Programando por diversión

Por Luis Alfonso Rey. Danysoft Internacional.

# Prólogo

Una de las cosas que más me atrajo hacia el mundo de la programación fueron los juegos, o más bien los video juegos. En el incipiente mercado de la micro informática a mí me gustaba asumir la identidad de Guybrush Threepwood, Larry o Simon The Sorcerer y pasarme las horas muertas jugando. Tampoco le hacía ascos al RtcW, Doom o Quake.

Pero si algo me motivó sobre manera fue la posibilidad de crear juegos. Desde entonces he creado muchos, no con orientación comercial, y menos ahora en donde hace falta una multinacional para crearlos y otra para lanzarlos; si no para consumo propio: para que mis hijos practiquen inglés, para probar tal o cual tecnología e incluso para aprender lenguajes; pero sobre todo porque me gusta.

Me gusta programar, me gustan los juegos y tengo un poco (o un mucho según a quien se pregunte) de friki.

Por otro lado la vida me ha llevado a dedicarme profesionalmente a la programación, lo cual agradezco profundamente pese a los sinsabores que pueda tener en algún momento, y hete aquí que mi labor es también promover y dar a conocer aspectos de la programación de diferentes lenguajes y tecnologías. Así que uno más uno pues dos, he terminado haciendo un tutorial que aúna las dos cosas, FireMonkey y el Tetris.

En este tutorial podrás encontrar estilos, Live Bindings, composición de interfaces, control de teclas, etc…

Pero al final te podrás echar una partida al Tetris, que tampoco está mal.

# Creando el proyecto

Antes de empezar es obligado comentar de antemano los requisitos, evidentemente se necesita conocimiento de Delphi y una versión de Delphi con Firemonkey, esto está hecho con Delphi 10.1 Berlin, pero es probablemente trasladable a versiones anteriores seguramente hasta XE7 sin grandes cambios.

Si además se quiere desplegar en Android e IOS como se explica aquí, también sería necesario disponer de un móvil Android compatible, con su cable de conexión al PC y de un Mac para la ejecución en el emulador. Aquí no se explicarán los pasos necesarios para configurar este sistema, pero si se indicaran recursos para hacerlo y poder ejecutar en estos entornos.

Bien hechas estas salvedades vamos a comenzar el proyecto, no sin antes establecer el requisito mínimo de adaptabilidad al entorno en que se ejecute.

Bien teniendo eso en mente he decidido crear una “Multi-Device Application” y en el selector de tipo optar por una plantilla Header/Footer. La razón es porque tanto los componentes de cabecera como de pie ya establecen un área delimitada donde posicionar los controles de apoyo, mientras que el área central será la de juego. En la imagen podemos ver la selección de proyecto.

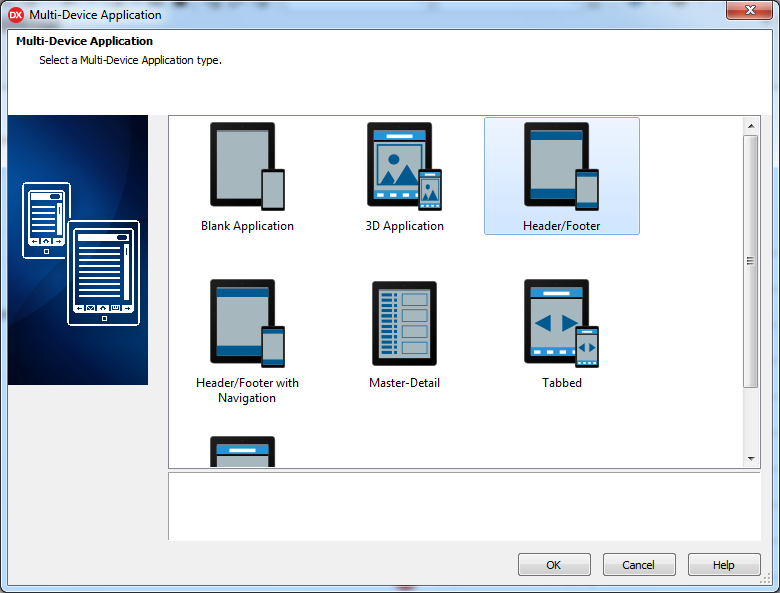


Ilustración 1 - Selección de proyecto

## La cabecera

Al ser un proyecto destinado a móvil y escritorio, además de control por teclado también necesitaremos botones en pantalla. De esta manera el pie quedará ocupado por estos controles y reservaremos la cabecera para mostrar la puntuación y la siguiente pieza.

Siendo así y si nos fijamos en lo que el componente header contiene veremos que es nada más que una etiqueta alineada a *Client*. Se ha modificado para situarla a la izquieda con la alineación *MostLeft* porque de esa manera ocupará el lado izquierdo, pero con una prioridad superior a cualquier otra alineación, excepto evidentemente las que también sean de tipo Most.

De esta manera a mano derecha podemos situar un *TImageControl* destinado a mostrar la pieza siguiente, con alineación MostRight y la propiedad HitTest deshabilitada para que no nos aparezca el desagradable diálogo de selección de imagen en ejecución, cada vez que hagamos clic sobre él.

Con estos dos controles situados, ya podemos desplegar dos etiquetas en el header una con un rótulo fijo “Score” y alineada a *Top*, y otra alineada a *Client* e inicializada a 0 para mostrar la puntuación. Como los controles anteriores están situados con alineamientos tipo “most” estas etiquetas quedan confinadas al espacio que ambos controles les dejan. Por último reseñas que las fuentes de las etiquetas han sido también retocadas asignándoles una fuente a través de la sub propiedad “Font”, de la propiedad “TextSettings”. En este caso se ha asignado la fuente *Segoe Print*, en varios tamaños. La imagen final sería esta:



Ilustración 2 Cabecera de la aplicación

## El pie

El pie, destinado a contener los controles, es de una confección más simple. Tan sólo hemos depositado cinco botones alineados MostLeft, Left, Client, Right y MostRight, respectivamente.

La complejidad proviene de que hemos decidido personalizar el aspecto de estos botones y aunque se podría haber de varias maneras, la manera más canónica, correcta o simplemente interesante ha sido hacerlo por estilos.

Dado que FireMonkey es un motor de representación vectorial, y al igual que otros motores de representación similares, sigue la estrategia de crear “primitivas de dibujo” simples para “componer” elementos más complejos. Así un botón puede estar compuesto de rectángulos, por ejemplo. Es una estrategia ingeniosa y poderosa ya que permite “redefinir” cada elemento desde la base, pero ciertamente muy laboriosa si tenemos en cuenta la cantidad de elementos que puede tener Delphi.

Por eso esta estrategia de componer fue complementada por otra que permitía cargar un bitmap y definir zonas en el que correspondían con cada uno de los controles de la paleta en sus diferentes estados, habilitado, deshabilitado, pulsado en el caso de los botones, etc… Esta estrategia era mucho más ágil a la hora de componer estilos.

Ambos mecanismos siguen el mismo principio, sólo que el primero utiliza muchos más componentes. Además el segundo permite cambiar el estilo tan solo rediseñando el bitmap asociado, siempre que se respeten la áreas definidas.

En cualquier caso nosotros lo que vamos a hacer es modificar el estilo de los botones, no crear un nuevo y completo estilo. Para ello basta hacer clic con el botón derecho sobre un botón y seleccionar la opción “Edit Custom Style”, también podríamos seleccionar Edit Default Style”. La principal diferencia es que la primera opción crea un estilo para el botón seleccionado y luego lo edita, mientras que la segunda edita el estilo por defecto, por lo que en principio cambiaríamos todos los botones a la vez.



Ilustración 3 Menú al contexto de un control en Firemonkey

Tras esta selección se abrirá el diseñador de estilos embebido en el IDE de Delphi. Este es sólo válido para Firemonkey (en el caso de los estilos VCL habrá que hacerlo utilizando una herramienta externa llamada “BitmapStyleDesigner”).

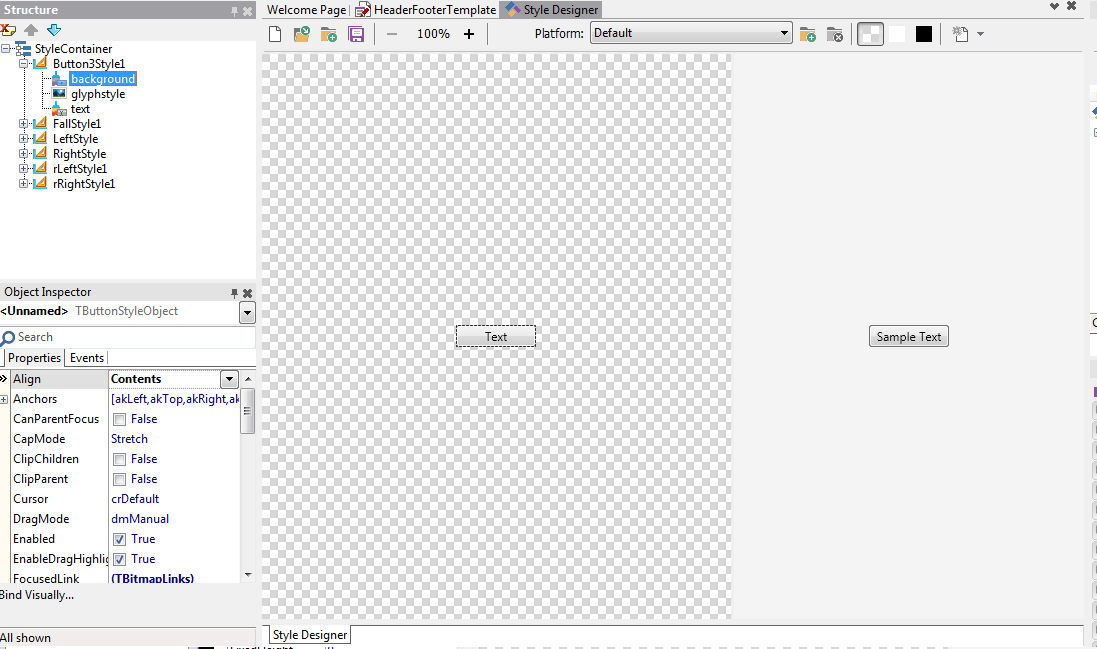


Ilustración 4 Diseñador de estilos Firemonkey embebido

Como se puede ver en la imagen además de una superficie de trabajo y una vista previa completamente nuevas, el diseñador utiliza la ventana de “Structure” y la de “Object Inspector”, para ayudar a manipular los estilos.

Además si observamos el estilo actualmente editado “Button3Style1” en la vista de “Structure”, podemos ver que contiene tres elementos un fondo que contiene la referencia a un bitmap del modo antes mencionado, Glyph y un control de texto; actualmente estos son los que posibilitan la funcionalidad y aspecto actual del botón.

Lo que hacemos parar cada uno de los botones situados en la botonera es borrar estos componentes y depositar una *TShape*, en concreto un *TPath*. Los “shapes” (formas) son las primitivas de dibujo más simples en Firemonkey. Con estas formas son con las que los estilos de Firemonkey tradicionalmente se componen. En el caso del *TPath*, es una forma un poco más compleja que *TCircle* o *TRectangle* ya que admite una definición de forma libre, en un formato SVG o XAML en su propiedad Data.

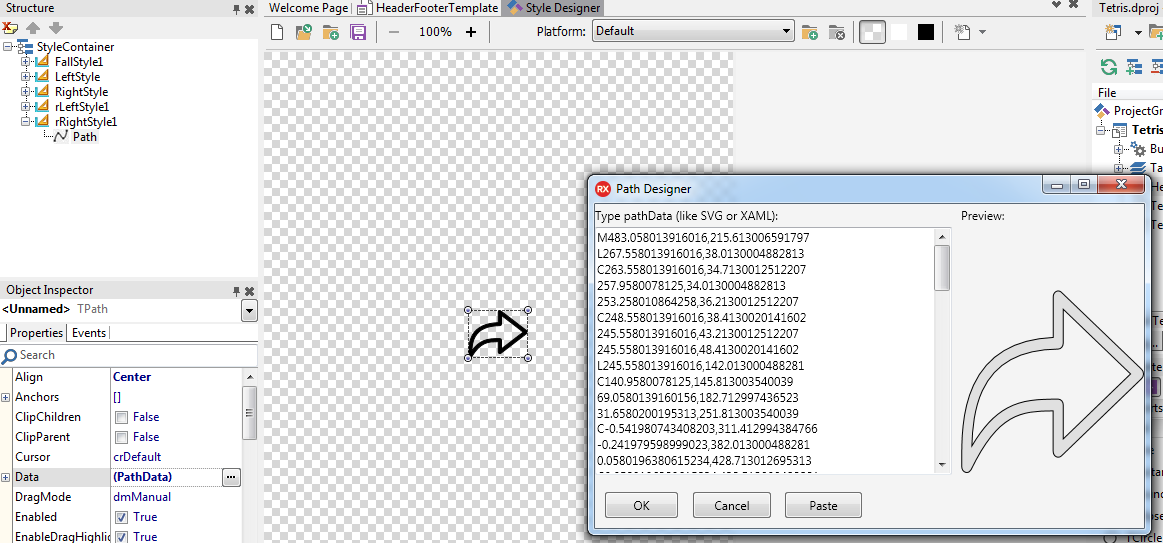


Ilustración 5 Definición del estilo Firemonkey. Detalle de la propiedad Data del TPath

Aprovechando esto y utilizando una página de internet donde descargar iconos SVG, obtenemos los iconos de manera gratuita a cambio de acreditar el origen, cosa que aprovecho a hacer aquí. Autor: Gregor Cresnar. Sítio: <http://www.flaticon.com>. Licencia: Creative Commons BY 3.0. <http://creativecommons.org/licenses/by/3.0/>.

Por último hemos de desmarcar la propiedad *HitTest*, para evitar que el control *TPath* capture los eventos de ratón, y sean sin embargo transmitidos al control *TButton*. Y con ello conseguimos la botonera perseguida.



Ilustración 6 Botonera de la aplicación

## Finalizando el interfaz

Como ya mencionamos antes en el espacio entre cabecera y pie vamos a situar un control *TImageControl*, con el alineamiento a *Client* y la propiedad *HitTest* desactivada, como en el anterior y por las mismas razones.

Evidentemente el juego debe de tener unos controles mínimos para poder comenzar una partida y cerrar el juego. Además también ha de tener un mecanismo de comunicación de mensajes.

Para solucionar esto lo que vamos a hacer es situar un panel en el formulario con el alineamiento *VertCert*, que nos permite centrar el panel en la aplicación y ajustarlo para que ocupe toda la sección central de esta, sólo variando la altura para hacerlo más grande o pequeño.

En este panel vamos a situar una etiqueta para mostrar mensajes y un par de botones para las opciones principales del juego. Trabajo ya conocido con la excepción de que en ese caso no vamos a crear estilos personales para los botones, bien al contrario utilizando la propiedad *StyleLookup* asignaremos uno por defecto concretamente “toolbutton”, que aplana el botón y lo reduce a todo texto.

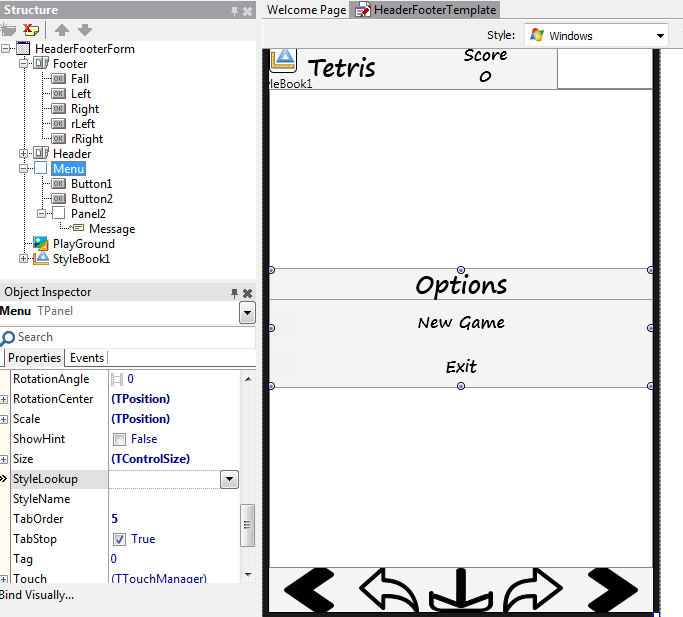


Ilustración 7 Diseño final. Detalle del árbol de estructura

# El Código

Aunque modelos de desarrollo como el de FireMonkey sugieren arquitecturas como MVVM o similares, nosotros hemos escogido una arquitectura más clásica, de componente, que es la más tradicional en Delphi.

A vista general hemos creado una clase TScreen que es la encargada de realizar todo el trabajo, mientras que hemos agregado una unidad para constantes y tipos de apoyo, dejando la unidad del formulario para tareas de interfaz.

## La unidad “Tetris.Pieces”

Esta unidad es muy simple, contiene un enumerado con los seis tipos de piezas soportadas, un tipo base que define un array de cuatro por cuatro bloques en el que definir cada una de las “vistas” de las piezas, y por último una constante que efectivamente las define.

Hemos elegido este enfoque, en el que todas las posibles formas están ya dibujadas, puesto que simplifica significativamente el código. Además no son demasiadas, cuatro formas posibles por seis piezas, veinticuatro formas posibles.

Hemos elegido un array definido en vez de dinámico porque suelen ser más rápidos, aunque haya piezas que rotadas no varían.

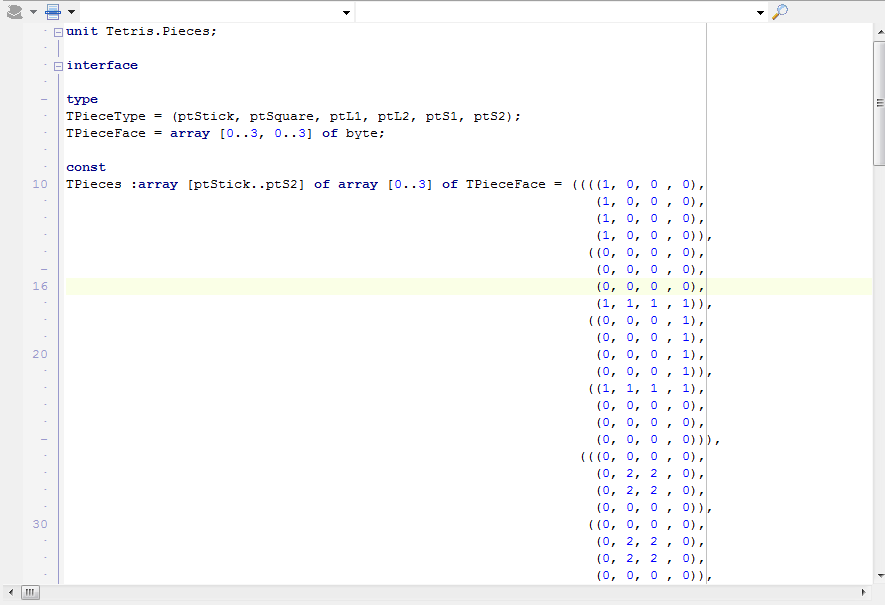


Ilustración 8 Vista de la unidad Tetris.Pieces

## La unidad HeaderFooterTemplate

En esta unidad como ya dijimos antes, se producen los enlaces del interfaz, por lo que la mayoría de los métodos de la única clase de la unidad, “THeaderFooterForm”, son invocaciones directas a métodos de la clase TScreen.

Salvedades hechas del método “FormKeyDown” que es un case para solucionar el control por teclado. Aquí hay dos elementos reseñables la instrucción:

key := key + Ord(KeyChar);

Que hay que hacer así porque, por alguna razón el evento detecta las teclas o a través del parámetro key o del KeyChar, así que con este truquito podemos ponerlas todas en un *case*. Por otro la línea

vkMenu, vkEscape: begin

Que establece la tecal vkMenu, que es la tecla de menú en móviles y tabletas.

Pero la salvedad mayor es el método Button1Click, que es a la sazón el método que permite comenzar una nueva partida. En el preparamos el objeto TScreen que vamos a utilizar para jugar, asignamos su evento OnGameTerminate, para reaccionar cuando perdamos, pero además creamos dos expresiones de binding que permiten enlazar las propiedads *Score* y *NextPieceBmp* (puntuación y la imagen de la siguiente pieza) con la etiqueta y el control de imagen de la cabecera, destinados a mostrar esta información.

Por último este mismo método también asigna el evento OnIdle de la aplicación. En esencia la clase TScreen representa la pantalla de juego, que a cada impulso se refresca. Estos impulsos se ejecutan con el método Refresh de esta clase, pero este método ha de ser llamado periódicamente. En vez de usar un “Timer” hemos decidido usar esté método mucho mas económico en términos de recursos de sistema.

## La unidad Tetris.Artifacts

Esta unidad es la más compleja de todas. Contiene sólo dos tipos *TCurrentPiece* y *TScreen*.

El primero es muy simple, un registro (tipo por valor) que contiene la información necesaria de una pieza en juego, las coordenadas cartesianas, el tipo de pieza y la cara (rotación) que tiene en ese momento.

La clase TScreen es más compleja. En esencia define un área de juego *FPlayGround* que sirve para situar los elementos estáticos del juego, piezas ya jugadas. Cada vez que se invoca su método Refresh y si el tiempo transcurrido desde la última operación de refresco es mayor que el establecido en *FTimeStep* (en milisegundos), el método intenta hacer descender la pieza del juego *FCurrentPiece*, copiándola modificando su altura cartesiana y comprobando si no colisiona con nada, en ese caso convierte en perdurable el cambio y si no evalúa si el campo de juego está lleno en cuyo caso para la operación de refresco *FStopped* y lanza el evento correspondiente *FOnGameOver*. Si el campo no estuviera lleno fija la pieza y crea otra nueva tomando la almacenada en la siguiente *FNextPiece* y generando otra aleatoria como la posterior.

Además la clase toma la altura y anchura tanto del campo como de los diferentes cuadrados que componen las piezas, para que así se pueda ajustar a los diferentes “Form Factors” de los diferentes dispositivos.

El resto de métodos se dividen en dos, los de movimiento como *MoveLeft, RotateLeft, etc.* Que al igual que Refresh copian la pieza, la transforman en el sentido de cada método y comprueban su validez. Y los de mecánica de juego, como:

* *PlayGroundFull*, que comprueba si existe algún “cuadrado” ocupado en la línea más alta del área de juego cuando se va a sacar una pieza, determinado así si el juego a terminado.
* SetNextPiece, que copia la pieza en el área de juego cuando ya no se puede mover más y establece la siguiente.
* SearchForCompleteRows, que recorre el área en búsqueda de las filas completas, incrementa la puntuación y aumenta, cada cincuenta puntos, un diez por ciento la velocidad del juego.
* getNextPieceBmp, que genera un bitmap con la imagen de la siguiente pieza a jugar.
* SetCanvasBrushColor, que en función de un número cambia el color de la brocha para pintar en los bitmaps, consiguiendo piezas de diferentes colores.
* Render, que genera un bitmap con el estado actual del juego.