como pequeños círculos centrados en sus coordenadas {x,y} enteras.



Consideremos el caso de referencia: $0 \le \Delta y \le \Delta x$. Pintamos los pixeles de a uno, incrementando *x*. Acabamos de pintar el *i*–ésimo, pixel en $\{x_i, y_i\}$, pues la recta pasa por $\{x_i, y(x_i)\}$. Nos preguntamos cual píxel hay que pintar ahora.

Sabiendo solamente que se pintó el pixel $\{x_i, y_i\}$ de una recta con pendiente entre cero y uno; la zona indicada es el conjunto de lugares que puede seguir la línea (analizarlo).

El redondeo equivale gráficamente a preguntar si la línea (de tendencia horizontal) pasa por debajo o por encima del punto medio entre dos pixeles (≡ regla de los diamantes). ¿Pintamos el pixel E (Este) o el NE (Noreste)?

La variable real $d_i = y(x_i) - y_i$ es el "error" vertical cometido, el desplazamiento de la recta respecto al pixel recién pintado; mientras que $h = y(x_i+1) - y_i = d_i + \Delta y/\Delta x$, también real, mide el próximo desplazamiento respecto al mismo y_i. El próximo error no puede ser mayor que medio pixel. Entonces, si $h \le 1/2$ se pinta el pixel E, provocando un error $d_{i+1} = h$; si $h > 1/2$ se pinta el NE, con error $d_{i+1} = y(x_i+1) - (y_i + 1) = (y(x_i+1) - y_i) - 1 = h - 1$ (negativo).

El test fundamental consiste, entonces, en averiguar si *h* es o no es mayor que 1/2 y, dependiendo del resultado, las variables se actualizan de un modo o de otro.

$$h_i = d_i + \Delta y/\Delta x > 1/2? \begin{cases} \text{Si: pintar NE y hacer: } d_{i+1} = h - 1 = d_i + \Delta y/\Delta x - 1 \\ \text{No: pintar E y hacer: } d_{i+1} = h = d_i + \Delta y/\Delta x \end{cases}$$

Para deshacernos del 1/2 y de las divisiones por Δx, tanto en la consulta como en la actualización de datos, **multiplicamos <u>todo</u> por $2\Delta x$, el múltiplo común de todos los denominadores**:

$$2\Delta x\, d_i + 2\Delta y > \Delta x? \begin{cases} \text{Si: pintar NE y hacer: } 2\Delta x\, d_{i+1} = 2\Delta x\, d_i + 2\Delta y - 2\Delta x \\ \text{No: pintar E y hacer: } 2\Delta x\, d_{i+1} = 2\Delta x\, d_i + 2\Delta y \end{cases}$$

Reagrupando y usando una variable de consulta D_i (relacionada con h_i, que ya no usaremos):

$$D_i = 2\Delta x\, d_i + 2\Delta y - \Delta x > 0? \begin{cases} \text{Si: pintar NE y hacer: } D_{i+1} = D_i + 2\Delta y - 2\Delta x \\ \text{No: pintar E y hacer: } D_{i+1} = D_i + 2\Delta y \end{cases}$$

Ya no hay divisiones. Las variables necesarias son:

$D_i = 2\,\Delta x\, d_i + 2\,\Delta y - \Delta x$,	la variable de decisión, que se compara con 0 y se incrementa con
$E = 2\,\Delta y$,	para cualquier caso y así queda si pintamos el pixel E; pero se le resta
$NE = 2\,\Delta x$,	cuando pintamos el pixel NE.

Los incrementos E y NE, dependen de Δx y Δy pero no del punto, son números fijos que se calculan al principio y se van sumando dentro del loop; en lugar de tener que recalcular D; con más operaciones.

Las variables son reales, pero si **redondeamos las coordenadas de los puntos extremos** inicial y final, logramos que Δy y Δx sean enteros, de modo que la variable de decisión se incrementa de a saltos enteros. Solo nos queda real el primer error d_0, cometido al redondear el punto inicial y que, por ser menor que medio píxel, también podemos redondear a cero.

No habiendo divisiones ni variables reales, todo el algoritmo se realiza con operaciones en enteros.

El redondeo inicial es una desviación virtual de la línea en no más de medio pixel al inicio y al final, por lo tanto, en el medio, nunca habrá un desvío mayor que medio píxel. Sumado al medio pixel de error aceptado previamente, este algoritmo tiene un error total menor que un pixel, pero no puede mejorarse sin trabajar con números reales. La ventaja de las operaciones enteras es la que lo justifica.

Desarrollaremos solo el caso de referencia ($|\Delta x| \geq |\Delta y|$ 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;
    if (abs(dx)>=abs(dy)) { // tendencia horizontal
        if (dx>=0) { // x avanza
            if (dx==0) {pinta(x1,y1); return;} // solo aquí, para que pinte al menos un punto
            int ystep=1; if (dy<0) {ystep=-1; dy=-dy;} // ystep es si sube o baja y
            int x=x1, y=y1, NE=2*dx, E=2*dy, D=E-dx; // asumimos error inicial nulo => D=2dy-dx
            while (x<x2) { // notar el < y no <= (no se pinta el ultimo)
                pinta(x,y);
                if (D>0) {y+=ystep; D-=NE;} // NE solo se resta si y cambia: si pintamos NE (o SE, si ystep=-1)
                x++; D+=E; // siempre avanza x y siempre se agrega E
            }
        } else { … } // x retrocede
    } else { … } // tendencia vertical
}
void linea_Bresenham(float x1, float y1, float x2, float y2) { // si vienen extremos float los pasamos a enteros
    linea_Bresenham(round(x1), round(x2), round(y1), round(y2));
}
```

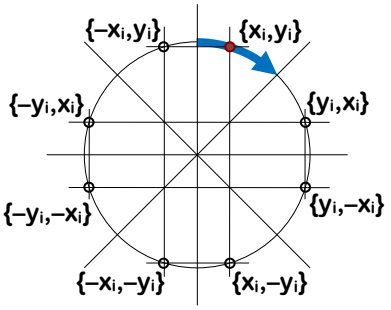
Como puede verse, multiplicando por los denominadores ($2\Delta x$) y redondeando los valores iniciales, se logró que no haya divisiones y que todas las operaciones sensibles se realicen en enteros. Solo quedaron multiplicaciones por 2 (≡ corrimiento de un bit hacia la izquierda: $\ll 1$) y fuera del lazo.

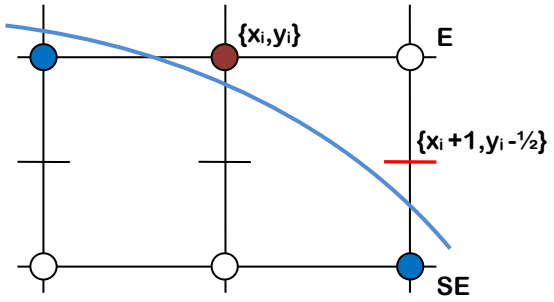
Este es el algoritmo que emplean las placas gráficas para rasterizar los segmentos rectilíneos.

Notar que pinta al menos un punto y no pinta el último. Así lo exige OpenGL, porque es preferible un error de un pixel a que una sucesión de segmentos rectos duplique fragmentos, generando problemas en las operaciones *per-pixel* como el *blending* (mezcla de colores).

Rasterización de Circunferencias

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





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 definen con el mismo par de enteros intercambiados y/o cambiados de signo. Al pintar los puntos hay que trasladarlos desde el origen al centro real.

Iremos directamente al método del punto medio con otro algoritmo debido a 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 pixel 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 $\{x_i + 1, y_i - .5\}$ (horizontal, x crece, y decrece), está o no dentro de la circunferencia:

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

Esa ecuación define la variable de decisión, que se compara con cero para pintar condicionalmente el pixel E o SE. Con eso ya alcanza para rasterizar la circunferencia; pero es una función cuadrática e involucra números reales. Podemos simplificarla:

$D_i = (x_i + 1)^2 + (y_i - 1/2)^2 - r^2 = x_i^2 + 2\,x_i + 1 + y_i^2 - y_i + 1/4 - r^2$ (D es función cuadrática de x e y)

Si resulta pintado E, el valor D_{i+1} futuro, en $\{x_i+1, y_i\}$, 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, en $\{x_i+1, y_i-1\}$, D_{i+1} 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 de la variable de decisión para el próximo paso:

$E_i = 2\,x_i + 3$ (los incrementos de D son

$SE_i = 2\,(x_i - y_i) + 5$ funciones lineales de x e y)

Para simplificar las cosas, asumimos que el radio y las coordenadas del centro están redondeados. Así, partimos del punto de coordenadas enteras $\{0, r\}$ y los valores iniciales son:

$D_0 = (0 + 1)^2 + (r - 1/2)^2 - r^2 = 1 + r^2 - r + 1/4 - r^2 = 5/4 - r$

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

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

Para evitar el valor fraccional 5/4 en D podríamos multiplicar todo por 4, pero ese 5/4 es constante, de modo que usaremos otro truco: cambiamos la variable D por una variable D':

$D' = D - 1/4 \quad \Rightarrow \quad D'_0 = D_0 - 1/4 = 5/4 - 1/4 - r \quad = 1 - r;$

Dado que $D > 0 \Leftrightarrow D' > -1/4$, la comparación se debería hacer con –1/4, pero el primer valor y los incrementos son enteros: En cada paso, D' será mayor que –1/4 si es mayor o igual que cero.

Ya conseguimos aritmética entera, pero los incrementos dependen de x e y; por lo tanto, hay que calcularlos en cada paso. Ahora, los incrementos son funciones lineales de x e y, pero sus propios incrementos son constantes, podemos calcularlos y sólo sumar. Calcularemos los **incrementos de los incrementos de la variable de decisión** para que dentro del *loop* principal solo haya sumas.

Cuando pasamos hacia el pixel 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:

$E_{i+1} = 2\,(x_i + 1) + 3 = 2\,x_i + 3 + 2 = E_i + 2$ (igual que al E) $\Rightarrow \quad SEE = 2$

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

Es decir que en el paso actual se actualizan D', E y SE, cada uno con su incremento precalculado. Los incrementos a aplicar dependen de que el próximo paso sea al E o al SE.

Estos cambios complican muy poco el código, pero lo hacen mucho más eficiente:

Inicial:		x = 0	y = r	D = 1 – r	E = 3	SE = 5 – 2 r
Loop: ¿D ≥ 0?	Si: pinta SE	x ++	y --	D += SE	E += 2	SE += 4
	No: pinta E	x ++		D += E	E += 2	SE += 2

El mismo estilo de cálculo de las segundas diferencias se puede utilizar para rasterizar todo tipo de funciones cuadráticas, que siempre tendrán una variable de decisión también cuadrática. En particular las elipses son las curvas cuadráticas más utilizadas en los programas de dibujo, siendo la circunferencia un caso especial. También se podrían utilizar terceras diferencias constantes para curvas de tercer grado como las polinómicas (NURBS y Bezier, que veremos en detalle más adelante) pero la falta de simetría y los cambios de tendencia horizontal/vertical hacen que no sea el método utilizado.

Nota: La finalidad de esta nota es mostrar la relación entre diferencias y diferenciales.

La función que define la variable de decisión es cuadrática en x e y. Por Taylor, podemos aproximar la diferencial por medio de las derivadas evaluadas en un punto intermedio. Elegimos el punto medio, del cual conocemos $x_m = x_i + ½$, pero no y_m que puede disminuir o no.

$D(x,y) = x^2 + 2\,x + 1 + y^2 - y + 1/4 - r^2 \qquad D_{i+1} - D_i \cong \partial_x D\,\Delta x + \partial_y D\,\Delta y$

$\partial_x D = 2x_m + 2 = 2(x_i + ½) + 2 = 2x_i + 3$ (x_m es siempre $x_i + ½$) $\qquad \partial_y D = 2y_m - 1$ (distinto y_m si pinta E o NE)

E ($\Delta x = 1, \Delta y = 0$): $E \cong (2x_i + 3)\,\Delta x + (2y_m - 1)\,\Delta y = (2x_i + 3)\,1 + (2y_i - 1)\,0 = 2x_i + 3 \qquad (y_m = y_i)$

SE ($\Delta x = 1, \Delta y = -1$): $SE \cong (2x_i + 3)\,\Delta x + (2(y_i - ½) - 1)\,\Delta y = 2x_i + 3 - 2y_i + 2 = 2(x_i - y_i) + 5 \qquad (y_m = y_i - ½)$

Lo mismo que hicimos para D podemos hacer para E y SE:

$E_{i+1} - E_i \cong \partial_x E\,\Delta x + \partial_y E\,\Delta y \qquad SE_{i+1} - SE_i \cong \partial_x SE\,\Delta x + \partial_y SE\,\Delta y$

$\partial_x E = 2 \qquad \partial_y E = 0 \qquad \partial_x SE = 2 \qquad \partial_y SE = -2$

EE ($\Delta x = 1, \Delta y = 0$): $EE \cong \partial_x E\,\Delta x + \partial_y E\,\Delta y = 2$

ESE ($\Delta x = 1, \Delta y = 0$): $ESE \cong \partial_x SE\,\Delta x + \partial_y SE\,\Delta y = 2$

SEE ($\Delta x = 1, \Delta y = -1$): $SEE \cong \partial_x E\,\Delta x + \partial_y E\,\Delta y = 2$

SESE ($\Delta x = 1, \Delta y = -1$): $SESE \cong \partial_x SE\,\Delta x + \partial_y SE\,\Delta y = 4$

Cabe aclarar que esta forma de hacer las diferencias y segundas diferencias no es la correcta para saltos enteros, sino la anterior: aritmética y exacta; aquí elegimos arbitrariamente la derivada en el punto medio y casualmente da valores correctos (porque la segunda derivada está acotada y la tercera es nula). En el Cálculo Numérico se enseñan estas técnicas de "Diferencias Finitas".

Algoritmo de subdivisión

Una curva paramétrica plana viene dada por dos funciones {x(t), y(t)} con un parámetro t real ∈ [t_a, t_b].

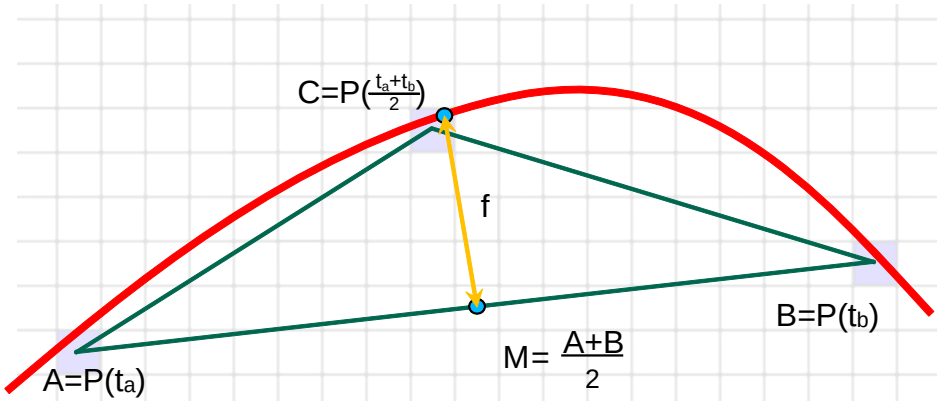
Este algoritmo consiste en encontrar una subdivisión del intervalo total $\Delta t = t_b - t_a$ en intervalos más pequeños Δt_i; de modo que, en cada uno, la curva sea prácticamente recta; o más formalmente: que, en cada tramo pequeño, la distancia entre la curva y el segmento secante sea menor que medio pixel.

Comencemos por el planteo simple y veremos la necesidad de refinarlo. Supongamos que ya tenemos un sub-intervalo Δt_i donde sabemos que la curva no zigzaguea.

El primer problema que analizaremos consiste en medir el error de considerar recto a este intervalo. El máximo error es la distancia entre la secante y una línea paralela que sea tangente a la curva (recordar el teorema del valor medio). Para medir ese error hay que encontrar el punto en el cual el vector tangente $\{\dot{x}, \dot{y}\} \equiv \{dx/dt, dy/dt\}$ es paralelo a $\{\Delta x, \Delta y\}$: para ello, se busca t, de modo que $\dot{y}\,\Delta x = \dot{x}\,\Delta y$. Aún con polinomios eso requiere de un prohibitivo cálculo numérico.

Una aproximación "razonable" consiste en calcular la distancia del punto medio de la secante al punto de la curva con t medio (distinto del punto medio de la curva, que también es muy difícil de calcular).

Esta aproximación no es ni acota el error; pero es un esquema útil y que se adopta, aun sabiendo que hay un potencial error. Si ese error fuese inaceptable, habrá que hacer más cuentas, pero se puede resolver.



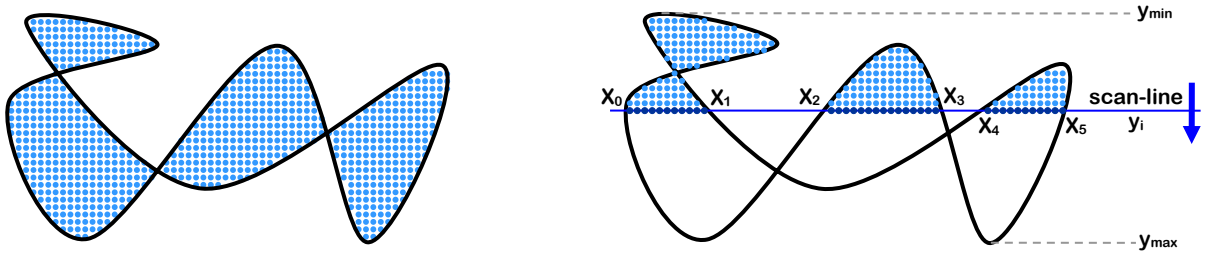
El siguiente problema es cómo llegar a una subdivisión sin zigzagueos. La curva puede ser cerrada, de modo que no se puede comenzar sin ninguna subdivisión previa. También puede ser un 8 o algo parecido, de modo que hace falta un número importante de subdivisiones iniciales ¿Cuántas? Una curva paramétrica y polinómica (como las que se usan en CG) zigzaguea tanto como el grado, eso permite fijar la cantidad de subdivisiones iniciales en, digamos, 2 veces el grado, para estar un poco más seguro. Si no se conoce la variabilidad de la curva, unas cien subdivisiones iniciales pueden calcularse rápidamente.

Si alguno de los dos problemas encontrados (precisión y variación) no se puede resolver, habrá que elegir otro método; pero éste sirve en general.

Se comienza definiendo una subdivisión adecuada del Δt y calculando el punto P de la curva para cada valor intermedio. Se almacenan los pares (P_i,t_i) (con x e y redondeados a enteros) en una lista enlazada. Se toman los extremos de la primera subdivisión: A = P(t_a) y B = P(t_b) se calculan $t_m = (t_a+t_b)/2$, C = P(t_m) y el punto medio de la secante M = (A+B)/2. Si la distancia entre C y M es menor que medio pixel, se pasa al tramo siguiente: (A,t_a) = (B,t_b) y (B,t_b) = next(B,t_b). En caso contrario, se almacena el par (C,t_m) entremedio de A y B: next(A,t_a) = (C,t_m) y next(C,t_m) = (B,t_b) repitiendo el tramo. El proceso termina cuando no hay tramo siguiente (next(B,t_b) == NULL). Finalmente, se rasteriza (Bresenham) la secuencia de tramos rectos consecutivos.

Rasterización de Figuras Planas Cerradas

Rasterizar una figura plana consiste en pintar (generar los fragmentos) del interior. Puede ser de igual o distinto color que la frontera y también puede ser en un solo proceso o en procesos separados.



El obvio problema es la determinación eficiente del conjunto de pixeles interiores. En general (no solo al rasterizar) para averiguar si un punto dado está dentro de una figura, se suele usar el test del rayo o de paridad: Se tira un rayo, en cualquier dirección y sentido, desde el punto hasta un punto fuera de la caja que encierra la figura. Si el punto de partida es interior el rayo corta un número impar de veces a la curva y viceversa. En forma equivalente una línea que viene desde el exterior "entra" en la figura al cortar por primera vez a la frontera, sale al cortarla por segunda vez y así sucesivamente hasta que sale definitivamente por la última intersección (siempre par).

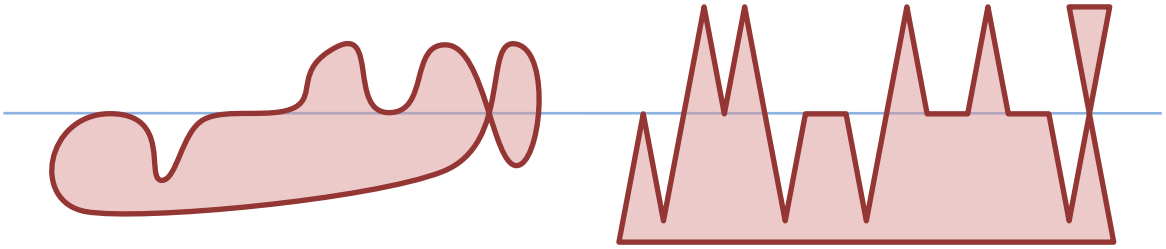
Para curvas y poligonales se puede diseñar, con esa idea, un algoritmo conceptualmente simple, que permite rasterizar figuras cóncavas, disconexas y con huecos. Se rasteriza primero la frontera; manteniendo, para cada valor de y, una secuencia ordenada con las coordenadas x de los píxeles pintados: $X(y) = \{x_0, x_1, x_2, x_3, \ldots, x_{n-1}\}$. Luego, para cada y_i, se recorre la lista $X(y_i)$, que tiene un número par n_i de valores, pintando los segmentos impares:

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

La primera y última línea contienen borde y no interior, por eso se guardan y recorren $dy-2$ listas.

El *buffer* de memoria de la imagen se almacena como una sucesión de líneas horizontales; por lo tanto, con un algoritmo que barre *scan−lines* horizontales se optimiza el uso de la memoria *cache*. Las impresoras 3D utilizan un procedimiento similar, pero son *scan–planes* y en cada uno de ellos se rasterizan, a su vez, las figuras planas que se obtienen al cortar el sólido que se está construyendo, mediante el plano de barrido actual.

El problema se da en las tangencias y auto intersecciones; y esos problemas afectan muchísimo el método, pues hay que distinguir casos: 1) los extremos (máximo o mínimo), donde la línea toca tangencialmente a la curva; al encontrar un extremo no debe cambiar la paridad que traía la línea 2) las inflexiones, donde la recta también es tangente a la curva, pero la atraviesa y hay que cambiar la paridad y 3) las auto intersecciones donde la frontera atraviesa más de una vez a la línea y no debe cambiar la paridad. El problema es el mismo si se trata de una poligonal o de una curva. Esos puntos especiales complican mucho el planteo del algoritmo.



Para estos problemas hay soluciones en la bibliografía y en internet. Es importante destacar aquí que casi siempre, un algoritmo conceptualmente simple, se vuelve complicado por algunos casos particulares o por la necesidad de evitar divisiones por cero o por problemas similares.

Para triángulos (la única figura que rasteriza OpenGL) el método es trivial, pues hay una sola entrada y una sola salida y, para cada y_i se calculan trivialmente (Bresenham) el x_{min} y el x_{max} en función de los valores en y_{i-1} del paso anterior. Es decir que se puede rasterizar el interior y la frontera a la vez.

Interpolación de variables – Coherencia.

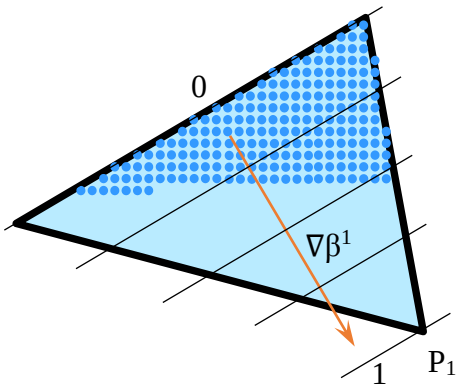
La rasterización se realiza en el espacio de la imagen; por lo tanto, son esas las coordenadas que se conocen del centro del fragmento y los vértices de la primitiva. Existen dos opciones para interpolar linealmente las variables definidas en el espacio visual: 1) Anti-transformar, para obtener las coordenadas visuales del fragmento y allí realizar una interpolación lineal y 2) Utilizar interpolación hiperbólica. Resulta mucho menos costosa la segunda opción y es esa la que utiliza OpenGL.

A cada vértice del espacio visual se le aplica la matriz de proyección, luego se divide {x.y,z} por el w resultante y el resultado se "estira" al viewport para armar la imagen. El resultado total, para una proyección ortogonal, es una transformación afín; por lo tanto, la interpolación lineal en el espacio de la imagen es correcta en ese caso. Para una perspectiva, en cambio, es una transformación proyectiva; la profundidad (*depth*) resulta $z = a+b/z_v$, donde z_v es el visual, a y b dependen de *znear* y *zfar*.

La profundidad de cada fragmento rasterizado se almacena en el *depth-buffer*, para oclusión visual. Como la profundidad ya está dada en el espacio de la imagen, se interpola linealmente allí. Se utilizan las coordenadas baricéntricas β^i, en el espacio de la imagen, las longitudes y las áreas ($\beta^i = a^i/a$, siempre positivas) se calculan con las coordenadas {x,y} de los centros de fragmentos.

Luego se aplican las fórmulas de interpolación perspectiva, para encontrar las coordenadas baricéntricas en el espacio visual: $\alpha^i = (\beta^i/z_{vi})/(\Sigma\ \beta^j/z_{vj})$ e interpolar linealmente las otras variables asignadas a cada vértice; por ejemplo, color o coordenadas de texturas: $v = \Sigma\ \alpha^i\ v$.

Vimos que, para *rasterizar* segmentos rectos en el plano, se aprovecha la constancia de la derivada. En 3D sucede lo mismo, pero con las derivadas parciales. Para calcular los β^i se utilizan los cocientes de áreas mediante productos vectoriales ordenados, pero la constancia de las derivadas parciales (la linealidad) nos permite actualizar con más eficiencia el valor de β^i en un fragmento vecino, sin tener que recalcular. Solo habrá que calcular una sola vez las derivadas parciales de las coordenadas baricéntricas en el espacio de la imagen (es sencillo, solo requiere calcular longitudes y no áreas) con eso se hacen pocas cuentas y en forma muy eficiente.



La técnica general de prever lo que va a suceder, asumiendo que será igual o no muy distinto de lo que sucede ahora, se conoce con el nombre de **coherencia** y se utiliza siempre en computación gráfica.

En este caso, conociendo la derivada y sabiendo que es constante, se conoce exactamente la profundidad del próximo fragmento.

Pero para otros asuntos puede suceder que la derivada no sea constante, en tal caso se puede suponer que no será muy distinta de la anterior (primeras diferencias). Saber aproximadamente lo que va a suceder es mejor que no tener ninguna idea. Esto se aprovecha en múltiples situaciones; por ejemplo: en una película, al buscar un punto o un objeto, si en el cuadro anterior estaba "aquí", en el actual debe estar cerca, si se venía moviendo "así", probablemente se haya movido "algo así"; no hay que buscar en toda la imagen, sino prever, más o menos, por donde puede estar.

Para un procedimiento secuencial es muy beneficioso aplicar la coherencia, en este caso la constancia de las derivadas, pero cabe aclarar que la GPU trabaja con múltiples procesadores ejecutando una misma tarea en paralelo y, por lo tanto, un fragmento no tiene información del "anterior", pues no hay secuencia. Si bien es un valiosísimo concepto, en las placas gráficas el paralelismo impide utilizarlo.

Antialiasing

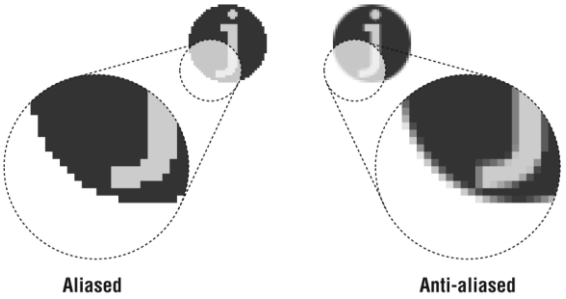
En la teoría del muestreo, al tomar una secuencia discreta de datos de una señal continua, se denomina "*aliasing*" a la falsa interpretación de una frecuencia por otra. Aquí, en Computación Gráfica, es el efecto de serruchado de las imágenes rasterizadas.

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

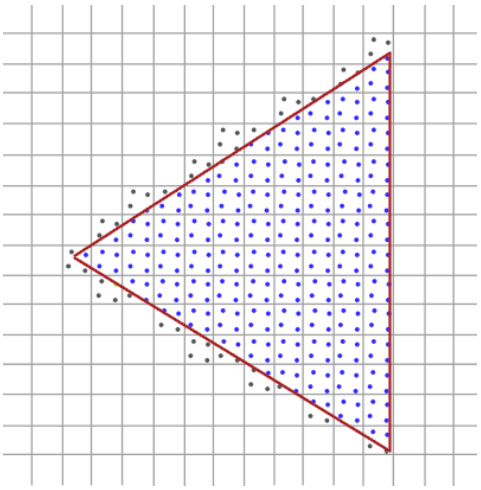
Hay muchas técnicas para atenuar el defecto visual. En general se basan en mezclar el color de la línea con el del fondo o el color de lo que haya "debajo" o "detrás".

Hay una técnica muy simple basada en el factor de cubrimiento del pixel (*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 pixel. 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 más 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 preexistente.

El problema se presenta también con los bordes del texto, las figuras y 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.



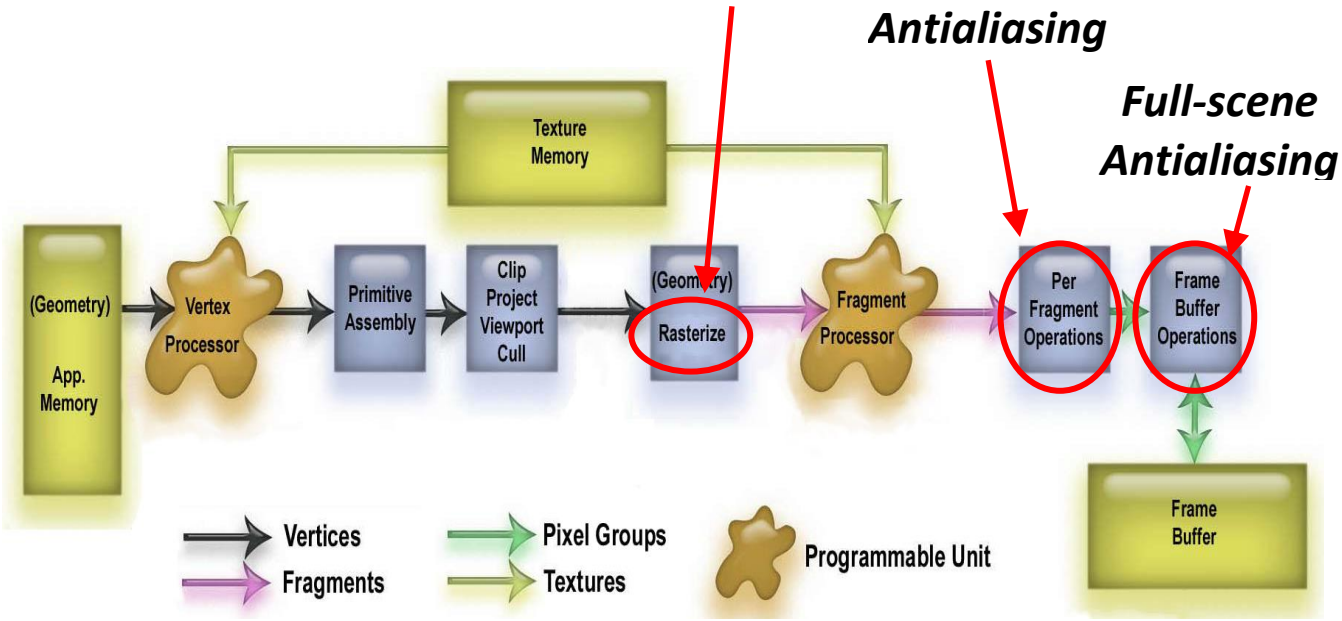
Los mejores resultados se obtienen renderizando en una imagen mayor (2, 4 u 8 veces mayor) y luego interpolar pixeles vecinos al reducir la imagen para mostrar. Esta técnica se denomina *supersampling* y *full scene antialiasing*. El *supersampling* es muy caro pues implica un número cuadráticamente mayor de cálculos. La solución estándar actual (para tiempo real) consiste en el *multisampling*, que "parece lo mismo": en lugar de analizar sólo si la primitiva cubre el centro del pixel se analizan varios puntos de muestreo (*samples*) por pixel y se promedian los resultados. No es lo mismo porque un control inteligente puede hacer eso solamente en los fragmentos inicial y final de cada *scan−line*, es decir: solo en los bordes de las figuras. La desventaja de eso es que si se utilizan texturas con transparencias (reja o alambrado) los bordes de la figura no son lo mismo que los bordes de la zona opaca de la textura; para la primitiva, estos fragmentos son interiores. Algunas placas (Nvidia) permiten avisar para que se haga *multisampling* o *supersampling* en todo el polígono, cuando se sabe que este lleva una textura con transparencias, ese modo de trabajo se habilita o deshabilita como es habitual.



En OpenGL está definido un buffer especial para *multisampling* que permite averiguar si el fragmento cubre cada uno de los *sample−points* definidos en el pixel e interpolar valores en esos puntos, para luego realizar un promedio ponderado. El antialiasing de la vieja funcionalidad fija de OpenGL es francamente inaceptable, este modifica el valor *alpha* del color (RGBA) del fragmento de acuerdo al cubrimiento del pixel y mezcla eso con el color que ya hay en el pixel del *color buffer*.

Usted está aquí:

Rasterización,
supersampling,
multisampling
e interpolación



Notas para el estudio del tema:

La intención didáctica de esta selección particular de temas consiste en mostrar las definiciones básicas (ej: *raster* vs. *vector*), los requisitos de extrema eficiencia para los programas implementados en hardware (ej: Bresenham, coherencia) y las dificultades que se presentan (siempre) con las situaciones especiales de los algoritmos simples (ejs: división por cero, puntos especiales al usar *scan−line*).

Por lo tanto, es necesario entender los pasos, pero no es necesario recordar de memoria los algoritmos complicados, ni siquiera se requiere deducirlos en un examen, con sus ecuaciones y simplificaciones; solo hay que entenderlos y recordar sus lineamientos. Por ejemplo, de Bresenham para segmentos rectos, una de las cosas que hay que saber es que para pasar todo a enteros, multiplica todo el sistema por el producto de todos los denominadores y redondea los valores iniciales. En este caso se multiplica todo por $2\Delta x$ del redondeo y las ecuaciones; pero esto no es necesario saberlo, pues son detalles que solo sirven para este caso y se puede encontrar la receta en cualquier libro.

En general se requiere aprender (entender y recordar) las definiciones, los lineamientos y los trucos que pueden servir en otras ocasiones, pues generalmente los exámenes contienen otras situaciones.

Ejemplos de preguntas de examen:

Responda las preguntas como un humano y no como un libro. Por ejemplo: "Para rasterizar una circunferencia con ancho 2a+1, en los octavos de tendencia horizontal dibujo segmentos verticales desde y-a hasta y+a del pixel en x,y calculado por Bresenham. En los tramos de tendencia vertical pinto segmentos horizontales. Eso deja espacios sin rellenar en las uniones a 45º y pixeles pintados dos veces. Después pienso un método para arreglar eso." Esa respuesta es el 75% del puntaje asignado (porque no explica que hacer en los 45º y no ve que la línea quedaría más delgada a los 45°). Si hubiese hecho scan−lines en forma adecuada, valdría como método correcto.

A continuación, algunos ejemplos de preguntas de parcial:

• ¿Qué es rasterizar? (líneas/figuras; continuidad/contigüidad)

• Describa los lineamientos del algoritmo DDA para rasterizar curvas paramétricas y en qué casos no puede utilizarse.

• Para curvas paramétricas definidas por polinomios, justifique la conveniencia de utilizar DDA en lugar de subdivisiones.

• Suponga que tiene un monitor 3D, o si lo prefiere una imagen tridimensional (una pila de imágenes planas) de WxHxD vóxeles (width, height, depth).

a. Describa un algoritmo para rasterizar eficientemente una curva paramétrica:
   $\{x(t), y(t), z(t)\} \quad t \in [t_0,t_1]$,
   donde las tres funciones son polinomios (se pueden calcular sus derivadas).
b. Describa como haría con un segmento rectilíneo (lineamientos).
c. Describa un algoritmo para rasterizar eficientemente un triángulo relleno (lineamientos).

• Describa el algoritmo del punto medio para rasterizar circunferencias.

• Describa un algoritmo de subdivisiones binarias para rasterizar elipses alineadas con los ejes (no inclinadas) con centro en {xc,yc}, semieje horizontal rh y vertical rv. Las ecuaciones paramétricas son:

$x = xc + rh \cos(t)$; $y = yc + rv \operatorname{sen}(t)$; con $t \in [0, 2\pi)$