Creación de componentes VCL (y IV)

Componentes gráficos

Y ya para terminar, nos quedan aprender cómo crear nuestros propios componentes gráficos, dibujando directamente en la pantalla a través del objeto Canvas.

Índice

Introducción

En los números anteriores hemos cubierto los aspectos más utilizados de los componentes: no-visuales, para la realización de tareas de cálculo o que no tienen representación en pantalla, y de los visuales, para que el usuario final interactúe con el componente.

Sin embargo, existe otro tipo de componentes, en los que somos los responsables del dibujo del componente en la pantalla.

Componentes que no son ventanas

Cuando explicamos los componentes visuales, dijimos que existían básicamente dos tipos: los que permitían la interactuación del usuario y los que no. Los primeros, que ya hemos explicado, permiten recibir eventos de teclado por una razón muy importante: son ventanas, aunque no lo parezca a simple vista.

Para el sistema operativo Windows, el concepto de "ventana" es mucho más amplio que para un usuario normal. Hasta ahora, para los simples mortales como nosotros, una ventana era ese rectángulo con una barra azul arriba, que aparece en la pantalla y muestra dentro los elemento de la aplicación con que estamos trabajando: cuadros de texto, campos de edición, árboles jerárquicos, botones, etc. Sin embargo, para el sistema operativo hay dos tipos de ventanas: las que parece (y se comportan) como una ventana de verdad (llamadas en inglés *overlapped*), y las que parecen (y se comportan) como un componente, llamadas "controles". Así, para Windows, un botón, un campo de edición, un cuadro combinado y casi cualquier componente es también una ventana de tipo control.

Las ventanas (ya sean de un tipo o de otro) tienen una característica muy importante: están identificadas por un número único que permite que el sistema operativo pueda comunicarse con ellas. Este número recibe el nombre de *handle*, o en castellano: descriptor, manejador, identificador, etc. y la comunicación, como ya vimos en su día, se consigue a través de los denominados mensajes. Esto tiene una consecuencia lógica: el sistema operativo solo puede comunicarse con una ventana a través de su descriptor de ventana. Otra consecuencia es que, lógicamente, un descriptor de ventana consume recursos. Podría parecer un consumo mínimo, casi despreciable, pero en situaciones de estrés, el sistema puede colapsarse por tener demasiados descriptores creados. En tiempos de los 16 bits (cuando trabajábamos con Windows 3.x), existía una barrera física de 65.535 descriptores como máximo, lo cual no era difícil de alcanzar, teniendo en cuenta que cada componente creado y ventana abierta consume un descriptor. Sin embargo, a partir de Windows 95 se permiten muchos más descriptores, aunque en el mundo de los ordenadores, el infinito no existe. Un descriptor de ventana, además de consumir recursos, hace que el componente funcione de forma más lenta, ya que hay que llamar a funciones del sistema para crear y destruir el descriptor, así como la recepción de los posibles mensajes.

Sabiendo todo esto, ya podemos adivinar por qué hay dos tipos de componentes gráficos: unos son ventanas, es decir: tienen un *handle* de ventana, y otros no lo son: no tienen descriptor.
Los componentes que son ventanas, heredan de `TWinControl`, sin embargo, los componentes gráficos que no son ventanas, heredan directamente de uno de los hijos de `TControl`: `TGraphicControl`. Como veis, los nombres sugieren bastante bien para qué sirve cada uno de los componentes.

¿Y por cual me decido?

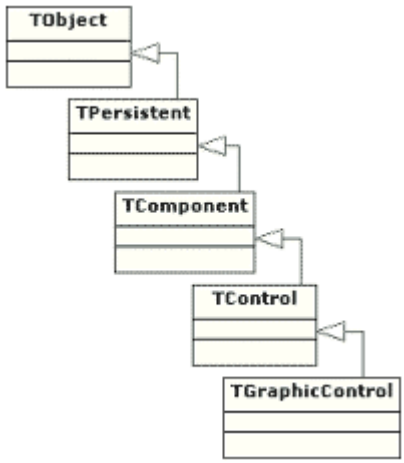
Una de los problemas típicos en la creación de componentes gráficos suele ser que no se elige convenientemente el padre del componente a crear. Para decidir el padre de nuestro componente, debemos saber el tipo de componente que queremos crear. Algunas preguntas que pueden ayudarnos son:

- ¿Nuestro componente necesita recibir eventos de teclado?
  Necesitas un `TWinControl`
- ¿Nuestro componente debe tener foco?
  Necesitas un `TWinControl`
- ¿Nuestro componente debe requerir dibujar formas o imágenes lo más rápido posible?
  Necesitas un `TGraphicControl`
- ¿Vamos a crear muchos componentes (cientos o miles) a la vez?
  Necesitas un `TGraphicControl`
- ¿Nuestro componente va a contener otros componentes dentro de él?
  Necesitas un `TCustomControl`.

La mayoría de los componentes de la paleta de Delphi descienden de `TWinControl`, ya que permiten que el usuario interactúe con ellos. Sin embargo, algunos muy concretos, como `TShape` o `TLabel`, son descendientes directos de `TGraphicControl`, ya que su misión no es que el usuario los maneje, sino que muestren algo en la ventana.

TGraphicControl será nuestro padre

Como ya sabemos, la creación de un componente requiere conocer lo mejor posible la clase padre y el resto de clases antecesoras. En nuestro caso, debemos conocer, entre otras, la clase `TGraphicControl`, tal y como podemos ver en la imagen de la derecha.



Muchas de estas clases ya nos resultarán familiares, ya que `TGraphicControl` se basa en las mismas clases de la VCL que hemos visto hasta ahora: `TComponente` y `TControl`. Otras clases que podemos ver son `TPersistent`, que lo dejaremos para dentro de un rato, y `TObject`, de la no hemos hablado hasta ahora, aunque ya va siendo hora.

`TObject` es la base para cualquier clase que definamos en Object Pascal. Los componentes, por el mero hecho de ser clases, también descienden de `TObject` y heredan todas sus características. Normalmente no es necesario utilizar los métodos heredados de `TObject`, ya que estos son de un nivel demasiado bajo, y no nos solucionarán grandes problemas (excepto en contadas ocasiones).

El componente `TGraphicControl` no es muy complejo, así que no hay mucho que decir. Simplemente se trata de un envoltorio sobre la clase `TCanvas`, que es la encargada de dibujar directamente en pantalla, a través de un grupo de funciones del API llamado GDI (*Graphic Independient Interface*). Esta clase, en realidad, es la que debemos conocer muy bien para saber dibujar en la pantalla.

El único detalle importante que debemos conocer sobre `TGraphicControl` es que nos proporciona un método (llamado `Paint`) que debemos sobrescribir, y donde codificaremos lo necesario para dibujar el aspecto de nuestro componente. Dentro de este método podemos acceder a la propiedad `Canvas`, con la que dibujaremos las imágenes o figuras que sean necesarias. La VCL sólo llamará a este método cuando sea necesario dibujar el componente: cuando se muestra la ventana, si se maximiza, cuando reaparece la ventana detrás de otra, etc.

La clave TCanvas

La clase `TGraphicControl` contiene una propiedad de tipo `TCanvas`, que utilizaremos para dibujar el aspecto del componente. Con un objeto de la clase `TCanvas`, podemos dibujar textos con cualquier fuente instalada en el sistema, o bien líneas, polígonos, arcos, rellenar superficies de color, copiar zonas de imagen, combinar imágenes, etc.

Una de las características más interesantes del GDI, y por lo tanto de la clase `TCanvas`, es que funciona independientemente del tipo de dispositivo sobre el que dibujan. Es decir: las funciones están diseñadas de tal forma que podemos dibujar con ellas en la pantalla, en la impresora, en un plotter, o en cualquier otro dispositivo gráfico, sin preocuparnos demasiado de las características de este dispositivo.

La clase `TCanvas` tiene tres atributos importantes:

- **Font**: se trata de un objeto de tipo `TFont` que representa la fuente que se utilizará cuando se dibuje un texto (a través de los métodos `TextOut` o `TextRect`). Antes de dibujar un texto debemos asegurarnos de que este objeto contienen los valores correctos.
- **Pen**: se trata del pincel (pen en inglés) que se utilizará para dibujar las líneas. El pincel es un objeto de tipo `TPen`, que permite configurar el color con que dibujará, el ancho, el estilo (raya continua, puntos...), etc. Antes de dibujar cualquier línea o polígono (con los métodos `LineTo`, `Polygon`, `Rectangle`, etc.) debemos configurar el tipo de pincel que queremos utilizar.
- **Brush**: se trata de la brocha (brush en inglés) que se utilizará para pintar zonas de relleno. Cuando estamos dibujando polígonos, podemos rellenar el interior con un color o tramado, utilizando para ellos las características que defina la brocha seleccionada. La brocha permite configurar el color de relleno, o el estilo de relleno (vacío, sólido, tramado, etc.). Antes de dibujar un polígono con relleno, debemos configurar apropiadamente los valores de la brocha.

Además de estos tres atributos, la clase `TCanvas` cuenta con varios métodos para dibujar distintas figuras, tal y como podemos ver en la siguiente tabla:

Método	Qué dibuja
`Arc`	Un arco a partir de las coordenadas de una elipse
`Ellipse`	Una elipse a partir de sus coordenadas
`FillRect`	Rellena un rectángulo
`FloodFill`	Rellena un area, delimitada por líneas de un color distinto. Hace el mismo efecto que la herramienta "Bote de pintura" de muchos programas gráficos
`LineTo`	Una línea recta
`FrameRect`	Un rectángulo con borde
`Polygon`	Un polígono cerrado a partir de un conjunto ordenado de puntos
`PolyLine`	Igual que el anterior, pero no rellena el interior con la brocha
`Rectangle`	Un rectángulo
`RoundRect`	Un rectángulo con los bordes redondeados
`TextOut`	Un texto
`TextRect`	Un texto restringido al tamaño de un rectángulo

El tablero

Ahora que ya sabemos quien va a ser el padre de nuestro componente, tenemos que decidir qué propiedades y eventos vamos a proporcionar a nuestros usuarios del componente. Como ya viene siendo habitual, vamos explicar con un ejemplo la creación de este tipo de componentes, y así veremos que es más sencillo de lo que parece. En esta ocasión, vamos a crear un componente gráfico, usando el `Canvas` para dibujar un tablero de ajedrez. Este componente nos puede servir como base para crear un ajedrez completo, o unas damas si no queremos complicarnos la vida.

Crear este componente, como cualquier componente gráfico, no es complejo, sino que la dificultad dependerá del algoritmo necesario para dibujarlo en pantalla. En nuestro caso, no es más que dibujar una serie de filas y columnas, alternando los colores de cada una de las casillas.

En el siguiente listado, podéis ver una primera aproximación del componente:

```
TTablero = class(TGraphicControl)
protected
   procedure DibujarTablero(filas, columnas: integer;
                        color1, color2: TColor);
   procedure DibujarBorde(color: TColor);

   function CanResize(var NewWidth, NewHeight: Integer): Boolean; override;
   procedure Paint; override;
end;
```

En este listado vemos cuatro métodos básicos:

- **DibujarTablero**: contiene todo el algoritmo de dibujo para pintar un tablero a través del `Canvas`. Si nos fijamos en los parámetros, permiten configurar el número de filas y columnas, así como los colores para las casillas blancas y negras.

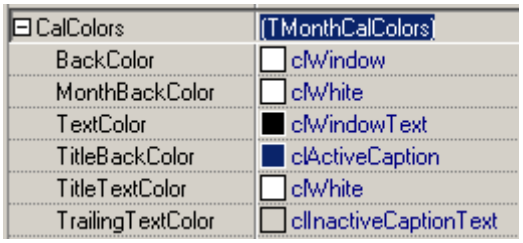
- **DibujarBorde**: dibuja un rectángulo que actuará de borde del tablero. Su único parámetro configura el color que se utilizará para dibujar este borde.

- **CanResize**: se trata de un método sobrescrito que sirve para permitir o bloquear el cambio de tamaño del componente. Cada vez que se intente cambiar el tamaño del componente, ya sea en diseño o a través de código, se llamará a este método, con el que podemos parar la operación de cambio de tamaño.

- **Paint**: se trata método principal, donde se iniciará toda la operación de dibujo. El código de este método es tan sencillo como llamar a `DibujarTablero` (para dibujar las casillas) y después llamar a `DibujarBorde` (para dibujar el recuadro exterior).

El código de cada uno de estos método podéis verlo en los ejemplos que acompañan al artículo.

Agrupando propiedades



Muchas de las propiedades de un componente se suelen referir al mismo aspecto. Por ejemplo, el componente `TMonthCalendar` (de la pestaña Win32) tiene una propiedad llamada `CalColors` que permite configurar los colores con que se muestrará el calendario en pantalla. Esta propiedad, aparece de una forma especial en el Inspector de Objetos, pudiéndose expandir, viendo las sub-propiedades dentro de la principal (como podéis ver en la imagen de la derecha). De este modo, podemos agrupar las propiedades que se refieren al mismo aspecto del componente.

Nosotros, como no podía ser de otra forma, también podemos hacer esto, simplemente sabiendo un par de cosas.

Nuestro amigo TPersistent

Ya hace tiempo que nos venimos encontrando con un personaje llamado `TPersistent`, aunque hasta ahora no hemos tenido el gusto de conocerlo. Bien, se trata una clase básica de la VCL, que nos ofrece todo el mecanismo de persistencia (es decir: almacenamiento) que utiliza Delphi para guardar las propiedades de los formularios. Pero vamos por partes. Todos sabéis que cuando estamos diseñando un formulario desde el entorno de Delphi, y pulsamos el botón "Save", se crea un archivo con extensión "dfm" que contiene la definición del formulario. Esta definición no es más que texto plano, como podéis ver en el Listado de la derecha, en la que se va describiendo los componentes que aparecen en el formulario. Esto, aunque parezca magia, tiene un culpable: la clase `TPersistent`, que es un antecesor de todos los componentes:

```
object Form1: TForm1
  Left = 385
  Top = 188
  Width = 523
  Height = 410
  Caption = 'Form1'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'MS Sans Serif'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Panel1: TPanel
    Left = 48
    Top = 16
    Width = 313
    Height = 193
    Caption = 'Panel1'
    TabOrder = 0
    object MonthCalendar1: TMonthCalendar
      Left = 68
      Top = 23
      Width = 197
      Height = 153
      Date = 38103.5638734259
      TabOrder = 0
    end
    object Panel2: TPanel
      Left = 280
      Top = 24
      Width = 185
      Height = 41
      Caption = 'Panel2'
      TabOrder = 1
      object Button1: TButton
        Left = 16
        Top = 8
        Width = 75
        Height = 25
        Caption = 'Button1'
        TabOrder = 0
      end
    end
  end
end
```

Podemos decir que cualquier objeto, por el mero hecho de heredar de `TPersistent`, puede almacenar el valor de sus propiedades de la sección `published` dentro del archivo DFM.

Bien, pues nosotros también podemos crear nuestros descendientes

de `TPersistent`, precisamente para eso, para que aparezcan en el DFM como una propiedad más.

El Inspector de objetos, además de mostrarnos las propiedades para que las veamos y modifiquemos, nos facilita la edición de ciertos tipos de propiedades, bien sea mostrando listas de selección (para los valores enumerados, booleanos, etc.), o bien agrupando las propiedades de los tipos descendientes de `TPersistent`.

No está bien copiar, pero sí coger ideas

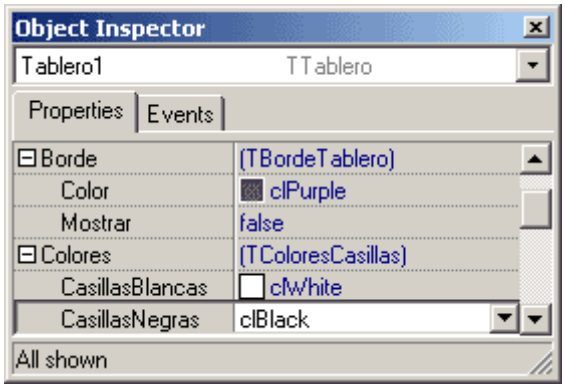
Pues visto que el componente TMonthCalendar hace esto, y que nosotros ya sabemos cómo, vamos a aplicarlo en nuestro tablero de ajedrez. En nuestro caso también podemos configurar los colores de las casillas, ya que en ningún sitio dice que tengan que ser obligatoriamente de color blanco y negro. Además, podríamos configurar el color del borde, incluso no sólo el color, sino también podríamos configurar si debe aparecer o no.

Todo esto lo vamos a hacer a través de dos clases: una para los colores de las casillas, llamada TColoresCasillas, y otra para la configuración del borde, llamada TBordeTablero, tal y como podéis ver en el siguiente listado:

```
TColoresCasillas = class(TPersistent)
private
   FTablero: TTablero;

   FCasillasBlancas: TColor;
   FCasillasNegras: TColor;

   procedure SetCasillasBlancas(value: TColor);
   procedure SetCasillasNegras(value: TColor);

public
   constructor Create(tablero: TTablero);

published
   property CasillasBlancas: TColor
            read    FCasillasBlancas
            write   SetCasillasBlancas
            default clWhite;

   property CasillasNegras: TColor
            read    FCasillasNegras
            write   SetCasillasNegras
            default clBlack;
end;
```

```
TBordeTablero = class(TPersistent)
private
   FTablero: TTablero;

   FMostrar: boolean;
   FColor: TColor;

   procedure SetMostrar(value: boolean);
   procedure SetColor(value: TColor);

public
   constructor Create(tablero: TTablero);

published
   property Mostrar: boolean
            read    FMostrar
            write   SetMostrar
            default true;

  property Color:  TColor
           read    FColor
           write   SetColor
           default clBlack;
end;
```

Ya en el componente `TTablero`, simplemente debemos definir dos propiedades, una para cada clase que acabamos de crear, y automáticamente, el Inspector de Objetos mostrará las propiedades agrupadas, tal y como podéis ver en la imagen de la derecha.



Uno de los detalles más importantes es la codificación de los métodos de escritura de las propiedades. Cuando somos los responsables del dibujado del componente, el cambio del valor de algunas propiedades provocan que el aspecto del componente cambie. Para ello, debemos lanzar la operación de pintado cuando el valor de la propiedad cambie, por ejemplo: cuando se cambie el color que debemos utilizar para pintar las casillas. Para lanzar esta operación, debemos hacer uso del método `Invalidate`, que viene heredado de `TControl`.

Con este método, provocado un repintado completo del componente, llamándose al método `Paint`, y lanzándose así todo el código que hemos escrito para dibujar en la pantalla. Podéis ver cómo se llama a este método en el siguiente listado:

```
procedure TColoresCasillas.SetCasillasBlancas(value: TColor);
begin
   if value <> FCasillasBlancas then
   begin
      FCasillasBlancas := value;
      FTablero.Invalidate;
   end;
end;
```

Es conveniente utilizar este método con cautela, ya que cada vez que se llame provocaremos un repintado del componente, aunque varias llamadas consecutivas desembocarán en un único repintado. En el último listado, podéis ver un ejemplo en que evitamos el uso de `Invalidate`, en un caso en que es suficiente pintar el borde, y no el componente completo.

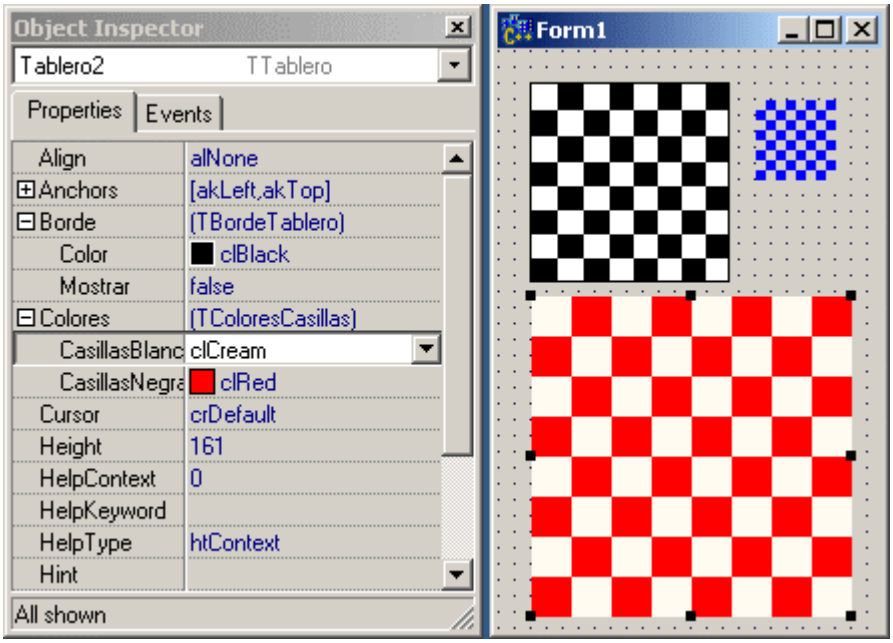
```
procedure TBordeTablero.SetMostrar(value: boolean);
begin
   if value <> FMostrar then
   begin
      FMostrar := value;

      if FMostrar then
         FTablero.DibujarBorde(FColor) // sólo se dibuja el borde
      else
         FTablero.Invalidate; // se dibuja todo
   end;
end;
```

Los ejemplos

Podéis descargar un ejemplo completo en el código fuente del tablero de ajedrez que hemos ido explicando durante el artículo:

http://users.servicios.retecal.es/sapivi/src/TableroAjedrez.pas

Una vez que tengáis el componente correctamente instalado en el sistema, podéis insertarlo en cualquier formulario, pudiendo mostrar tableros de ajedrez como los de la siguiente imagen:



¡Espero que os sea de utilidad!

Autor: JM

Este artículo apareció publicado por primera vez en el número 4 de la revista Todo Programación, editada por Studio Press, S.L. y se reproduce aquí con la debida autorización.