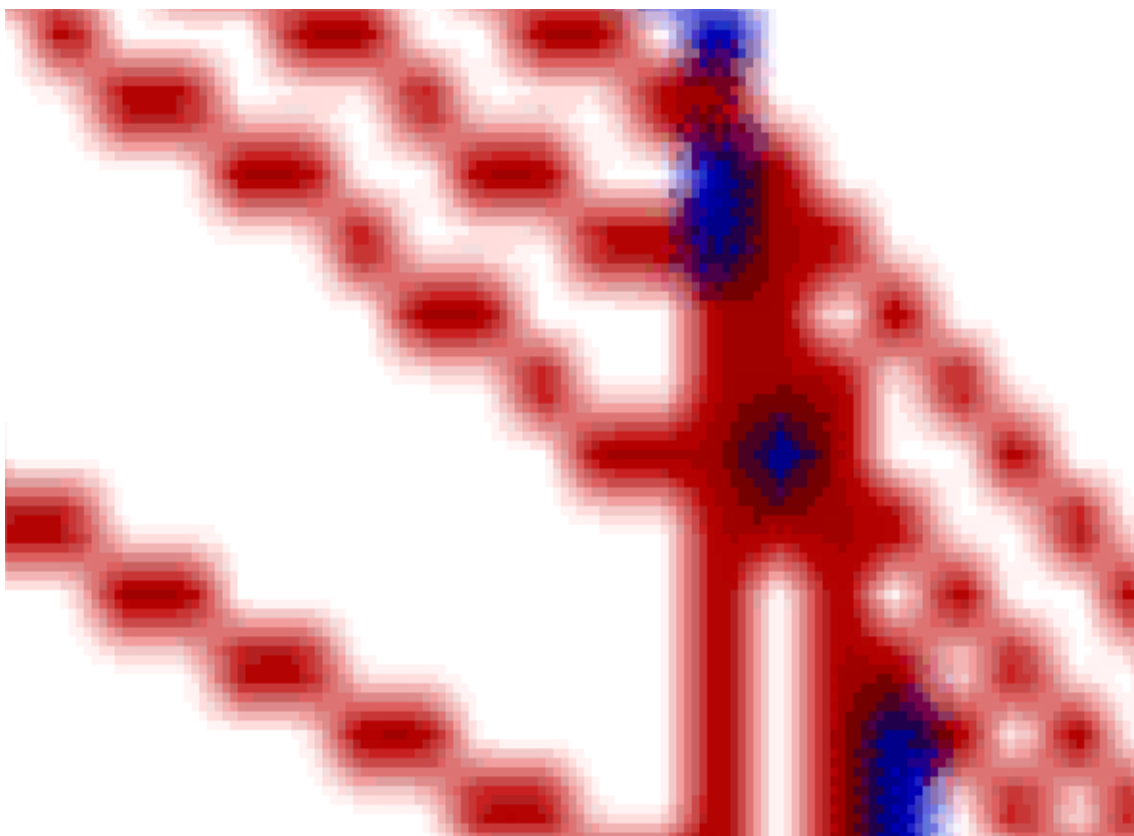


---

# 2

## Algoritmos y Conceptos Básicos

---



## 2.1 Dispositivos Gráficos

Los resultados gráficos de una aplicación pueden mostrarse en una gran variedad de dispositivos de salida. Normalmente estos dispositivos son o bien de pantalla o bien de impresión. Sin embargo, desde el punto de vista de la Computación Gráfica, es importante otra clasificación, referida al modo en que los mismos son manejados por la computadora. De esa manera, podemos ver que existen dispositivos de los siguientes tipos:

- **Dispositivos de vectores**, los cuales reciben de la computadora la información geométrica de la localización y tamaño de las primitivas que soportan, de las cuales producen una reproducción “caligráfica”.
- **Dispositivos de raster**, los cuales reciben de la computadora la información de una serie de pixels, los cuales son posicionados en forma contigua.

Los dispositivos de vectores fueron los primeros en desarrollarse, pero luego del vertiginoso descenso en el costo de la memoria volátil, a partir de la década del 70 se hicieron más baratos los dispositivos de raster. Como veremos en este Capítulo, ésto implica un cambio en la manera de representar las primitivas gráficas (usualmente dichas primitivas son el punto, el segmento de recta y la circunferencia o el círculo).

### 2.1.1 Dispositivos de vectores

Actualmente estos dispositivos son más caros, pero tienen ciertas ventajas que los hacen únicos. Por ejemplo, tienen mucha mejor resolución y precisión que los dispositivos de raster, y requieren un ancho de banda de comunicación mucho menor dado que no reciben la discretización completa de las primitivas sino solamente su posición.

**Plotters:** Grafican en una hoja (que en algunos casos puede ser de gran tamaño) sobre la cual se desliza una pluma movida por motores de pasos de gran precisión. En los plotters de tambor, la pluma se desliza en sentido horizontal y el papel en sentido vertical. En los plotters planos (más económicos), el papel está fijo y la pluma realiza todos los movimientos. Son usuales las resoluciones del orden de los  $10000 \times 10000$ . Es posible utilizar colores por medio de varias plumas. Son ideales para la graficación rápida y precisa de planos.

**Displays de almacenamiento:** Al igual que televisores y monitores, estos dispositivos son pantallas de rayos catódicos, pero difieren en ciertos aspectos tecnológicos. Esencialmente, la pantalla tiene cierta “memoria” electrostática que mantiene visibles los elementos graficados con muy alta precisión y sin la necesidad de refresco. Por lo tanto, una imagen muy compleja a la cual se van agregando elementos en orden es idealmente representada por estos dispositivos. Un elemento se representa “pintándolo” por medio de una serie de recorridas del cañón electrónico. El borrado, sin embargo, no puede hacerse en forma selectiva, por lo que no se puede alterar la posición de un elemento sin tener que borrar y redibujar todos los demás. Sin embargo, su precisión y velocidad sin necesidad de memoria volátil los hace ideales para la representación de imágenes de radares.

### 2.1.2 Dispositivos de raster

**Impresoras de matriz:** Era hasta hace poco el dispositivo de impresión más común. Recibe de la computadora la información gráfica como una secuencia de líneas, las cuales va reproduciendo con una cabeza impresora (por medio del golpe de martillos o el rocío de tinta).

**Impresoras Laser:** Recibe de la computadora la información gráfica como una secuencia de líneas, las cuales almacena en una memoria local. Dicha memoria es utilizada para comandar la intensidad de un haz laser que recorre línea por línea el papel, mientras es expuesto al contacto del toner. Donde el haz incide con gran intensidad, el papel se dilata por el calor y absorbe el toner.

**Monitores:** Se han popularizado enormemente a partir del descenso en el precio de la memoria volátil y el incremento constante en la calidad de las prestaciones (resolución, color, precisión). Esencialmente se comportan de una manera similar a un receptor de televisión, excepto por el hecho de que reciben la señal de video y sincronismo en forma directa de la computadora y no a través de una portadora de radio. Al igual que con las impresoras de matriz, la imagen se construye línea por línea, en sentido horizontal primero (de izquierda a derecha) y vertical después (de arriba abajo). Debe existir un refresco de la imagen en memoria, la cual es recorrida por la tarjeta gráfica de la computadora para producir las líneas de barrido. Por lo tanto, el elemento esencial en estos dispositivos es la tarjeta gráfica, la cual veremos en detalle más adelante. Los monitores más populares pueden tener resoluciones de hasta  $1200 \times 1024$  (aunque este límite avanza día a día), con una cantidad de colores limitada por las prestaciones de la tarjeta gráfica. Esto representa una calidad más que aceptable para la mayor parte de las aplicaciones.

### 2.1.3 Hardware gráfico para monitores

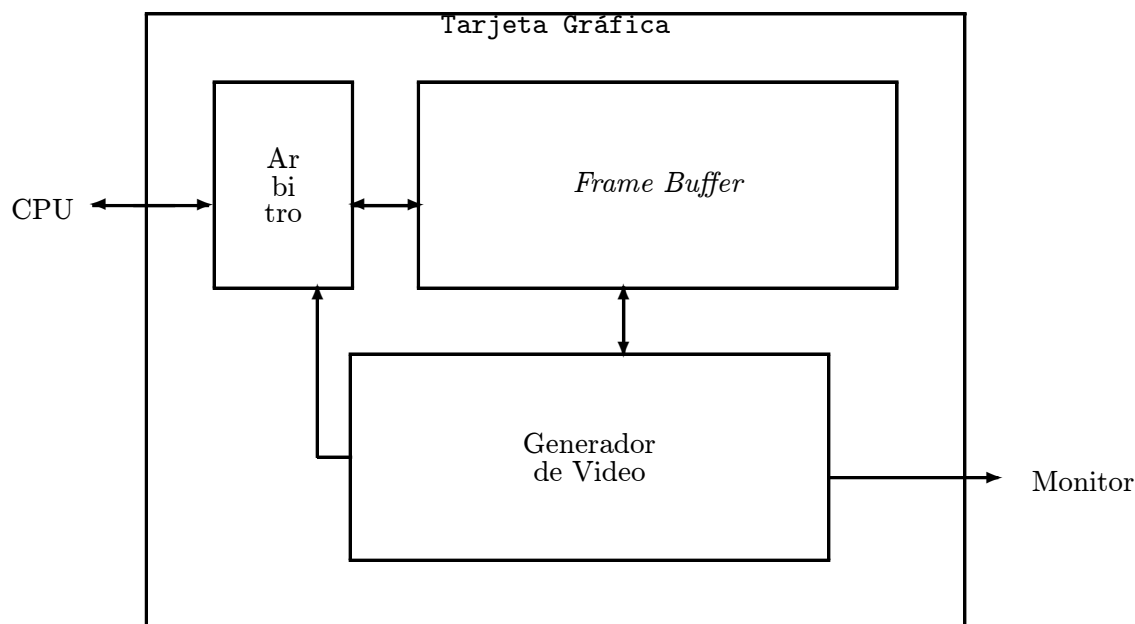


Figura 2.1 Componentes básicos de una tarjeta gráfica de raster.

Los dispositivos de raster requieren un refresco permanente de la discretización de la salida gráfica. En el caso de los monitores, dicho refresco se realiza en un segmento de la memoria volátil de la computadora denominada *frame buffer* o buffer de pantalla, que usualmente se implementa por medio de memoria RAM de alta velocidad localizada dentro de la tarjeta gráfica. El buffer de pantalla es accedido en forma rítmica por el generador de video, que es el encargado de “componer” la señal de video que va hacia el monitor. Al mismo tiempo, al producirse una salida gráfica por parte de la CPU de la computadora, la misma debe ser discretizada y almacenada en el buffer de pantalla. Este acceso debe ser permitido solamente en los momentos en los que el generador de video no está accediendo al buffer, y por lo tanto se requiere el uso de un árbitro que mantenga abierto el acceso al buffer solo en esos casos (ver Figura 2.1).

El temporizado es crítico en el manejo del buffer de pantalla, por lo que se requiere memoria RAM de alta velocidad, mucho mayor que la velocidad requerida para la RAM de la CPU. Por ejemplo, en una norma de video de 1024 pixels por línea, la pantalla es refrescada 35 veces por segundo a una tasa de aproximadamente un millón de pixels por pantalla. Esto significa que en promedio el buffer de pantalla es accedido 35 millones de

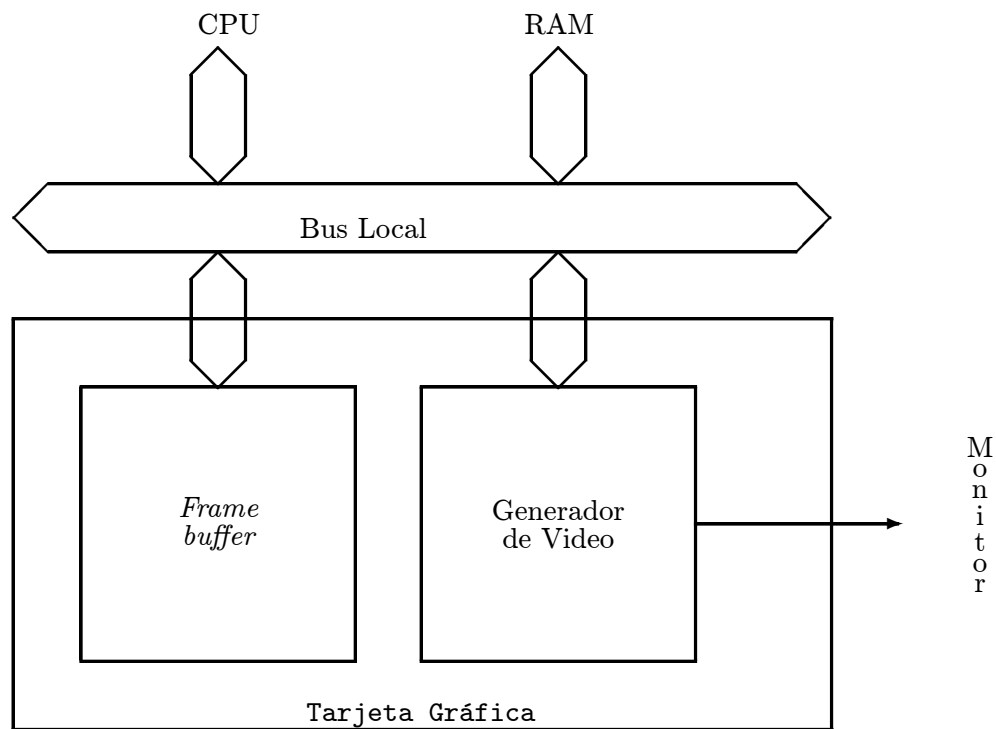


Figura 2.2 Tarjeta gráfica basada en tecnología local bus.

veces por segundo por el generador de video, lo cual requiere una velocidad de acceso a memoria de aproximadamente 30ns para cumplir solo con el refresco de pantalla. En una situación como ésta, utilizar memoria de 25ns. para el buffer de pantalla permite utilizar solamente un pico de 5 millones de accesos por segundo para la CPU, lo cual en muchos casos es insuficiente si se tiene en cuenta que el acceso entre la CPU y la tarjeta gráfica por el bus ISA debe cumplir cierto protocolo que hace más lenta la comunicación.

Otro esquema posible para manejar la memoria de pantalla es utilizar la tecnología de *bus local* (difundida alrededor de 1993 con las motherboard 486 y tarjetas Vesa Local Bus). Básicamente la idea es evitar el uso del bus de datos ISA para interconectar la tarjeta gráfica con la CPU. De ese modo se utiliza un segundo bus (llamado bus local), normalmente de 32 bits en vez de 16, con la velocidad del reloj externo del microprocesador (50Mhz. en vez de 8.33) y con capacidad de acceso directo a memoria (ver Figura 2.2). Este tipo de configuraciones permitió una mejor utilización del ancho de banda marginal de la memoria del frame buffer, y por lo tanto, en determinadas apli-

caciones, como por ejemplo animaciones, la prestación de un mismo hardware aumenta en un orden de magnitud solamente al modificar la configuración de acceso.

La evolución del hardware gráfico en PC siguió esquemas similares, utilizándose primero el port PCI y luego el AGP. Las tarjetas gráficas PCI tenían una configuración similar a las tarjetas ISA dado que el port está pensado para dispositivos genéricos. Sin embargo, el acceso es en 32 bits, a frecuencia de reloj externo, y en algunos casos con posibilidades de acceso directo a memoria. El port AGP, por su parte, permite un acceso en 64 bits a frecuencia interna del micro (1Ghz. en algunos casos) directamente al bus interno y al caché. Por lo tanto es esperable un rendimiento cientos de veces mayor que con las tarjetas ISA.

Un tema que evolucionó paralelamente a la velocidad de acceso y procesamiento en las tarjetas gráficas, fue la implementación por hardware de gran parte de los algoritmos de la *tubería de procesos gráficos* (ver Secc. 3.4.3). En efecto, la mayor parte de los algoritmos básicos que veremos en los siguientes dos capítulos se caracterizan por ser directamente implementables en hardware, lo que acelera no solamente su ejecución sino también la comunicación entre la CPU y la tarjeta gráfica. Dependiendo del sistema operativo, esta comunicación se establece a través de un protocolo que permite abstraerse al desarrollador de aplicaciones de los detalles de implementación de estos algoritmos. Un ejemplo destacado son las actuales tarjetas G-Force, que implementan por hardware no solo los algoritmos básicos 2D y 3D sino gran parte de los modelos de iluminación, sombreado, mapeo de texturas y demás. De esa manera, una de estas tarjetas está en condiciones de manipular escenas muy complejas en tiempos interactivos.

Otro interesante nivel de abstracción con el que contamos actualmente para el desarrollo de sistemas de computación gráfica, son las *bibliotecas* (también llamadas *librerías*) gráficas, las cuales encapsulan también gran parte de las funciones y procedimientos necesarios en la tubería de procesos. En la última década, OpenGL fue adquiriendo hegemonía por ser lo suficientemente completa, versátil, y por permitir el desarrollo intermigrable entre diferentes plataformas. En la actualidad, paquetes y bibliotecas mucho más complejas y específicas como VTK, Fly3D y otras, asientan su funcionalidad sobre OpenGL.

Recapitulando, la clave del funcionamiento de la tarjeta gráfica no está en los requisitos de memoria, sino en la estructura del generador de video. El generador de video debe recorrer la memoria del buffer de pantalla y entregar las líneas de barrido al monitor dentro de una determinada norma de video. Dicha norma puede ser una norma de televisión (PAL o NTSC) o de monitor (1024×768, 800×600, etc.). Enton-

ces, el barrido es producido por un generador de barrido cuyas frecuencias horizontal y vertical son programables en función de la norma que se utiliza.

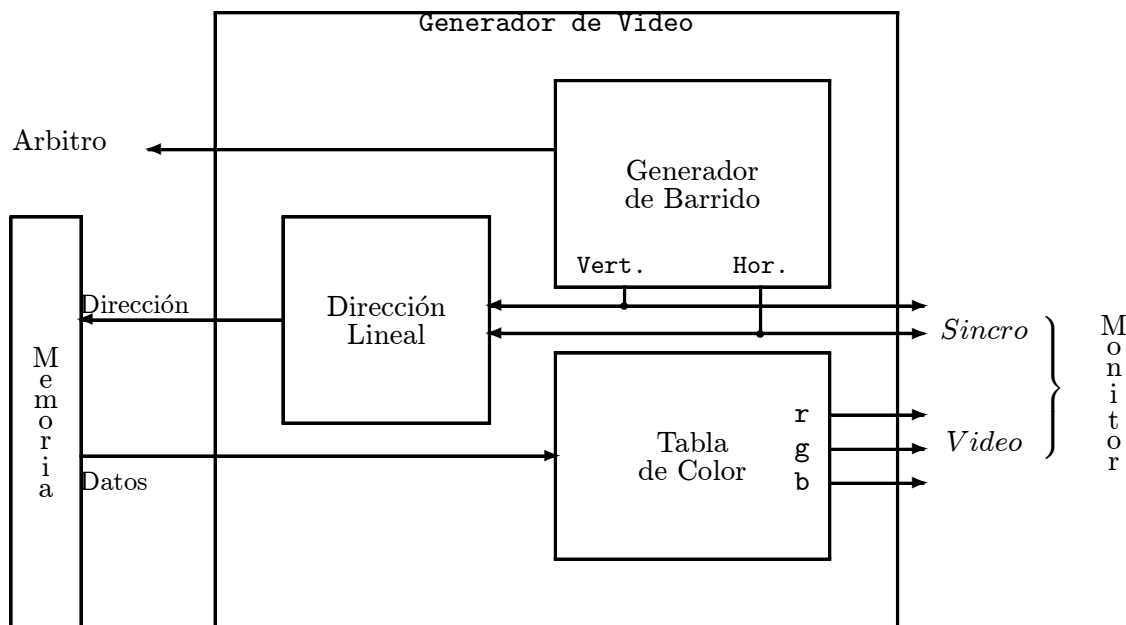


Figura 2.3 Estructura del generador de video.

Las señales de barrido son enviadas al monitor, pero también se utilizan para encontrar la posición de memoria en la cual está almacenada la información gráfica de cada pixel que constituye una línea de barrido. Esto se realiza por medio de una unidad aritmética que encuentra una dirección lineal a partir de los valores de la señal de barrido horizontal y vertical. La dirección lineal habilita la salida del valor almacenado en un lugar de la memoria del buffer de pantalla. Dicho valor es transformado en información gráfica por medio de una tabla de color (ver Figura 2.3), excepto en el modo *true color* que se explicará más adelante.

En las tarjetas gráficas se representa el color por medio del espacio cromático RGB (ver el Capítulo 5). Esto significa que el color de cada pixel se representa por medio de una terna de valores de las componentes en rojo, verde y azul, respectivamente, que tiene dicho color. Normalmente es innecesario almacenar dicha terna para cada pixel. En cambio se utiliza una tabla de colores predefinidos para lograr un uso más eficaz del buffer de pantalla, dado que en el mismo no se almacena el color del pixel sino el índice al color correspondiente en la tabla de colores. Por ejemplo, si se utilizan solamente

256 colores simultaneos, entonces es necesario almacenar solamente 1 byte de índice para cada pixel (ver Figura 2.4).

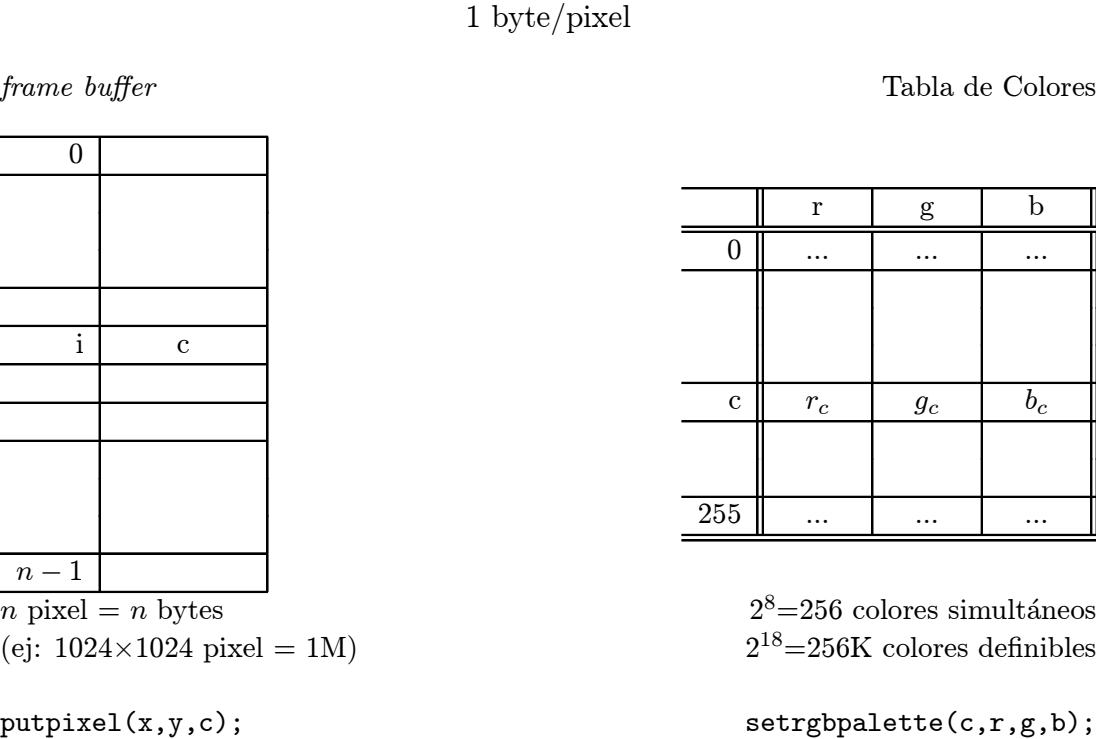


Figura 2.4 Modelo de memoria y tabla de colores a 1 byte/pixel.

En dicho modelo se utiliza la sentencia `putpixel(x,y,c)` para acceder al buffer de pantalla y setear el pixel en la posición (x, y) con el índice de color c, con x, y, c de tipo word. Para setear la tabla de colores se utiliza la sentencia `setrgbpalette(c,r,g,b)`, donde c es el índice del color a setear, y r, g, b son las componentes en el modelo RGB del color. En los modos gráficos VGA y super VGA, los parámetros r, g, b son de tipo word, pero se truncan los dos bit menos significativos, dado que el rango efectivo de cada componente es de 0 a 63. Esto permite definir 256 colores simultáneos de entre 256K colores definibles. Por lo tanto es conveniente utilizar una aritmética dentro de dicho rango para representar los colores, y multiplicar por 4 en el momento de la llamada.

El manejo se simplifica en el modelo *true color*, en el cual cada pixel tiene alocada la memoria para representar las componentes en RGB del color en que está seteado (ver Figura 2.5). Esto representa triplicar el tamaño del buffer de pantalla y el ancho de banda de la memoria asociada, lo cual tiene un costo bastante elevado, pero permite

*True Color*

0	
3i	$r_i$
3i+1	$g_i$
3i+2	$b_i$
3n - 1	

} pixel i

$n$  pixel =  $3n$  bytes  
(ej:  $1024 \times 1024$  pixel = 3M bytes)

$2^{24} = 16.7\text{M}$  colores simultáneos  
`picture.canvas.pixels[x][y] :=  $r_i + 256 * g_i + 65536 * b_i$ ;`  
`(i=x*picture.widht + y)`

Figura 2.5 Modelo de memoria true color.

representar 16.7M colores simultáneos (en el Capítulo 4 discutiremos si esta posibilidad de representación es suficiente). En realidad, las tarjetas modernas tienen 8 bytes por pixel (tres para el color, uno para el  $\alpha$  u opacidad, y cuatro para el *z-buffer*), y en algunos casos pueden haber otros 8 bytes más para buffer de texturas, iluminación, etc.

Como el modelo *true color* no está debidamente soportado en los modos antiguos de video, probablemente para poder acceder al buffer de pantalla se requiera acceder a la memoria por medio de interrupciones, reescribir los drivers de video, o instalar drivers especiales para tarjetas gráficas específicas. Afortunadamente, en los lenguajes visuales las componentes que tienen propiedades gráficas accesibles al programador son programables con el debido nivel de abstracción. Toda componente visual con propiedad *bitmap* tiene la posibilidad de que su apariencia sea modificada, a partir de cambiar el contenido de dicho bitmap. Por ejemplo, en la plataforma Delphi, las componentes de las clases `form`, `picture`, `panel`, etc., heredan la propiedad `canvas` que, como su nombre lo indica, permite que la apariencia visual de la componente sea modificada.

En particular, el bitmap de dicha propiedad es accesible por medio de la propiedad `pixels`, que es simplemente una matriz de tamaño idéntico al bitmap, y que en cada celda representa el color asociado al pixel respectivo. De esa manera, para modificar el contenido del pixel  $x, y$  de la componente, seteándolo al color  $(r, g, b)$ , es necesario modificar el contenido de dicha matriz por medio de una sentencia `picture.canvas.pixels[x][y] := r + 256 * g + 65536 * b;`.

## 2.2 Técnicas de Discretización

A partir de este punto, y en lo que resta del Capítulo, nos vamos a comprometer con el modelo de dispositivo de raster. Para conseguir independencia de dispositivo, entonces, es necesario adoptar un conjunto de primitivas y establecer una serie de métodos que permitan representar dichas primitivas en nuestro dispositivo de raster satisfaciendo un conjunto de especificaciones.

### 2.2.1 El sistema de coordenadas físico

El primer paso para conseguir una representación adecuada de las primitivas es caracterizar matemáticamente el medio que nos permite representarlas. Las primitivas gráficas independientes de dispositivo (en la “imagen” mental del usuario) normalmente se representan en un espacio Euclídeo de una determinada dimensión. En dichas condiciones un punto es una entidad matemática  $p = (x, y)$ , donde  $(x, y) \in \mathbb{R}^2$ .

En el soporte aritmético de la computadora, dicha representación se efectúa con los tipos de datos provistos, que pueden ser números reales con punto flotante de simple o doble precisión. Este espacio se denomina *espacio de la escena* y es uno de los muchos espacios que se utilizarán para factorizar adecuadamente las diversas tareas de un sistema gráfico.

Por último, en el soporte gráfico del buffer de pantalla, un punto se representa con un pixel, y dicha representación se efectúa seteando una posición de memoria con un contenido dado. Este espacio se denomina *espacio de pantalla* y se direcciona a partir del sistema de coordenadas físico, cuyo origen es el vértice superior izquierdo. Es posible encontrar varias correspondencias posibles entre el sistema de coordenadas físico y un sistema de coordenadas arbitrario en el espacio de la escena. En la literatura normalmente se considera que un pixel es un “punto con extensión” en el espacio de la escena, y por lo tanto el origen de dicho espacio coincide con el vértice superior

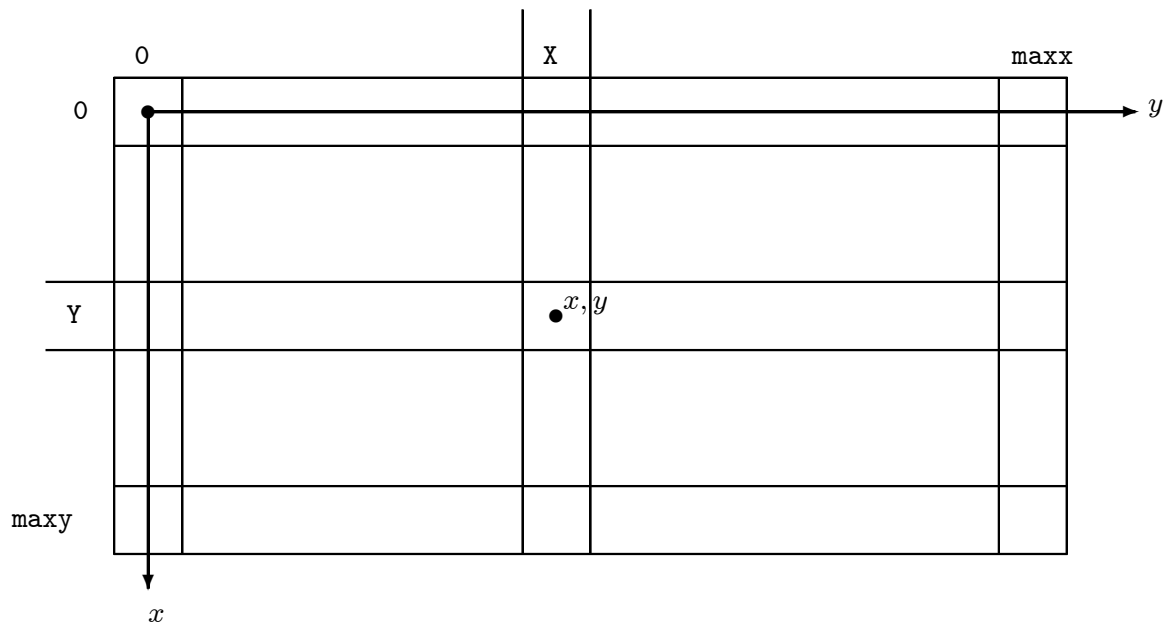


Figura 2.6 Sistema de coordenadas físico junto al espacio de la escena.

izquierdo del píxel  $(0,0)$  (ver por ejemplo [33, 40, 66, 84]). Desde nuestro punto de vista, esto no es del todo correcto, y parece más adecuado pensar que el origen del sistema de coordenadas de la escena está en el *centro* del píxel  $(0,0)$  (ver Figura 2.6).

De esa manera, el espacio de pantalla es un espacio discreto y acotado  $[0 \dots maxx] \times [0 \dots maxy]$ , con  $maxx, maxy \in \mathbb{N}$ , el cual está en correspondencia con el espacio de la escena (euclídeo)  $(x,y) \in \mathbb{R}^2$  a través de las operaciones

$X := \text{round}(x);$

$Y := \text{round}(y);$

Por dicha razón es que la operación de llevar una primitiva del espacio de la escena al espacio de pantalla se denomina *discretización*.

### 2.2.2 Primitivas gráficas

Es muy difícil escoger un conjunto de primitivas gráficas que sea adecuado para la representación de todo tipo de entidades gráficas. Sin embargo, el siguiente subconjunto en la práctica resulta suficiente:

**Puntos:** Se especifican a partir de su localización y color. Su discretización es directa, como se mencionó más arriba.

**Segmentos de recta:** Son esenciales para la mayor parte de las entidades. Se especifican a partir de un par de puntos que representan sus extremos.

**Circunferencias:** En algunos casos representar entidades curvadas con segmentos poligonales puede ser inadecuado o costoso, por lo que en la práctica las circunferencias o círculos se adoptan como primitivas. Se especifican con la posición de su centro y su radio.

**Polígonos:** Son indispensables para representar entidades sólidas. Se representan a partir de la secuencia de puntos que determina la poligonal de su perímetro.

### 2.2.3 Especificaciones de una discretización

En el momento de escoger un método de discretización para una primitiva gráfica, es indispensable contar con criterios que permitan evaluar y comparar las ventajas y desventajas de las distintas alternativas. Entre todas las especificaciones posibles, podemos mencionar las siguientes:

**Apariencia:** Es la especificación más obvia, aunque no es fácil describirla en términos formales. Normalmente se espera que un segmento de recta tenga una “aparición recta” más allá de que se hallan escogido los pixels matemáticamente más adecuados. Tampoco debe tener discontinuidades o puntos espúreos. Debe pasar por la discretización del primer y último punto del segmento. Debe ser uniforme, etc.

**Simetría e invariancia geométrica:** Esta especificación se refiere a que un método de discretización debe producir resultados equivalentes si se modifican algunas propiedades geométricas de la primitiva que se está discretizando. Por ejemplo, la discretización de un segmento no debe variar si dicho segmento se traslada a otra localización en el espacio, o si es rotado, etc.

**Simplicidad y velocidad de cómputo:** Como los métodos tradicionales de discretización de primitivas se desarrollaron hace tres décadas, en momentos en que las posibilidades del hardware y software eran muy limitadas, los resultados eran muy sensibles al uso de memoria u operaciones aritméticas complejas. Por lo tanto, los métodos tienden a no depender de estructuras complejas y a ser directamente implementables en hardware específico de baja complejidad.

#### 2.2.4 Métodos de discretización

Dada una primitiva gráfica a discretizar, debemos encontrar los pixels que la representen de la manera más correcta posible. Para ello, lo más adecuado es caracterizar matemáticamente a dicha primitiva, de modo que su discretización pueda efectuarse en forma sencilla. Entre los diversos métodos que pueden plantearse destacamos los dos siguientes:

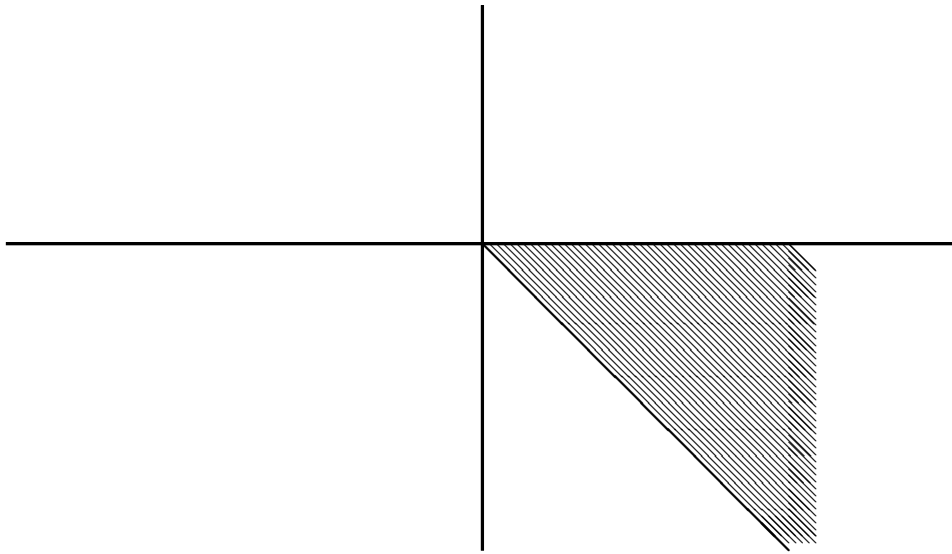
**Evaluar su ecuación diferencial a diferencias finitas:** Este método, denominado DDA (discrete difference analyzer) consiste en plantear la ecuación diferencial de la primitiva a discretizar, y luego evaluar dicha expresión a intervalos adecuados.

**Análisis del error:** Estos métodos fueron desarrollados por Bresenham [18] y se basan en analizar, dado un pixel que pertenece a la discretización de la primitiva, cuál es el próximo pixel que minimiza una determinada expresión que evalúa el error que comete la discretización.

### 2.3 Discretización de Segmentos de Rectas

El análisis de los métodos de discretización de rectas parte de considerar el comportamiento esperado en determinados casos particulares. Dichos casos surgen de suposiciones específicas que simplifican el problema, pero que al mismo tiempo se pueden generalizar a todos los demás casos por medio de simetrías. Dado un segmento de recta que va de  $(x_0, y_0)$  a  $(x_1, y_1)$ , se supone que

- $\Delta x = (x_1 - x_0) \geq 0$ ,
- $\Delta y = (y_1 - y_0) \geq 0$ , y
- $\Delta x \geq \Delta y$ .



**Figura 2.7** Situación particular para derivar los algoritmos de discretización de segmentos de recta.

Esto equivale a trabajar en el “octavo” del espacio de pantalla sombreado en la Figura 2.7, donde el origen es el pixel que corresponde a la discretización del punto  $(x_0, y_0)$  y la zona sombreada a los lugares donde puede ubicarse el punto  $(x_1, y_1)$ .

### 2.3.1 Segmentos de recta DDA

Como ya se mencionara, los métodos DDA evalúan la ecuación diferencial de la primitiva a intervalos finitos. En el caso particular de los segmentos de recta, la ecuación diferencial es

$$\frac{dy}{dx} = \frac{\Delta y}{\Delta x} = m.$$

El método busca encontrar una secuencia de  $n + 1$  puntos tales que  $(x_0, y_0) = (x^0, y^0), (x^1, y^1), \dots, (x^n, y^n) = (x_1, y_1)$ . La discretización de cada uno de ellos son los pixels de la discretización del segmento. Esta propiedad, si bien es trivial, es de gran importancia porque determina que la discretización de un segmento de recta es invariante frente a transformaciones afines. Esto significa que es equivalente transformar los extremos del segmento y discretizar el segmento transformado, o discretizar primero y transformar cada punto obtenido. Sin embargo, la primera alternativa es mucho más eficiente.

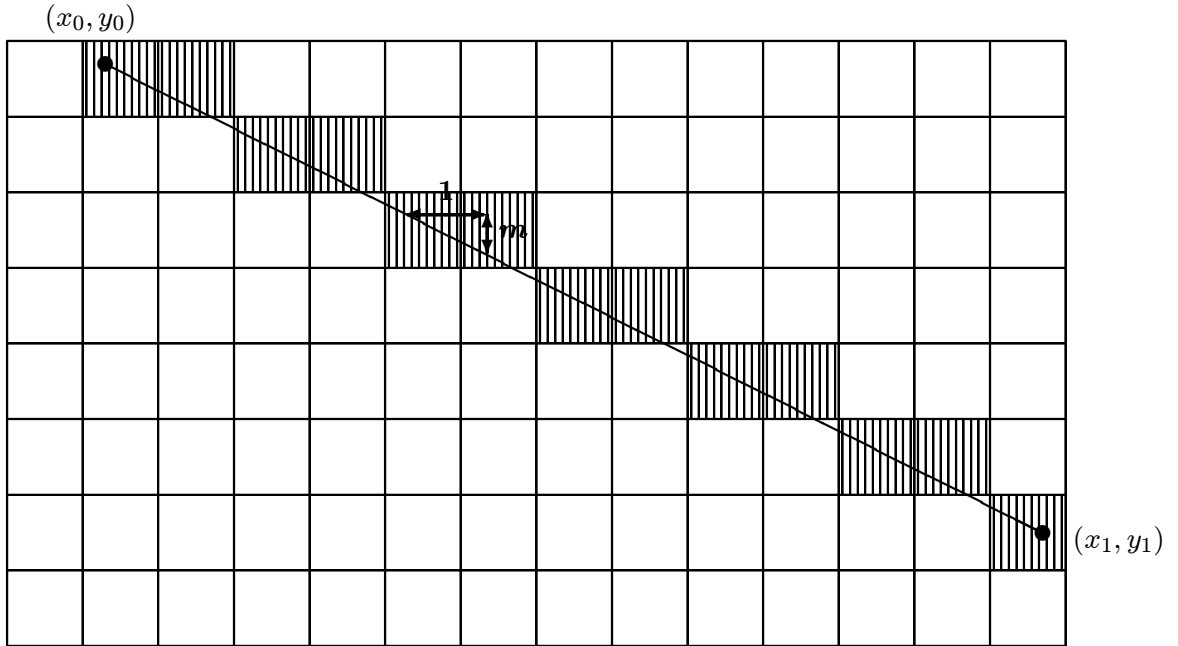


Figura 2.8 La discretización de un segmento por DDA.

Dada la ecuación diferencial y un incremento finito arbitrario  $\varepsilon = \frac{1}{n}$ , podemos pasar de un pixel dado de la secuencia al siguiente por medio de la expresión

$$\begin{cases} x^{k+1} &= x^k + \varepsilon \Delta x \\ y^{k+1} &= y^k + \varepsilon \Delta y \end{cases}$$

$\varepsilon$  determina la “frecuencia” de muestreo del segmento. Un valor muy pequeño determina que muchas muestras producirán puntos que serán discretizados al mismo pixel. Por el contrario, un valor muy grande determinará que el segmento aparezca “punteado” en vez de ser continuo como corresponde. Un valor práctico es elegir  $\varepsilon \Delta x = 1$  y por lo tanto  $n = \Delta x$ , es decir, la discretización tiene tantos pixels como longitud tiene el segmento en la variable que más varía (más uno, dado que la secuencia tiene  $n + 1$  puntos). Al mismo tiempo es fácil ver que

$$\varepsilon \Delta y = \frac{\Delta y}{\Delta x} = m.$$

En la Figura 2.8 es posible ver el resultado de discretizar un segmento particular por el método DDA. Obsérvese que los puntos extremos  $(x_0, y_0)$  a  $(x_1, y_1)$  son en efecto

```

procedure linea(x0,x1,y0,y1:real;col:integer);
var x,y,m:real;
begin
    m := (y1-y0)/(x1-x0);
    x := x0; y := y0;
    while x<=x1 do begin
        putpixel(round(x),round(y),col);
        x := x+1;
        y := y+m;
    end;
end;

```

Figura 2.9 Algoritmo DDA para segmentos de recta.

puntos y por lo tanto están ubicados en cualquier lugar dentro del pixel que corresponde a su discretización. Un algoritmo sencillo que computa la discretización de un segmento de recta por el método DDA se muestra en la Figura 2.9. Obsérvese que es necesario computar las variables en punto flotante, y que además se requiere una división en punto flotante.

Para poder discretizar un segmento de recta en cualquiera de las otras siete posibilidades ( $\Delta x < 0$ ,  $\Delta y < 0$ , o  $|\Delta x| < |\Delta y|$ ), es necesario considerar las simetrías que se aplican. Si por ejemplo no se cumple que  $\Delta y = (y_1 - y_0) \geq 0$ , entonces hay que considerar pendientes negativas (simetría A), caso que el algoritmo de la Figura 2.9 realiza automáticamente. En cambio, si  $\Delta x = (x_1 - x_0) < 0$ , entonces es necesario *decrementar* a  $x$  en el ciclo e iterar mientras no sea *menor* que  $x_1$  (simetría B). Por último, si no se cumple que  $\Delta x \geq \Delta y$ , entonces es necesario intercambiar los roles de las variables  $x$  e  $y$  (simetría C). Cualquier combinación de situaciones se puede resolver con combinaciones de simetrías (ver Figura 2.10).

### 2.3.2 Segmentos de rectas por Bresenham

En el algoritmo DDA para segmentos de recta es necesario computar sumas entre las variables en punto flotante, y además se requiere una división en punto flotante para computar la pendiente. El mérito del algoritmo que vamos a presentar consiste en que todas las operaciones se realizan en aritmética entera por medio de operaciones sencillas, y por lo tanto, su ejecución es más rápida y económica, y es de fácil implementación con hardware específico [18].

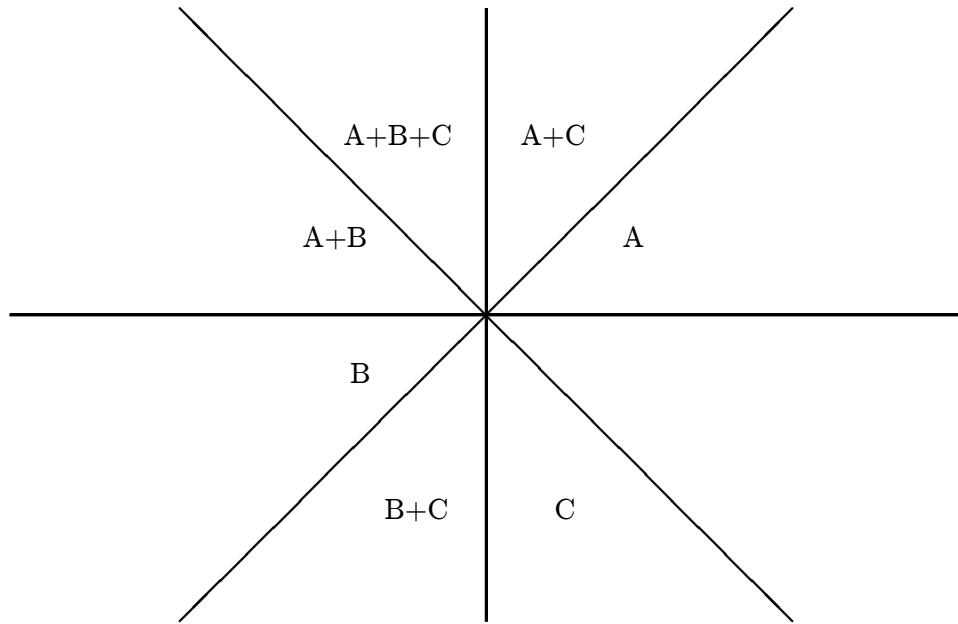


Figura 2.10 Simetrías para los demás casos.

El punto de partida del análisis es el siguiente. Si la discretización de los puntos extremos del segmento debe pertenecer a la discretización del segmento, entonces es conveniente efectuar la llamada al algoritmo luego de discretizar los extremos. Esto significa que  $(x_0, y_0)$  y  $(x_1, y_1)$ , y por lo tanto  $\Delta x$  y  $\Delta y$  son enteros. Luego, si  $p$  es un pixel que pertenece a la discretización del segmento, entonces en las condiciones particulares mencionadas más arriba, el proximo pixel solamente puede ser el ubicado a la derecha (**E** o “hacia el este”), o el ubicado en diagonal hacia la derecha y hacia abajo (**D** o “en diagonal”, ver Figura 2.11).

La decisión de ir hacia el paso **E** o **D** se toma en función del error que se comete en cada caso. En este algoritmo se considera que el error es la distancia entre el centro del pixel elegido y el segmento de recta, medida en dirección del eje **Y** positivo del espacio de pantalla (es decir, hacia abajo). Si el error en  $p$  fuese cero, entonces al ir hacia **E** el error pasa a ser  $m$  (la pendiente del segmento), y en **D** el error pasa a ser  $m - 1$  (ver Figura 2.12).

En general, si en  $p$  el error es  $e$ , la actualización del error es

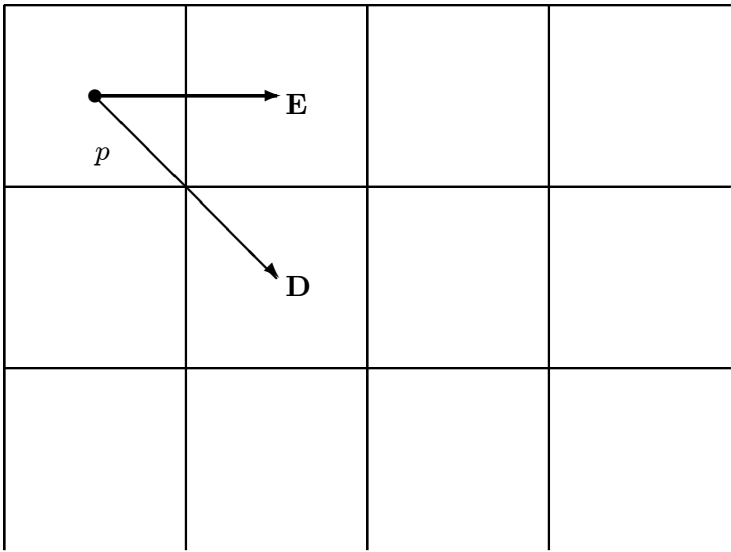


Figura 2.11 Si  $p$  pertenece a la discretización, el próximo pixel solo puede ser **E** o **D**.

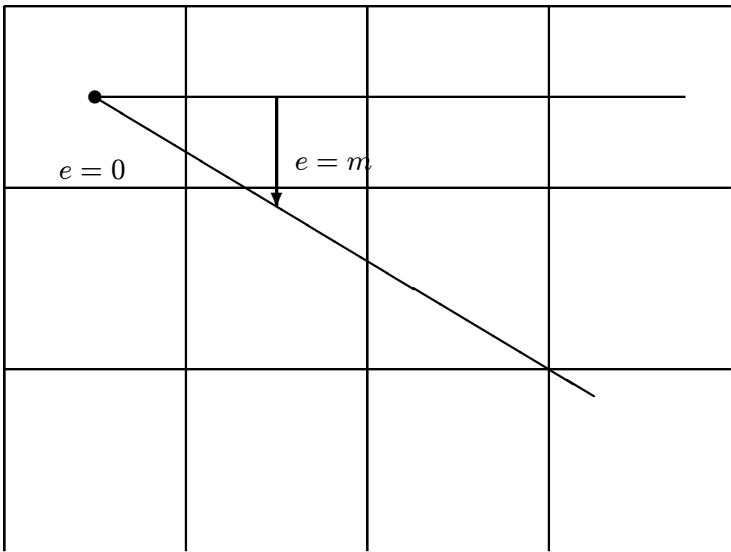


Figura 2.12 Actualización del error.

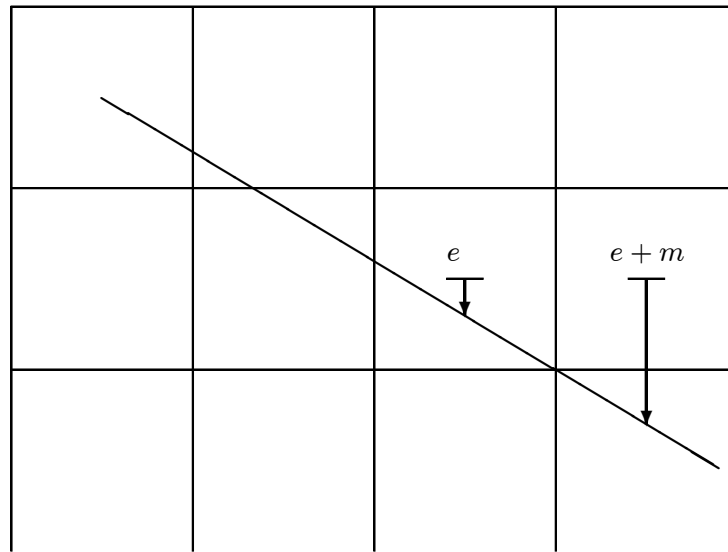


Figura 2.13 Elección del próximo píxel.

- Paso **E**  
 $e := e + m$
- Paso **D**  
 $e := e + m - 1$

Por lo tanto, la elección del paso **E** o **D** depende de que el valor absoluto de  $e + m$  sea o no menor que el valor absoluto de  $e + m - 1$ . Expresado de otra manera, sea  $e$  el error en un determinado píxel. Si  $e + m > 0.5$  entonces el segmento de recta pasa más cerca del píxel **D**, y si no, pasa más cerca del píxel **E** (ver Figura 2.13).

Una de las economías de cómputo del método proviene de poder testear el error preguntando por cero. Es fácil ver que si se inicializa el error en

$$e_o = m - 0.5,$$

entonces en cada paso hay que chequear  $e > 0$  para elegir **D**. La otra economía proviene de realizar manipulaciones algebraicas para efectuar un cómputo equivalente pero en aritmética entera. Como se testea el error por cero, multiplicar el error por una constante no afecta el resultado. Por lo tanto, multiplicamos el error por  $2\Delta x$ . A partir de dicho cambio, se constatan las siguientes igualdades:

```

procedure linea(x0,x1,y0,y1,col:integer);
var x,y,dx,dy,e,ix,iy:integer;
begin
  dx := x1 - x0; dy := y1 - y0;
  ix := 2*dx;    iy := 2*dy;
  y := y0;      e := iy - dx;
  for x := x0 to x1 do begin
    putpixel(x,y,col);
    if e>0 then do begin
      y := y+1;
      e := e-ix;
    end;
    e := e+iy;
  end;
end;

```

Figura 2.14 Algoritmo de Bresenham para segmentos de recta.

- $e_o = 2\Delta x(\frac{\Delta y}{\Delta x} - 0.5) = 2\Delta y - \Delta x$ .
- Paso **E**  
 $e := e + 2\Delta y$ .
- Paso **D**  
 $e := e + 2(\Delta y - \Delta x)$ .

Todas las operaciones, entonces, se efectúan en aritmética entera.

La implementación del algoritmo de Bresenham para segmentos de recta se muestra en la Figura 2.14. Teniendo en cuenta que los productos por 2 en aritmética entera se efectúan con un desplazamiento de bits a izquierda, es posible observar que el mismo utiliza operaciones elementales e implementables con hardware específico muy sencillo.

## 2.4 Discretización de Circunferencias

Como en el caso de los segmentos de recta, en la discretización de circunferencias o círculos trabajaremos solamente en una parte de la misma, y obteniendo las demás

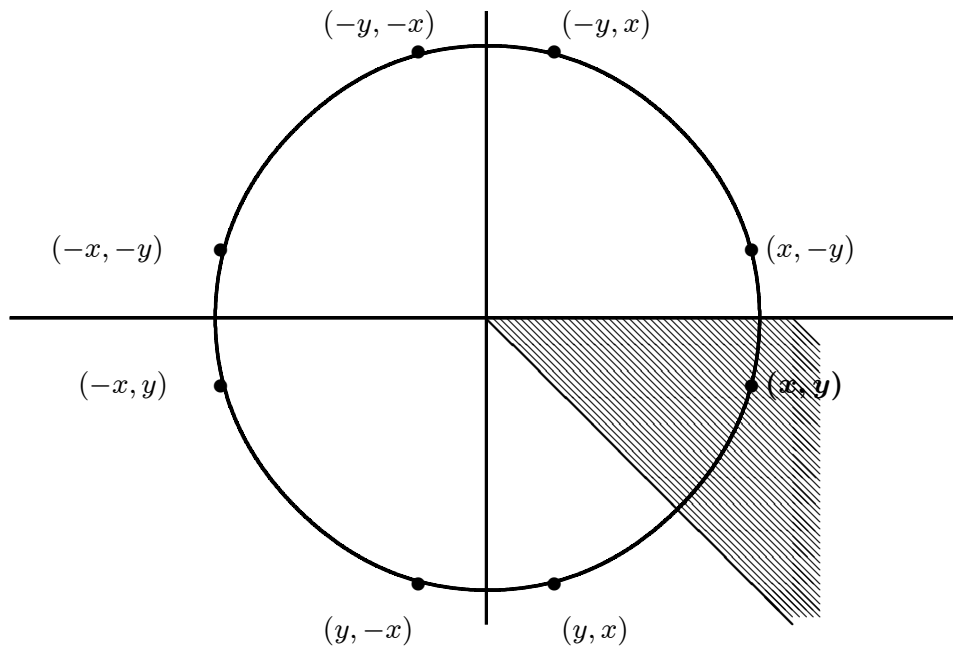
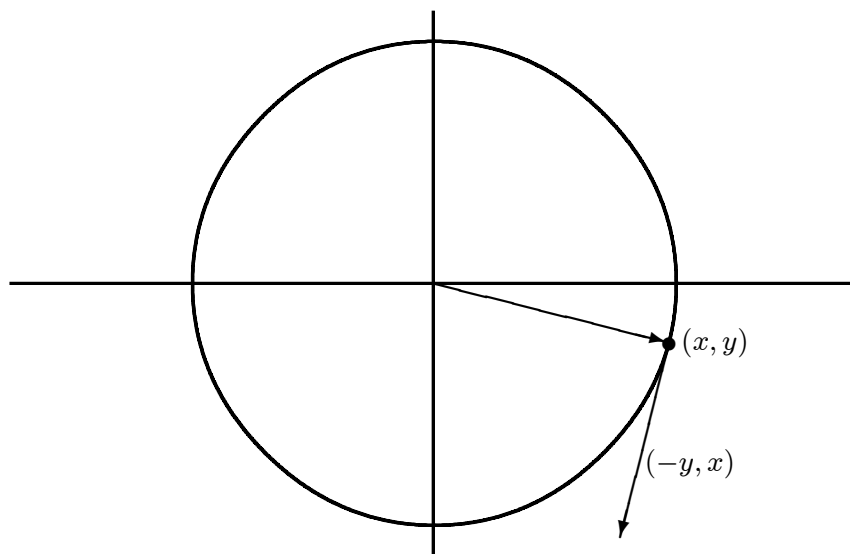


Figura 2.15 Simetrías para la discretización de circunferencias.

partes por simetría. Supongamos que la circunferencia a discretizar está centrada en el origen, y que  $p = (x, y)$  es un pixel de su discretización tal que

- $x \geq 0$ ,
- $y \geq 0$ ,
- $x \geq y$ .

Dicha situación describe un octavo de la circunferencia, pero las partes faltantes pueden encontrarse por medio de las simetrías mostradas en la Figura 2.15.



**Figura 2.16** Interpretación geométrica de la ecuación diferencial de la circunferencia centrada en el origen.

### 2.4.1 Discretización de circunferencias por DDA

Sea una circunferencia de radio  $r$  centrada en el origen y sea  $p = (x, y)$  un punto que pertenece a la misma. Entonces, la ecuación diferencial de la circunferencia en dicho punto (ver Figura 2.16) es

$$\frac{dy}{dx} = -\frac{x}{y}.$$

La evaluación de la ecuación diferencial por diferencias finitas, como en el caso de los segmentos de recta, consiste en encontrar una secuencia de valores, de modo que dado un pixel de la discretización  $(x_k, y_k)$ , el próximo pixel se encuentra con

$$\begin{cases} x_{k+1} &= x_k - \varepsilon y_k \\ y_{k+1} &= y_k + \varepsilon x_k \end{cases} \quad (2.1)$$

$\varepsilon$  determina la “frecuencia” de muestreo de la circunferencia. Valores pequeños determinan un cómputo redundante, y valores muy grandes determinan un cubrimiento desperejo. Un valor práctico es elegir  $\varepsilon x = 1$  y por lo tanto  $\varepsilon y = \frac{y}{x}$  (ver Figura 2.17).

El primer pixel de la secuencia es  $p_0 = (r, 0)$ , el cual pertenece a la discretización de la circunferencia y además se computa en forma directa.

```

procedure circ(radio:real;col:integer);
var x,y:integer;
    rx:real;
begin
    rx:=radio;
    x:=round(rx); y:=0;
    while (y<x) do begin
        putpixel(x,y,col);  putpixel(y,x,col);
        putpixel(-x,y,col); putpixel(-y,x,col);
        putpixel(x,-y,col); putpixel(y,-x,col);
        putpixel(-x,-y,col); putpixel(-y,-x,col);
        rx:=rx-y/rx;
        x:=round(rx);
        y:=y+1;
    end;
end;

```

Figura 2.17 Algoritmo DDA para discretización de circunferencias.

Es importante tener en cuenta que el sistema de ecuaciones 2.1 puede representarse como una transformación:

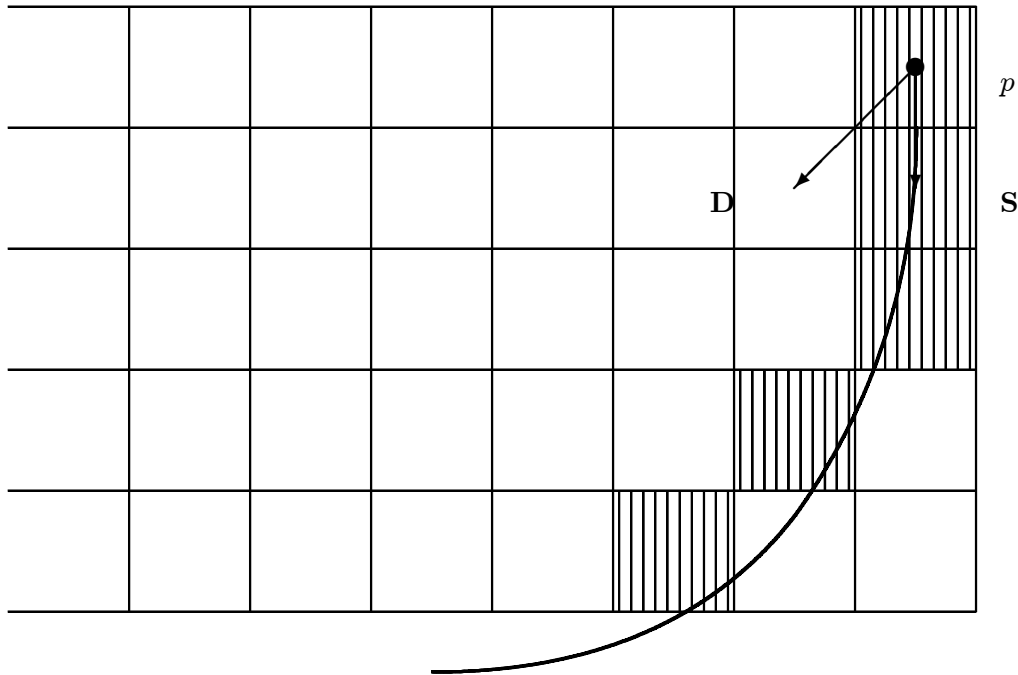
$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & -\varepsilon \\ \varepsilon & 1 \end{bmatrix} \begin{bmatrix} x_k \\ y_k \end{bmatrix},$$

la cual tiene determinante  $1 + \varepsilon^2$ . Esto significa que la discretización es levemente espi-  
ralada hacia afuera. Una forma de solucionar este problema es utilizar una recurrencia  
cuya transformación sea

$$\begin{bmatrix} 1 & -\varepsilon \\ \varepsilon & 1 - \varepsilon^2 \end{bmatrix}.$$

Esta transformación representa en realidad una elipse y no una circunferencia, pero  
la excentricidad de la misma es del orden de  $\varepsilon^2$  y por lo tanto la diferencia es más  
reducida. Utilizar dicha expresión altera la evaluación de  $y_{k+1}$ :

$$\begin{aligned} y_{k+1} &= y_k(1 - \varepsilon^2) + \varepsilon x_k \\ y_{k+1} &= y_k - \varepsilon(\varepsilon y_k - x_k) \\ y_{k+1} &= y_k - \varepsilon x_{k+1}. \end{aligned}$$



**Figura 2.18** Si  $p$  pertenece a la discretización, el próximo píxel solo puede ser **S** o **D**.

Por lo tanto, la recurrencia para computar la discretización de una circunferencia centrada en el origen con DDA queda modificada a

$$\begin{cases} x_{k+1} &= x_k - \varepsilon y_k \\ y_{k+1} &= y_k + \varepsilon x_{k+1} \end{cases}$$

De todas maneras, la desventaja más importante de este algoritmo es que debe realizar una división en punto flotante para cada paso.

#### 2.4.2 Discretización de Bresenham para circunferencias

Como en el caso planteado para segmentos de recta, este método se basa en analizar el error entre la verdadera circunferencia y su discretización. Si  $p$  pertenece a la discretización, el próximo píxel de la discretización solo puede ser **S** (“ir hacia el sur”) o **D** (“ir en diagonal”, ver Figura 2.18).

El error (en este caso la distancia al cuadrado) de un pixel  $p = (x, y)$  a una circunferencia de radio  $r$  con centro en  $(0, 0)$  es:

$$e = x^2 + y^2 - r^2.$$

Si elegimos el paso **S** entonces el próximo pixel es  $p_S = (x, y + 1)$ , y el error pasa a ser

$$e_S = (x)^2 + (y + 1)^2 - r^2 = e + 2y + 1.$$

Si elegimos el paso **D** entonces el próximo pixel es  $p_D = (x - 1, y + 1)$ , y el error pasa a ser

$$e_D = (x - 1)^2 + (y + 1)^2 - r^2 = e_S - 2x + 1.$$

La elección de un paso **S** o **D** dependerá de cuál error tiene menor módulo:

$$\text{Si } |e + 2y + 1| > |e + 2y + 1 - 2x + 1| \text{ entonces } \mathbf{D}.$$

Teniendo en cuenta que tanto en **D** como en **S** se incrementa  $y$ , entonces se actualiza el error a  $e_S$  antes de la comparación.

$$\text{Si } |e_S| > |e_S - 2x + 1| \text{ entonces } \mathbf{D}.$$

Por último, teniendo en cuenta que  $e_S - 2x + 1$  es siempre negativo (recordar que  $x > 0, x > y$ ), entonces:

$$\text{Si } 2e_S > 2x - 1 \text{ entonces } \mathbf{D}.$$

De esa manera, el algoritmo queda planteado exclusivamente con operaciones enteras y de aritmética sencilla (ver Figura 2.19.)

## 2.5 Discretización de Polígonos

El objetivo de la discretización de polígonos es encontrar el conjunto de pixels que determinan el área sólida que dicho polígono cubre en la pantalla. Si bien existen diversos métodos, aquí presentaremos el más económico y difundido, que se basa en encontrar la intersección de todos los lados del polígono con cada línea de barrido (a y constante), por lo que el método se denomina *conversión scan* del polígono. Este método es de gran importancia porque se generaliza a una clase de algoritmos denominados *scan-line* para resolver determinados problemas de iluminación y sombreado en tres dimensiones (ver Capítulo 7).

Todo polígono plano puede descomponerse en triángulos. Por lo tanto el triángulo será la base del análisis de la conversión scan de polígonos en general. Para computarla es necesario dimensionar dos arreglos auxiliares de enteros **minx**, **maxx** (llamados *bucket* de pantalla), que para cada línea de barrido almacenarán el menor y mayor  $x$  respectivamente (ver Figura 2.20).

```
procedure circ(radius:real;col:integer);
var x,y,e:integer;
begin
  x:=radius; y:=0;
  e:=0;
  while (y<x) do begin
    putpixel(x,y,col); putpixel(y,x,col);
    putpixel(-x,y,col); putpixel(-y,x,col);
    putpixel(x,-y,col); putpixel(y,-x,col);
    putpixel(-x,-y,col);putpixel(-y,-x,col);
    e:=e+2*y+1;
    y:=y+1;
    if ((2*e)>(2*x-1)) then do begin
      x:=x-1;
      e:=e-2*x+1;
    end;
  end;
end;
```

**Figura 2.19** Algoritmo de Bresenham para circunferencias.

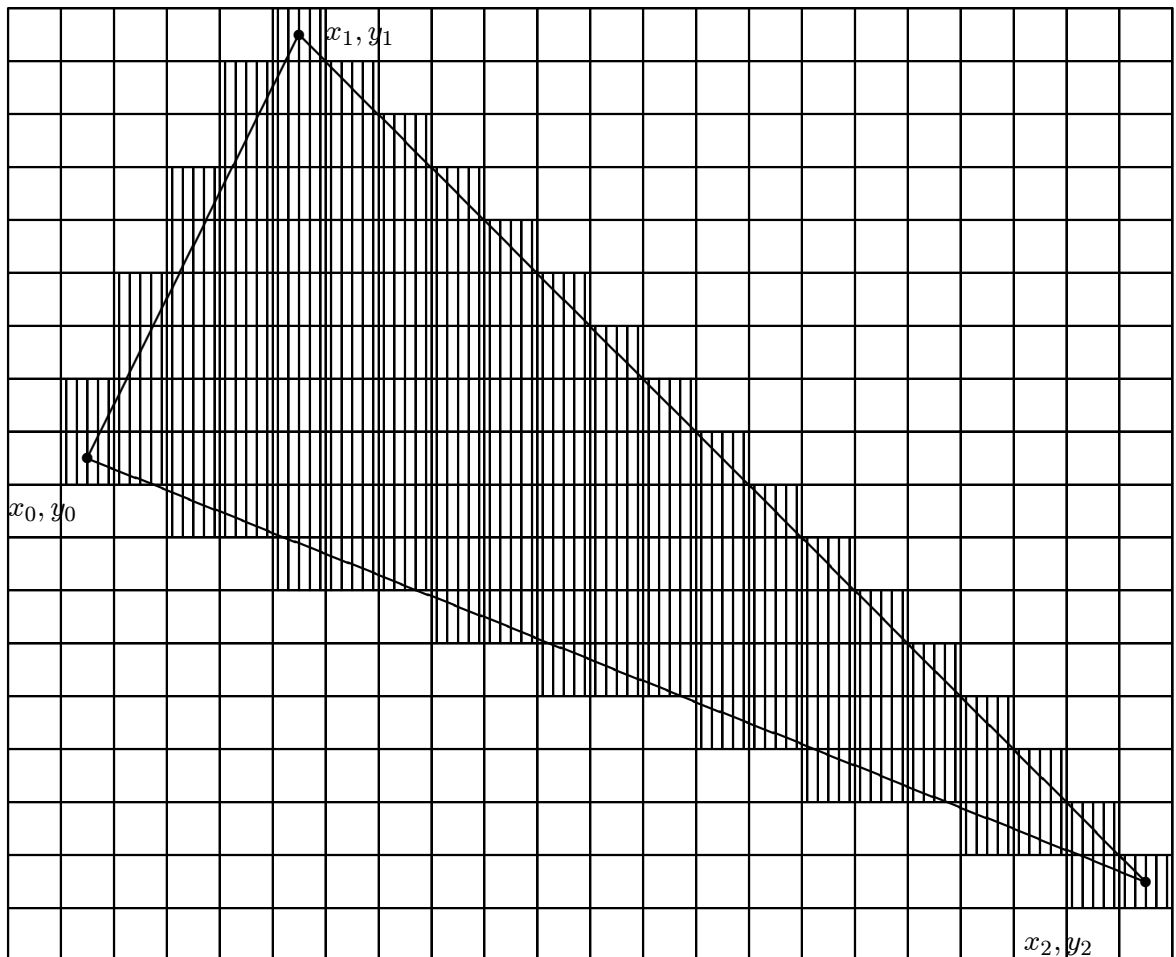


Figura 2.20 Conversión scan de un polígono.

1. Inicializar `minx` a infinito y `maxx` a menos infinito.
2. Discretizar cada arista del triángulo (con DDA o Bresenham) reemplazando la sentencia  
`putpixel(x,y,col);`  
o su equivalente, por  
`if x>maxx[y] then maxx[y]:=x;`  
`if x<minx[y] then minx[y]:=x;`
3. Para cada `y` activo graficar una línea de `minx[y]` a `maxx[y]`.

Es interesante ver cómo se puede extender este algoritmo para generar polígonos con color interpolado. Este método, denominado *sombreado de Gouraud*, del cual nos ocuparemos en el Capítulo 7, genera un color para cada vértice de cada polígono, el cual es luego interpolado en cada pixel interior por medio de interpolación (bi-)lineal. La primera interpolación se realiza en el perímetro del polígono, donde el algoritmo incremental de discretización de segmentos genera los colores a lo largo del perímetro, colores que son almacenados en un bucket de pantalla modificado a tal efecto. Por último, en el paso final de la conversión scan, se genera la línea que corresponde a cada línea de barrido, realizando una segunda interpolación lineal entre los colores almacenados en el bucket para dicha línea.

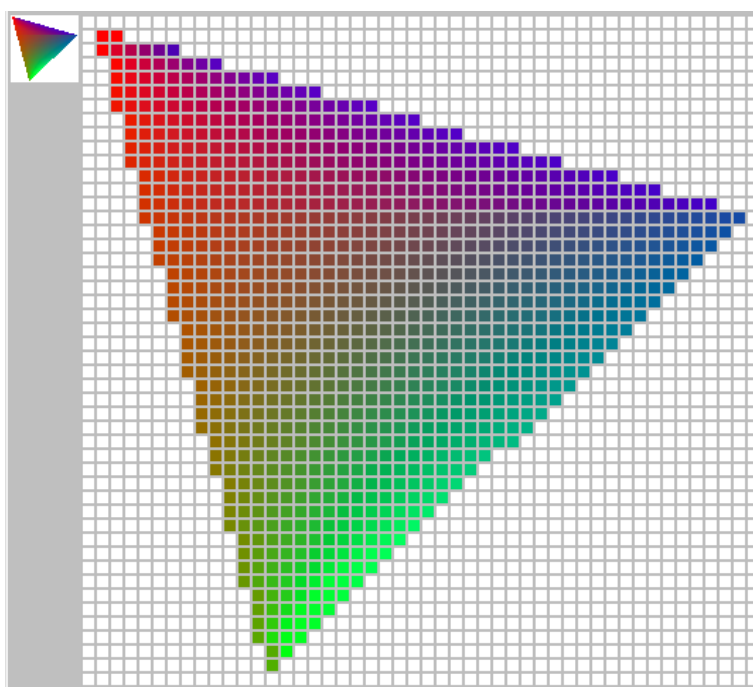


Figura 2.21 *Polígono con color interpolado*

## 2.6 Ejercicios

1. Implementar los algoritmos de discretización de segmentos de recta por DDA y Bresenham con todas las simetrías.
2. Implementar los algoritmos de discretización de circunferencias por DDA y Bresenham. Modificar el DDA para minimizar el error. Modificar ambos algoritmos para que reciban las coordenadas del centro.
3. Implementar el algoritmo de conversión scan para triángulos. ¿Es necesario modificarlo sustancialmente para cualquier polígono convexo?

## 2.7 Bibliografía recomendada

La mayor parte de los trabajos que dieron origen a los algoritmos presentados en este Capítulo pertenecen a publicaciones virtualmente inaccesibles (excepto en colecciones de *reprints* de trabajos clásicos, ver por ejemplo [36]).

Una descripción interesante de los dispositivos gráficos puede encontrarse en los Capítulos 2 y 19 del libro de Newman y Sproull [66], y en el Capítulo 7 del libro de Giloi [40]. Una descripción más moderna puede encontrarse en el Capítulo 4 del libro de Foley et. al. [33]. Los interesados en conocer arquitecturas avanzadas para sistemas gráficos pueden consultar además el Capítulo 18 del mismo libro. La discusión relacionada con la correspondencia entre el espacio de los objetos y el espacio de pantalla está inspirada en las conclusiones de un trabajo de Jim Blinn [12].

La descripción más directa de los algoritmos de discretización de segmentos y circunferencias puede encontrarse en el Capítulo 2 del libro de Newman y Sproull. Aunque difieren de la presentada aquí por adoptarse distintas convenciones, los desarrollos son similares. En el Capítulo 3 del libro de Foley et. al. es posible encontrar una detallada descripción de varios algoritmos que generalizan los presentados en este Capítulo (por ejemplo, la discretización de segmentos de un determinado grosor, o el “llenado” de polígonos con patrones repetitivos). La conversión scan de polígonos es tratada en el Capítulo 16 del libro de Newman y Sproull.

