Rasterización (Scan-Conversion)

La versión continua de una línea (recta o curva) en dos dimensiones es un conjunto infinito de pares de números reales {x(t), y(t)}, ambos son función 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 cota puede ser infinita.

Los dispositivos vectoriales (ideales) pueden asumir a t como el tiempo y mover dos motores independientes para trazar la línea. En la práctica, con motores paso a paso (*stepper*) se aproximan las curvas de acuerdo a la precisión de los motores. Como ejemplo de esto tenemos los pantógrafos para cortar chapas y los *plotters* de corte para autoadhesivos utilizados en carteles y gigantografías.

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 o píxeles del soporte. Lo mismo pasa con los monitores que constan de una matriz de puntos brillantes. Las líneas trazadas en estos dispositivos deben representarse como una sucesión finita de puntos contiguos, de coordenadas enteras y de modo que no haya espacio entre dos puntos sucesivos de la línea: $\max(\Delta x, \Delta y)=1$.

Podría pensarse en utilizar el conjunto {int(x(t)+0.5), int(y(t)+0.5)} de coordenadas redondeadas al entero mas cercano. Éste es un conjunto contiguo, discreto y finito de pares. Pero no se puede definir un algoritmo que recorra el parámetro t como una variable continua.

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. De ese modo se puede dibujar la curva como se requiere: una sucesión de píxeles contiguos.

Para calcular el incremento de t que hace 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 de un salto unitario de y se necesita y' = dy/dt. Se calcula el salto de t que haga que el máximo salto de coordenada, x o y, sea uno.

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

Esto es: cuando la curva es de tendencia horizontal ($x' > y'$) , se mueve de a un paso en x y si es de tendencia vertical, 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 comienza calculando las derivadas en t_0 y pinta o marca un punto en $\{x_0=x(t_0), y_0=y(t_0)\}$. 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=±1. Con dt=1/| x´| calcula dy=y´dt y pinta $\{x_0+dx, y_0+dy\}$. Ahora calcula la derivada en t+dt y repite del mismo modo, hasta que y´ sea mayor que x´, en cuyo caso altera el método y así hasta llegar a t_1.

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.

Rasterización de Segmentos Rectilíneos

Algoritmo DDA

El método antes planteado se simplifica si las derivadas x' e y' son constantes, es decir: para segmentos rectilíneos. 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:

```
static int redondea(float a) {return int(a+.5);}

static void intercambia (float &a, float &b) {float tmp=a; a=b; b=tmp;}

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));
```

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

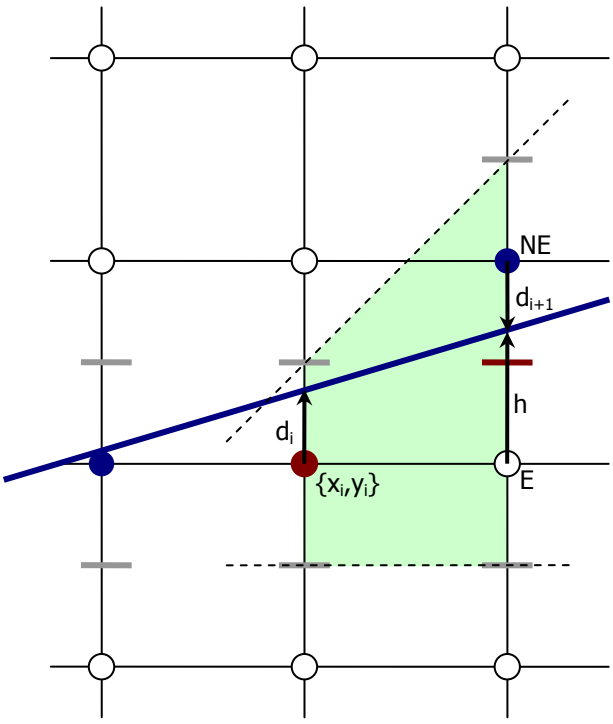
Algoritmo de Bresenham

La gran mejora consiste en utilizar solo aritmética de enteros, en el que se conoce como algoritmo de Bresenham o de punto medio.

Para deshacernos de las divisiones hay que multiplicar todo por 2 (para el redondeo) y por el denominador de la pendiente. Luego hay que procurar que todos los valores sean enteros. Pero veámoslo gráficamente, para que no quede el frío código sino la geometría como teoría del algoritmo:

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

Consideremos el caso de referencia: 0<dx, 0<dy y dy<dx. Vamos pintando píxeles de a uno, incrementando x. Acabamos de pintar el píxel i–ésimo, 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: $y_{i+1} = y_i+1 \Rightarrow d_{i+1} = (y - y_{i+1}) = (y - (y_i + 1)) = (y - y_i) - 1 = h - 1.$

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 y de las divisiones, tanto en la consulta como en la actualización de datos, multiplicamos 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:

$D = 2\,dx\,d_0 + 2\,dy - dx$,	la variable de decisión inicial, 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 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 ésta. Dado que se realiza solo al principio (y no dentro del lazo) no se atenta sensiblemente contra la velocidad del algoritmo.

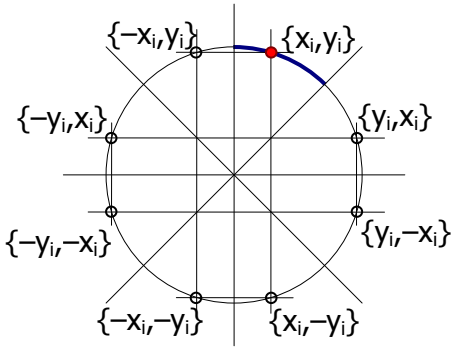
Desarrollamos solo el caso de referencia, el resto se obtiene por simetrías:

```
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.

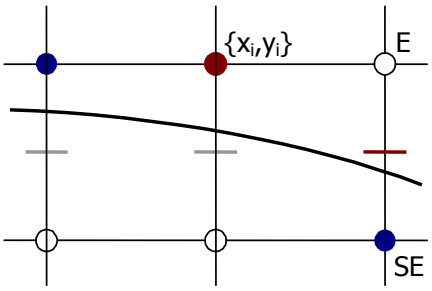
Rasterización de Circunferencias

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, transforman enteros en enteros; son sólo cambios de signo e intercambio de coordenadas, 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. Hay que averiguar si un dado punto medio está dentro o fuera de la circunferencia. Dado que x crece, si el punto medio está dentro pintamos el píxel E y si esta fuera pintamos el SE, como en el ejemplo que se muestra debajo.



Un punto cualquiera de coordenadas {x,y} está fuera del círculo si $x^2 + y^2 > r^2$. Para el punto medio habrá que ver si:

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

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:

$h = d - 1/4 \quad \Rightarrow h_0 = d_0 - 1/4 = 1 - r; \quad d > 0 \Rightarrow h > -1/4;$

la comparación se hace entonces 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 es bastante eficiente, pero puede mejorarse aprovechando que la segunda derivada es constante: el incremento de los incrementos (diferencias de segundo orden) es constante, con ello podemos evitar las multiplicaciones por dos.

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

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

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

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

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

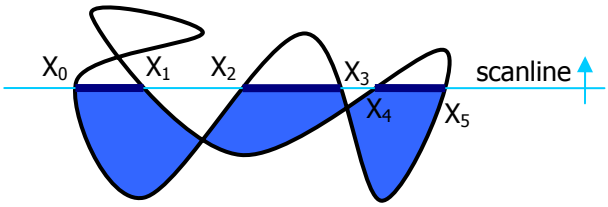
$E_0 = 2 \times 0 + 3 = 3$

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

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

Rasterización de Polígonos

Rasterizar una figura cerrada consiste en rasterizar su contorno y pintar los píxeles que queden en el interior. El obvio problema es como determinar eficientemente el conjunto de píxeles interiores.

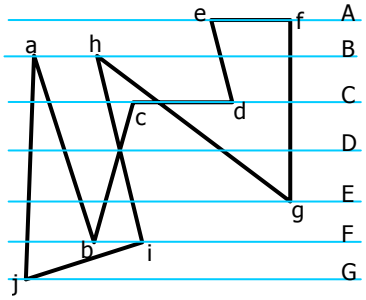


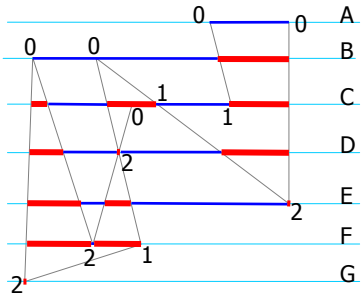
El algoritmo usual consiste rasterizar primero la frontera, manteniendo para cada línea horizontal o *scan-line* (cada y = constante entre y_{min} e y_{max}), una secuencia ordenada de menor a mayor con las coordenadas x de los píxeles pintados. Luego, para cada valor de y se recorre la lista correspondiente $\{X_0, X_1, X_2, X_3, \ldots, X_{n-1}\}$ pintando los segmentos impares:

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

El barrido se hace horizontal porque el ***framebuffer***, que se interpreta como una matriz de WxH píxeles, se guarda en memoria como un *array* unidimensional, una sucesión de *scan-lines*. De ese modo, se puede optimizar el uso de la memoria *cache* del sistema.

Este método permite pintar figuras cóncavas o aún auto interceptadas, pero no está exento de algunos problemas que pueden verse en 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 de x.

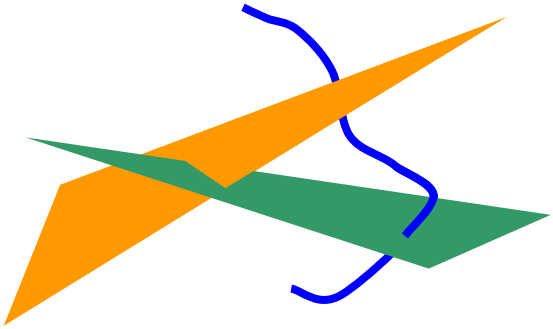
Para solucionar el problema de los vértices y las intersecciones, se permite repetir valores de x y los vértices sólo se cuentan cuando corresponden al y mínimo (estricto) del segmento rasterizado. Es decir que están duplicados los puntos como b, j y g, mientras que puntos como el i y el d se cuentan 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 g-h está una sola vez. Un x duplicado puede significar entrada/salida o salida/entrada.

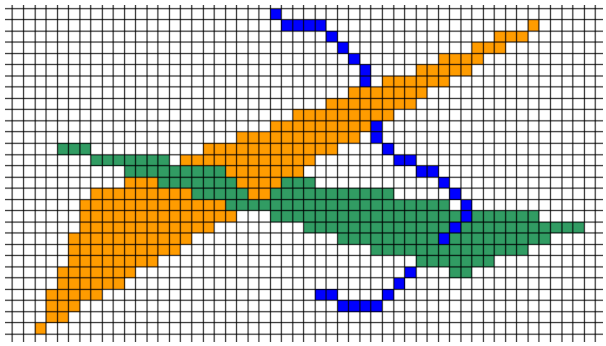
Cualquier curva suave puede ser tratada como una poligonal de segmentos de un píxel de longitud.

Z-buffer

Al renderizar varios objetos, algunos tapan la visual de los otros. Hay varios métodos para resolver los problemas de oclusión visual, pero cuando se dispone de suficiente memoria, y se descansa en el hardware especializado para la *rasterización*, se utiliza el algoritmo basado en el *z-buffer* o *depth-buffer*. Básicamente consiste en pintar el píxel si la distancia al ojo (z) es menor que la almacenada en un *buffer* especial; en tal caso se actualiza ese *buffer* con el valor de z recién calculado.

El *color-buffer* es la porción de memoria que almacena los WxH píxeles de cada componente de color (en general, un *byte* por cada componente RGBA, seguidos). Ese es el *buffer* que se pinta o actualiza cuando un píxel es visible. El *depth-buffer* se mantiene en paralelo y consiste en WxH valores de punto flotante. La porción de espacio visible está limitada mediante un plano cercano al ojo y uno más alejado, las distancias *znear* y *zfar* del ojo a los planos sirve para cuantizar el rango de profundidades.





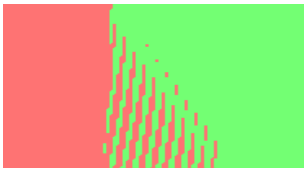
El algoritmo fue desarrollado por Ed Catmull (uno de los fundadores de Pixar) en 1974 y es algo así:

Inicializar el *z-buffer* con el z máximo, el del *far-plane*: depth(x,y)=zfar
Para cada primitiva dentro del espacio visual:
Rasterizar calculando z (la "profundidad" del píxel).
Si z < depth(x,y)
Actualizar el *color-buffer* (pintar)
Actualizar el *z-buffer*: depth(x,y)=z

Este es el algoritmo que utilizan OpenGL y la mayoría de las APIs gráficas a partir del abaratamiento de la memoria. Es muy rápido, eficiente y general, no se calcula ninguna intersección, no requiere ningún ordenamiento de los objetos ni subdivisión del espacio porque *rasteriza* todas las primitivas del volumen visual en cualquier orden. Pero adolece de algunos defectos que pueden ser importantes:

1) No sirve para manejar las transparencias múltiples: si el objeto que se va a dibujar es semi-transparente, debe combinarse su color con el color actual del píxel. Pero, si aparece un tercer objeto intermedio y también transparente, ya no se dispone de la información necesaria para calcular bien el color resultante. Cuando hay transparencias, el resultado depende el orden de rasterización.

2) Precisión: los planos coincidentes (o casi) suelen resultar en visualizaciones defectuosas debido a la cuantización de z. Esto obliga a ubicar con ajuste preciso los planos *near* y *far* para aumentar la precisión del resultado, que aún así puede ser malo. Este defecto se llama *z-fighting* (pelea de z) y es muy visible por la mezcla ruidosa del color de las piezas. También aparece cuando se dibujan los bordes de las primitivas sobre las mismas (*wireframe* y *filled* superpuestos); para resolver esto, OpenGL trae una función (glPolygonOffset) que perturba el *z-buffer* cuando se activa.



3) Resulta más complicado realizar un *antialiasing*: Cuando un píxel esta formado por varias fuentes distintas (bordes) el *antialiasing* (en computación gráfica léase: suavizado) se realiza pintando el píxel con una combinación ponderada de las fuentes. Con el *z-buffer* se pinta de acuerdo a una única fuente.

4) El algoritmo de *z-buffer* requiere *rasterizar* muchas primitivas (todas las del *viewport*) y realiza múltiples accesos no secuenciales a la memoria (*cache misses*).

Para escenas especialmente complejas se puede combinar con distintas técnicas de ordenamiento y/o descarte masivo (*clipping* y *culling*) de objetos o grupos de objetos invisibles, tanto para reducir la labor como para optimizar el uso de la memoria (*cache*).

Extensión a 3D del Algoritmo de Bresenham

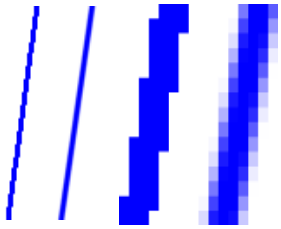
Vimos que para *rasterizar* segmentos con el algoritmo de Bresenham, se aprovecha la constancia de la derivada. En 3D sucede lo mismo, pero con las derivadas parciales, y eso permite calcular z en cada paso de *rasterización* y así rellenar el *depth-buffer*.

Si estamos *rasterizando* un segmento, avanzamos de a un píxel en x o y, la coordenada z del píxel actual es igual que la del anterior más una constante: la derivada parcial de z respecto de x o y. Haciendo el cálculo de antemano, en el *loop* de *rasterización* solo se suma un factor constante.

En el caso de polígonos u otras figuras planas, que rasterizamos mediante *scan-lines*, el proceso es igual: siempre se suma la derivada parcial de z respecto de x, que es constante para el plano.

Antialiasing

Se denomina "*aliasing*" a la falsa interpretación de una frecuencia por otra distinta al muestrear (tomar una secuencia discreta de datos de) una señal continua. Aquí, en Computación Gráfica, se le llama así al 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*): En el algoritmo de Bresenham, el desplazamiento d_i nos brinda un indicador de la proporción de línea que pasa por el píxel. Podríamos pintar el píxel, no con el color de la línea sino con una mezcla proporcional y lo mismo para el píxel vecino. 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.

Manipulación del *frame-buffer*

OpenGL trabaja con varios *buffers*, ya conocemos el *color-buffer*, donde se arma la imagen que vemos y el *depth-buffer*, que sirve para realizar las oclusiones visuales. Internamente no son más que *arrays* de valores asignados a los píxeles. Hay más *buffers* y en conjunto conforman el *framebuffer*. Los valores pueden leerse, manipularse y sobrescribirse desde el programa, OpenGL trae además algunas rutinas propias de manipulación.

Los *buffers* estándar de OpenGL son los siguientes:

- *Color buffers: front-left, front-right, back-left, back-right* y algunos otros *auxiliary color buffers*
- *Depth buffer*
- *Stencil buffer*
- *Accumulation buffer*

Se puede ver que hay más de un *buffer* de color. El *front* y el *back* se utilizan para *renderizar* con la técnica de *double-buffering*: se *renderiza* en *background* mientras se visualiza el *front*, cuando la imagen está completa se intercambian los roles de ambos *buffers* (*swap buffers)*, de esa forma se evita el parpadeo (*flickering*) que se ve en las animaciones con un solo *buffer*. *Left* y *right* son dos juegos de *buffers* de color utilizados para estereoscopía, en cada uno se genera la imagen para cada ojo. El resto de los *buffers* de color (*aux*) se utilizan como *layers* o capas o también se pueden utilizar para almacenamiento temporario de imágenes generadas.

Un ejemplo sencillo de utilización del *color-buffer* consiste en averiguar el color del píxel que hay "debajo" del cursor para saber si está sobre un objeto o sobre el fondo. Si se utiliza iluminación en lugar de un color fijo el color es variable, en tal caso se puede hacer un segundo renderizado en un buffer auxiliar (normalmente en el *back* y sin *swappear*) con un color fijo y distinto para cada objeto, leyendo el color del píxel se puede conocer el objeto debajo del el cursor.

Se pueden lograr algunos trucos útiles manipulando el *z-buffer*. Un ejemplo es la determinación de siluetas buscando altos gradientes de z entre píxeles. También se puede declarar *read-only* al *z-buffer* (no se actualiza) o cambiar la función que compara los valores de z entrante y almacenado, con ello pueden hacerse varios pasos de *renderizado* para dibujar en uno las líneas invisibles de una manera y en el siguiente las visibles de otro modo. Otro truco estándar es la proyección de sombras, se basa en una manipulación del *depth-buffer* que se genera al mirar la escena desde el foco de luz.

El *stencil buffer* se utiliza para muchísmos trucos, el nombre proviene del uso principal que consiste en enmascarar la zona de dibujo. Por ejemplo en un juego de carreras de autos, el parabrisas sirve de máscara para *renderizar* dentro toda la escena externa.

El *buffer* de acumulación se utiliza para acumular datos de color, como si fuesen distintas manos de pintura con efecto acumulado (con ciertas reglas de enmascaramiento y sobreescritura) sobre una misma superficie. Sirve por ejemplo para hacer el borroneado por movimiento (*motion blur*) dando la impresión de velocidad o para hacer antialiasing o para simular la profundidad de campo fotográfica.

La presencia o no de los distintos *buffers* depende del hardware en cuestión. Normalmente se solicitan al inicializar y luego se consulta a la placa gráfica si efectivamente están disponibles por hardware.

Los *buffers* que se utilizan se escriben, borran y enmascaran mediante sencillas reglas que pueden verse en el *Red Book* u otro manual de OpenGL. Lo importante de saber es que los datos pueden manipularse logrando una miríada de efectos útiles y trabajando directamente con los píxeles (fragmentos) en modo *raster*. Estas técnicas se denominan *image-precision* en contraposición a las técnicas *model precision* que se realizan en el espacio (vectorial) de la escena, como por ejemplo el cálculo de intersecciones entre objetos.

Una imagen completa puede leerse de un *buffer* como un rectángulo de píxeles; por el contrario, una imagen leída, por ejemplo de un archivo, puede escribirse en algún *buffer*. OpenGL trae un juego especial de funciones para leer, escribir y manipular imágenes y un *pipline* especial para hacer esas operaciones en forma eficiente.