Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación

Lecturas en Ciencias de la Computación
ISSN 1316-6239

Despliegue de Primitivas 2D

Prof. Rhadamés Carmona.

ND 2006-03

Laboratorio de Computación Gráfica
Caracas, Mayo, 2006.

UNIVERSIDAD CENTRAL DE VENEZUELA
FACULTAD DE CIENCIAS
ESCUELA DE COMPUTACIÓN

Lecturas en Ciencias de la Computación
ISSN 1316-6239

Despliegue de Primitivas 2D

Carmona Rhadamés

ND 2006- 03

Laboratorio de Computación Gráfica
Caracas, 08 de Mayo de 2006

N

ND 2006-03
Lectura en Ciencias de la Computación
ISSN 1316-6239

DESPLIEGUE DE PRIMITIVAS 2D

Rhadamés Carmona

Mayo 2006

Resumen

En esta guía se estudian los algoritmos incrementales para el despliegue de primitivas gráficas, como la línea, el círculo, la elipse, rectángulos y triángulos. Un algoritmo incremental en este contexto consiste en expresar la solución algorítmica a través de sumas y restas, y en lo posible con aritmética entera. El análisis de los algoritmos es presentado de manera tal, que el estudiante pueda desarrollar algoritmos incrementales para primitivas gráficas 2D menos comunes en los paquetes gráficos como la parábola y la hipérbola, y más aún, mejorar la eficiencia de programas que realizan cálculo redundante, deduciendo el incremento en variables o expresiones entre iteraciones consecutivas.

Laboratorio de Computación Gráfica
Escuela de Computación, Facultad de Ciencias, Universidad Central de Venezuela
Los Chaguaramos. Apartado 47002 Caracas 1041-A Venezuela
rcarmona@strix.ciens.ucv.ve

Notas preliminares

Este material va dirigido a los estudiantes de Introducción a la Computación Gráfica, y corresponde al primer tema de la materia: despliegue de primitivas gráficas 2D. Los algoritmos aquí presentados no están totalmente optimizados. Sin embargo, se sugieren las herramientas para mejorar la eficiencia de los algoritmos. Entre estas se encuentran las operaciones de corrimiento de registros, diferencias de segundo orden, factorización del algoritmo y simplificación de fórmulas. Durante el desarrollo de estas notas, se asignan tareas para que el estudiante implemente los algoritmos eficientemente.

Objetivos

- Que el estudiante esté en capacidad de implementar los algoritmos de despliegue de primitivas 2D fundamentales, como lo son: línea, círculo, elipse, triángulo, rectángulo, de manera eficiente y con aritmética entera.
- Realizar e implementar algoritmos incrementales basados en sumas y restas, e incentivar al estudiante a aplicar estas técnicas a otros problemas cotidianos de la computación, o bien otras primitivas gráficas menos utilizadas en los paquetes CACG como la parábola y la hipérbola.

Contenido

Introducción

El despliegue de una primitiva 2D consiste por lo general en determinar cuales píxeles del monitor activar, para lograr una aproximación visual a la misma. Cada píxel en pantalla puede ser visto como un cuadrado 1x1, haciendo abstracción del radio aspecto, y tiene un color RGB asociado. Estos píxeles tienen a su vez un centro, el cual por lo general tiene coordenadas enteras, que van desde el (0,0) hasta (*width*-1, *height*-1), en donde *width*height* es la resolución en píxeles del monitor. Los centros de píxeles están igualmente espaciados y distanciados en una unidad; esto nos permite diseñar algoritmos incrementales con aritmética entera para acelerar la ejecución del despliegue de primitivas gráficas bidimensionales.

Los algoritmos incrementales más famosos, son los propuestos por Bresenham, quien en 1965 propuso el algoritmo incremental para despliegue de líneas. En esta guía se estudian los algoritmos de despliegue de líneas, círculos, elipses, rectángulos y triángulos.

1.- Despliegue de líneas

Una línea en pantalla la podemos definir como el segmento de recta delimitado por dos puntos (x_0,y_0) y (x_1,y_1), en donde $x_0, y_0, x_1, y_1 \in \boldsymbol{Z}^+$. Por conveniencia, asumiremos que los extremos del segmento de recta tienen coordenadas enteras, puesto que por lo general son introducidos vía clic del ratón. Debido a que cada píxel es un cuadrado, los índices enteros (x_i,y_i) se refieren al centro de un píxel. El píxel (x_i, y_i) está delimitado por el cuadrado cuya área abarca los (x,y) tal que $x\in[x_i\text{-}0.5, y_i\text{-}0.5)$ y $y\in[y_i\text{+}0.5, y_i\text{+}0.5)$.

Analizaremos el caso en que la pendiente de la recta $m\in[0,1]$. El resto de los casos pueden deducirse a partir de este, o sencillamente ser derivados como espejos de este caso.

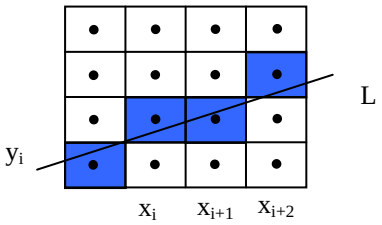

Figura 1.1: Píxeles asociados a una línea con pendiente entre 0 y 1

Dado que en nuestro caso de estudio $m\in[0,1]$, para cada x entero entre los límites del segmento va a existir un único valor discreto en y pero no viceversa (ver Fig. 1.1). Entonces el problema se reduce a realizar un ciclo variando x entre x_0 y x_1, y para cada valor de x buscar su imagen $y=mx+b$, realizando redondeo para acotar el error a 0.5 píxeles. Asumiendo que los vértices del segmento se ordenan de manera tal que $x_0<x_1$, podemos hacer un algoritmo inicial con aritmética real como sigue:

```
Putpixel(x0,y0);
m := (y1-y0)/(x1-x0);
b := y0-m*x0;
For x:=x0+1 To x1 DO
      PutPixel(x,Round(m*x+b));
EndFor;
```

El producto puede ser eliminado sabiendo que $y_{i+1}=m*x_{i+1}+b=m*(x_i+1)+b=y_i+m$, por lo tanto el algoritmo puede escribirse como:

```
Putpixel(x0,y0);
Y:=y0;
m = (y1-y0)/(x1-x0);
```

```
For x:=x0+1 To  x1 Do
      PutPixel(x,Round(y));
      y := y+m;
EndFor;
```

Ahora bien, la aritmética que está siendo utilizada es real. *Bresenham* en 1965 tomó la iniciativa de escribir un algoritmo de despliegue de líneas con aritmética entera. Este algoritmo es llamado algoritmo de punto medio, *o Midpoint Line Algorithm*. La idea es la siguiente; supongamos que en un paso dado del algoritmo se decidió desplegar (x_i,y_i). Las alternativas para el próximo píxel a pintar son E(x_i+1, y_i) y NE(x_i+1,y_i+1). Si la ecuación de la recta la escribimos en su forma implícita $f(x,y)=0$, la respuesta está en verificar el signo de la función f al evaluarla en el punto medio M entre ambos píxeles. Esto es justamente evaluar el signo de $f(M)=f(x_i+1, y_i+0.5)$. Como es bien sabido, si un punto (x,y) está arriba de la recta, entonces $f(x,y)>0$; si está debajo, $f(x,y)<0$, y si está justo sobre la recta, $f(x,y)=0$. Para nuestro caso, si $f(x_i+1, y_i+0.5)>0$, el pixel que vamos a tomar es E(x_i+1,y_i), y en caso contrario NE(x_i+1,y_i+1). En la Fig. 1.2, vemos que si evaluamos $f(x_i+1,y_i+0.5)$, el valor resultante es mayor que cero, ya que M está por arriba de la recta. Por consiguiente, el punto seleccionado es E (el punto hacia el este de (x_i,y_i)).

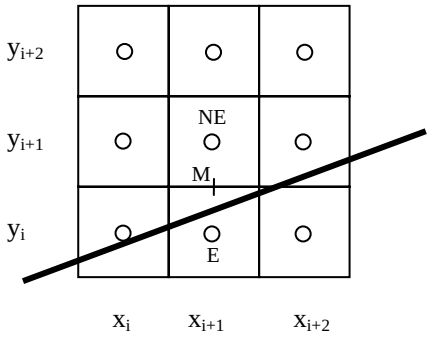

Figura 1.2: *Midpoint Line Algorithm*. La decisión a tomar en cada paso es prender el pixel NE ó E.

Para comenzar, expresemos la ecuación explícita de la recta como una ecuación implícita.

$$y\text{-}y_0=\Delta y/\Delta x*(x\text{-}x_0) \Rightarrow (y\text{-}y_0)\Delta x=\Delta y*x\text{-}\Delta y*x_0 \Rightarrow \Delta x*y\ \text{-}\Delta x*y_0\text{-}\Delta y*x+\Delta y*x_0 = 0.$$

Evaluando en (x_i,y_i), obtenemos

$$f(x_i,y_i) = \Delta x*y_i\ \text{-}\Delta x\ *y_0\ \text{-}\Delta y*x_i+\Delta y*x_0$$

Entonces, nuestra variable de decisión d, viene dada por

$$d=f(M)=f(x_i+1, y_i+0.5)= \Delta x*(y_i+0.5)\ \text{-}\Delta x\ *y_0\ \text{-}\Delta y*(x_i+1)+\Delta y*x_0$$

Caso 1: $d>0$
Tal y como lo muestra la Fig. 1.2, si $d=f(M)>0$ nos quedamos con el pixel "E", i.e. el pixel centrado en (x_i+1,y_i). Para el próximo paso, calcularemos $d'=f(x_i+2,y_i+0.5)$

$$d'=f(x_i+2,y_i+0.5)= \Delta x*(y_i+0.5)\ \text{-}\Delta x*y_0\text{-}\Delta y*(x_i+2)+\Delta y*x_0$$
$$=\Delta x*(y_i+0.5)\ \text{-}\Delta x*y_0\text{-}\Delta y*(x_i+1)+\Delta y*x_0\text{-}\Delta y =f(x_i+1,y_i+0.5)\text{-}\Delta y = d\text{-}\Delta y$$

Por lo tanto, el problema se reduce a realizar en cada iteración la operación con aritmética entera $d'=d\text{-}\Delta y$, y verificar el signo de d' para determinar qué píxel pintar.

Caso 2: $d<0$
Si $d=f(M)<0$ nos quedamos con el pixel "NE", i.e. el pixel centrado en (x_i+1,y_i+1). Para el próximo paso, calcularemos $d'=f(x_i+2,y_i+1.5)$

$$d'=f(x_i+2,y_i+1.5)= \Delta x*(y_i+1.5)\ \text{-}\Delta x*y_0\ \text{-}\Delta y*(x_i+2)+\Delta y*x_0$$

$=\Delta x*(y_i+0.5) - \Delta x*y_0 - \Delta y*(x_i+1)+\Delta y*x_0+\Delta x-\Delta y = f(x_i+1,y_i+0.5)+\Delta y+\Delta x = d+\Delta x-\Delta y$

Por lo tanto, el problema se reduce a realizar en cada iteración la operación con aritmética entera $d'=d+\Delta x-\Delta y$, y verificar el signo de d' para determinar que píxel pintar.

Caso 3: $d=0$
En este caso, cualquiera de las dos alternativas anteriores puede utilizarse.

Ya el algoritmo incremental está prácticamente resuelto. El problema ahora a resolver es inicializar adecuadamente la variable d. Antes de realizar el ciclo iterativo se sabe que el pixel (x_0,y_0) debe prenderse por ser un punto que pertenece a la recta, i.e. $f(x_0,y_0)=0$. Sin embargo, en la primera iteración, al calcular $f(M)=f(x_0+1,y_0+0.5)=f(x_0,y_0)+ 0.5\Delta x-\Delta y=0.5\Delta x-\Delta y$, aparece el valor de $0.5\Delta x$, y si Δx no es par, nos resulta un número real, arrastrando la aritmética real al algoritmo en casi todas sus operaciones. Debido a que nos interesa el signo de d y no su valor como tal, podríamos trabajar con $g(x,y)=2*f(x,y)$ en vez de $f(x,y)$. De esta manera, el primer valor de d vendría dado por

$d=2*(0.5\Delta x-\Delta y)= \Delta x-2*\Delta y$

Se puede demostrar fácilmente que al hacer el estudio de casos para la función $g(x,y)$, los incrementos quedan multiplicados por el factor 2. Entonces:

Caso 1: $d>0$
Se pinta el pixel E, y se actualiza $d'=d-2*\Delta y$
Caso 1: $d<0$
Se pinta el pixel NE, y se actualiza $d'=d+2*(\Delta x-\Delta y)$

El algoritmo incremental (solo operaciones + y – de números enteros) *Midpoint Line* es mostrado a continuación. Cabe destacar que los productos de la forma 2*x pueden ser implementados como una operación de corrimiento de registro en aritmética entera, por lo que no son contados como producto en sí (ejemplo, $2*x$ es equivalente en Lenguaje C a $x<<1$).

```
Procedure LineaPendienteEntre0y1(x0,y0,x1,y1: Integer; c: Color);
      Var dx,dy,x,y,d,IncE,IncNE: Integer;
      // Se asume x0<=x1, y0<=y1, (dy/dx)∈[0,1]
      dx     := x1-x0;
      dy     := y1-y0;
      d      := dx-2*dy;
      IncE   := 2*dy;
      IncNE  := 2*(dx-dy);
      x      := x0;
      y      := y0;
      PutPixel(x,y,c);
      While (x<x1) Do
            If (d<=0) Then
                  d := d+incrNE
                  y := y+1;
            Else
                  d := d+incE;
            EndIf;
            x := x+1;
            PutPixel(x,y,c);
      EndWhile;
EndProc;
```

Tarea:
Realizar el análisis para los casos en que la pendiente no esté en [0,1). Hay en total 3 casos restantes. Implementar el algoritmo definitivo. Nota: Para saber cual es el caso, no hace falta evaluar la pendiente, ni realizar operaciones aritméticas. Por ejemplo, para determinar que $0<=m<1$, puede evaluarse la condición $0<=dy/dx<1$ o **equivalentemente** $0<=dy<dx$, ya que $x_0<x_1$ por hipótesis y $dx=x_1-x_0$ es positivo.

2.- Despliegue de círculos

Asumamos, que el círculo tiene como centro el (0,0), y radio *r*. Entonces la ecuación viene dada por $x^2+y^2=r^2 \Rightarrow f(x,y)=x^2+y^2-r^2=0$. Por razones de eficiencia, se toma en cuenta que el círculo es simétrico en octantes, tal y como lo muestra la Fig. 2.1.

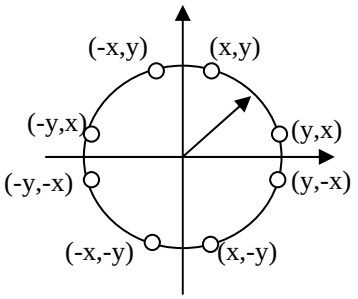

Figura 2.1: Ocho puntos simétricos sobre una circunsferencia

Para empezar, supongamos que existe una acción Dibuja8Puntos que dado un (*x*,*y*), dibuja los 8 puntos simétricos del círculo tal y como se muestra en la Fig. 2.1. Como puede notarse, sólo es necesario realizar el algoritmo para desplegar 1/8 del círculo. Si comenzamos en (0,*r*), terminamos en el primer (*x*,*y*) tal que *x*>*y*. En cada paso del algoritmo se debe tomar una decisión: tomar el punto E o el SE. Ver Fig. 2.2.

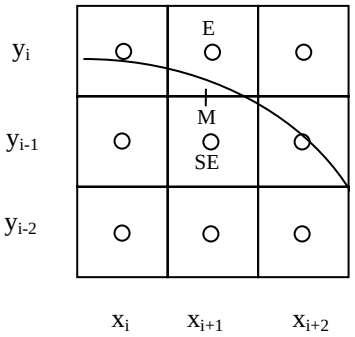

Figura 2.2: *M* está justo entre los píxeles E y SE, y el signo de *f*(*M*) nos indica cuál tomar

Asumamos que el punto (x_i,y_i) es el punto actual en el algoritmo iterativo, y se quiere saber cuál es el próximo punto a desplegar, es decir, $E(x_i+1,y_i)$ ó $SE(x_i+1,y_i-1)$. El signo resultante al evaluar la ecuación implícita en el punto del medio $M(x_i+1,\ y_i+0.5)$, nos indica cuál de los dos puntos está más cerca al círculo, y por ende, cuál pintar.

$$d = f(M) = f(x_i+1,y_i+0.5) = (x_i+1)^2+(y_i-0.5)^2-r^2$$

Caso 1: $d > 0$

Se toma el pixel $SE(x_i+1,\ y_i-1)$. Para el próximo paso $M'=(x_i+2,y_i-1.5)$, y

$d' = f(M') = f(x_i+2,y_i-1.5) = (x_i+2)^2+(y_i-1.5)^2-r^2 = x_i^2+4x_i+4+y_i^2-3y_i+9/4-r^2$

$= x_i^2+2x_i+1+y_i^2-y_i+1/4-r^2 +2x_i+3-2y_i+8/4 = f(x_i+1,y_i+0.5) +2(x_i-y_i)+5 = d+2(x_i-y_i)+5$

Caso 2: $d < 0$

Se toma el pixel $E(x_i+1,\ y_i)$. Para el próximo paso $M'=(x_i+2,y_i-0.5)$, y

$d' = f(M') = f(x_i+2,y_i-0.5) = (x_i+2)^2+(y_i-0.5)^2-r^2 = x_i^2+4x_i+4+y_i^2-y_i+1/4-r^2$

$= x_i^2+2x_i+1+y_i^2-y_i+1/4-r^2 +2x_i+3 = f(x_i+1,y_i+0.5) +2x_i+3 = d+2x_i+3$

Caso 3: $d = 0$.

Se toma cualquiera de los otros dos casos.

Veremos qué ocurre en el caso de borde: la inicialización de *d*. La primera vez, el punto (x_i,y_i) es (0,*r*), y además, $f(x_i,y_i)=f(0,r)=0$, ya que (0,*r*) está exactamente sobre la circunferencia. Entonces, el primer valor

de d viene dado por $f(1, r\text{-}0.5) = 1+(r\text{-}0.5)^2\text{-}r^2 = 1+r^2\text{-}r+¼\text{-}r^2 = 5/4–r=1\text{-}r+¼$. Note que esto obligaría a utilizar aritmética real en todo el algoritmo. Sin embargo, dado que los incrementos $2x_i+3$ y $2(x_i\text{-}y_i)+5$ son números enteros, el valor ¼ restante en la inicialización no afectaría el signo de las evaluaciones de $f(x,y)$ para los puntos de interés. Así, la inicialización más idónea sería $f(1, r\text{-}0.5) = 1\text{-}r$. A continuación se presenta el algoritmo de despliegue de círculos.

```
Procedure Circulo(r: Integer; c:color);
      Var x,y,d: Integer;
      x:=0;
      y := r;
      d = 1-r;
      Dibuja8Puntos(x,y,c);
      While (y>x) Do
            If (d<0) Then
                  d := d+2*x+3      // 2*x se implementa con shift
            Else
                  d := d+2*(x-y)+5; // Implementar el '*' como shift
                  y:=y-1;
            EndIf;
            x := x+1;
            Dibuja8Puntos(x,y,c);
      EndWhile;
EndProc;
```

Tarea: Qué modificaciones hay que hacer para desplegar círculos no centrados en el origen?. Demuestre que basta con modificar la acción `Dibuja8Puntos`.

3.- Despliegue de elipses

Se realizará el análisis para una elipse de eje mayor en x. El caso restante puede obtenerse por simple espejo de este. La ecuación implícita de la elipse centrada en (0,0) cuyo eje mayor en x es $2a$ y cuyo eje menor en y es $2b$ puede escribirse como:

$$f(x,y) = b^2x^2 + a^2y^2 – a^2b^2 = 0$$

La idea es realizar el algoritmo de despliegue para un cuadrante (x positivo, y positivo). Los otros puntos se obtienen por simetría. El algoritmo tiene dos modalidades:

1. Cuando la pendiente de la curva está en [–1, 0], el ciclo iterativo se realiza en x.
2. Cuando la pendiente está entre (–∞, –1], el ciclo se hace sobre y (Ver Fig. 3.1).

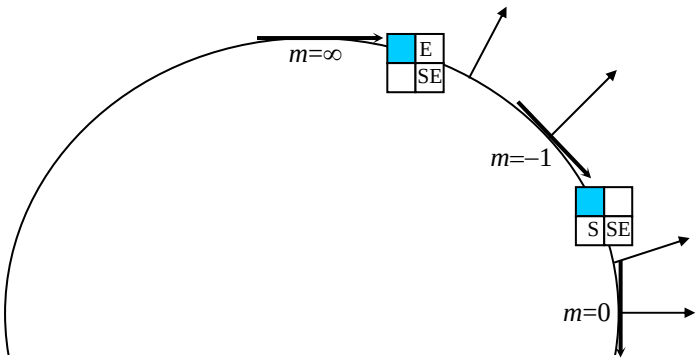

Figura 3.1: Las dos modalidades del algoritmo de despliegue de elipses. Observe como cambia la pendiente de la recta tangente así como el gradiente en cada punto.

Modalidad 1:
La decisión que hay que tomar es, desplegar el píxel E ó el SE. Esto se resuelve de la misma forma que para el círculo y la línea. Primero, evalúe el punto medio entre ambos puntos:

$$d = f(M) = f(x_i+1,y_i-0.5) = b^2(x_i+1)^2 + a^2(y_i-0.5)^2 - a^2b^2$$

Caso 1: $d>0$:
Tomamos SE(x_i+1,y_i-1)
$d' = f(x_i+2,y_i-1.5) = b^2(x_i+2)^2 + a^2(y_i-1.5)^2 - a^2b^2 = d + b^2(2x_i+3) + a^2(-2y_i+2) = d+\Delta_{SE}$

Caso 2, $d<0$
Tomamos E(x_i+1,y_i)
$d' = f(x_i+2,y_i-0.5) = b^2(x_i+2)^2 + a^2(y_i-0.5)^2 - a^2b^2 = d + b^2(2x_i+3) = d + \Delta_E$

Caso 3: $d=0$
Este caso puede ser absorbido por cualquiera de los casos anteriores

Ahora debemos verificar que sucede en el caso de borde. El primer punto que se despliega es el (0,*b*). Luego de esto, se calcula el primer valor de *d*, dado por:

$$d = f(1, b-0.5) = b^2 + a^2(b-0.5)^2 - a^2b^2 = b^2 + a^2(-b+1/4)$$

De nuevo aparece la indeseable fracción. Si a es par, entonces $a^2/4$ no tiene parte decimal, y puede utilizarse división entera. Pero si a no es par, la solución será utilizar la función $g(x,y)=4*f(x,y)$ en vez de $f(x,y)$, y por ende los incrementos Δ_{SE} y Δ_E se multiplican por 4. Así se utiliza aritmética entera en todo el algoritmo.

Ahora bien, hasta ahora sabemos que esta modalidad comienza para el punto (0,*b*), pero cuándo termina el ciclo?. Hemos hablado que esta modalidad llega hasta que la pendiente de la curva sea –1. Esto equivale a decir, que el vector gradiente tenga igual magnitud en *x* e *y*.

El vector gradiente en el punto *x*,*y* viene dado por $\Delta f(x,y) = \begin{pmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{pmatrix} = \begin{pmatrix} 2b^2x \\ 2a^2y \end{pmatrix}$

Algorítmicamente podemos escribir que mientras se cumpla que $2b^2x < 2a^2y$, estamos en la modalidad 1. Para reducir el error en la decisión de cambio de modalidad, trabajamos con el punto medio. Mientras $2b^2(x_i+1) < 2a^2(y_i-0.5)$ estamos en la modalidad 1. Para forzar la aritmética entera, rescribimos adecuadamente $2a^2(y_i-0.5)$ como $a^2(2y_i-1)$. Finalmente se muestra el algoritmo que dibuja la mitad de la elipse, y en simetría de 4 puntos.

```
Procedure MidPointEllipse(a,b: Integer; c: Color);
      Var x,y,d: Integer;
      // Modalidad 1
      x := 0;
      y := b;
      d := b*(4*b – 4*a*a) + a*a;
      // Dibuja 4 puntos simétricos de la elipse
      EllipsePoints(x,y,c);
      While (b*b*2*(x+1) < a*a*(2*y-1)) Do
            If (d < 0) Then
                  d  := d+4*(b*b*(2*x+3))
            else
                  d  := d+4*(b*b*(2*x+3)+a*a*(-2*y+2));
                  y := y-1;
            EndIf;
            x := x+1;
```

```
            EllipsePoints(x,y,c);
      EndWhile;
      //Modalidad 2 ...
EndProc;
```

Aunque no está explícito en el algoritmo, muchas optimizaciones pueden hacerse. Por ejemplo, realizar el producto b*b y a*a una sola vez y colocarlo en una variable para su posterior uso. Igualmente, se deben simplificar las fórmulas, y todos los productos de 2*, 4*, y 8*, realizarse con "left shit" con parámetro 1, 2, 3 respectivamente. Esto mejoraría la eficiencia del algoritmo. También debería factorizarse el algoritmo, y simplificar algunos cálculos, tomando en cuenta de que en cada iteración podría actualizarse d con tan sólo una suma. Por ejemplo, al utilizar las diferencias de segundo orden, podemos estudiar cómo varía d por cada incremento de x en una unidad en cada caso.

Modalidad 2:

La decisión que hay que tomar es, desplegar el pixel S o el SE?. Igual que siempre, evaluemos en el punto medio de ambos pixeles.

$d = f(M) = f(x_i+0.5, y_i-1) = b^2(x_i+0.5)^2 + a^2(y_i-1)^2 - a^2b^2$

Caso 1, $d>0$: Tomamos $S(x_i,y_i-1)$

$d' = f(x_i+0.5, y_i-2) = b^2(x_i+0.5)^2 + a^2(y_i-2)^2 - a^2b^2 = d + a^2(-2y_i+3) = d+\Delta_S$

Caso 2, $d<0$: Tomamos $SE(x_i+1,y_i-1)$

$d' = f(x_i+1.5, y_i-2) = b^2(x_i+1.5)^2 + a^2(y_i-2)^2 - a^2b^2 = d + b^2(2x_i+2) + a^2(-2y_i+3) = d + \Delta_{SE}$

Caso 3, $d=0$: este caso puede ser absorbido por cualquiera de los anteriores

Verifiquemos ahora el caso de borde: cuál es el primer valor de d, para tomar la primera decisión?. Tomemos en cuenta que el primer ciclo termina cuando $2b^2(x_i+1) >= 2a^2(y_i-0.5)$, y que el punto actual es (x_i,y_i). El primer valor de d en esta modalidad es tomar el punto medio entre los pixeles S y SE del pixel actual (x_i,y_i). Entonces $M=(x_i+0.5, y_i-1)$. Al evaluar $f(x,y)$ en este punto tenemos:

$$d = f(M) = f(x_i+0.5, y_i-1) = b^2(x_i+0.5)^2 + a^2(y_i-1)^2 - a^2b^2$$
$$= b^2(x_i^2+x_i+1/4)+ a^2(y_i^2-2y_i+1) - a^2b^2$$

De nuevo surge la indeseable fracción. Por ello multiplicamos todo por 4, i.e. trabajar con $g(x,y)=4f(x,y)$ y sacar de nuevo las cuentas. Así, tenemos:

Valor inicial de d:	$b^2(4x_i^2+4x_i+1)+ a^2(4y_i^2-8y_i+4) - 4a^2b^2$
Parada del ciclo:	Cuando ($y<=0$)
Δ_S:	$4a^2(-2y_i+3)$
Δ_{SE} :	$4(b^2(2x_i+2) + a^2(-2y_i+3))$

```
Procedure MidPointEllipse(a,b: Integer; c: Color);
      Var x,y,d: Integer;
      // Modalidad 1 ...
      // Modalidad 2:
      d := b*b*(4*x*x+4*x+1)+a*a*(4*y*y-8*y+4) -  4*a*a*b*b;
      While (y>0) Do
            If (d < 0) Then Begin
                  d  := d+4*(b*b (2*x+2)+a*a *(-2*y+3));
                  x := x+1;
            End Else
                  d  := d+4*a*a*(-2*y+3);
            EndIF
            y := y-1;
            EllipsePoints(x,y,c);
      EndWhile
EndProc;
```

De nuevo, este algoritmo no está optimizado. Hay operaciones que pueden factorizarse para mejorar la eficiencia del algoritmo. Por ejemplo, en la instrucción `d := d+4*a*a*(-2*y+3);` nos damos cuenta que en cada iteración "**y**" tiene el mismo valor anterior menos 1, así que la diferencia entre el valor de **d** anterior y el valor de **d** actual sería 8*a*a, que puede ser calculado una sola vez. De esa forma el ciclo iterativo puede manejarse de forma incremental. Además, las operaciones de productos por potencia de dos, deben realizarse con operaciones de corrimiento de registros, por tratarse de números enteros.

Tarea:
1. Realizar el algoritmo de la elipse minimizando el número de operaciones (sin productos dentro de los ciclos, y factorizando los cálculos en lo posible).
2. Muestre que para desplegar una elipse con centro (Cx, Cy) bastaría sólo con modificar la acción EllipsePoints.
3. Cómo haría para rellenar la elipse?.

4.- Despliegue de rectángulos

Un rectángulo con ejes paralelos a *x* e *y*, está definido por un par de puntos: (xmin, ymin), y (xmax, ymax). Esto corresponde a las coordenadas superior izquierda, e inferior derecha del rectángulo respectivamente. Así que para desplegar una rectángulo con estos extremos, bastaría con desplegar dos líneas verticales y dos horizontales.

```
For x:=xmin To xmax Do
      Putpixel(x,ymin,c);
      Putpixel(x,ymax,c);
EndFor;
For y:=ymin+1 To ymax-1 Do
      Putpixel(xmin,y,c);
      Putpixel(xmax,y,c);
EndFor;
```

Si se tiene acceso directo a la memoria de video (cambiar la rutina de putpixel por un acceso a memoria), seguramente podamos utilizar otro tipo de algoritmo que explote la localidad espacial de los datos. Las filas de píxeles están continuas en memoria, por lo que las líneas horizontales pueden ser dibujadas con rutinas eficientes en ensamblador, o su equivalente (i.e. memset o setmem en C, C++). En cualquier caso, el despliegue de una línea horizontal suele ser más rápido, puesto que los píxeles en la línea están continuos en memoria, explotando la localidad espacial de los datos.

5.- Despliegue de triángulos

Un triángulo está delimitado por tres puntos (x_0,y_0), (x_1,y_1) y (x_2,y_2). Para su despliegue bastaría con tres llamadas al algoritmo de línea:

```
Línea(x0,y0,x1,y1,c);
Línea(x1,y1,x2,y2,c);
Línea(x2,y2,x0,y0,c);
```

6.- Despliegue círculos rellenos

La forma más trivial de rellenar un círculo es que al ejecutarse `Dibuja8Puntos` se hagan las siguientes líneas horizontales:

```
Procedure Dibuja8Puntos(x,y: Integer; cOut, cIn: Color);
      HorizLine(-x+1, y, x-1, cIn);
      HorizLine(-y+1, x, y-1, cIn);
      HorizLine(-y+1, -x, y -1, cIn);
      HorizLine(-x+1, -y, x-1, cIn);
```

```
        Putpixel(x,y,cOut);
        Putpixel(y,x,cOut);
        // .... otros 6 putpixels ...
    EndProc;

    Procedure Horizline(xMin, yMin, xMax: Integer; c: Color);
        Var i: Integer;
        For i := xMin To xMax Do
            PutPixel(i, yMin, c);
        EndFor;
    EndProc;
```

Sin embargo, en varias llamadas consecutivas el valor de ***y*** es el mismo, y vamos a estar redibujando casi todos los píxeles de una línea dibujada con anterioridad. Esto es evidentemente redundante. Así que debe realizarse un control de este hecho, y sólo dibujar una línea horizontal por cada valor de *y*. Además, los píxeles de las primeras líneas del círculo y de las últimas líneas tienen errores, porque el círculo externo de color cOut no es continuo.

Tarea: Realice el algoritmo de relleno de círculos solventando los problemas planteados. Por eficiencia, evite hacer dos llamadas al algoritmo, ya que la implementación trivial es desplegar el círculo relleno y luego el borde del círculo.

7.- Despliegue elipses rellenas

Dado que se dibujan 4 píxeles por iteración, podemos dibujar entonces 2 líneas por cada iteración del algoritmo de despliegue.

```
    Procedure EllipsePoints(x,y: Integer; cOut, cIn: Color);
        HorizLine(-x+1,y,x-1, cIn);
        HorizLine(-x+1,-y,x-1, cIn);
        PutPixel(x,y,cOut);
        PutPixel(x,-y,cOut);
        PutPixel(-x,y,cOut);
        PutPixel(-x,-y,cOut);
    EndProc;
```

Igual que para rellenar círculos, esta manera es ineficiente ya que generalmente hay varias llamadas a EllipsePoints con el mismo valor de *y*. Igualmente, las primeras y últimas líneas de la elipse no tienen continuidad en el borde.

Tarea: realice el algoritmo de relleno de elipse solventando los problemas planteados. Utilice los tips vistos en el relleno de círculos.

8.- Despliegue de rectángulos rellenos

En este caso, basta con trazar una línea horizontal por cada valor de *y* en el intervalo (ymin,ymax). A continuación se presenta el algoritmo:

```
  Procedure Rectangulo(xmin,ymin,xmax,ymax: Integer; cOu, cIn: Color);
        Var i: Integer;
        HorizLine(xmin, ymin, xmax, cOut);
        HorizLine(xmin, ymax, xmax, cOut);
        For i:=ymin+1 To ymax-1 Do
            PutPixel(xmin, i, cOut);
            HorizLine(xmin+1, i, xmax-1, cIn);
            PutPixel(xmax, i, cOut);
        EndFor;
   EndProc;
```

9.- Despliegue de triángulos rellenos

Como se dijo anteriormente, un triángulo está delimitado por tres puntos, (x_0,y_0), (x_1,y_1), (x_2,y_2). La idea es realizar el relleno por líneas horizontales, al igual que todas las primitivas, para explotar la localidad espacial en el acceso a los datos. Para ello tomemos dos casos generales:

Asumamos que los puntos están ordenados tal que $x_0 \leq x_1 \leq x_2$. Entonces, desplegar el triángulo $\Delta(P_0,P_1,P_2)$ equivale a desplegar los triángulos $\Delta(P_0,P_1, P_3)$ y $\Delta(P_1,P_3,P_2)$, en donde el punto P_3 es la intersección de la recta $y=y_1$ con el segmento de recta P_0P_2.

Para desplegar el triángulo $\Delta(P_0, P_1, P_3)$, se despliegan segmentos horizontales partiendo del segmento P_1P_3, hasta llegar al punto P_0. El primer segmento se despliega desde el punto P_1 hasta P_3. Para el siguiente, que llamaremos $P_1^1P_3^1$, se intersecta la línea $y=y_1+1$ con los segmentos P_0P_1 y P_0P_3, se redondean las intersecciones, y se procede a dibujar el segmento. Sin embargo, esta claro que $P_1^1.x=P_1.x+(x_1-x_0)/(y_1-y_0)$. Similarmente, $P_3^1.x=P_3.x-(x_3-x_0)/(y_3-y_0)$. Así que los puntos sucesivos pueden obtenerse de forma incremental, sumando la constante $1/m$, en donde m es la pendiente de la recta respectiva.

El otro enfoque es obtener los puntos $P_1^iP_3^i$ "ya redondeados" utilizando el algoritmo de Bresenham para despliegue de líneas, tomando en cuenta que no se actualizan los puntos hasta tanto no haya variado el valor de y en el algoritmo (esto es porque según la pendiente de la recta, Bresenham puede generar varios puntos consecutivos con el mismo valor de y). Lo importante de este enfoque es que se puede utilizar aritmética entera en todas las operaciones.

Si se desea que el relleno sea de distinto color que el borde, entonces deben desplegarse los pixeles extremos del segmento de color de borde, y los internos de color de relleno.

Tarea: realizar el algoritmo incremental para el despliegue de parábolas e hipérbola

10. BIBLIOGRAFÍA

- Foley James. Vam Dam Andries. Feiner Steven. Hugkea John. *"Computer Graphics. Principles and Practice"*. Addison-Wesley. 2da. Edición. 1990.