Canvas.

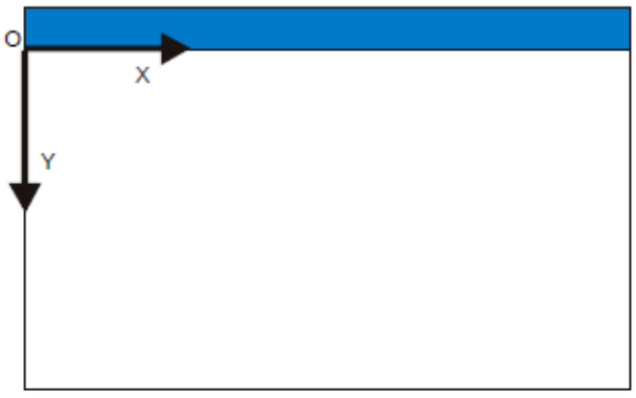
Dibujar en Windows

Para graficar en Delphi trabajamos con un objeto llamado *Canvas* (se podría traducir como “Lienzo”). Representa la superficie de graficación, con propiedades tales como la pluma o el pincel que vamos a usar. Además, tiene métodos que nos permiten dibujar primitivas -rectángulos, círculos, elipses, texto, etcdirectamente sobre su superficie.

Coordenadas

Es importante entender cómo se toma el sistema de ejes coordenados en la pantalla. Para cualquier componente, la coordenada X corre desde el borde izquierdo hacia el derecho, igual que en matemática; en cambio, la coordenada Y se toma desde arriba hacia abajo, al contrario de la norma seguida usualmente por los matemáticos. Esto significa que el centro de coordenadas está en la esquina superior izquierda del componente.

Para definir un punto se necesitan dos coordenadas, *x* e *y*. Se pueden pedir las dos coordenadas por separado, como dos parámetros de tipo entero, o bien utilizar una estructura creada al efecto: el registro Tpoint.



Está definido en la unit *Windows*, de la siguiente manera:

```
type TPoint = record X: Longint;
Y: Longint; end;
```

Como vemos, nada más que una forma cómoda de tener las dos coordenadas “en un solo paquete”. Existe incluso una función que toma dos números y los devuelve encapsulados en un Tpoint:

```
function Point(AX, AY: Integer): TPoint;
```

Lo mismo sucede con los rectángulos: podemos definirlos con cuatro coordenadas, las de la esquina superior izquierda y las de la esquina inferior derecha. Existe también una estructura, llamada en este caso Trect.

```
TRect = record case Integer of
        0: (Left, Top, Right, Bottom: Integer);
        1: (TopLeft, BottomRight: TPoint);
end;
```

Se trata de un registro variante. Según como se lo pidamos, nos entregará su contenido en cuatro números enteros o en dos registros Tpoint. Algo así como la dualidad onda-partícula de la mecánica cuántica, o los discursos de los políticos.

Supongamos que tenemos las coordenadas de un rectángulo en una estructura como la anterior, llamada R; en tal caso, las siguientes sentencias son equivalentes:

```
Canvas.Rectangle(R.Left, R.Top, R.Right, R.Bottom);

Canvas.Rectangle(R.TopLeft.X, R.TopLeft.Y, R.BottomRight.X, R.BottomRight.Y);
```

También en este caso existe una función que toma cuatro números y los devuelve empaquetados en un registro Trect:

```
function Rect(ALeft, ATop, ARight, ABottom: Integer): TRect;
```

En dónde dibujamos

Cada componente que muestra una cara en la pantalla tiene su propio Canvas, que tendremos que utilizar para dibujar en ese componente. ¿Cuándo? Cuando lo deseemos, puede ser al presionar un botón o al mover el ratón o en cualquier otro momento.

Hay sin embargo un evento especial para dibujar la ventana; este evento se produce cada vez que Windows determina que hay que actualizar el contenido de la ventana, por ejemplo si se mueve otra ventana que tapaba una parte de la nuestra. Este evento existe solamente en la clase tForm, y se denomina OnPaint. Todos los ejemplos en que no se indique lo contrario se deberían escribir en este evento.

Para que nuestros dibujos nos queden como queremos, hay tres propiedades esenciales: TFont, TPen y TBrush. La primera es la conocida propiedad para el texto,

en la que establecemos su tamaño, tipo, color, etc. La propiedad Color es del tipo TColor y admite colores predefinidos (por ejemplo : clBlue, clRed) o un Valor en hexadecimal que represente el color. Este tendrá la forma $00bbggrr. El símbolo del dólar ($) indica que se trata de un hexadecimal. Los dos siguientes ceros están ahí en todos los colores, sin que cumplan ninguna función (aparte de hacer que el número tenga ocho cifras en total). Después se sitúan los valores del azul (bb), verde (gg) y rojo (rr) que compongan el color. Estos valores serán en hexadecimal, por supuesto, y tomarán un valor entre cero (00) y 255 (FF). Sin embargo es más fácil usar la función RGB.

La siguiente propiedad de la fuente es Height, que indica el ancho de las letras. Esto está relacionado con Size, el tamaño total. La fórmula que las relaciona es:
Font.Size = -Font.Height * 72 / Font.PixelsPerInch

O lo que es lo mismo:
Font.Height = -Font.Size * Font.PixelsPerInch / 72

La propiedad Name indica la fuente que se está usando, siendo la normal MS Sans Serif, pero puede tomar el nombre de cualquiera que haya instalado en Windows. Para obtener las fuentes disponibles, debe usar la propiedad Screen.Fonts, donde están listadas. Si las quiere incluir todas en una lista hay un procedimiento bastante rápido:

```
var
ListFonts : TStrings;
begin
ListFonts := Screen.Fonts;
ListaFuentes.Items := ListFonts;

end;
```

La última propiedad de Font es Style, donde indica cómo se dibujará. Puede tomar una combinación de: fsBold: Negrita, fsItalic: Cursiva, fsUnderline: Subrayada y fsStrikeOut: Tachada.

Para añadir o quitar algunos de ellos se hace de la siguiente forma:
TCanvas.Font.Style := TCanvas.Font.Style + [fsBold]; //(Por ejemplo)

TCanvas.Font.Style := TCanvas.Font.Style - [fsItalic]; //(Por ejemplo)

Para empezar, la forma más simple de dibujar algo es haciéndolo de píxel en píxel (aunque no es la más fácil ni la más cómoda), pero también se pueden leer. Esto se hace con la propiedad TCanvas.Pixels, que, en realidad, es una matriz de dos

dimensiones, por lo que hay que pasarle dos valores si queremos acceder a alguno de sus elementos. En relación con la pantalla, el primer valor corresponde a la coordenada X, y el segundo, a la Y. Esta matriz es del tipo TColor, por lo que solamente sirve para leer o establecer el color de un punto.

```
TCanvas.Pixels [10,2] := clRed;

TColor := TCanvas.Pixels [11,5];
```

Primitivas

Se llama Primitivas a las formas básicas de todo dibujo: rectángulos, círculos, líneas, puntos. Dependiendo del sistema de graficación tendremos más o menos primitivas - por ejemplo, bajo un entorno de graficación 3D podríamos tener una primitiva cubo. Windows nos da acceso a sus primitivas a través de funciones de la API. No obstante, Delphi encapsula estas funciones dentro de la clase TCanvas, y en especial las funciones de la API son métodos de la clase TCanvas. Todos los componentes que permiten que les dibujemos encima tienen una instancia de esta clase, como una propiedad llamada *Canvas* (después dicen que los programadores no tenemos imaginación para elegir nombres).

En muchas de las funciones para dibujar primitivas gráficas necesitaremos expresar un rectángulo; normalmente lo haremos con las coordenadas de los puntos superior izquierdo e inferior derecho. Empecemos por las primitivas más sencillas: por ejemplo, un punto.

No hay una primitiva que grafique un punto.

¿¿Cómo?? El gráfico más simple posible, y no hay forma de hacerlo? Claro que hay formas de dibujar un punto, sólo que no hay un procedimiento exclusivo para eso.

Podemos dibujar un punto de por lo menos dos maneras diferentes:

- accediendo al pixel individual para cambiarle el color directamente (propiedad *Pixels)*
- haciendo una *recta* de un punto de largo.

Lo que sí nos permite la API de Windows es el mover la pluma *sin graficar,* algo así como decirle a la máquina “apoya el lápiz en las coordenadas *(x,y)* para empezar a dibujar”. *No se grafica nada con esta orden,* solamente se posiciona la punta del lápiz. La declaración de este método es por supuesto muy simple:

- **procedure TCanvas.MoveTo(x,y: integer);**

¿Y cómo sabemos si la pluma se movió realmente? Pues resulta que en cualquier momento podemos saber en qué lugar está la “punta del lápiz” leyendo la propiedad PenPos, que es de tipo Tpoint.

Como un ejemplo, cambiaremos la pluma a la posición (100,45) y comprobaremos inmediatamente que se realizó el movimiento:

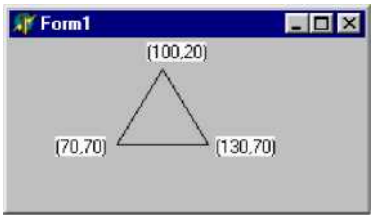
```
Canvas.MoveTo(100,45);
if (Canvas.PenPos.x=100) and (Canvas.PenPos.y=45) then ShowMessage('Todo OK, colega!') else
      ShowMessage('La pluma no se encuentra en la posición indicada');
```

La capacidad de mover la pluma es importante porque no tenemos disponible una función para dibujar una línea entre dos puntos; en su lugar, existe la función LineTo que toma las coordenadas del punto *final* de la línea... tomando como inicial el punto donde esté la pluma en el momento de dibujar.

- **procedure LineTo(x,y: integer);**

Esto nos permite hacer una línea continua partiendo de un punto, simplemente llamando a LineTo por cada tramo.

Ejemplo 1: dibujar un triángulo como el mostrado en la figura:




```
Canvas.MoveTo(100,10);
Canvas.LineTo(130,70);
Canvas.LineTo(70,70);
Canvas.LineTo(100,10);
```

Ejemplo 2: crear una función que permita dibujar una línea especificando los dos puntos extremos.

Podemos definir la función Linea como sigue:

```
function Linea(ElCanvas: Tcanvas; P0,P1: Tpoint);
begin
      ElCanvas.MoveTo(P0.x,P0.y);
      ElCanvas.LineTo(P1.x,P1.y);
end;
```

Notemos la necesidad de incluir el canvas sobre el cual se desea dibujar como un parámetro; si esta función estuviera definida en la clase Tcanvas esto no sería necesario. Les propongo como un ejercicio interesante la creación de una clase descendiente de Tcanvas que implemente la función anterior.

Ejemplo 3: dibujar el mismo triángulo anterior, ahora utilizando la nueva función *Linea.*

```
Linea(Canvas,Point(100,20),Point(130,70));
Linea(Canvas,Point(130,70),Point(70,70));
Linea(Canvas,Point(70,70),Point(100,20));
```

De la misma manera que en el ejemplo, podemos dibujar un rectángulo o en general un polígono de N lados. Pero hay formas más fáciles de hacerlo. Tenemos la función Rectangle, que dibuja un rectángulo, y la función Polygon que dibuja un polígono.

- **procedure Rectangle(x1,y1,x2,y2: integer);**

Dibuja un rectángulo tomando (x1,y1) como la esquina superior izquierda y (x2,y2) como la esquina inferior derecha.

- **procedure Polygon(points: array of tPoint);**

Dibuja una serie de líneas conectando los puntos del array, *uniendo el último punto con el primero.* Para indicar los puntos en la línea de comandos es muy útil la función Point de Delphi.

Ejemplo 4:

Para dibujar un cuadrado con vértices en los puntos (10,10); (100,10); (100,100); (10,100) podemos hacer lo siguiente:

```
Canvas.Polygon([Point(10,10),point(100,10),point(100,100),point(10,100)]);
```

o bien, utilizando Rectangle:

```
Canvas.Rectangle(10,10,100,100);
```

o bien, usando líneas:

```
with Canvas do begin
      MoveTo(10,10);
      LineTo(100,10);
      LineTo(100,100);
      LineTo(10,100);
      LineTo(10,10);
end;
```

Pero ¡CUIDADO! Hay una diferencia fundamental entre hacer una figura utilizando las funciones Rectangle o Polygon y hacerlas simplemente dibujando líneas: las funciones mencionadas generan figuras *cerradas,* por lo que automáticamente se pintan con el pincel activo (por defecto, color blanco sólido). Usando líneas, por más que éstas se intersecten y formen una figura cerrada, no se pintará esa superficie porque en lo que a Windows concierne se trata de una serie de líneas y nada más.

En lo que sigue indicaremos entonces para cada método de Canvas si se genera una figura abierta o cerrada. Recordemos que en el último caso se pinta automáticamente la superficie interior, por lo que si no tomamos recaudos se tapará lo que había debajo.

Hay funciones que producen variantes sobre las anteriores:

- **procedure FrameRect(const rect:tRect);**

Dibuja un rectángulo sin relleno. Note el uso de una estructura Trect, a diferencia de Rectangle. Genera una figura cerrada pero con “relleno transparente” (a todos los usos prácticos, *sin* relleno). *Se utiliza el color de pincel (Brush) como color del cuadro.*

- **procedure RoundRect(x1,y1,x2,y2,ancho,alto: integer);**

Dibuja un rectángulo entre los puntos (x1,y1) y (x2,y2), redondeando las puntas con cuartos de elipse de anchura *Ancho* y altura *Alto.* Genera una figura cerrada.

- **procedure Polyline(points: array of tPoint);**

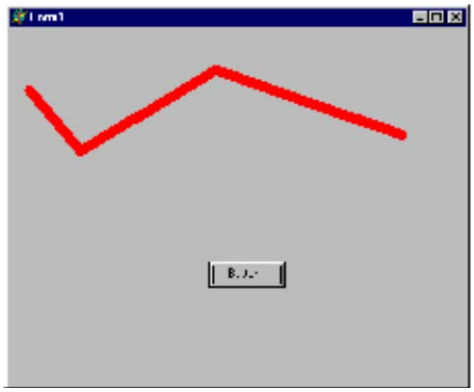
Dibuja una serie de líneas conectando los puntos del array. Genera una figura abierta.

Notemos que no hay una función especial para graficar un cuadrado, ya que éste es un caso especial de rectángulo.

Otra serie de funciones se ocupan de las figuras *curvas:* elipses, arcos, etc.

```
var
A: array [0..3] of TPoint;
B : TPoint;
begin
Canvas.Pen.Width := 10;
Canvas.Pen.Color := clRed;
B.X := 20;
B.Y := 60;
A [0] := B;
B.X := 70;
B.Y := 120;
A [1] := B;
B.X := 200;
B.Y := 40;
A [2] := B;
B.X := 380;
B.Y := 104;
A [3] := B;
Canvas.Polyline (A);

End;
```

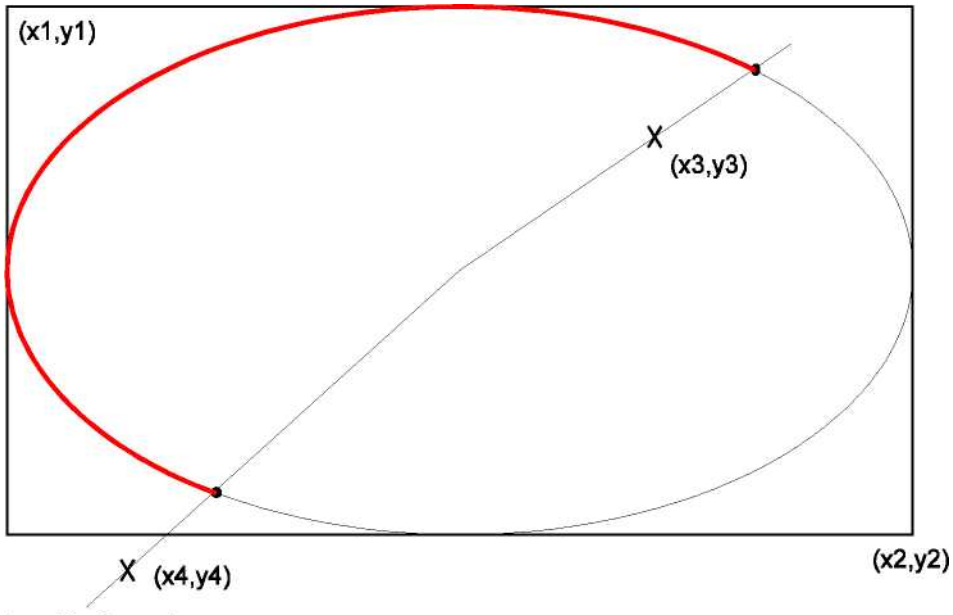


- **Procedure ellipse(x1,y1,x2,y2: integer);**

Dibuja una elipse inscripta en el rectángulo definido por los dos puntos (figura cerrada). También se utiliza para dibujar círculos.

- **procedure arc(x1,y1,x2,y2,x3,y3,x4,y4: integer);**

Dibuja un arco de elipse. La elipse se toma encerrada por el rectángulo (especificado por los dos primeros puntos, (x1;y1) y (x2;y2)); el comienzo y final del arco son determinados por la intersección de una línea desde el centro de la elipse, que pase por los puntos 3 (x3;y3) y 4 (x4;y4) respectivamente.

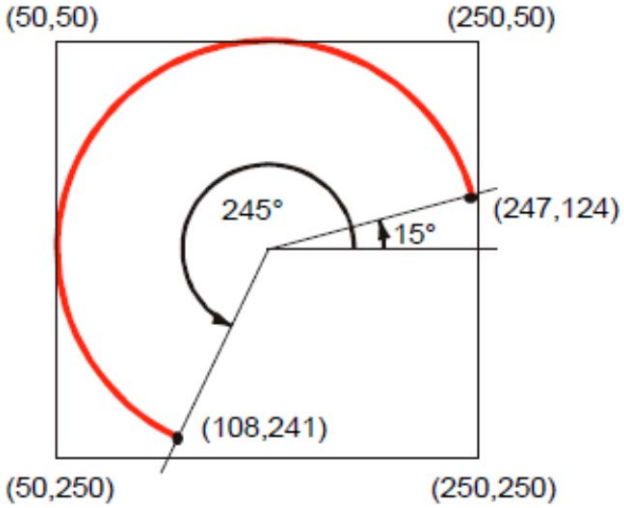


Especificaciones de un arco

Ejemplo 6:

Graficar un arco de circunferencia de radio 100 pixels, que comience a los 15 grados y termine a los 245 grados.

Usando un poquito de trigonometría básica (esa que aprendimos a los 10 años, más o menos) podemos llegar a una situación como la siguiente:



Usando la información del gráfico anterior, llegamos a la orden necesaria para dibujar la elipse (aproximadamente, recordemos que tenemos que redondear a nros. enteros):

Canvas.Arc(50,50,250,250,247,124,108,241);

Ejemplo 7:

Grafiquemos un arco de una elipse de eje mayor igual a 100 pixels y eje menor igual a 70 pixels, que vaya desde los 90 grados hasta los 180 (un cuarto de elipse). La situación es más fácil que la anterior, ya que ni siquiera es necesario calcular las funciones trigonométricas de esos ángulos:

Canvas.Arc(0,0,100,70,50,0,0,35);

Hay otra instrucción muy parecida, pero esta vez se cierra el arco con una linea entre los extremos y se rellena con el pincel activo (propiedad Brush):

- **procedure chord(x1,y1,x2,y2,x3,y3,x4,y4: integer);**

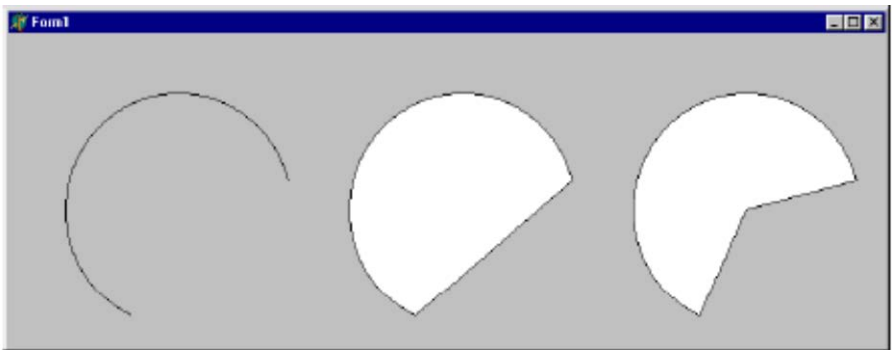
Haciendo los mismos ejemplos que antes cambiando Arc por Chord podemos notar la diferencia entre los dos procedimientos. Y todavía hay una variante más:

- **procedure Pie(x1,y1,x2,y2,x3,y3,x4,y4: Longint);**

Dibuja una porción de una tarta, definida de la misma manera que para Chord o Arc.

La diferencia entre estas dos últimas funciones está en la manera en que cierran la figura, como podemos comprobar dibujando con las mismas

coordenadas. En la figura siguiente se muestra el resultado del ejemplo ?, realizado con las tres funciones anteriores.



Otro grupo de funciones se utilizan para pintar:

- **Procedure fillRect(const rect:tRect);**

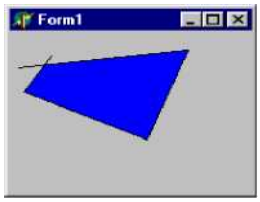
Rellena el rectángulo especificado con el pincel actual (sin borde).

- **procedure FloodFill(x,y:integer; color:tColor; FillStyle:tFillStyle);**

Rellena la superficie que contiene al pixel (x,y).

si FillStyle es fsBorder, se rellena hasta que se encuentra el color especificado.

Si FillStyke es fsSurface, se rellena la superficie mientras el color sea igual al especificado en el parámetro *Color;* se detiene cuando encuentra un color distinto.



Ejemplo 8:Queremos rellenar con color azul la superficie limitada por un polígono; pero resulta que no usamos Polygon para construir éste, sino que lo hicimos con líneas sueltas. No nos queda más remedio que usar FloodFill:

```
with Canvas do begin
      Pen.Color:= clBlack; //Color de las líneas de borde MoveTo(30,20);
      LineTo(10,50);
      LineTo(100,90);
      LineTo(130,15);
      LineTo(5,30);
      Brush.Color:= clBlue; //Con este color se pinta
FloodFill(20,40,clBlack,fsBorder); end;
```

Aquí he utilizado un poco los objetos de pluma (Pen) y pincel (Brush) para asegurarme que el resultado será el esperado; enseguida estudiaremos estos objetos.

Métodos para trabajar con texto • procedure TextOut(x,y: integer; const text: string);

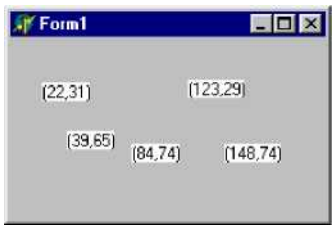
Escribe el string comenzando en la posición (x,y) con el font actual.

Ejemplo 9:

Hagamos un programa que cuando se presione el botón izquierdo del ratón en la superficie de la ventana nos escriba *ahí mismo* las coordenadas en que se presionó el botón.

El primer evento que se nos viene a la mente es el OnClick; bueno, pues destiérrenlo de sus mentes inmediatamente porque este evento no nos brinda la información de la posición del cursor en el momento de producirse. En su lugar, debemos usar OnMouseDown o OnMouseUp. He aquí el código, y una muestra del resultado:

```
procedure TForm1.FormMouseUp(Sender: TObject;
Button: TMouseButton; Shift: TShiftState;
    X, Y: Integer); begin
canvas.TextOut(x,y,Format('(%d,%d)',[x,y]));
end;
```



• **procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);**

Esta función escribe un texto limitándolo a un determinado rectángulo. Si el texto se sale de los límites, no aparece en pantalla.

Ejemplo 10:

Modificaremos el ejemplo anterior para que solamente se escriban las coordenadas si estamos dentro de un rectángulo de esquinas (50,50) y (200,150):

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer); begin
    canvas.TextRect(Rect(50,50,200,150),x,y,Format('(%d,%d)',[x,y]));
end;
```

Ahora se dibuja el rectángulo primero (se borra todo lo que está debajo) y si el texto se sale de los límites del rectángulo, se recorta.

- **function TextHeight(const Text: string): Integer;**

Esta función devuelve la altura en pixels del texto enviado como parámetro.

- **function TextWidth(const Text: string): Integer;**

Esta función devuelve el ancho en pixels de una determinada cadena.

• function TextExtent(const Text: string): TSize;

Devuelve los dos datos anteriores juntos en una estructura Tsize, definida como type

```
TSize = record cx: Longint; cy: Longint; end;
```

Ejemplo 11:

Vamos a escribir otra variación al ejemplo anterior de las coordenadas; ahora queremos que en lugar de escribir las coordenadas escriba “Hola, mundo!” recuadrado como se ve en la imagen. El código podría ser algo así:

```
procedure TForm1.FormMouseUp(Sender: TObject; Button: TMouseButton
Shift: TShiftState; X, Y: Integer); var
    s: string; r: tSize; begin
    s:= 'Hola, mundo!'; canvas.TextOut(x,y,s); r:= canvas.TextExtent(s);
    Canvas.FrameRect(Rect(x-3,y-3,x+r.cx+3,y+r.cy+3));
end;
```

Note que en el código anterior se deja un margen de 2 pixels vacíos entre el texto y el recuadro (el recuadro mismo ocupa un pixel de ancho, por lo que separamos uno de otro en 3 pixels).

Los colores

Los colores en Delphi se representan con números enteros de cuatro bytes, donde cada uno tiene un significado preciso. Para representar colores se ha definido el tipo Tcolor, que no es más que un subrango de los enteros.

Hay una serie de constantes predefinidas en la unidad Graphics, que representan colores comunes de la paleta de sistema o colores de elementos comunes definidos por el usuario en el Panel de Control (por ejemplo, clBtnFace es el color que elegimos en el Panel de Control para el frente de los botones -usualmente gris). Estas constantes empiezan con las letras cl, y en la ayuda se puede ver un listado con sus equivalencias buscando Tcolor en el índice.

Pero no estamos restringidos a las constantes predefinidas; podemos crear colores intermedios trabajando con el modelo RGB (Rojo, Verde, Azul).

Veamos el significado de cada byte de una variable TColor:

Bytes	Significado
3 (más peso)	Paleta. 0= paleta del sistema 1= paleta actual 2= paleta lógica del dispositivo
2	Color Azul
1	Color Verde
0 (menos peso)	Color Rojo

así, el valor hexadecimal $00FF0000 representa azul puro, $0000FF00 verde puro y $000000FF rojo puro. $00000000 es negro y $00FFFFFF es blanco. Los valores intermedios se “mapean” (se busca la mejor coincidencia) en la paleta de colores indicada por el byte más alto.

Sin entrar en detalles técnicos, los dispositivos gráficos pueden mostrar generalmente una cantidad limitada de colores distintos a la vez; por ejemplo, un adaptador de video puede ser capaz de representar cualquier color pero solamente 256 colores diferentes a la vez. Entonces se define una *paleta,* una lista de 256 colores que serán los que se verán simultáneamente. Podemos cambiar las entradas de esta paleta, pero no la cantidad. Cuando estamos ante este caso (el más común) se necesita el *mapeo* de colores: al presentar un color al sistema, éste lo busca en la paleta correspondiente. Si no lo encuentra, determina por algún método el color más cercano y selecciona éste.

Por suerte, en general no necesitaremos preocuparnos por este trabajo ya que el sistema lo realiza en forma automática y bastante bien. Delphi además modifica la paleta cuando se muestra una imagen por ejemplo en un componente TImage, para que los colores de ésta se vean sin distorsión. Es posible que los colores del resto de la pantalla cambien (a veces drásticamente), pero algo hay que resignar...

En todos los ejemplos que siguen utilizaremos las constantes predefinidas.

Cambiar el pincel, la pluma o la letra

La clase tCanvas tiene cuatro propiedades fundamentales que representan otros tantos objetos de dibujo:

- Pen: objeto que modela la pluma utilizada para graficar líneas, ya sean bordes de figuras o líneas sueltas.
- Brush: objeto que modela el pincel utilizado para pintar figuras.

- Font: objeto que representa la fuente con la que se escribe.
- Pixels: array bidimensional de colores. Cada entrada de la matriz representa el color de un pixel del canvas.

Los tres primeros son objetos en si mismos, con propiedades que hacen muy fácil cambiar las características del canvas. El último es un arreglo de elementos de tipo TColor.

Propiedades del objeto Pen (clase tPen)

- *Color:* indica el color de la pluma. El color se aplica a todas las líneas que se dibujen a continuación, hasta que se vuelva a cambiar o se destruya el Canvas.

Por ejemplo, para dibujar en rojo haríamos (antes de dibujar):

```
Canvas.Pen.Color:= clRed;
```

- *Mode:* indica el modo de combinar los colores de la pluma y de la pantalla.

Para dibujar algo, debemos cambiar el color de algunos puntos de la pantalla; pero estos puntos ya tienen un color. Podemos decidir ser absolutistas y reemplazar directamente el color anterior con el nuevo, o ser un poco más democráticos y permitir que se mezclen de alguna manera.

La propiedad Mode indica la forma de mezclar los colores.

Los valores posibles son:

Modo	Color resultado
pmBlack	Siempre negro, no importa el color que demos a la línea
pmWhite	Siempre blanco
pmNop	Sin cambio; en la práctica, es como si la línea fuera transparente
pmNot	Inverso del color de fondo (resultado = NOT fondo)
pmCopy	Color de la pluma (valor por defecto)
pmNotCopy	Inverso del color de la pluma (resultado = NOT pluma)
pmMergePenNot	resultado = pluma AND (NOT fondo)
pmMaskPenNot	Combinación de colores comunes a la pluma y el inverso del fondo
pmMergeNotPen	resultado = fondo AND (NOT pluma)
pmMaskNotPen	Combinación de colores comunes al fondo y el inverso de la pluma
pmMerge	resultado = fondo AND pluma
pmNotMerge	Combinación inversa a pmMerge (NOT (fondo AND pluma))
pmMask	Combinación de colores comunes al fondo y a la pluma
pmNotMask	Combinación inversa a pmMask
pmXor	resultado = fondo XOR pluma
pmNotXor	Inversa de pmXor

Width: ancho en pixels de la pluma.

Style: estilo de la pluma (continua, punteada, a trazos, etc) *cuando se trabaja con pluma de 1 pixel de ancho.* Si la pluma es más ancha, siempre aparece continua!

Las constantes permitidas están listadas en la ayuda.

Propiedades del objeto Brush (clase tBrush)

Color: indica el color del pincel.

Style: estilo del pincel (sólido, con rayas verticales, horizontales, oblicuas, etc). Las constantes están listadas en la ayuda, con una muestra de las mismas aplicadas a un rectángulo.

Bitmap: un BMP de 8x8 pixels que se usará como patrón para el relleno.

Propiedades del objeto Font (clase tFont)

Color: color de la fuente a usar.

Name: nombre de la fuente (Arial, Times New Roman, etc)

Size: tamaño en pixels de la fuente.

Style: estilo de la fuente: es un conjunto que puede tener alguno de los valores siguientes (pueden ser varios):

- fsBold: negrita
- fsItalic: itálica
- fsUnderline: subrayada
- fsStrikeOut: tachada.

Por ejemplo, dibujaré un rectángulo de bordes azules, y rallado en color amarillo.

```
PaintBox1.Canvas.Pen.Color := clBlue;
PaintBox1.Canvas.Brush.Style := bsHorizontal;
PaintBox1.Canvas.Brush.Color := clYellow;
PaintBox1.Canvas.Rectangle (0,0,300,220);
```