DELPHI AL LÍMITE

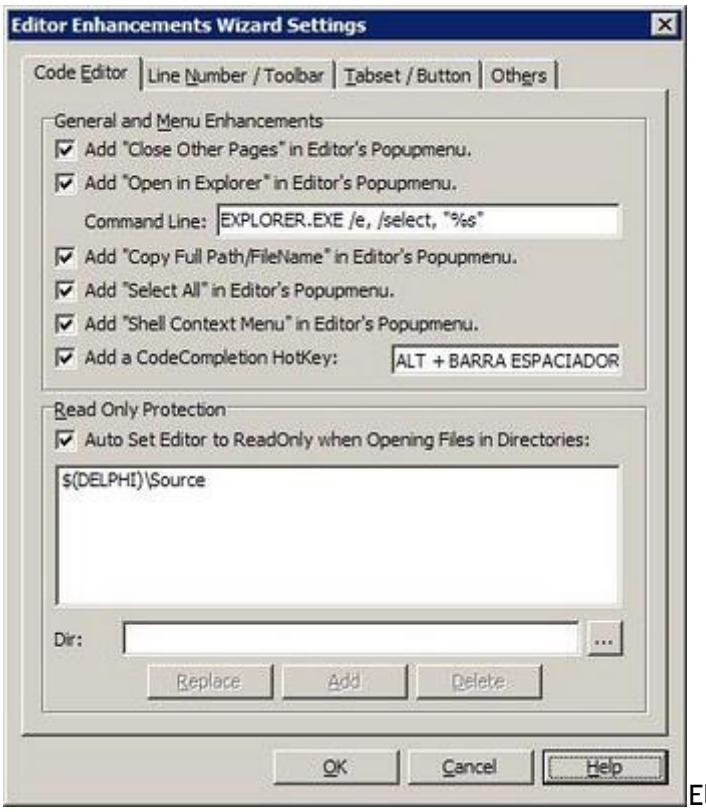
30 diciembre 2009

El experto cnPack (y 8)

Termino esta serie de artículos dedicados al experto **cnPack** viendo las últimas opciones que me quedan por comentar, como pueden ser los atajos de teclado o el asistente de escritura de código que viene a sustituir a Code Insight.

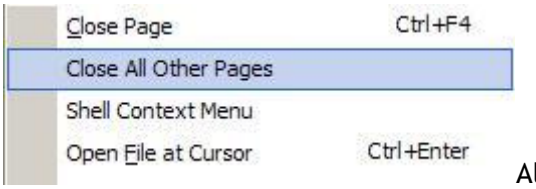
EDITOR ENHACEMENTS

Las opciones del editor de código proporcionan una gran cantidad de características tales como la barra de herramientas, los números de línea, los botones de selección, etc. Al pulsar el botón **Settings** vemos todas estas opciones de la pestaña **Code Editor**:



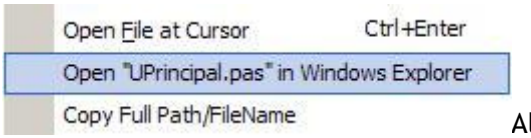
El apartado **General and Menu Enhacements** contiene todas estas opciones:

Add "Close Other Pages" in Editor's Popupmenu: Si esta opción está activada lo que hará será mostrar esta opción en el menú contextual cuando pinchamos con el botón derecho del ratón sobre el código fuente:

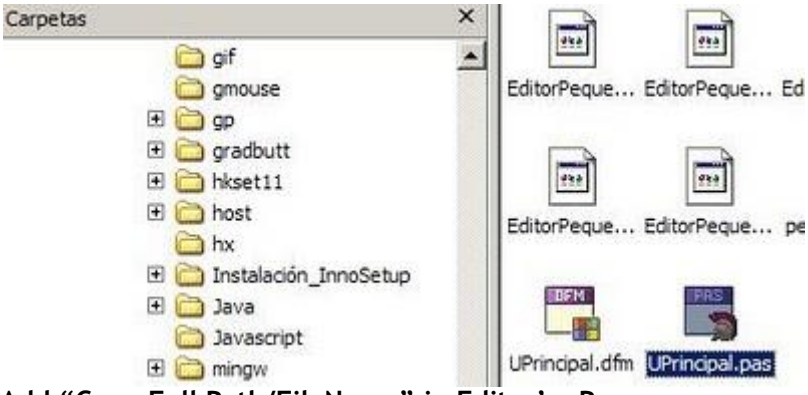


Al pulsar sobre esta opción cerrará todas las pestañas de código menos en la que estamos situados. Con ello liberamos memoria y nos concentramos en lo que estamos haciendo.

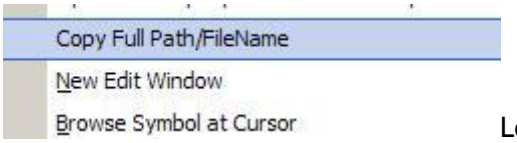
Add "Open in Explorer" in Editor's Popupmenu: esta opción también viene incluida en el mismo menú contextual y si por ejemplo estamos situados en la unidad **UPrincipal.pas** se vería esta opción:



Al pulsar sobre esta opción, lo que hace es abrir el Explorador de Windows y se va a la carpeta donde está esta unidad y además la selecciona:



Add "Copy Full Path/FileName" in Editor'ss Popupmenu: seguimos con otra opción del mismo menú que se muestra así:



Lo que hace esta opción es copiar en el portapapeles la ruta completa donde se encuentra la unidad que estamos editando.

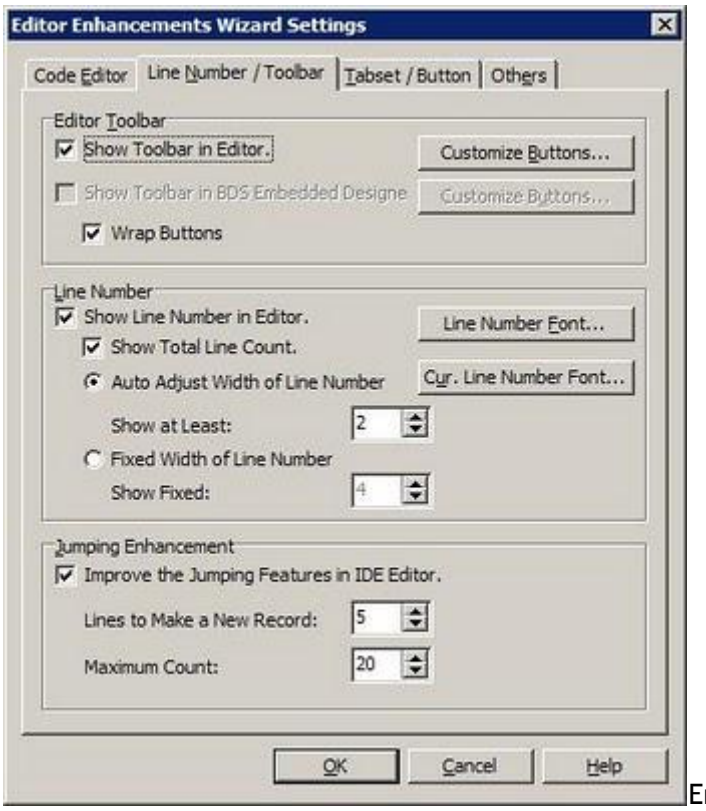
Add "Select All" in Editor's Popmenu: Activa la opción de seleccionar todo el código fuente sin tener que recurrir al menú **Edit** de Delphi.

Add "Shell Context Menu" in Editor's Popmenu: Al seleccionar esta opción lo que hacemos es provocar el menú contextual que aparece en el Explorador de Windows, por lo que podemos comprimir con RAR u editar con otro programa sin salir de Delphi.

Add CodeCompletion HotKey: Activa el completado automático de código pulsamos la combinación de teclas ALT + BARRA DE ESPACIO.

Y dentro del apartado **Read Only Protection** tenemos la opción **Auto Set Editor to ReadOnly when Opening Files in Directories**, la cual protege los archivos de código fuente que intenten editarse fuera de Delphi mientras estamos trabajando con ellos dentro del IDE. Por defecto, protege el directorio donde está alojado nuestro proyecto.

Pasemos a la pestaña **Line Number / Toolbar**:

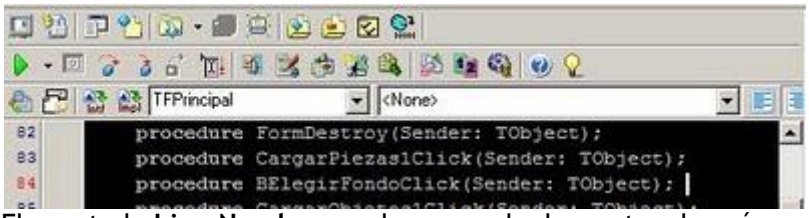


En el apartado **Editor Toolbar** tenemos estas opciones:

Show Toolbar in Editor: activa la visualización de la barra de herramientas principal que aparece en la parte superior del código:


Show Toolbar in BDS Embedded Designe: muestra la barra de herramientas en las versiones modernas de Delphi (RAD Studio).

Wrap Buttons: Si esta activa esta opción, lo que hace es partir la barra de herramientas en varios trozos cuando el editor se estrecha horizontalmente:



El apartado **Line Number** es el encargado de mostrar los números de línea en la parte izquierda del editor. Tiene estas opciones:

Show Line Number in Editor: Muestra los números de línea. Esta opción es conveniente desactivarla para las versiones modernas de Delphi que ya traen esta característica.

Show Total Line Count: Muestra en la esquina inferior izquierda del editor el número total de líneas de la unidad actual:

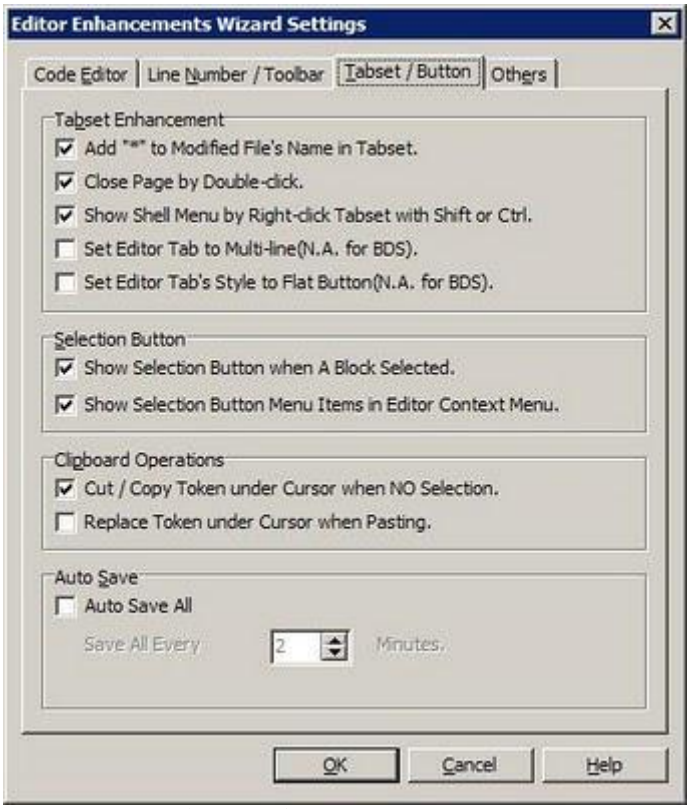
Auto Adjust Width of Line Number: Ajusta el ancho de la barra de números de línea con un ancho al número más grande.

Fixed Width of Line Number: El ancho de la barra de números será igual al número de dígitos que le pongamos (por defecto 4).

También tenemos a la derecha dos botones para seleccionar la fuente y el color de los números de línea e incluso del número donde estamos situados.

El apartado **Jumping Enhacement** no tengo ni idea para que sirve. Se supone que es para los saltos en las búsqueas o algo de eso, pero ni dice nada en el manual y por muchas pruebas que he realizado no noto la diferencia en activarlo o desactivarlo. La verdad es que a veces en el manual de cnPack se despachan agusto escribiendo la ayuda.

La tercera pestaña es **Tabset Button**:



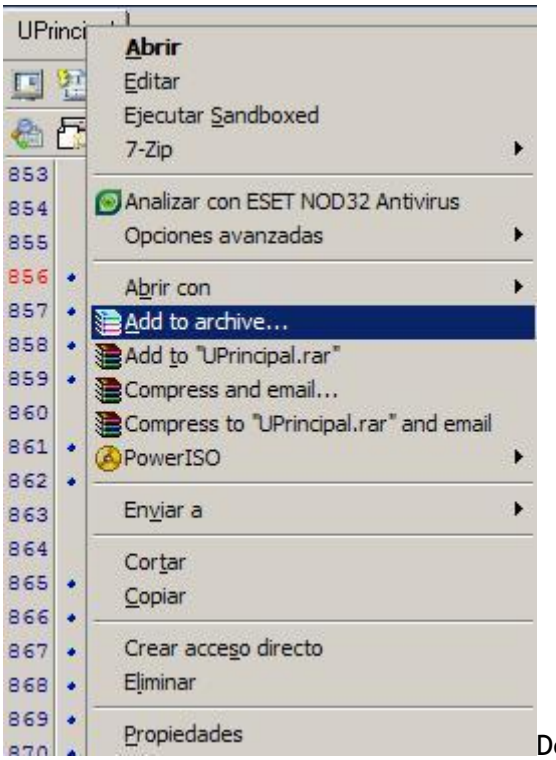
El apartado **Tabset Enhacement** contempla las opciones de las pestañas del editor de código y los botones con estas opciones:

Add '*' to Modified File's Name in Tabset: Al modificar el código fuente después de grabar en disco hará que se muestre un asterisco en la pestaña actual:

Una vez que grabamos desaparecerá.

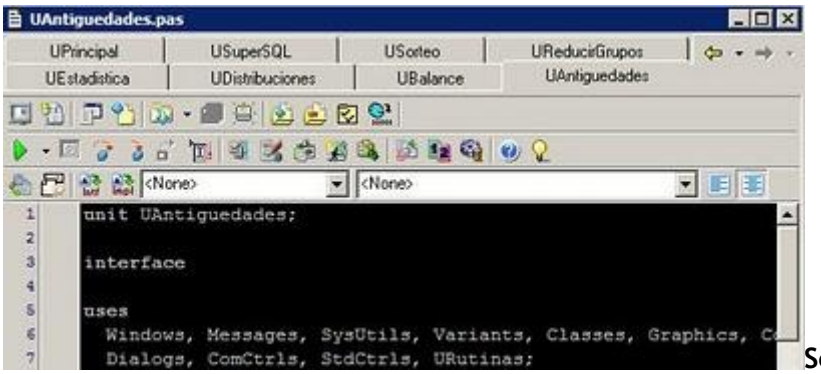
Close Page by Double-click: Podemos cerrar la pestaña de código actual haciendo doble clic sobre la misma.

Show Shell Menu by Right-click Tabset with Shift or Ctrl: Al pinchar una pestaña con el botón derecho del ratón y manteniendo pulsada la tecla SHIFT o CTRL mostrará el menú contextual del explorador de Windows:



De este modo, podemos comprimir a RAR la unidad sin salir del IDE.

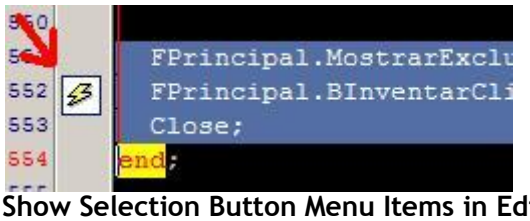
Set Editor Tab to Multi-line: Establece un sistema de pestañas multilinea para cuando tengamos abiertas muchas pestañas y el editor de código de estreche (no funciona en RAD Studio):



Set Editor Tab's Style to Flat Button: Pone las pestañas con estilo plano (no funciona en RAD Studio):

La sección **Selection Button** contiene dos opciones:

Show Selection Button when a Block Selected: Mostrar el botón de selección cuando se ha seleccionado un bloque de código:



Show Selection Button Menu Items in Editor Context Menu: Mostrar botón de selección en el menú contextual cuando se ha seleccionado un bloque de código.

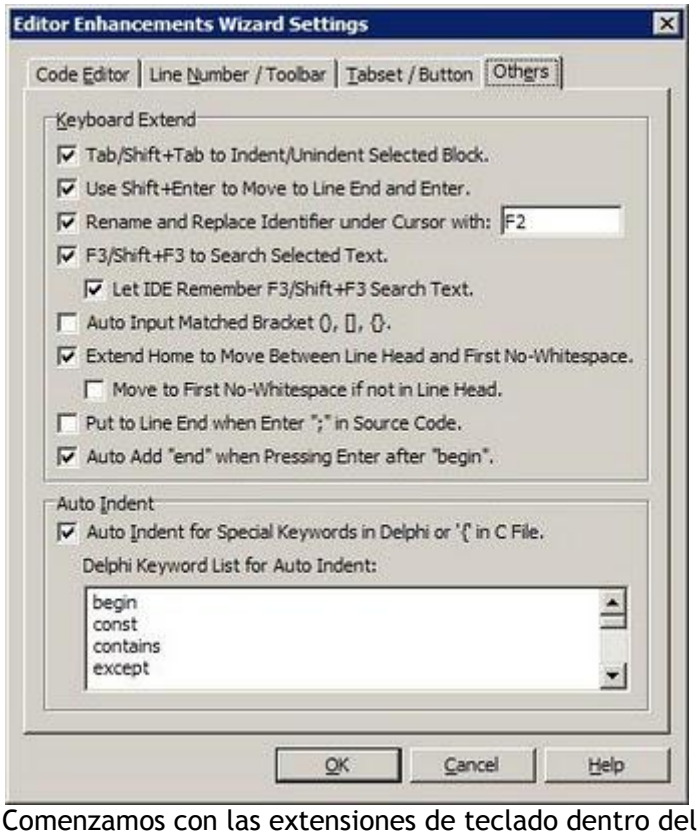
Luego tenemos el apartado **Clipboard Operations** con estas opciones para el portapapeles:

Cut/Copy Token under Cursor when NO Selection: Cuando estamos encima de una palabra y pulsamos CTRL + C ó CTRL + X, copiará o cortará dicha palabra si tener que seleccionarla. Es algo muy cómo para trabajar rápido.

Replace Token under Cursor when Pasting: Si activamos esta opción, al pegar cualquier palabra, reemplazará aquella donde tengamos situado el cursor.

El último apartado es **Auto Save** y permite guardar las últimas modificaciones de nuestro código cada x minutos.

Y la última pestaña de este formulario de opciones es **Others**:



Comenzamos con las extensiones de teclado dentro del apartado **Keyboard Extend**:

Tab/Shift + Tab to Indent/Unindent Selected Block: Al pulsar el TABULADOR desplazamos el bloque de código seleccionado dos espacios a la derecha y al pulsar el TABULADOR + SHIFT lo desplazamos a la izquierda. Esto es mucho más cómodo que las

combinaciones de teclas CTRL + K + I y CTRL + K + U que trae Delphi.

Use Shift + Enter to Move to Line End And Enter: Al pulsar la combinación de teclas SHIFT + INTRO cuando entamos en medio de una línea, hace como si hubiésemos pulsado la tecla FIN y luego la tecla INTRO para insertar una línea.

Rename and Replace Identifier under cursor with F2: Supongamos que estamos situados encima del nombre de esta función:

```
private
  { Private declarations }
  procedure GuardarEnTabla(Tabla: TIBQuery;
public
```

Al pulsar la tecla **F2** se nos abrirá un cuadro de diálogo para reemplazar esta palabra por otra:

F3/Shift + F3 to Search the Selected Text: Si tenemos un trozo de texto seleccionado y pulsamos la tecla F3, buscará hacia abajo la siguiente coincidencia. Si queremos que busque hacia arriba puslamos SHIFT + F3. Esta opción lleva incluida otra debajo llamada **Let IDE Remember F3/Shift + F3 Search Text**. Lo que hace es memorizar la última búsqueda que hemos hecho en el IDE de Delphi, por si se nos ocurre buscar con CTRL + F.

Auto Input Matched Bracket: Si activamos esta opción, al abrir un paréntesis, corchete o llave, automáticamente escribirá al lado la de cierre. Comodísimo.

Extend Home to Move Between Line Head and First and First No-WhiteSpace: si activamos esta opción y la que tenemos debajo, cuando pulsemos la tecla INICIO, el cursor se situará al comienzo de la línea, independientemente si está más desplazada a la derecha o a la izquierda. De otro modo, se iría a la izquierda del todo. Es ideal para ir al comienzo de una línea que esta bastante identada.

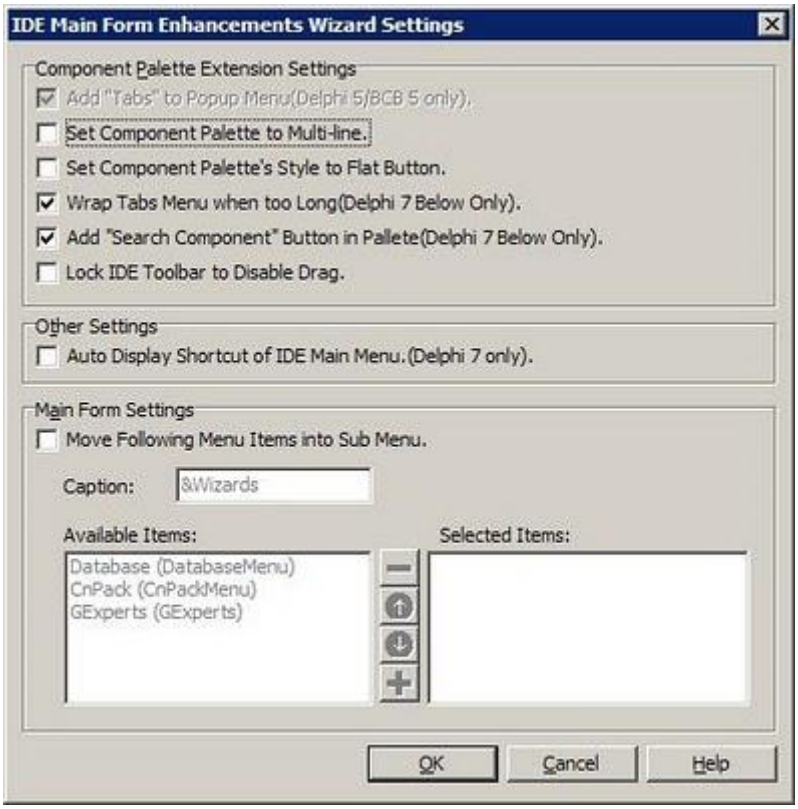
Put to Line End when Enter ";" in Source Code: Coloca el código de final de línea automáticamente cuando introducimos ";" al final de una línea que no es de comentario. Visualmente no se nota nada.

Auto Add "end" when Pressing Enter after "begin": Añade la palabra **end** cuando pulsamos la tecla INTRO después de la palabra **begin**. Lo mismo que hacen las últimas versiones de Delphi.

Y terminamos este formulario viendo el apartado **Auto Indent** donde tiene una opción que permite desplazar el cursor dos espacios a la derecha cuando pulsamos INTRO después de las palabra reservadas como **begin**, **const**,**private**, etc.

IDE MAIN FORM ENHACEMENTS

Esta opción añade características avanzadas a la paleta de componentes de Delphi 7:

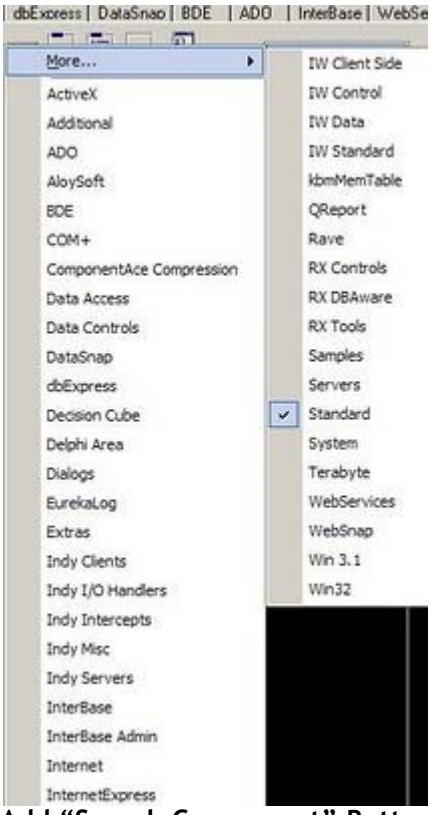


Dentro del apartado **Component Palette Extensión Settings** tenemos estas opciones:

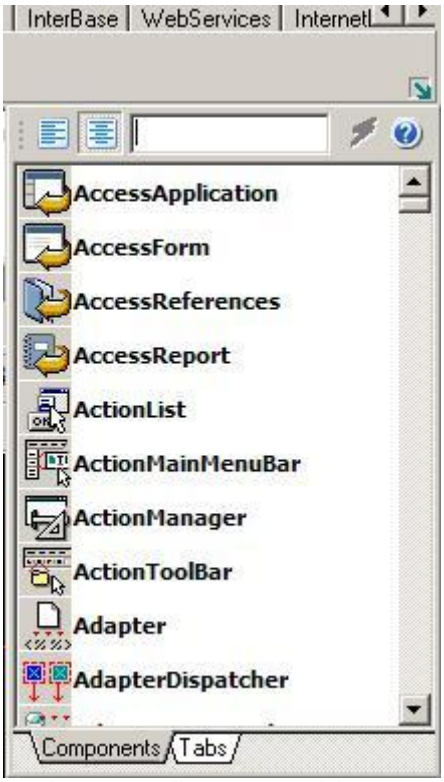
Set Component Palette to MultiLine: Al activar esta opción podemos visualizar a la vez todas las pestañas de los componentes:

Set Component Palette's Style to Flat Button: Cambia el estilo de las pestañas de la paleta de componentes y las deja planas:

Wrap Tabs Menu when too Long (Delphi 7 Below Only): En las versiones de Delphi 7 e inferiores hace que el menú**Tabs** que aparece al pinchar con el botón derecho del ratón sobre la pestaña de componentes se fragmente en varias partes para facilitar su visualización en monitores con poca resolución:



Add "Search Component" Button in Palette (Delphi 7 Below Only): Activa en versiones de Delphi 7 e inferiores la búsqueda avanzada que aparece en la parte derecha de la paleta de componentes:

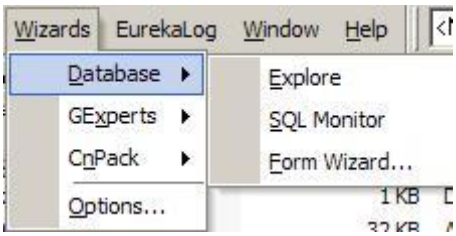


Lock IDE Toolbar to Disable Drag: Desactiva la posibilidad de mover las barras de herramientas que están junto al formulario que estamos editando para evitar el error de arrastrarlas y perder su visualización.

En el apartado **Other Settings** tenemos la opción **Auto Display ShortCut of IDE Main**

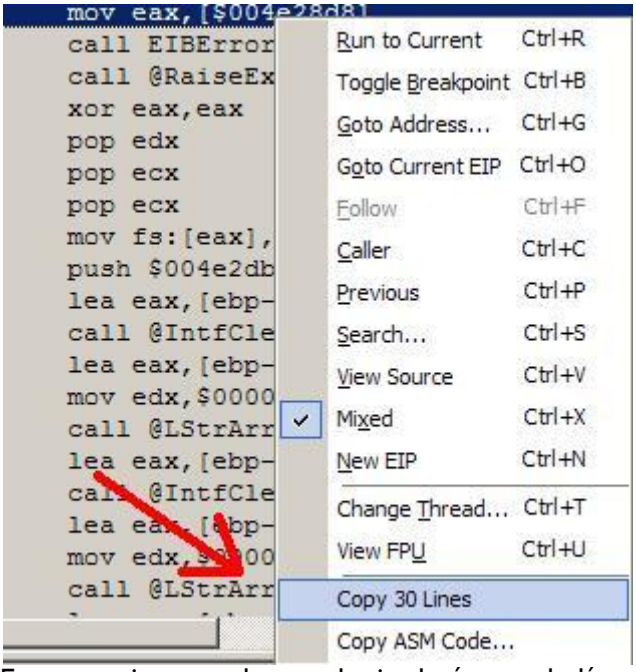
Menu que fuerza al menú de Delphi a mostrar todas las combinaciones de teclas que estén asociadas a cada opción.

Y por último, tenemos el apartado **Main Form Settings** con la opción **Move Following Menu Ítems into Sub Menu** que permite que aparezca en el menú principal de Delphi la opción Wizards para que llame a estos asistentes:

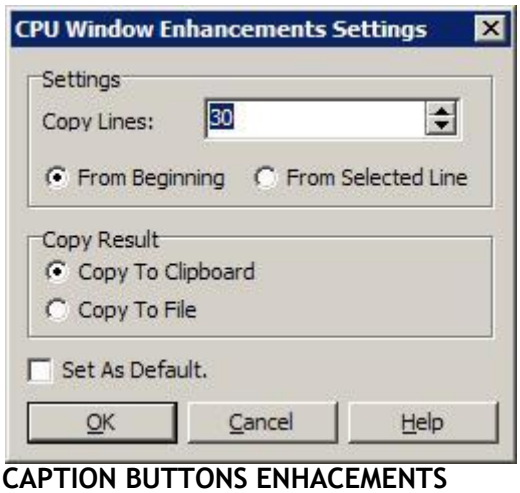


CPU WINDOW ENHACEMENTS

Esta opción proporciona características avanzadas relacionadas la ventana de la CPU. Añade dos opciones al menú contextual que aparece en la ventana de la CPU:

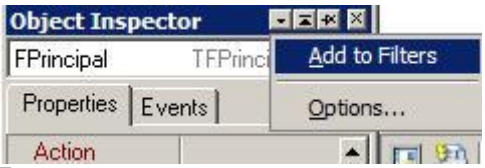


En sus opciones podemos elegir el número de líneas que queremos copiar y si queremos llevarlas al portapapeles:

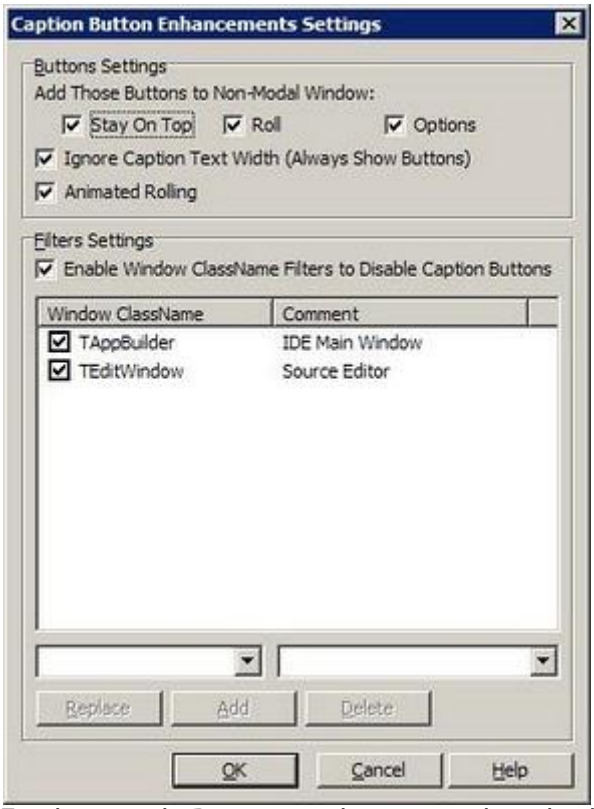


CAPTION BUTTONS ENHACEMENTS

Esta opción añade nuevos botones a las ventanas de Delphi que son del mismo tipo que el inspector de objetos:



Estas son sus opciones:



En el apartado **Buttons** podemos visualizar los botones:

Stay of Top: Hace que la ventana siempre esté delante de las demás.

Roll: Encoge la ventana para dejar sólo la barra de título:

Options: llama a esta ventana de opciones.

Luego están las opciones:

Ignore Caption Text Width: Independientemente de la longitud del título de la ventana siempre mostrará los botones:

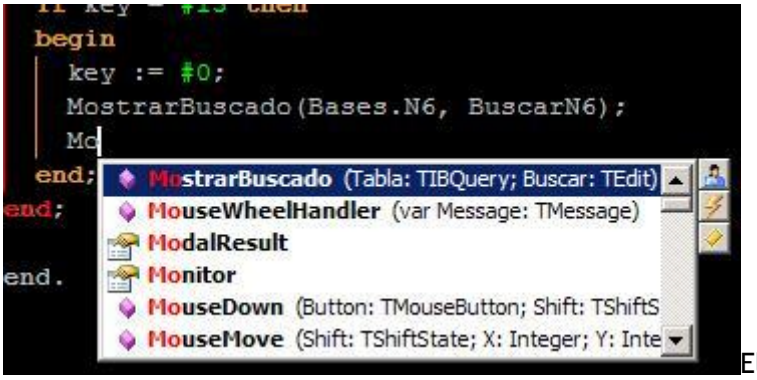
Si quitamos esta opción irá quitando botones intentando mantener siempre visible el título de la ventana:

Animate Rolling: hace que la opción de encoger el formulario verticalmente lo haga con una animación tipo persiana.

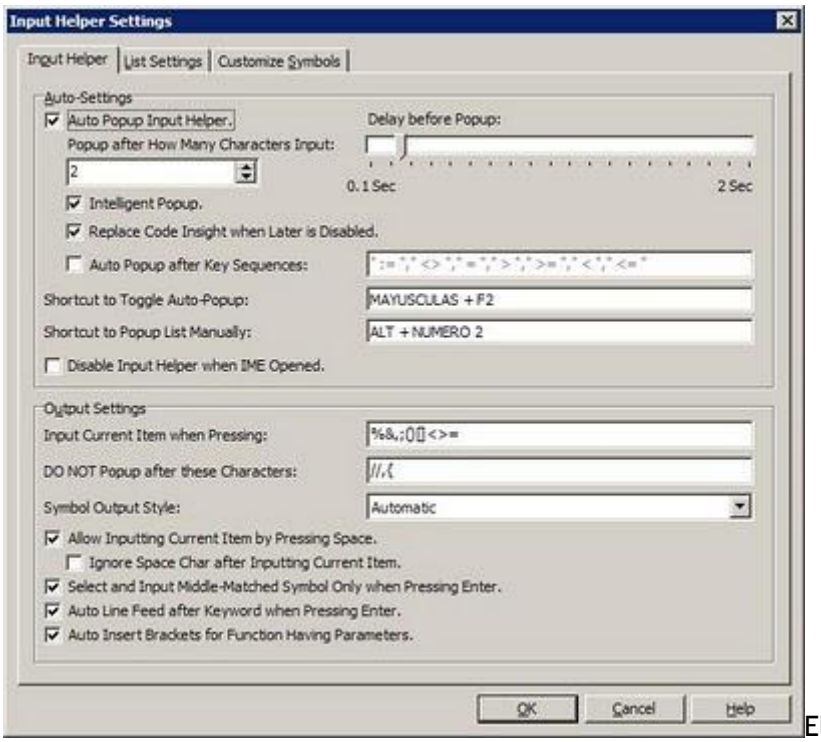
En el apartado **Filter Settings** podemos filtrar para que estos botones especiales no aparezcan en ciertas ventanas de Delphi como el formulario principal o el editor de código. Si desactivásemos esta opción aparecería esto:

CODE INPUT HELPER

Aquí tenemos todas las opciones relacionadas con el asistente que nos aparece al escribir el código fuente que viene a reemplazar a Code Insight de Delphi:



El asistente que trae por defecto Delphi sólo nos ayuda cuando escribimos un punto después del nombre de un objeto o registro, aunque también podemos pulsar la combinación de teclas CTRL + ESPACIO para que nos de sugerencias en cualquier momento. Pero con **cnPack**, podemos hacer que las sugerencias de código sean automáticas mediante estas opciones:



El apartado **Auto-Settings** contiene las opciones más importantes:

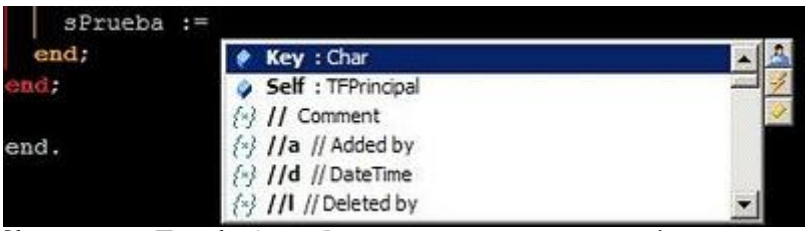
Auto Popup Input Helper: Al activar esta opción nos aparecerán todas las sugerencias de funciones, procedimientos u objetos a los pocos milisegundos de comenzar a escribir. Es importante que esta opción este activada para que funcione todo lo demás.

Popup alter How Many Characters Input: en este campo especificamos el número de caracteres que debemos escribir sin pausa para que se active el asistente de sugerencias de código.

Intelligent Popup: Si esta activada esta opción, conforme vayamos escribiendo un comando, si resulta que es el único que está en la lista desaparecerá el asistente. Si no es así, entonces seguirá activa la lista mostrando en rojo el candidato.

Replace Code Insight when Later is Disabled: Esta opción deshabilita el asistente Code Insight que trae el propio Dephi para que actúe **cnPack** en libertad.

Auto Popup alter key Sequences: permite activar el asistente después de escribir cualquier carácter de comparación (>,:=, <, etc.):



Shortcut to Toggle Auto-Popup: permite activar o desactivar este asistente con la combinación de teclas SHIFT + F2 a menos que elijamos otra.

Shortcut to Popup List Manually: Permite asociar una combinación de teclas para llamar al asistente sin tener que esperar o por si se ha cerrado porque hemos pulsado la tecla ESCAPE o hemos movido el puntero del ratón a otro lugar.

Disable Input Helper when IME Oponed: Deshabilita el asistente cuando está abierto el IME (Input Method Editor - Editor de entrada de métodos).

Pasamos al apartado **Output-Settings** con estas opciones:

Input Current Item when Pressing: Cuando tenemos abierta la lista de sugerencias a escribir, si tecleamos los siguientes caracteres las introducirá en el código fuente además del carácter tecleado: %&,;()[]<>=.

DO NOT Popup alter these characters: Al contrario de la opción anterior, aquí podemos elegir con que caracteres no queremos que se copie de la lista al código fuente.

Symbol Output Style: Esta opción determina como queremos que realice la búsqueda de sugerencias cuando estamos con el cursor encima de un indentificador.

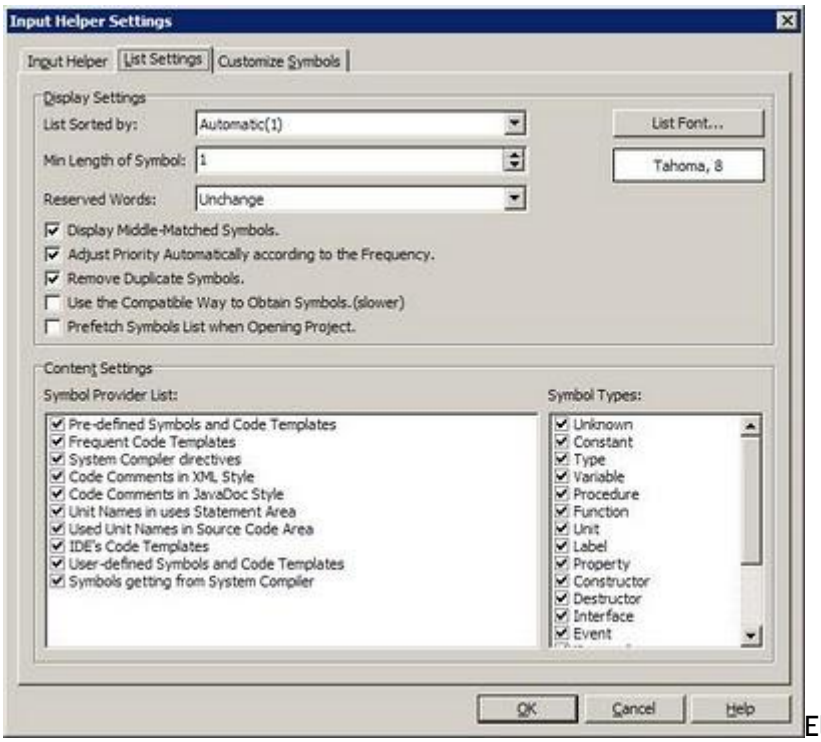
Allow Inputting Current Item by Pressing Space: Con esta opción podemos hacer que al estar la lista abierta, cuando pulsamos la barra de espacio la cierra e inserta lo que hemos elegido. Si además activamos la opción **Ignore Space Char after Inputting Current Item** hacemos que después de introducir en el código el elemento elegido no haga un espacio a continuación.

Select and Input Middle-Matched Symbol Only when Pressing Enter: Si el elemento actual coindide con el que estamos escribiendo, permitirle reemplazarlo por el de la lista al pulsar la tecla Intro.

Auto Line Feed after Keyword when Pressing Enter: Si hemos elegido un elemento de la lista pulsando la tecla Intro, insertamos una nueva línea debajo de la misma.

Auto Insert Brackets for Function Having Parameters: Si el elemento de la lista que hemos elegido es un procedimiento o una función, añade los paréntesis automáticamente.

Vamos con la segunda pestaña llamada **Line Settings**, que es la encargada de configurar como se muestra la lista de sugerencias:



El apartado **Display Settings** tiene estas opciones:

List sorted by: elegimos el criterio por el que vamos a ordenar los elementos de la lista.

Min Length of Symbol: Elegimos la longitud mínima (por defecto 1) de los elementos que queremos que aparezcan.

Reserved Words: Seleccionamos la capitalización (pasar a mayúsculas la primera letra) de las palabras reservadas: minúsculas, mayúsculas o sólo la primera letra.

Display Middled-Matched Symbols: Mostrar los símbolos asociados al tipo de elemento de la lista. Los símbolos son plantillas de código que permiten acelerar la inserción de código. Esta por ejemplo el símbolo **beg** que equivale a**begin .. end**.

Adjust Priority Automatically according to the Frequency: Se mostrarán los elementos más utilizados al principio de la lista.

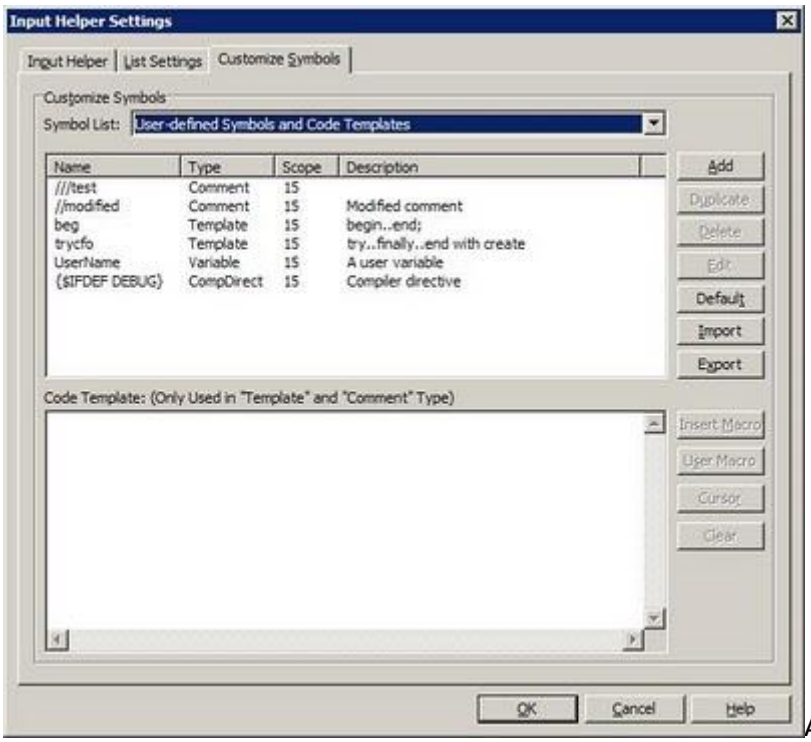
Remove Duplicate Symbols: Se eliminaran de la lista los elementos con el mismo nombre.

Use the Compatible Way to Obtain Symbols: Si el asistente crea problemas, cuelgues o incompatibilidades con Delphi, debemos activar esta opción.

Prefetch Symbols List when Opening Project: Si activamos esta opción hacemos que el asistente precargue todo lo que pueda para que luego su ejecución sea más rápida. Aunque es recomendable tener buen procesador y memoria RAM.

El apartado **Content Settings** permite seleccionar los tipos de objeto que queremos ver al abrir la lista de sugerencias. Por defecto están activados todos.

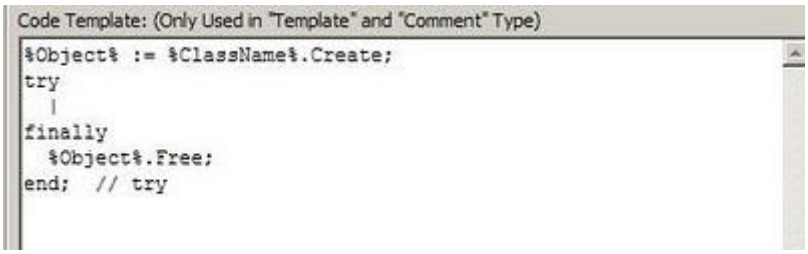
Y la última pestaña de este formulario es **Customize Symbols** y nos va a permitir personalizar la lista de sugerencias:



Aquí podemos pulsar el botón **Add** para añadir nuevos símbolos (plantillas) a las que tiene. Por ejemplo, si escribimos **beg** nos aparecerá este asistente:

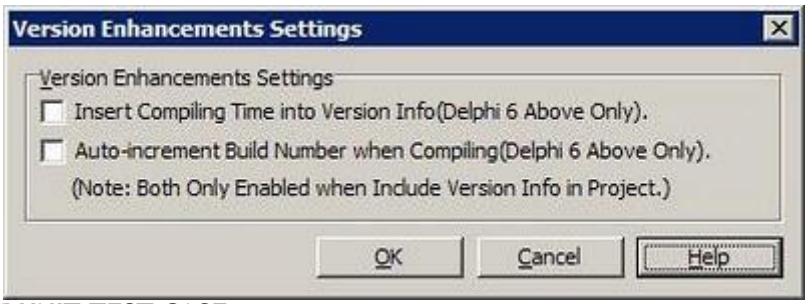
Al pulsar Intro nos introducirá automáticamente las sentencias **begin** y **end**:

Al seleccionar una plantilla de la lista podemos ver como se crea en la parte inferior de la ventana:



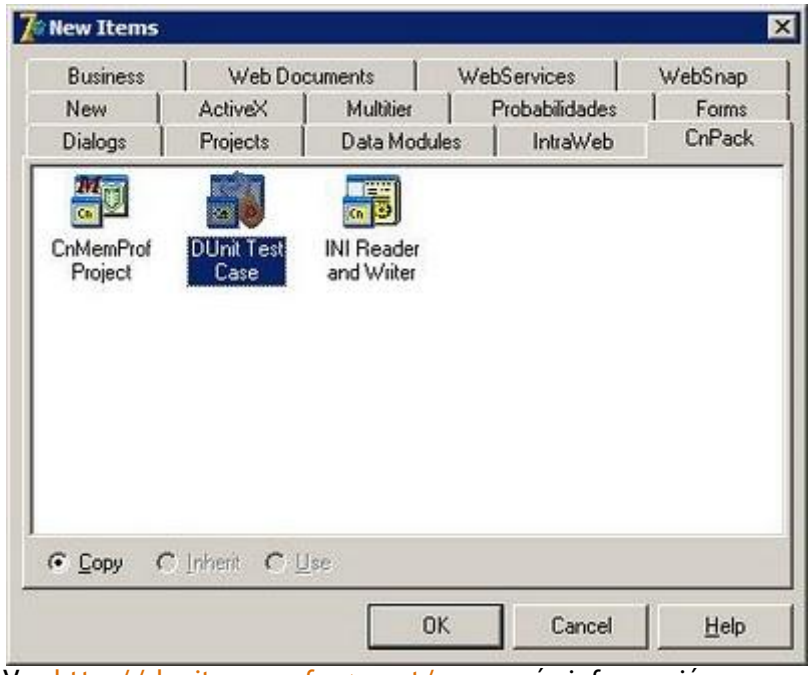
VERSION ENHACEMENTS

Estas opciones proporcionan mejoras para versiones de Delphi 6 o inferiores en lo que respecta a la compilación:



DUNIT TEST CASE

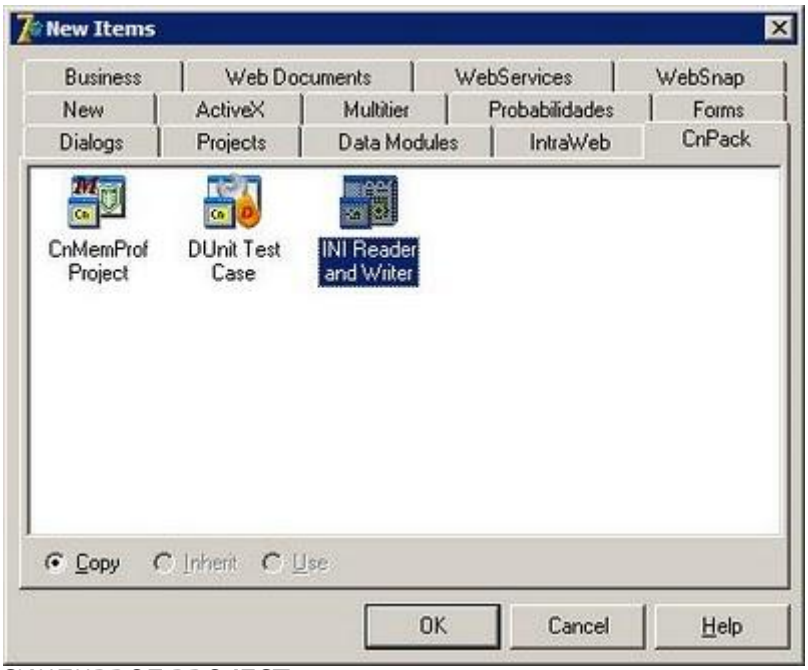
Permite crear nuevas unidades para realizar pruebas unitarias utilizando **DUnit**. Se crean seleccionando **File** -> **New**-> **Other**:



Ver http://dunit.sourceforge.net/ para más información aunque las últimas versiones de Delphi ya traen pruebas unitarias.

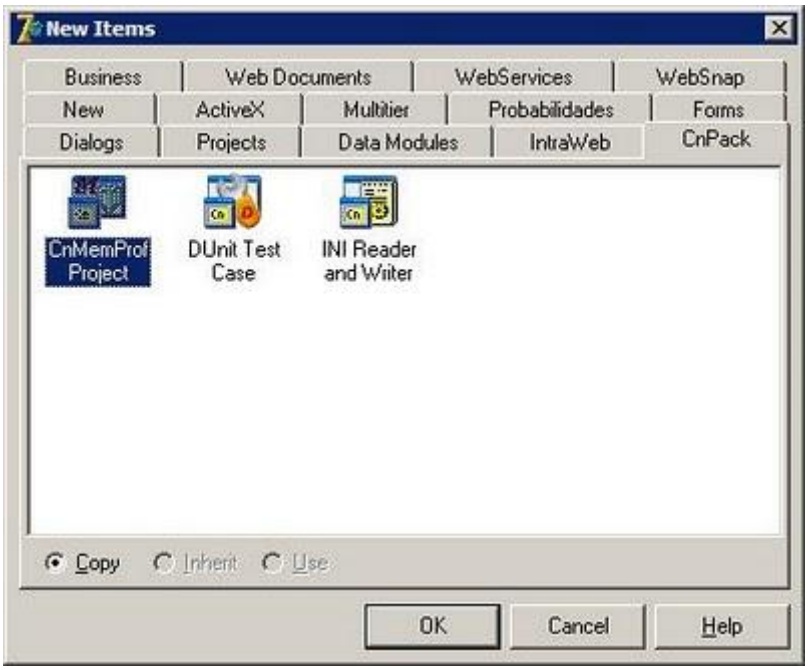
INI READER AND WRITER

Permite generar nuevas unidades para la escritura y lectura archivos INI cuando seleccionamos **File** -> **New** ->**Other**:



CNMEMPROF PROJECT

Activa la posibilidad de crear nuevos proyectos **CnMemProf**:



CONCLUSIONES

Seguro que me he dejado en el tintero alguna característica más que viene incluida en cnPack pero como habéis visto, este experto es gigante y no me da tiempo para más, ya que tengo ganas de abarcar nuevos temas. El único inconveniente que le he encontrado a este experto es que en máquinas con 1 GB de RAM o menos va bastante lento, sobre todo con proyectos que tienen cientos de formularios (también depende de la versión de Delphi). De todas formas, lo considero casi imprescindible para aumentar la productividad programando.

Pruebas realizadas en Delphi 7 y RAD Studio 2007.

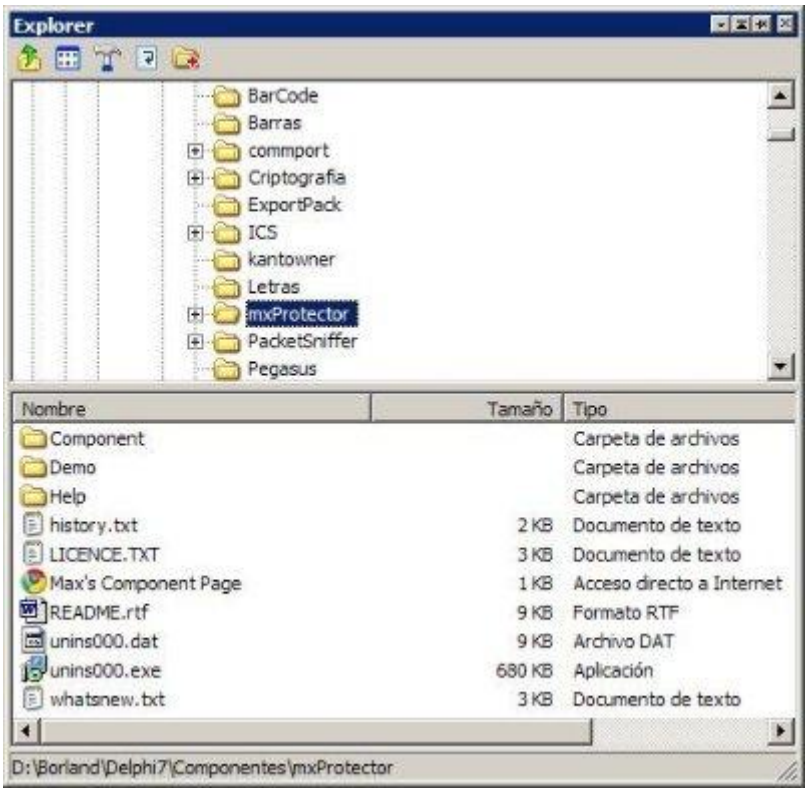
11 diciembre 2009

El experto cnPack (7)

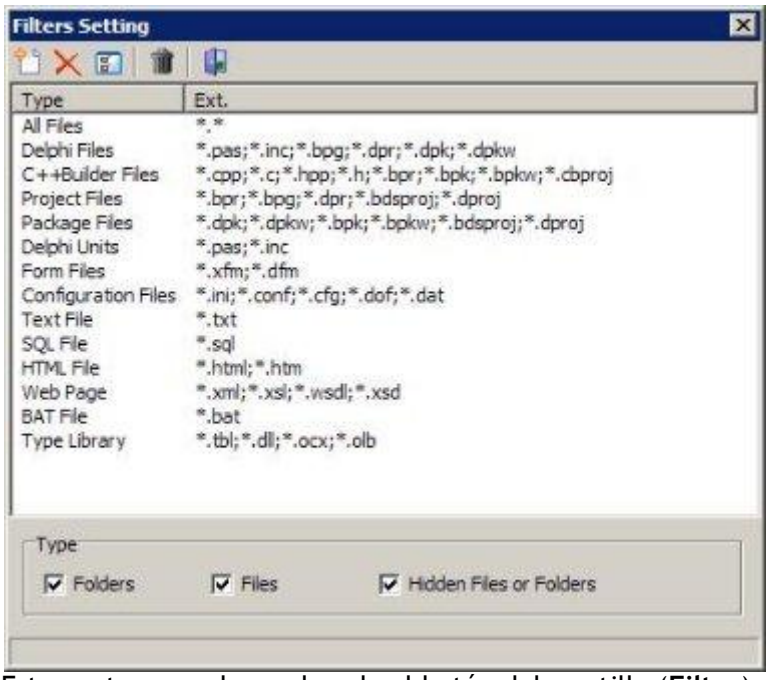
Sigamos viendo todas las posibilidades que nos ofrecen las opciones del experto **cnPack**.

EXPLORER WIZARD

Estas opciones permiten configurar el pequeño explorador de Windows que tiene **cnPack** y que podemos abrir seleccionando en el menú superior **cnPack** -> **Explorer…**



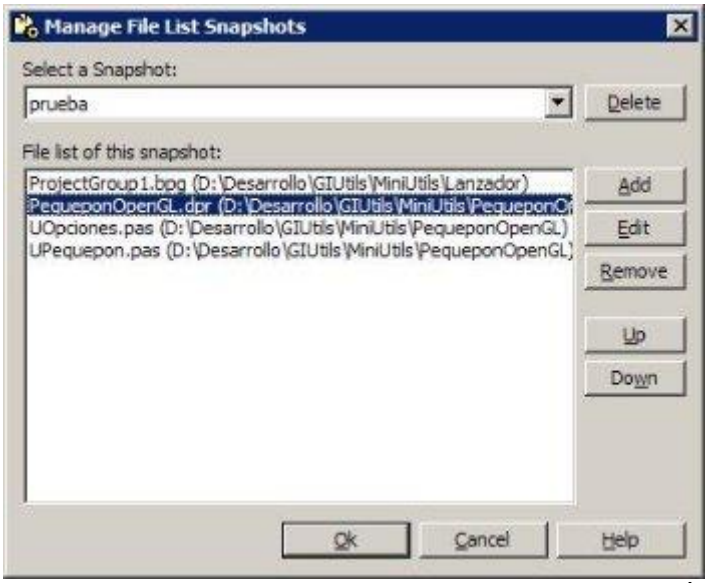
Como explorador, hay que reconocer que es bastante rudimentario, aunque es muy rápido a la hora de recorrer las carpetas del disco duro. Aunque al contrario de un explorador normal, en este explorador los archivos a buscar están filtrados según estas extensiones:



Esta ventana se abre pulsando el botón del martillo (**Filter**) y seleccionando la opción **Customize Filter**. Dentro del apartado de opciones de **Explorer Wizard** lo único que podemos configurar es una combinación de teclas para abrirlo rápidamente.

HISTORIAL FILES SNAPSHOT

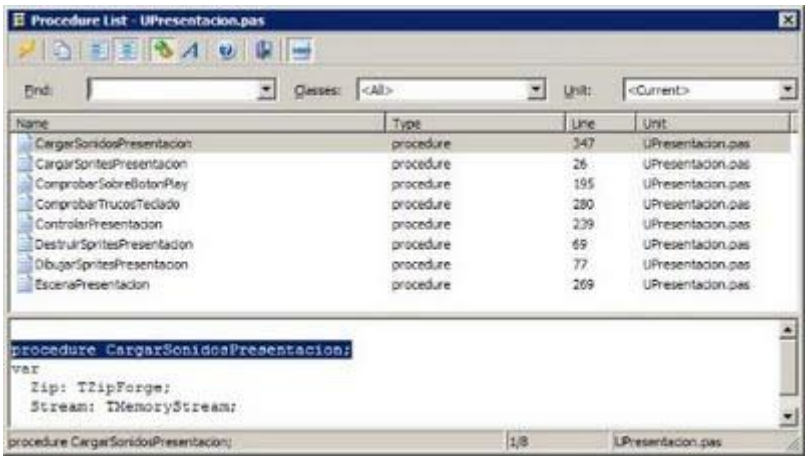
Aparte del histórico de archivos que puedan tener las versiones superiores a Delphi 7, **cnPack** también conserva un histórico de los archivos que hemos manejado en el IDE. Este histórico se ve pulsando el botón **Settings** de este apartado:



Podemos seleccionar todos los archivos y pulsar el botón **Remove** para limpiar el histórico.

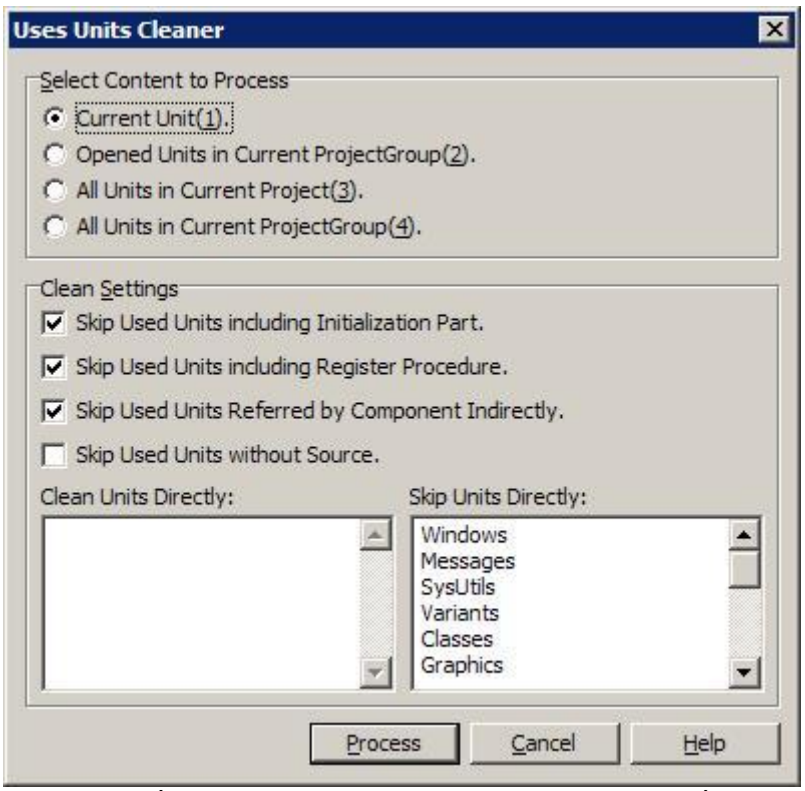
PROCEDURE LIST WIZARD

Mediante esta opción solo podemos asignar una tecla rápida para acceder a la ventana de búsqueda de procedimientos que vimos anteriormente:



USES UNIT CLEANER

El limpiador de unidades está dentro de la opción **cnPack** -> **Uses Cleaner** y como vimos anteriormente se utilizaba para eliminar del apartado **uses** aquellas unidades que ya no se utilizan, sobre todo, componentes insertados con anterioridad que han sido reemplazados por otros:



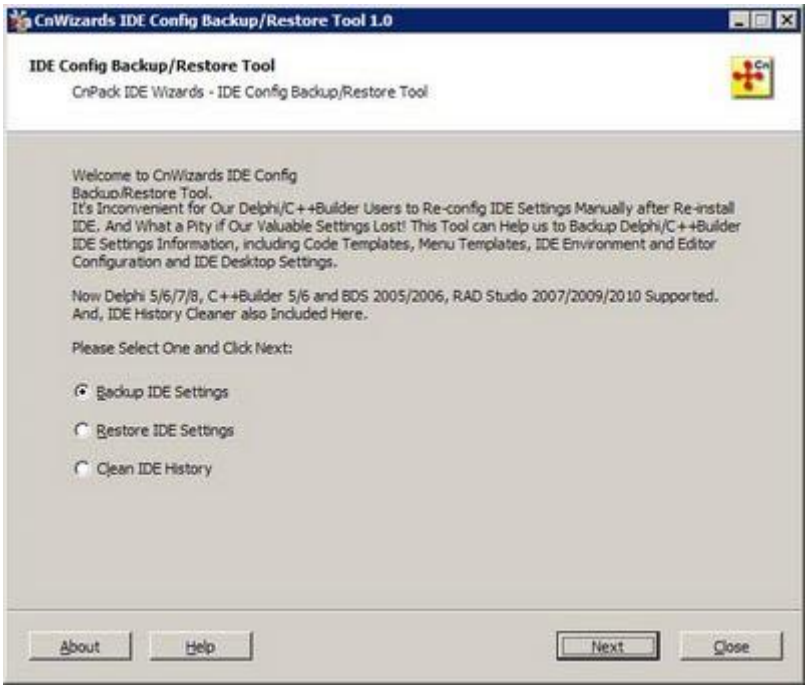
En esta opción tan solo podemos asignar una combinación de teclas.

IDE ENHACEMENTS SETTINGS

Activa las opciones avanzadas dentro del IDE. Lo he desactivado y realmente no sé lo que quita. En la ayuda tampoco viene nada sobre esta opción.

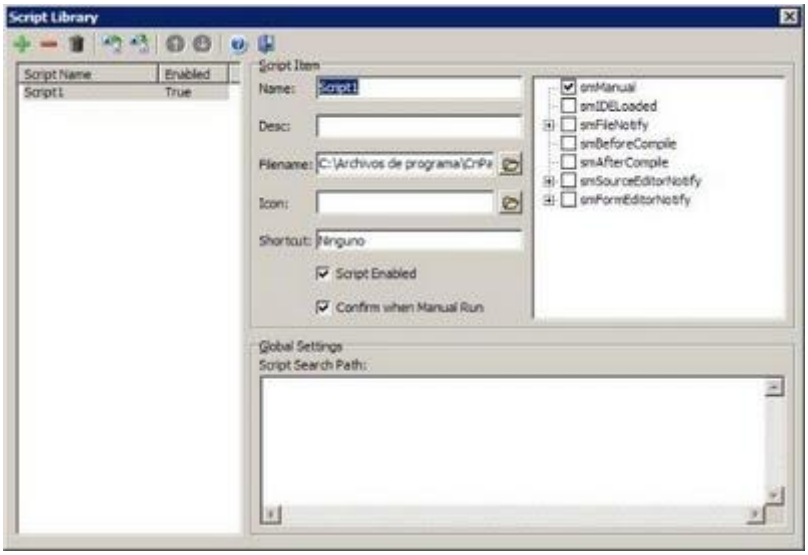
IDE CONFIG BACKUP/RESTORE BACKUP TOOL

Si asignamos una tecla rápida a esta opción podemos llamar rápidamente al asistente de creación de copias de seguridad:



SCRIPT WIZARD

En este apartado configuramos mediante el botón **Settings** las opciones de la librería de script:



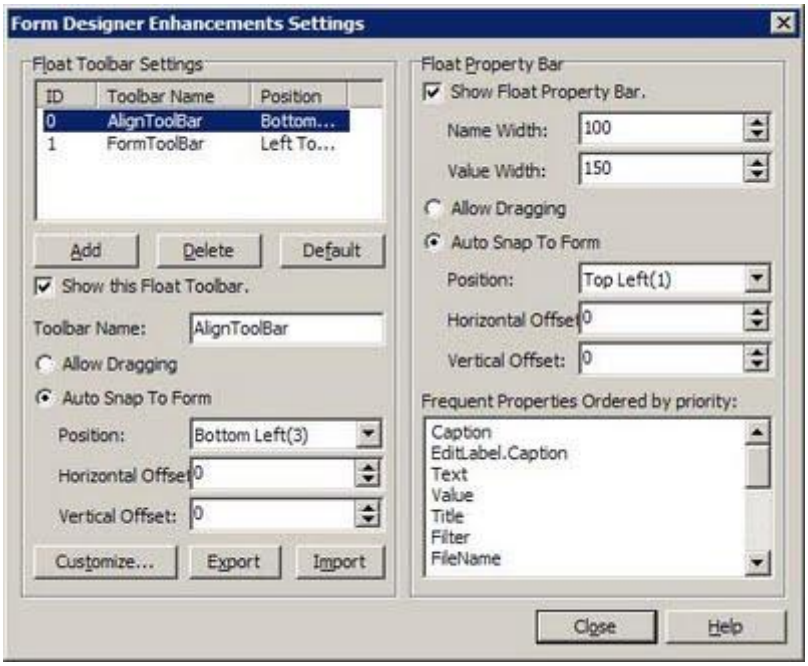
FAST CODE IDE OPTIMIZER

Si activamos esta opción entonces optimizará el programa que compilamos para que utilice una nueva versión del motor de memoria añadiendo además la posibilidad de utilizar las nuevas instrucciones de los procesadores: MMX, SSE, SSE2, etc. Aunque según sus autores, esta característica no funciona en la versión de Delphi 5.

FORM DESIGNER ENHACEMENTS

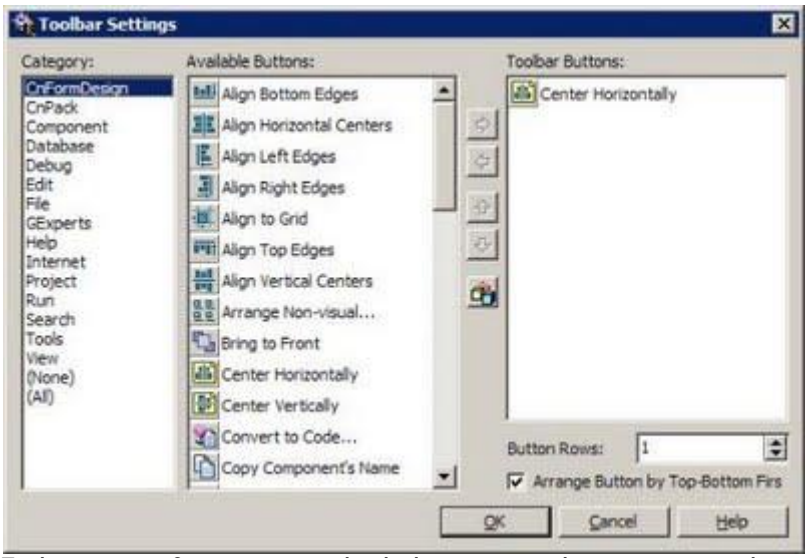
Esta opción activa o desactiva la barra lateral de botones que vemos en Delphi 7 a la izquierda del formulario:

Al entrar en su apartado de opciones podemos configurar todo esto:

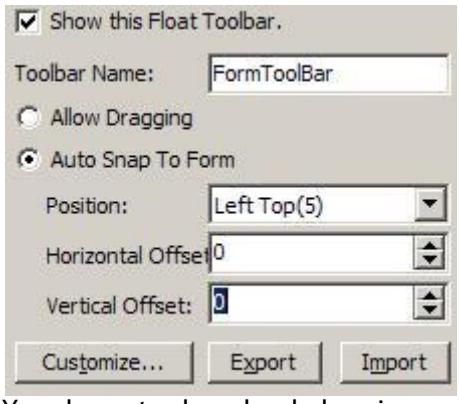


Estos son sus apartados:

Float Toolbar Settings: Mediante esta opción podemos añadir o quitar barras laterales al lado del formulario. Al pulsar el botón **Add** podemos crear una nueva barra de herramientas seleccionando todos los botones que queramos:



En la parte inferior izquierda de la ventana de opciones podemos hacer que la barra que hemos seleccionado en la lista de arriba sea visible o invisible además de especificar su posición y su separación respecto al formulario:

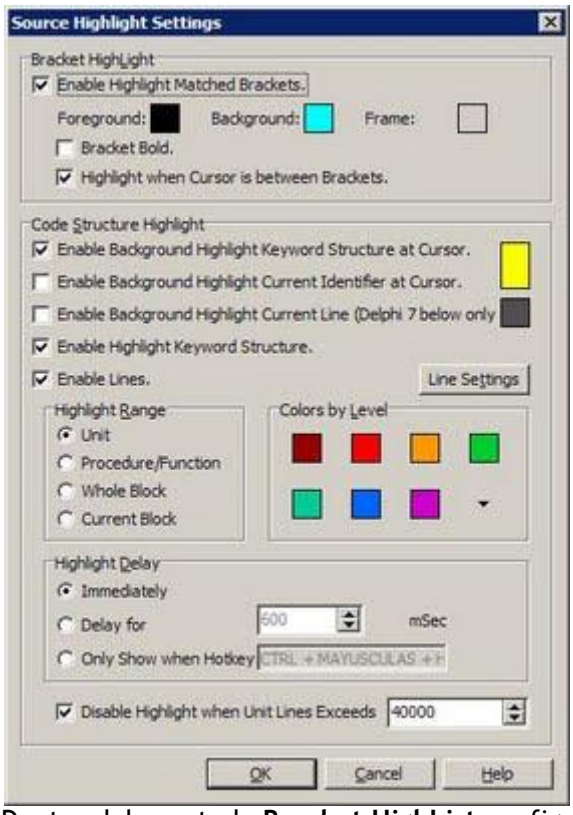


Y en la parte derecha de la misma ventana (**Float Property Bar**) elegimos las propiedades de la barra flotante que aparece en la parte superior del formulario:

Y al igual que con las barras de herramientas anteriores podemos elegir su posición, separación, etc.

SOURCE HIGHLIGHT ENHACEMENTS

Esta sección es la encargada de configurar el modo de iluminar con colores el código fuente del editor:



Dentro del apartado **Bracket HighList** configuramos la iluminación de los paréntesis:

Enable Highlight Matched Brackets: Si esta opción está activada se iluminarán los paréntesis o corchetes cuando estemos encima de ellos:

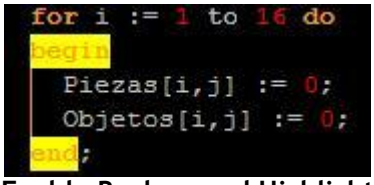
También se puede elegir el color de iluminación y el color de fondo.

Bracket Bold: Utilizará la fuente en negrita al dibujar el paréntesis.

Hightlight when Cursor is Between Brackets: Se iluminarán los paréntesis cuando el cursor está dentro de los mismos, aunque no esté encima del paréntesis de apertura o de cierre.

Dentro del apartado **Code Structure Highlight** configuramos la iluminación de las estructuras de código con estas opciones:

Enable Background Highlight Keyword Structure at Cursor: esta opción activa la iluminación de las palabras reservadas que engloban un bloque de código al situarnos en la misma línea:



Enable Background Highlight Current Identifier at Cursor: si activamos esta opción entonces también iluminará cualquier identificador (variables, procedimientos, etc.):

```
begin
  ETituloPantalla.Caption := '
  BBorrarTodoClick(Self);
  sArchivoPantalla := 'Nueva p
end;
```

Enable Background Highlight Current Line: ilumina toda la línea donde estamos situados (sólo en Delphi 7 porque en las versiones superiores ya lo hace el propio IDE):

```
  end;

  Result := Copy(sRuta,1,i)+'\';
end;
```

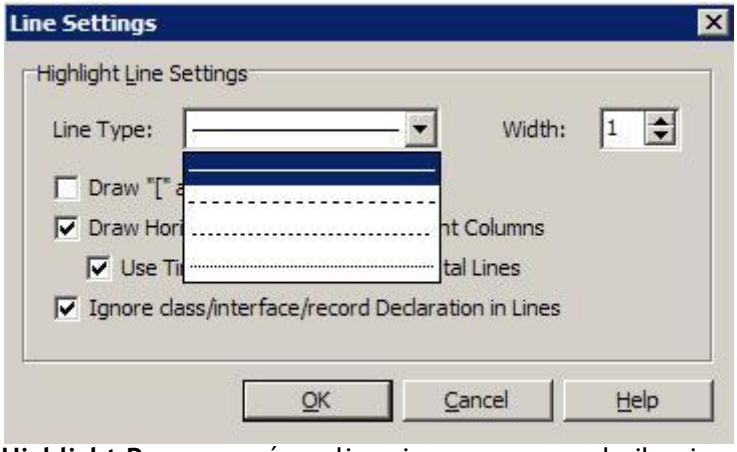
Enable Highlight Keyword Structure: ilumina de un color distinto a las palabras reservadas encargadas de las estructuras de datos. Por ejemplo, en la definición de una clase:

```
TVertice = class
  x, y, z: GLfloat;
  u, v: GLfloat;

  constructor Create;
  constructor Create(
end;
```

Enable Lines: las líneas verticales que dibuja permiten de un solo vistazo saber si la identación de código es correcta:

```
// ¿Tiene subsprites?
if bSubSprites then
begin
  Origen.x := iSubX*iAnchoSub;
  Origen.y := iSubY*iAltoSub;
  Origen.w := iAnchoSub;
  Origen.h := iAltoSub;
  Destino.x := x;
  Destino.y := y;
  Destino.w := iAnchoSub;
  Destino.h := iAltoSub;

  // Dibujamos el subsprite seleccion
  if SDL_BlitSurface( Superficie, @Or
  begin
    ShowMessage( 'Error al dibujar el
    bSalir := True;
    Exit;
  end;
end
```

A la derecha tenemos el botón **Line Settings** donde seleccionamos el tipo de líneas a dibujar:



Highlight Range: aquí se elige si queremos que la iluminación de colores sea en toda la unidad, en el procedimiento o función donde estamos situados, en el bloque de código actual, etc.

Colors by Level: se pueden elegir los colores en cada nivel de identación.

Highlight Delay: por defecto viene seleccionado para que se ilumine el código inmediatamente, pero también se puede configurar para que lo haga cada x milisegundos o si queremos hacerlo pulsando una combinación de teclas.

Disable Hightlight when Unit Lines exceeds: viene configurado para que si la unidad de código donde estamos situados supera las 40.000 líneas entonces desactive la iluminación para evitar que Delphi se ralentice (para máquinas antiguas con poca potencia).

En el próximo artículo terminaré de hablar de las opciones que nos quedan.

Pruebas realizadas en Delphi 7 y Delphi 2007.

Publicado por Administrador en 10:24 0 comentarios
Etiquetas: expertos

04 diciembre 2009

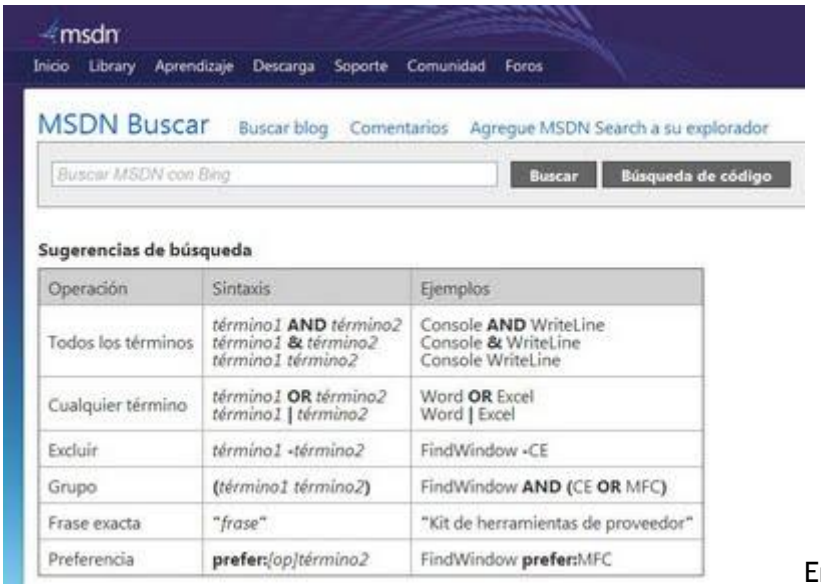
El experto cnPack (6)

Vamos a continuar viendo las opciones de **cnPack** comenzando con la ayuda que nos brinda Microsoft a través de su portal MSDN.

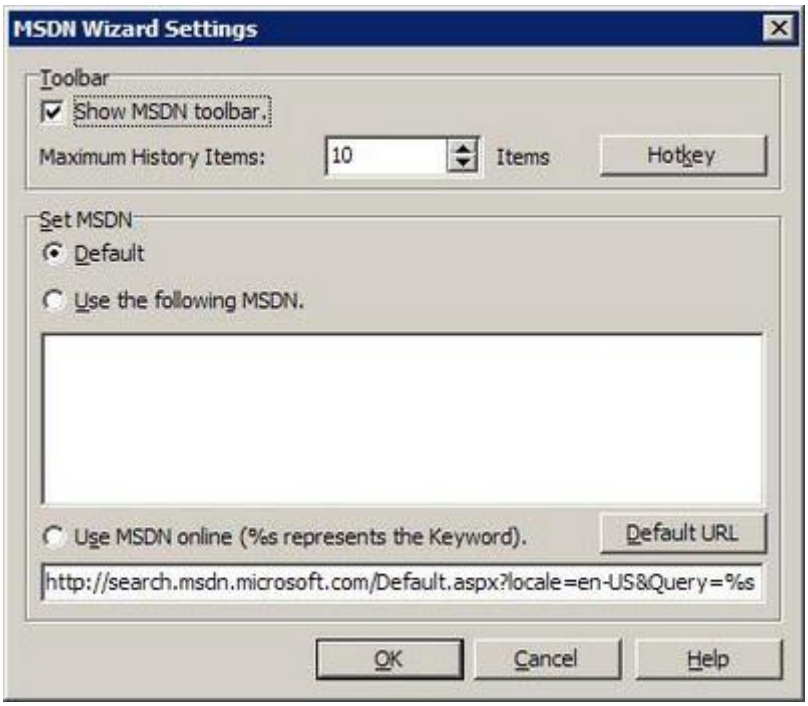
MSDN HELP WIZARD

Esta opción es por si queremos acceder a la ayuda de Microsoft por su portal MSDN. Esta pensada sobre todo para consultar las librerías estándar de la plataforma .NET y la API de Windows. Para consultar MSDN desde Delphi hay que seleccionar en el menú superior **cnPack** -> **MSDN Help Wizard**:

Según elijamos la opción **Help** o **Search** se irá a una página web u otra:

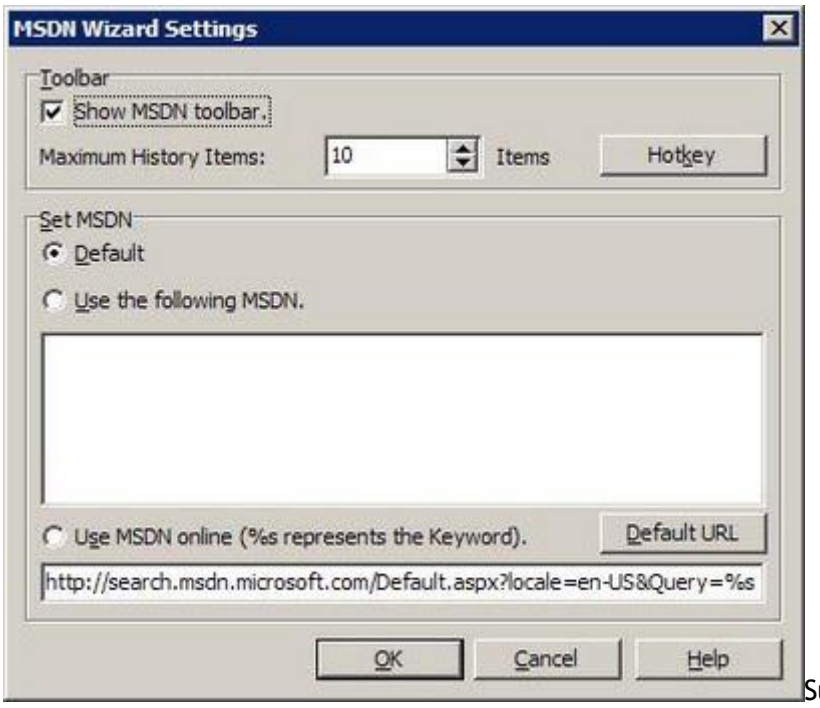


En su apartado de opciones podemos cambiar la página de inicio:

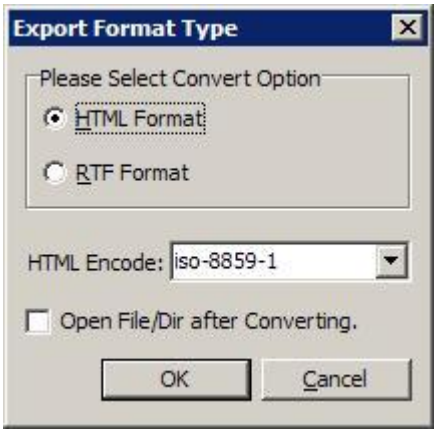


SOURCE FORMAT CONVERTED WIZARD

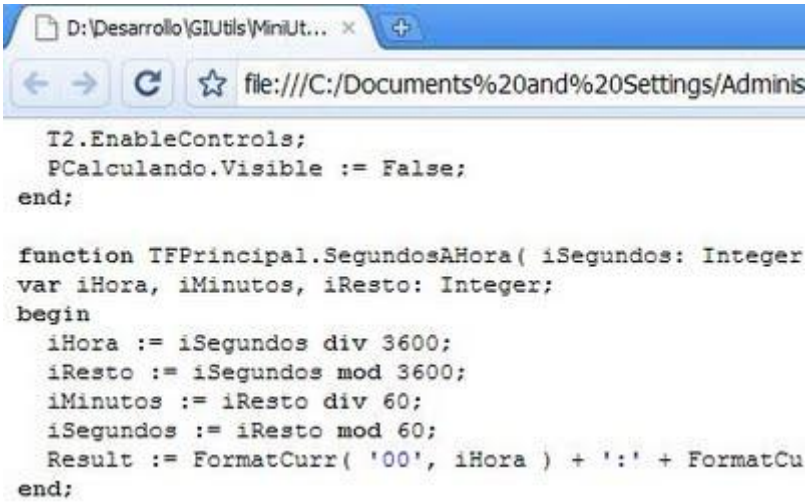
Estas opciones determinan como va a exportarse el código fuente a los formatos HTML o RTF. Su ventana de opciones nos permite asignar una combinación de teclas a cada tipo de exportación:



Supongamos por ejemplo que quiero exportar la unidad actual donde estoy a HTML. Entonces asigno la combinación de teclas **CTRL + H** y al pulsarla estando en el editor de código mostrará esta ventana:



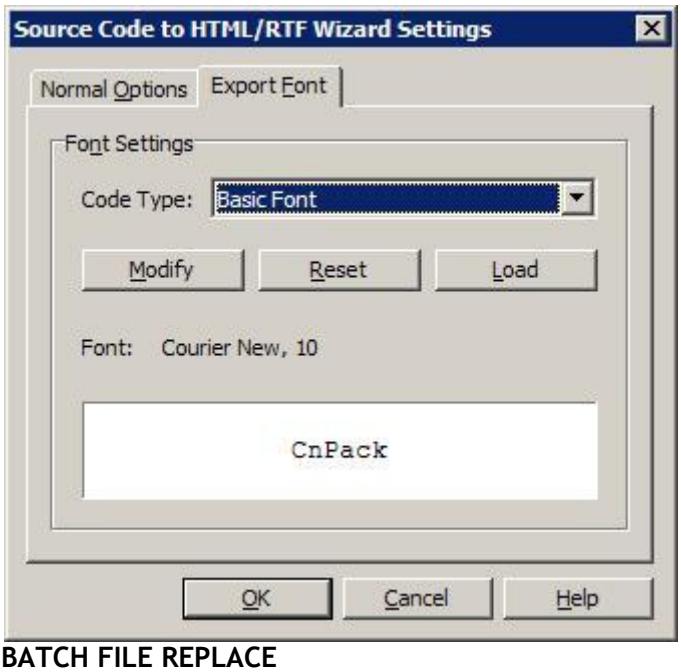
Entonces elegimos por ejemplo HTML y nos pedirá donde guardar la página web. Al abrirla veremos que la conversión ha sido perfecta:



Como podemos observar

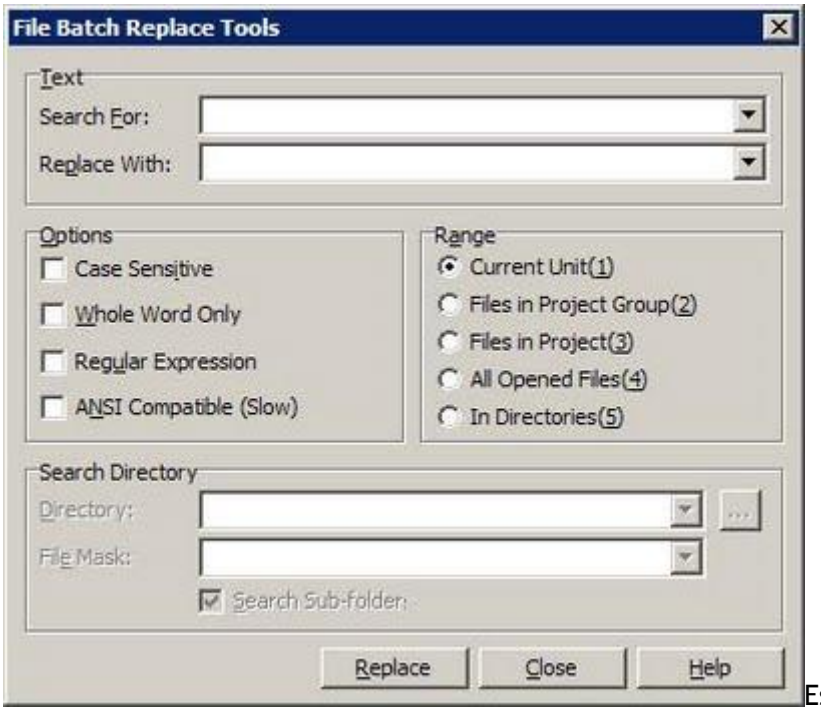
en la imagen, vemos que ha mantenido correctamente la identación de código. Lo mismo podemos hacer para exportanlo a RTF para crear documentación. Estas dos opciones son muy útiles para compartir código o crear manuales para el resto de usuarios de Internet.

En la segunda pestaña de opciones tenemos la posibilidad de cambiar la fuente para adaptarla a nuestras necesidades:



BATCH FILE REPLACE

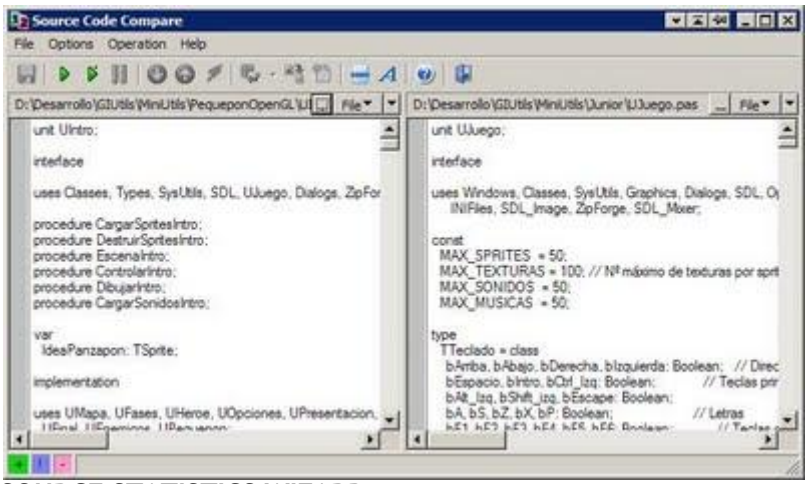
Esta opción pertenece a la búsqueda y sustitución de texto en todos los archivos del proyecto. Lo único que podemos hacer aquí es asignar una combinación de teclas a la búsqueda especial que tiene **cnPack** y que es muy parecida a la de Delphi:



Esta herramienta es muy parecida a la opción **Search** -> **Find in Files** que ya trae Delphi.

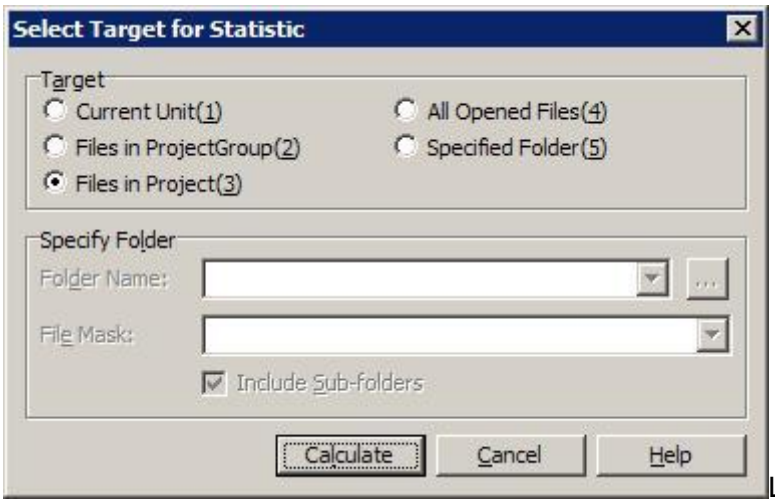
SOURCE COMPARE WIZARD

Aquí sólo podemos activar o desactivar la herramienta que nos permite comparar dos ficheros de código en busca de diferencias:

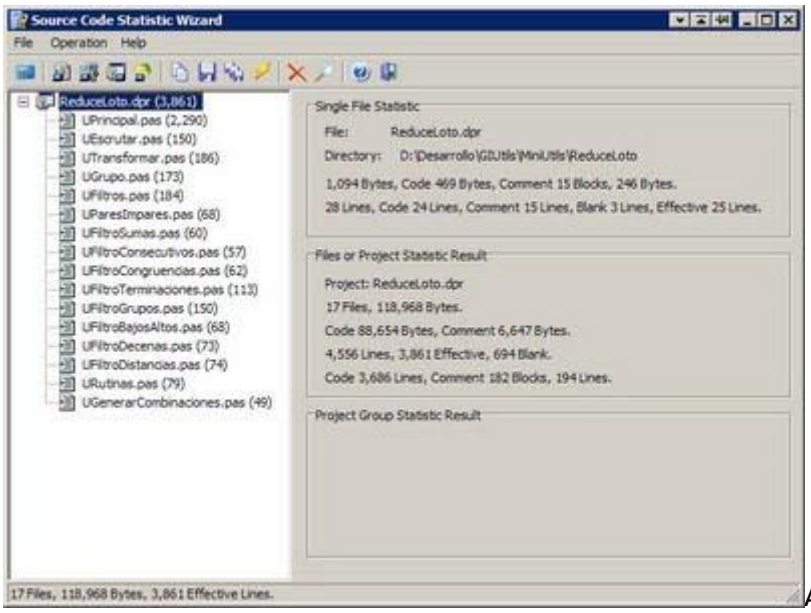


SOURCE STATISTICS WIZARD

cnPack incorpora una herramienta estadística que cuenta el número de líneas de código de todo el proyecto así como cada una de las unidades que incorpora. Esta información está disponible seleccionando **cnPack** -> **Source Statistics**. Aparecerá una ventana para elegir que es lo que queremos analizar:



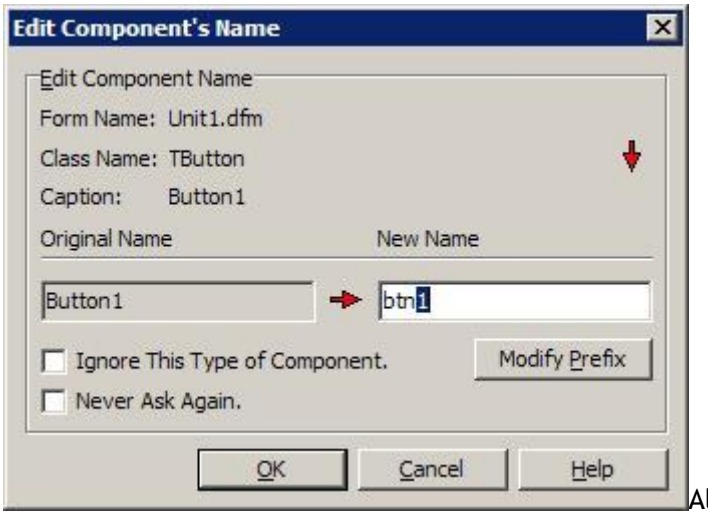
Lo mejor es seleccionar **Files in Project** para tener una visión global del proyecto:



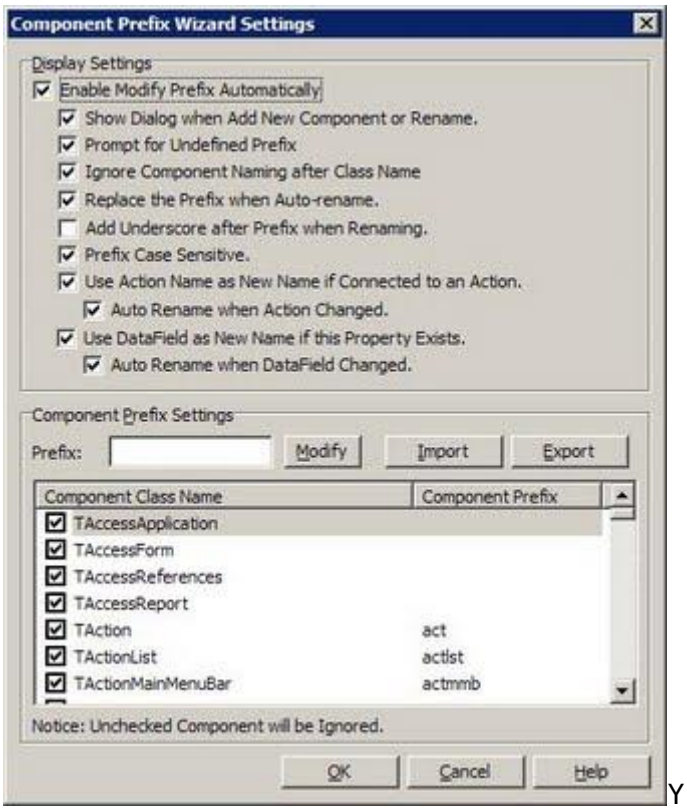
Además nos viene en la parte izquierda (entre paréntesis) el número de líneas global y el de cada unidad. En su apartado de opciones solo podemos activar o desactivar esta utilidad.

COMPONENT PREFIX WIZARD

Como vimos anteriormente, **cnPack** permite asignar un prefijo a cada componente que se inserta en su formulario según su clase:



Al pulsar el botón **Settings** de esta opción muestra todas estas posibilidades:

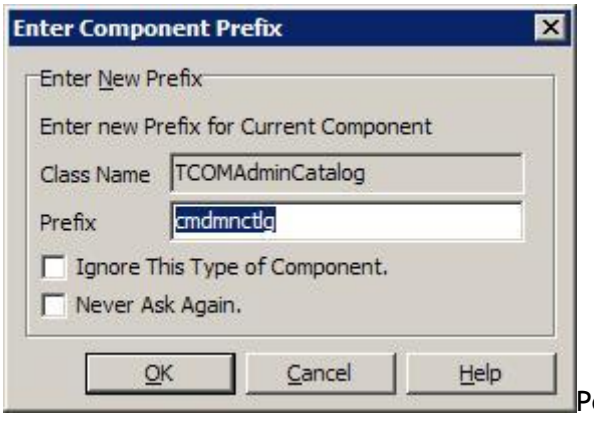


Y esto es lo que activan y desactivan:

Enable Modify Prefix Automatically: Esta es la opción general que activa o desactiva el asignar prefijos a los componentes.

Show Dialog when Add new Component o Rename: si desactivamos esta opción, no aparecerá la ventana de prefijos pero si lo asignará al componente insertado. Si por ejemplo insertamos tres botones a un formulario los llamará **btn1**, **btn2** y **btn3**.

Prompt for Undefined Prefix: Se desactivamos la opción anterior y encuentra un componente del que no sabe el prefijo entonces mostrará el cuadro de diálogo para que se lo asignemos:



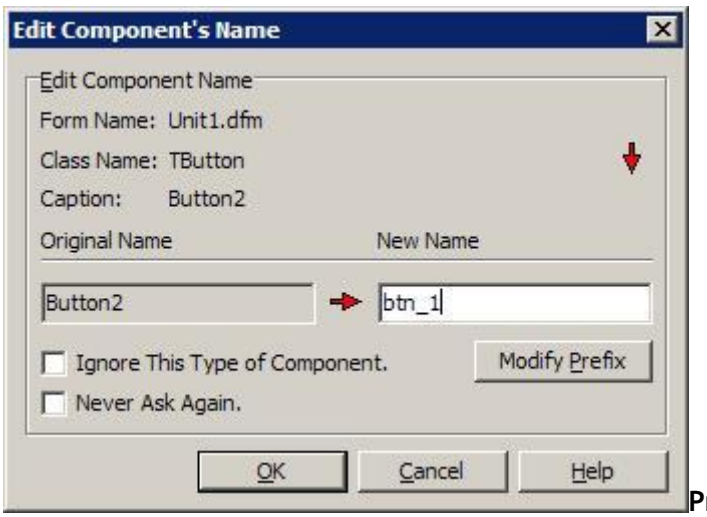
Pero si desactivamos ambas opciones entonces no volverá a preguntar el prefijo nunca.

Ignore Component Naming after Class Name: Ignora el nombre del componente para clases que no comienzan por T.

Replace the Prefix when Auto-Rename: Volver a reemplazar el prefijo al cambiarle el

nombre.

Add Underscore after Prefix when Renaming: Si activamos esta opción añadirá un guión bajo entre el prefijo y el nombre que le damos al componente:



Prefix Case sensitive: Si esta opción esta activada entonces nos obliga a que el prefijo tenga las minúsculas o mayúsculas como corresponde.

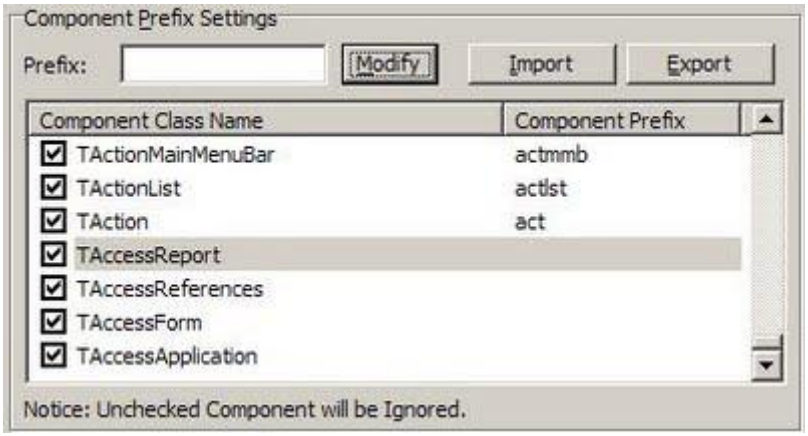
Use Action Name as New Name if Connected to an Action: Si el componente que estamos nombrando tiene asignada una acción (propiedad **Action** asignada a una acción del componente **ActionManager**) entonces el nombre del componente será el prefijo más el nombre de la acción asignada.

Auto Rename when Action Changed: Si además de la opción anterior esta activada esta opción, cuando renombremos el nombre de la acción se renombrará automáticamente el nombre de nuestro componente asignado a la misma. Aunque por las pruebas que he realizado, no se entera hasta que intentamos modificar la propiedad**Name** del componente asociado a la acción.

Use DataField as New Name if This Proterty Exists: Al insertar un componente asociado a un campo de una base de datos como por ejemplo el componente **TDBEdit** entonces al modificar su propiedad **DataField** le cambiará el nombre al componente con el prefijo de su clase mas el nombre del campo. De ese modo nos ahorramos de asociar el campo **APELLIDOS** a dicho componente y luego tener que ir a su propiedad **Name** y poner **dbedtAPELLIDOS**. Lo hace sólo.

Auto Rename when DataField Changed: Si además de la opción anterior está activada esta otra entonces cuando se modifique la propiedad **DataField** cambiará también el nombre del componente. Aunque me ha pasado igual que antes, no se entera hasta que intento cambiar la propiedad **Name**.

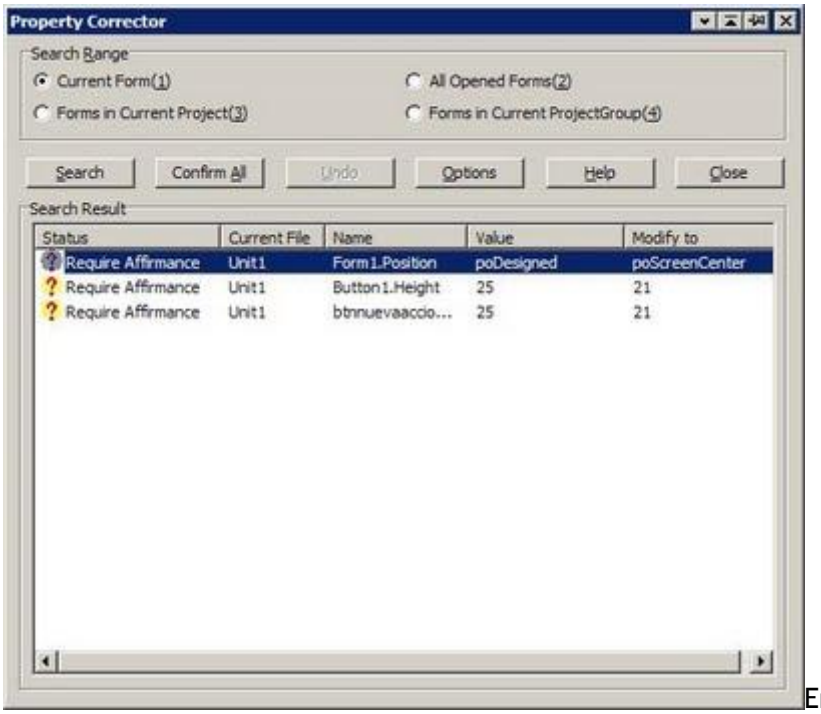
Luego tenemos en la parte inferior de la ventana todas las clases de los componentes de Delphi y su prefijo asignado. Podemos ver que hay muchos sin definir:



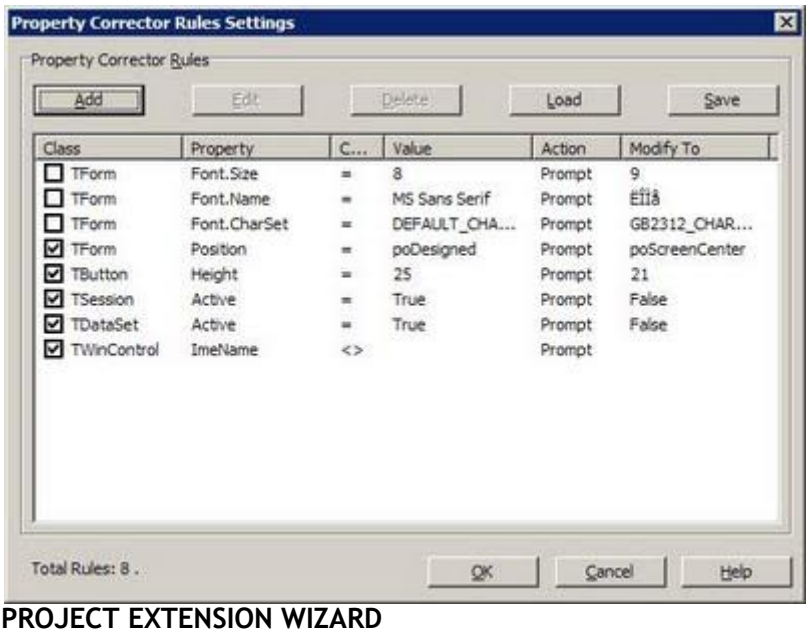
Si queremos asignar nuestros propios prefijos entonces tenemos que seleccionar una clase, escribir el nuevo prefijo en el campo **Prefix** y pulsar el botón **Modify**.

PROPERTY CORRECTOR

El corrector de propiedades lo vimos anteriormente cuando tratábamos de que todos los componentes visuales tuviesen el mismo tamaño según su clase. Este corrector también puede abrirse llamando al menú superior **cnPack** -> **Property Corrector**:

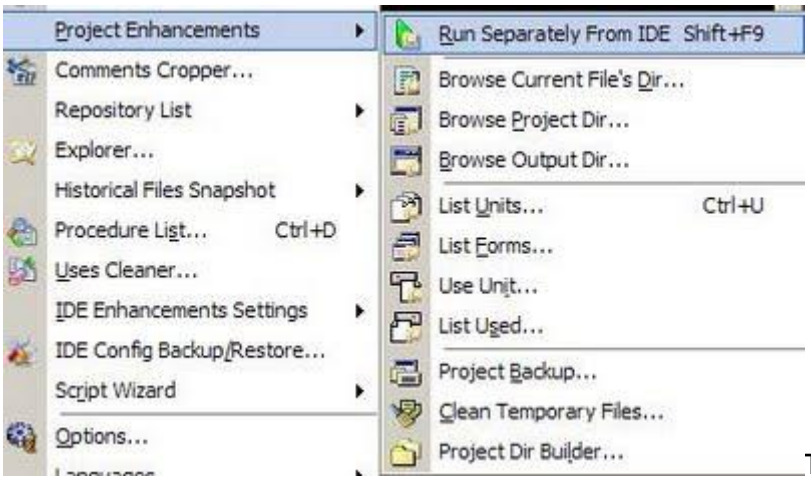


En su apartado de opciones podemos definir nuestras preferencias en ciertos componentes como puede ser la fuente, el tamaño, la posición, etc.:

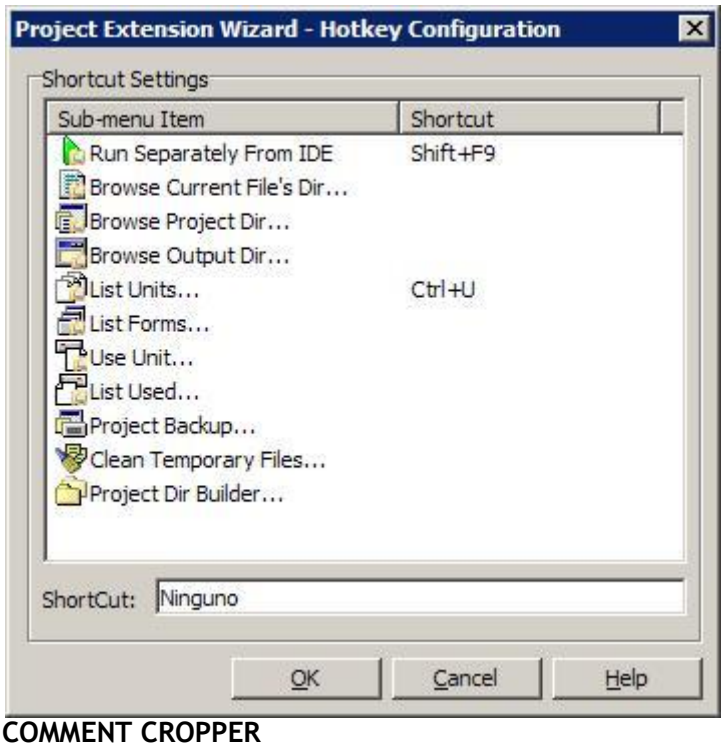


PROJECT EXTENSION WIZARD

Si seleccionamos la opción **cnPack** -> **Project Enhacements** podemos ver que aparece un nuevo menú para mejorar las opciones del proyecto con posibilidades como ejecutar el programa fuera de Delphi, ver los archivos del proyecto, etc:

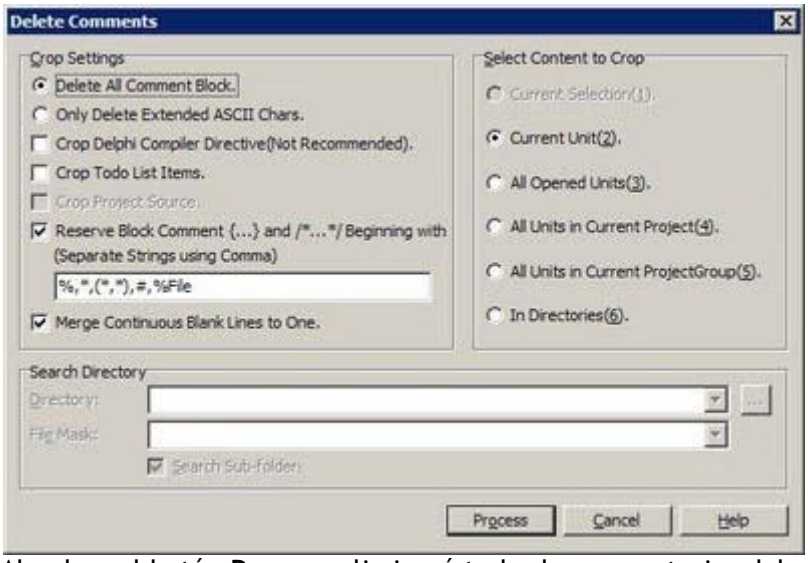


Todas estas herramientas ya las vimos anteriormente en la barra de botones. Pues el apartado **Settings** de esta opción permite asignar una combinación de teclas a todo este menú:



COMMENT CROPPER

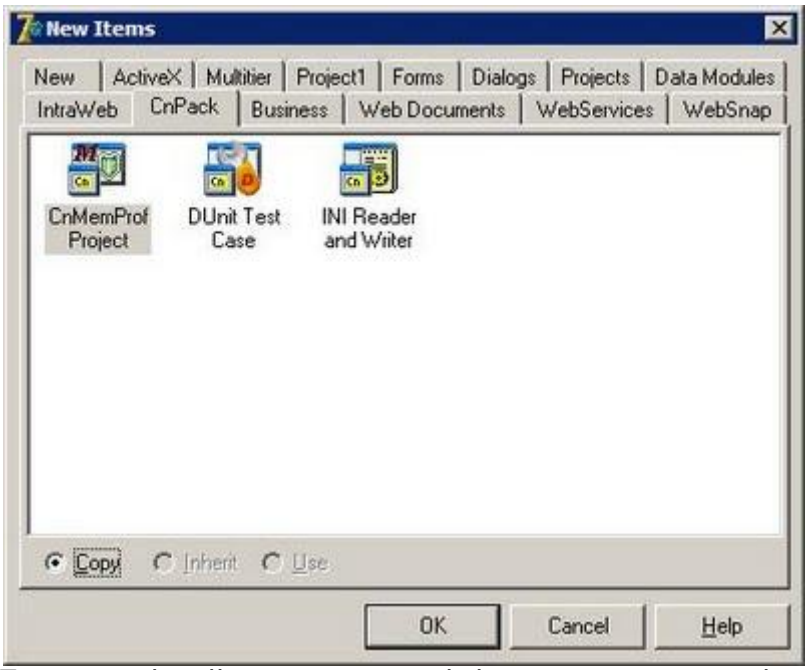
cnPack incorpora un asistente para eliminar comentarios tanto de la unidad actual como de todo el proyecto. Esta herramienta puede llamarse desde el menú **cnPack** - > **Comments Cropper**:



Al pulsar el botón **Process** eliminará todos los comentarios del código. En su apartado de opciones lo único que podemos hacer es asignarle una combinación de teclas o desactivarlo.

REPOSITORY LIST

En las opciones de Delphi para crear un nuevo proyecto u objeto, si seleccionamos **File** - > **New** - > **Other** podemos ver que cnPack a añadido su propia pestaña al repositorio:



Tiene tres plantillas para crear unidades para maninupar archivos INI o para crear test unitarios utilizando DUnit.

En el próximo artículo continuaré con las infinitas opciones de **cnPack**. Algún día se tienen que acabar.

Pruebas realizadas en Delphi 7 y Delphi 2007.

Publicado por Administrador en 18:42 0 comentarios
Etiquetas: expertos

27 noviembre 2009

El experto cnPack (5)

Después de ver toda la interfaz de trabajo que aporta el experto **cnPack** vamos a pasar a ver como podemos configurarlo mediante su botón de **Opciones**:

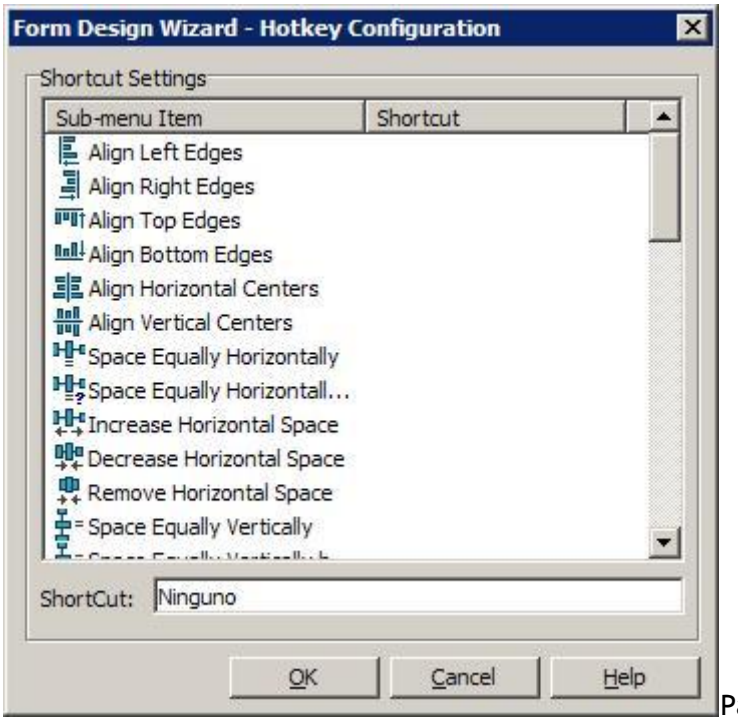

También podemos acceder a las mismas seleccionando en el menú superior **cnPack** -> **Options**, donde veremos esta ventana:

Vamos a ir viendo cada una de las opciones por el orden de la lista.

FORM DESIGN WIZARD

Mediante esta opción podemos asignar atajos de teclado a los botones de la barra de herramientas de **cnPack**encargados de la alineación y tamaño de los componentes seleccionados:


Al pulsar el botón **Settings** nos aparecerá la lista de botones con su tecla de acceso rápido a la derecha:

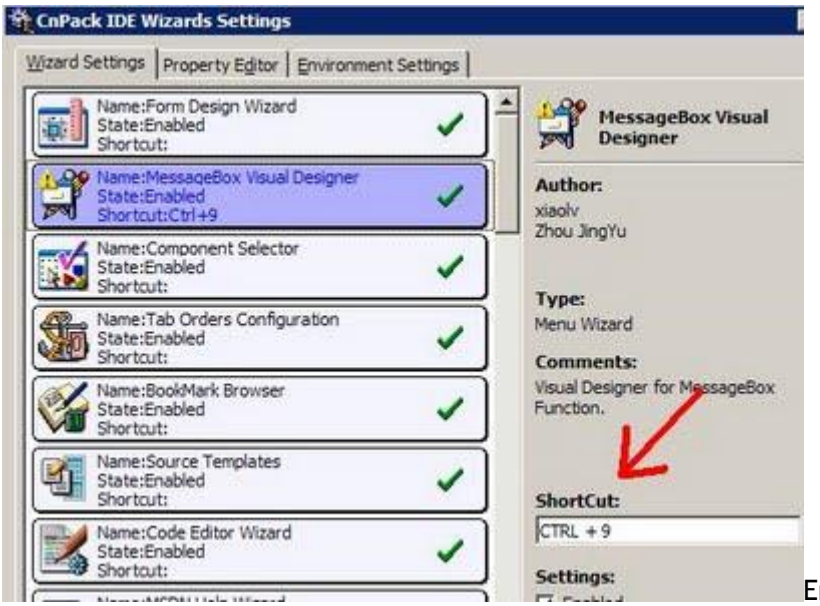


Para asignar un atajo de teclado

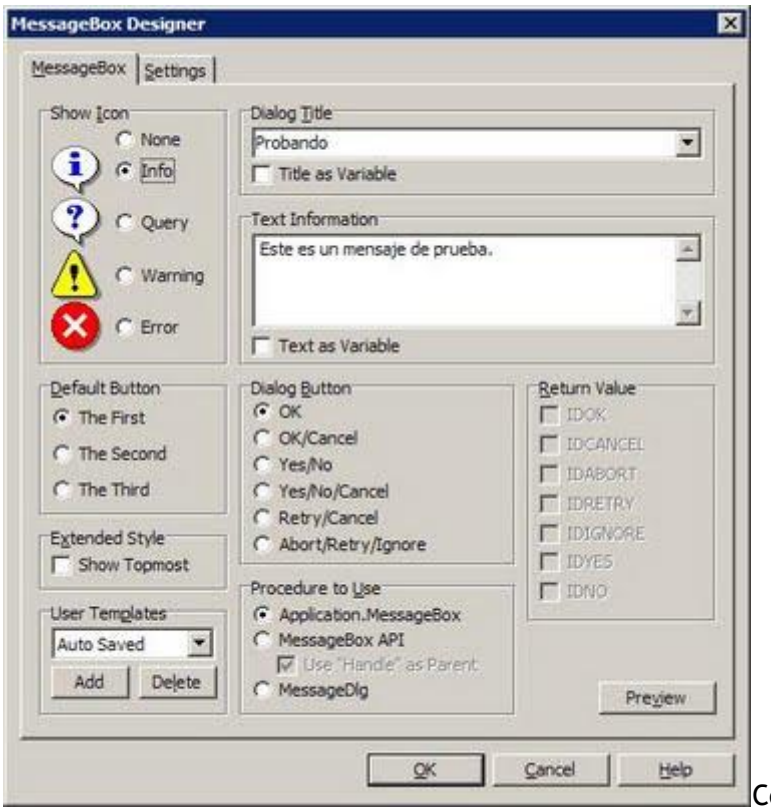
solo tenemos que seleccionar un botón de la lista, pinchamos con el ratón en el campo **Shortcut** y pulsamos aquella tecla que queremos asociar a dicho botón.

MESSAGE BOX VISUAL DESIGNER

Si estamos hartos de escribir la función de **Application.MessageBox** entonces podemos hacer que **cnPack** inserte el código de la misma automáticamente. Pero para que sea efectiva debemos asignar un atajo de teclado en el campo**Shortcut**, por ejemplo CTRL + 9:



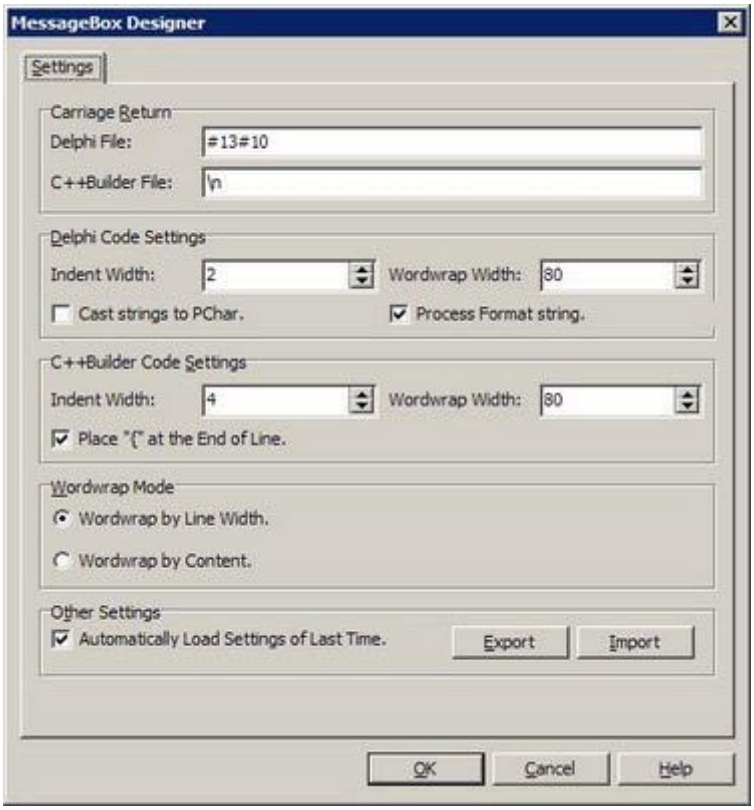
Entonces al pulsar dicha combinación de teclas estando en el editor de código aparecerá este asistente:



Como podéis ver la imagen, solo tenemos que teclear el título, el contenido del mensaje y configurar el tipo de mensaje (información, pregunta, advertencia, error, etc.). Nos insertará este código:

```
begin
  Application.MessageBox('Este es un mensaje de prueba.', 'Probando', MB_OK +
    MB_ICONINFORMATION);
end;
```

Si volvemos a las opciones y pulsamos el botón **Settings** veremos esta configuración:



Estos son sus apartados:

Carriage Return: podemos definir tanto para Object Pascal como para C++ los caracteres de retorno de carro.

Delphi Code Settings: En este apartado configuramos en ancho de la identación de código mediante el campo**Ident Witdh** y el ancho de la línea mediante **Wordwrap Width**. Esto afecta a la hora de crear el código de la función**MessageBox** así como la identación y ancho de sus parámetros. Si activamos la opción **Cast string to PChar**introduce la cadena de texto del mensaje dentro de una función **PChar**:

```
begin
  Application.MessageBox(PChar('Este es un mensaje de prueba.'),
    PChar('Probando'), MB_OK + MB_ICONINFORMATION);
end;
```

C++ Builder Setting: Lo mismo que lo anterior para el código C/C++.

Wordwrap Mode: Si activamos la opción **Wordwrap by Line Width** separará los parámetros en línea según el ancho que hemos especificado (80 por defecto) o por el contrario si queremos hacerlo por contenido. En ninguna de las dos opciones me ha respetado al máximo de 80 caracteres cuando he puesto mensajes largos:

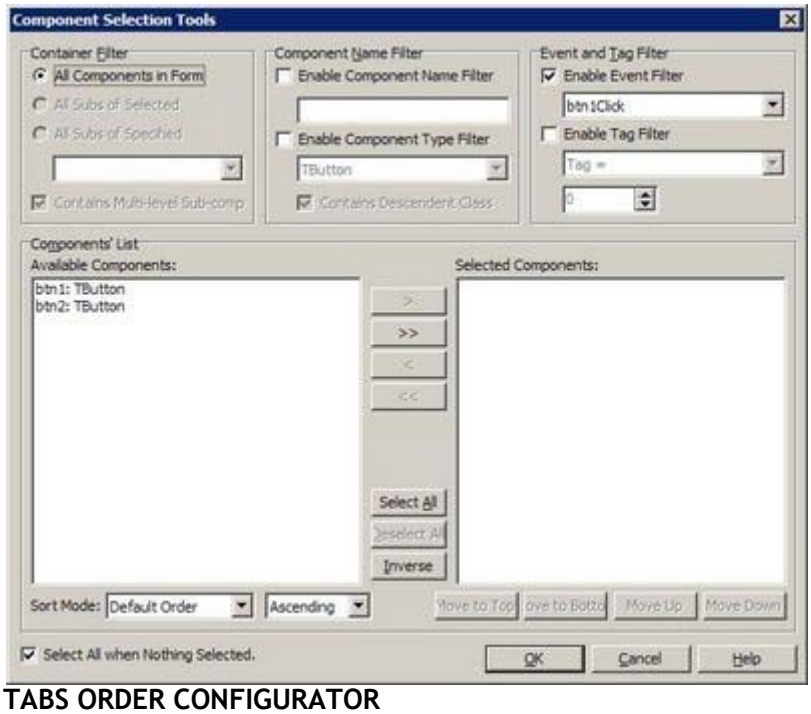
```
begin
  Application.MessageBox(
    'Este es un mensaje de prueba para probar si me respeta el ancho de texto al limite.',
    'Probando',
    MB_OK + MB_ICONINFORMATION);
end;
```

Other settings: podemos exportar o importar en un archivo INI la plantilla que utiliza para crear el código:

[Auto Saved]
MsgBoxCaption=Probando
MsgBoxCaptionIsVar=0
MsgBoxText=Este es un mensaje de prueba para probar si me respecta el ancho de texto al límite.
MsgBoxTextIsVar=0
MsgBoxIcon=1
MsgBoxButton=0
MsgBoxDefaultButton=0
MsgBoxResult=0
MsgBoxCodeKind=1
MsgBoxTopMost=0

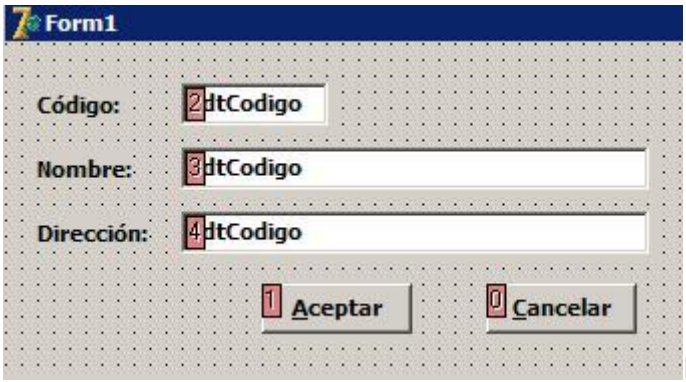
COMPONENT SELECTOR

Aquí podemos activar o desactivar la opción de selección múltiple de componentes que vimos anteriormente:

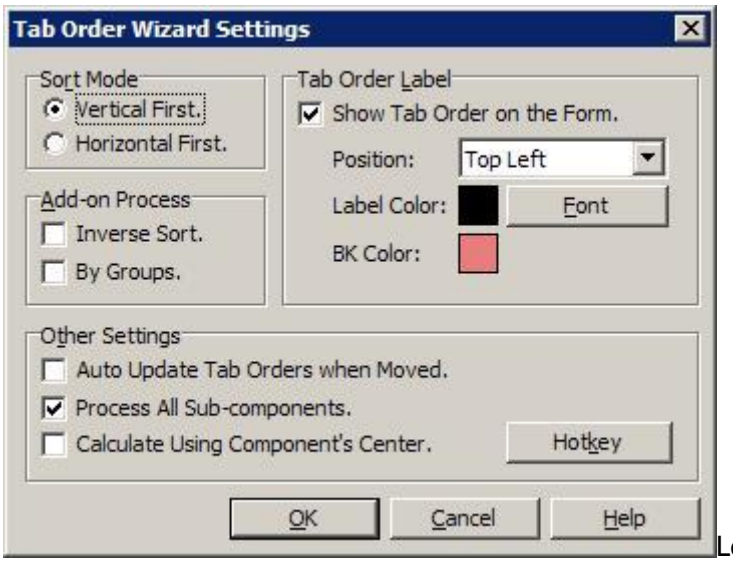


TABS ORDER CONFIGURATOR

Este apartado controla las opciones que afectan al orden de tabulación de los componentes visuales de un formulario:



La ventana de opciones es esta:



Los apartados que contiene son estos:

Sort Mode: podemos hacer que el orden de tabulación sea de arriba abajo (**Vertical First**) o de izquierda a derecha (**Horizontal First**).

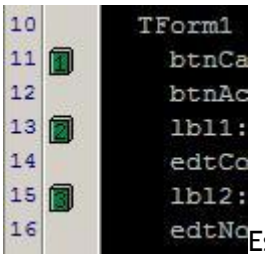
Add-on Process: Si está activada la opción **Inverse Sort** entonces el orden de tabulación irá al revés. Al activar la opción **By Groups** entonces considerará a los componentes con la misma anchura y altura como del mismo grupo, por lo que su tabulación será continua entre ellos (por lo menos es lo que dice la ayuda, pero por más pruebas que he realizado no he notado la diferencia).

Tab Order Label: Aquí configuramos como queremos ver el número de orden de tabulación en cada componente. Por defecto esta configurado para que se dibuje en la esquina superior izquierda del componente y con un color rojizo con borde negro. Pero si nos estorban podemos ponerlos a la derecha (**Top Right**) y en otro color:

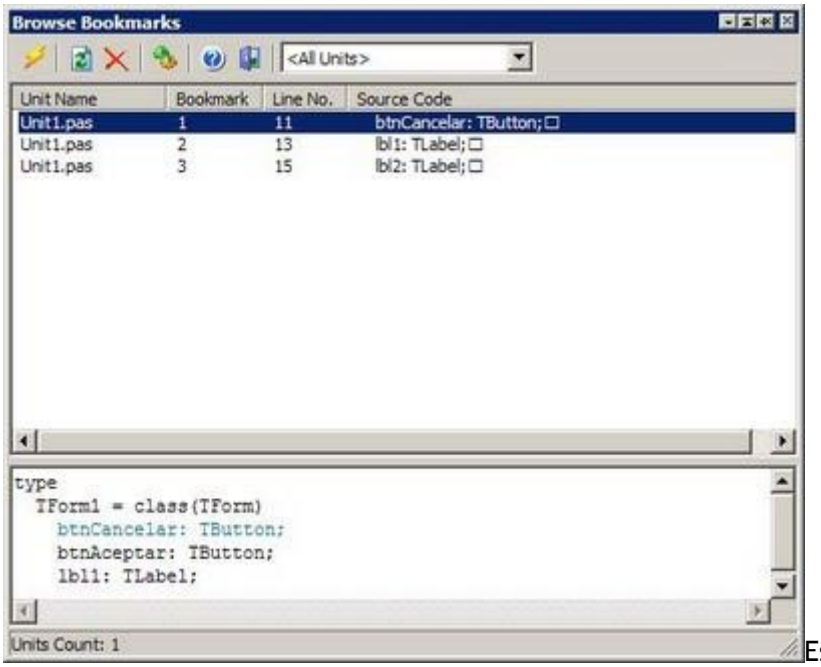
Other Settings: Tenemos tres opciones donde la primera nos permite corregir el orden de tabulación cuando se mueven los componentes (**Auto Update Tab Orders…**). La segunda opción ya viene activada y posibilita el orden de tabulación de unos componentes que están dentro de otros (como Frames, GroupBox, etc.). Y con la opción**Process All Sub-components** podemos hacer que calcule el orden de tabulación a partir de las coordenadas centrales de cada componente, en vez de esquina superior izquierda (**Calculate Using Component's Center**).

BOOKMARK BROWSER

En este apartado podemos configurar el navegador de marcadores de línea (**BookMark**) que incorpora **cnPack** y que solemos poner en nuestro código fuente:

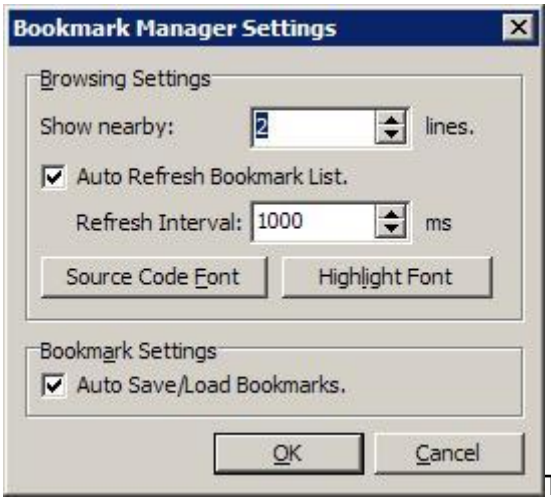


Este navegador puede verse seleccionando las opciones del menú superior **cnPack** -> **Bookmark Browser**. Mostrará esta ventana:



Esta es una lista de los marcadores que hemos colocado en la unidad actual que estamos editando y si hacemos doble clic sobre alguna o pulsamos el botón del rayo, irá directamente a la línea donde está.

También vemos que con solo seleccionar un marcador nos mostrará en la parte inferior las primeras 4 líneas de código que rodean a nuestra línea marcada. En su ventana de opciones podemos cambiar el campo **Show nearby**para decirle cuantas líneas queremos mostrar alrededor (por defecto 2):



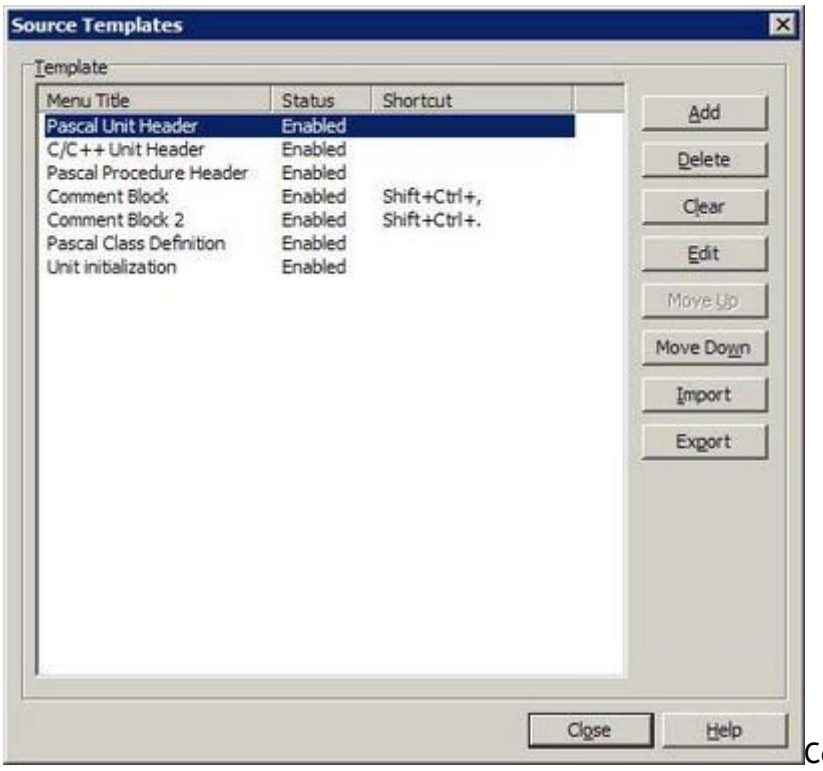
También está activada la opción de auto-refrescar la lista cada segundo y que se guarden en disco automáticamente. Yo creo que aquí les ha faltado la opción de poder dejar abierta la ventana de bookmarks aunque

hagamos doble clic sobre uno. Es muy incómodo tener que volver a la lista cada vez que queremos saltar a otro marcador.

Las otras opciones que lleva son para cambiar la fuente y el color de la previsualización de las líneas.

SOURCE TEMPLATES

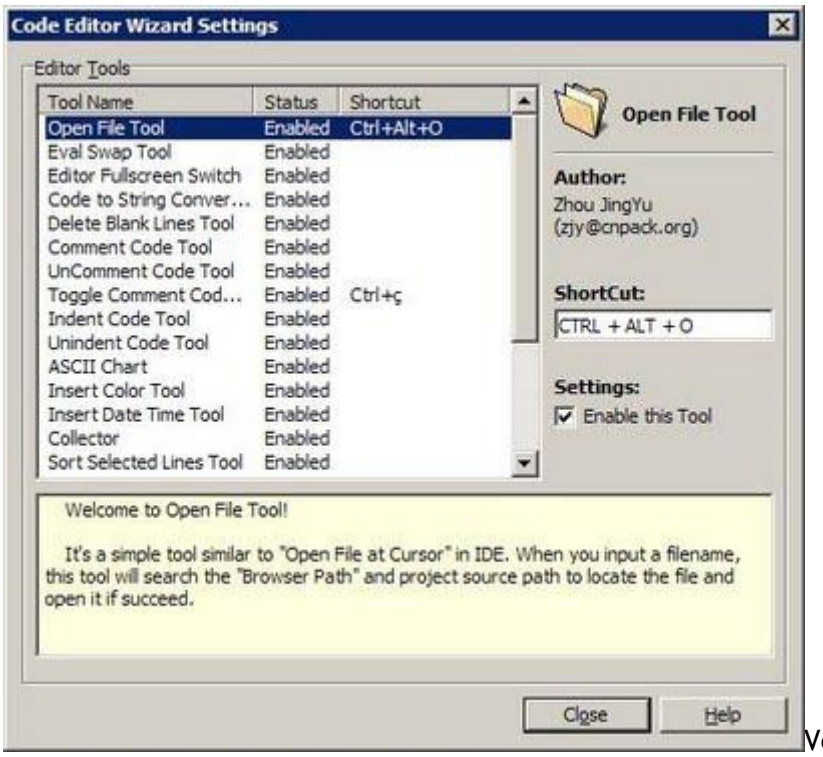
Esta opción abre la ventana que vimos anteriormente para crear, editar o eliminar las plantillas de código que tiene predeterminadas:



Contiene dos botones para importar y exportar plantillas que utilizan la extensión **.cdt** y se guardan en formato XML.

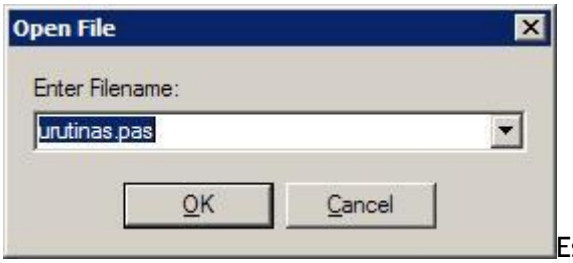
CODE EDITOR WIZARD

En este apartado podemos personalizar las teclas de acceso rápido que queremos para los asistentes del editor de código que tiene **cnPack,** así como habilitar o deshabilitar estos asistentes:



Veamos cada opción y lo que hace:

Open File Tool: abre un archivo cualquiera dentro del editor como si en Delphi hubiésemos elegido **File** -> **Open**pero con otra ventana:



Esta opción tiene predeterminado el atajo de teclado **CTRL + ALT + O**.

Eval Swap Tools: Este asistente lo utilizamos anteriormente para intercambiar lo que hay a la derecha o a la izquierda del operador de igualación, por ejemplo:

Clientes.FieldByName('NOMBRE').AsString := sNombre;

Después de utilizar esta opción quedaría así:

sNombre := Clientes.FieldByName('NOMBRE').AsString;

Lo que nos interesa aquí es asignarle una tecla rápida para no perdernos en las decenas de botones que tiene**cnPack**.

Editor Fullscreen Switch: esta opción es la que conmuta entre poner el editor de código a pantalla completa o dejarlo normal. Aquí le he asignado yo las teclas **CTRL + ALT + F** ya que es muy útil. Si pulsamos el botón **Settings**podemos configurar que se maximice el editor nada más arrancar Delphi y ocultar el formulario principal o restaurarlo a modo normal cuando volvamos de pantalla completa:

Delete Blank Lines Tools: lo único que podemos hacer con esta opción es asignar un atajo de teclado para eliminar las líneas vacías de código.

Comment Code Tool: este asistente lo utilizábamos para convertir las líneas de código seleccionadas en comentario.

Uncomment Code Tools: al contrario del anterior, quita los comentarios.

Toggle Comment Code Tools: pone o quita los comentarios. Yo he asignado a esta opción las teclas **CTRL + ALT + C** porque podemos anular muchas líneas de código rápidamente, ya que al convertir la línea en comentario pasa a la siguiente línea. Esta última característica la podemos desactivar si pulsamos sobre el botón **Settings**:

Ident Code Tool: Con esta opción movíamos dos espacios a la derecha el código fuente seleccionado. Aquí podemos asignar la tecla que queramos aunque yo prefiero seguir utilizando las de Delphi (CTRL + K + I).

Unident Code Tool: Mueve a la izquierda el código fuente y en Delphi sus teclas son CTRL + K + U.

ASCII Chart: esta es la opción que activa o desactiva la tabla de caracteres ASCII para ver su código en decimal o hexadecimal.

Insert Color Tool: aquí no vendría tampoco mal asignar un atajo de teclado al asistente que permite elegir un color y devuelve el número hexadecimal del mismo.

Insert Date Time Tool: esta opción pertenece al asistente que escribe en varios formatos la fecha y hora actual del sistema.

Collector: si vais a utilizar el pequeño bloc de notas que mostré anteriormente deberíais asignar una tecla rápida al mismo.

Sort Selected Lines Tool: Si asignaís un atajo de teclado a esta opción podéis ordenar las líneas de código alfabéticamente y a la velocidad del rayo.

Toggle Use/Include Tool: esta opción nos permite saltar directamente a la línea donde se encuentra la sección**uses**.

Toggle Var Field Tool: con esta otra opción saltamos a la sección **var** del procedimiento donde estamos situados.

Previous Message Line Tool y **Next Message Line Tool**: con las teclas rápidas **ALT + ,** y **ALT + .** podemos recorrer la ventana de mensajes de Delphi por si no nos apetece coger el ratón.

Jump to Interface Tool: Salta a la línea donde declaramos la interfaz (**interface**).

Jump to Implementation Tool: Salta a la línea de la implementación (**implementation**).

Jump to Matched Keyword Tool: Salta a la siguiente palabra que estamos buscando.

Espero que no asignéis a todas un atajo de teclado porque si no vais a necesitar una tabla periódica como la de los químicos para acordaros de todas. En el próximo artículo seguiremos exprimiendo las opciones.

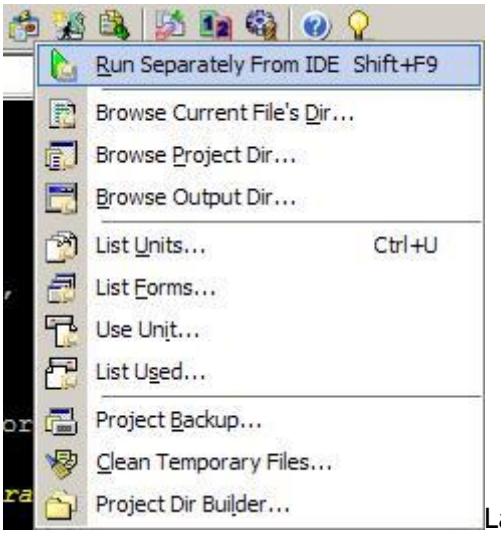
Pruebas realizadas en Delphi 7 y Delphi 2007.

20 noviembre 2009

El experto cnPack (4)

Continuando con la barra de botones que aparece encima del editor de código, vamos a ver las características del botón **Tools to Enhace Functions about Project**:

 Este botón está relacionado con los archivos de nuestro proyecto. Al pulsarlo nos aparecen estas opciones:



La opción **Run separately From IDE** ejecuta nuestro programa como si lo hubiésemos ejecutado fuera de Delphi. Esto suele utilizarme mucho cuando tenemos que hacer una comprobación rápida y no tenemos gana de desactivar el Debugger.

Luego vienen tres opciones que llaman al explorador de Windows y saltan a los siguientes directorios:

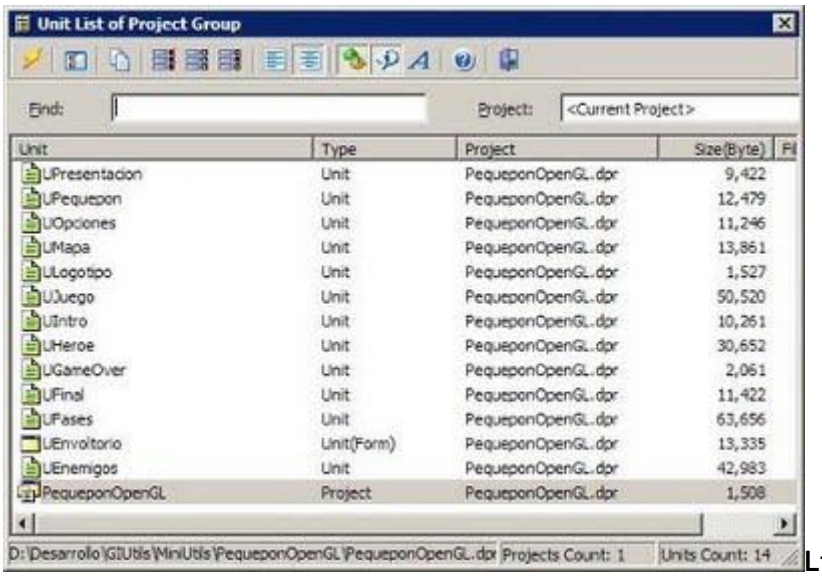
Browse Current File's Dir: Va a la carpeta donde están los archivos de nuestro programa.

Browse Project Dir: Va a la carpeta de nuestro proyecto.

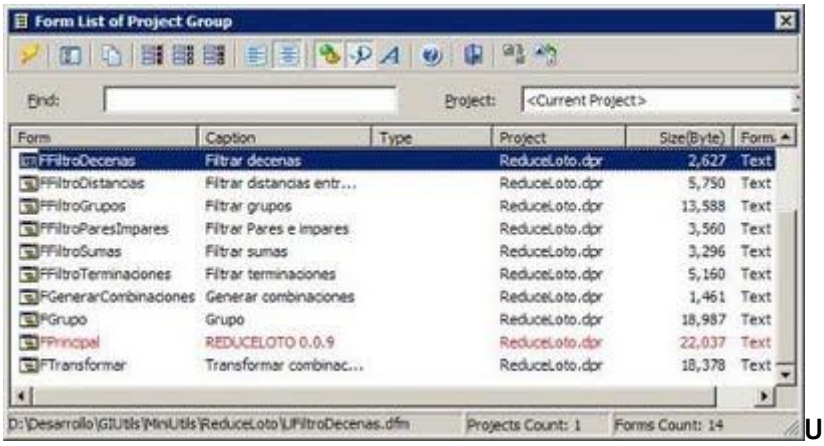
Browse Output Dir: Muestra el contenido de la carpeta donde se compila nuestro ejecutable.

Si las opciones de nuestro proyecto apuntan a la misma carpeta, estas tres opciones harán prácticamente lo mismo. Después vienen otro grupo de opciones que solo son útiles si estamos trabajando con el editor a pantalla completa:

List Units: abre la ventana de búsqueda de unidades:

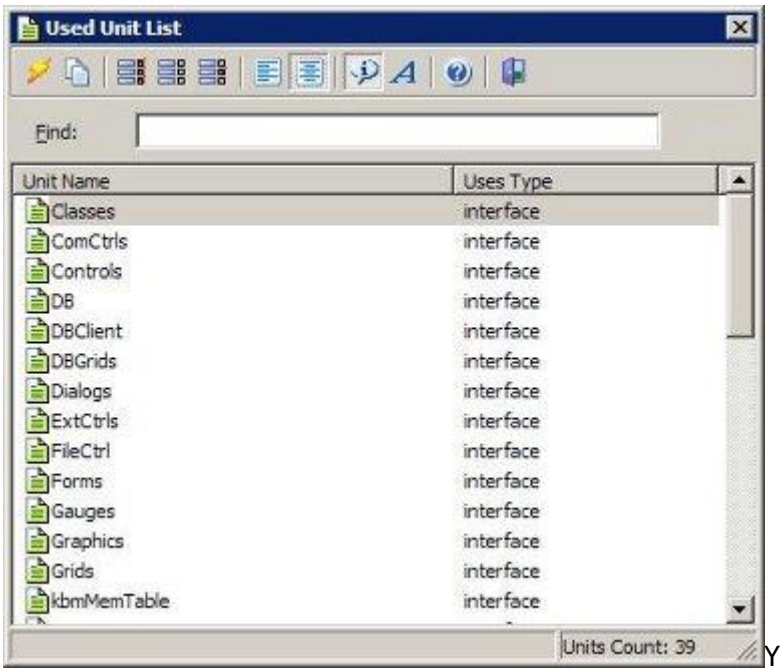


List Forms: abre la ventana de búsqueda de formularios:



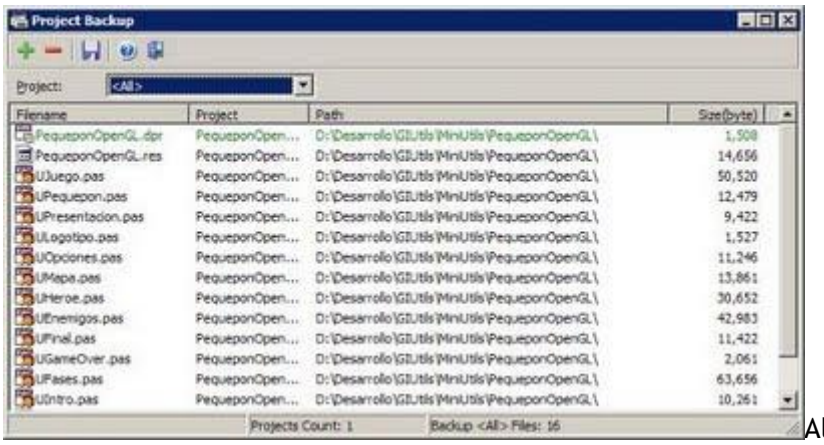
Use Unit: añade otra unidad a la unidad actual. Es exactamente la misma opción que **File** -> **Use Unit** de Delphi.

Used Unit List: muestra la lista de unidades utilizadas por nuestro proyecto:

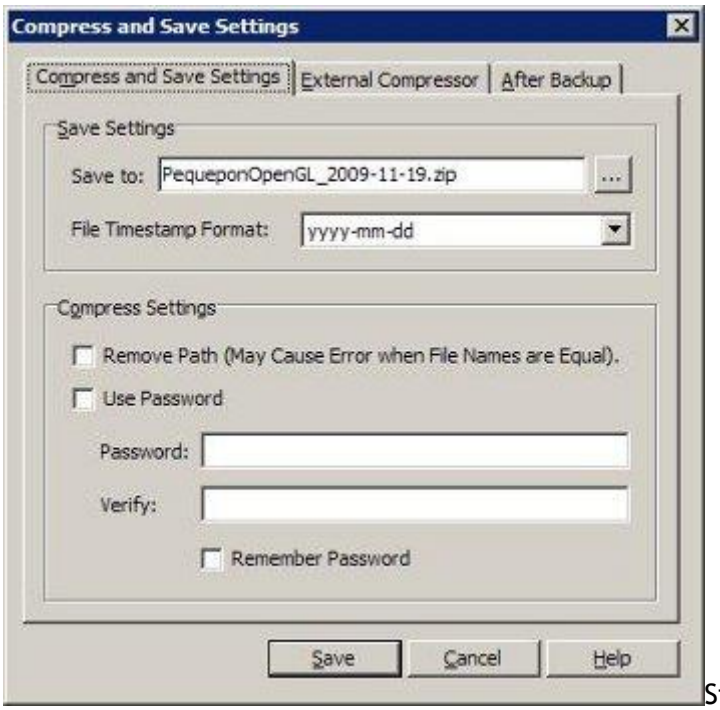


Y las tres últimas opciones de este menú están relacionadas con el proyecto:

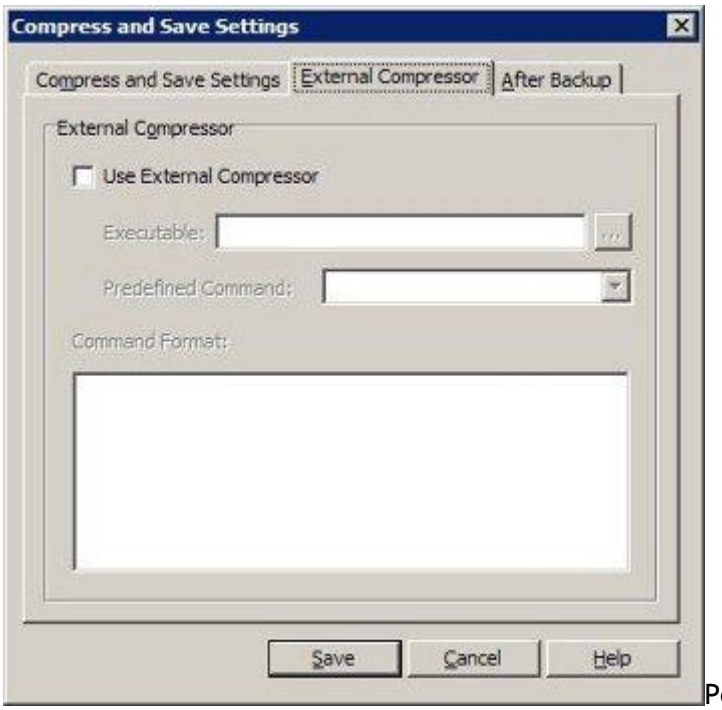
Proyect Backup: Permite hacer una copia de seguridad de nuestro proyecto en un archivo zip. Al pulsar sobre esta opción veremos una ventana con todas las unidades:



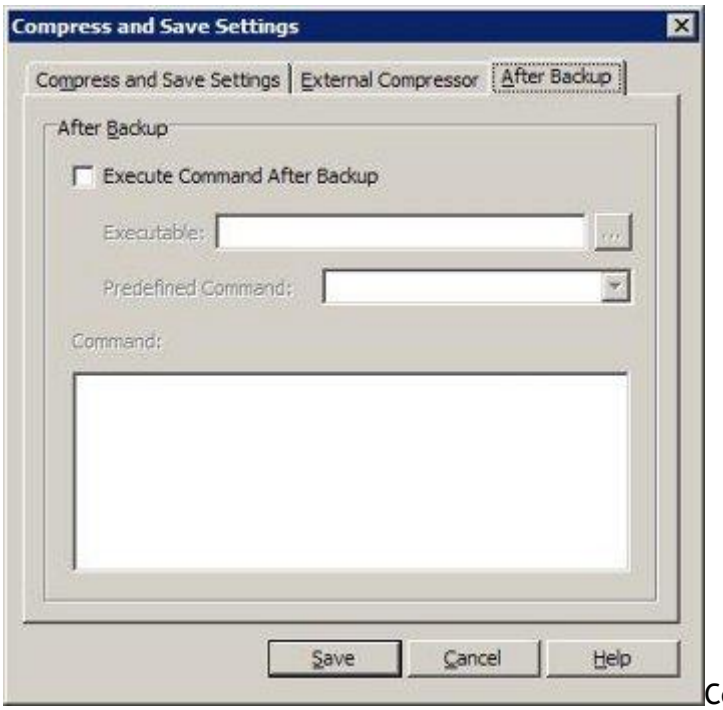
Al pulsar sobre el botón de guardar (**Compress and Save the Backup file**) aparecerá este cuadro de diálogo donde debemos elegir el nombre del archivo a guardar, las opciones de compresión e incluso podemos asignar una contraseña:



Si no nos gusta comprimir la copia en un archivo zip podemos llamar a otro compresor externo (como 7-zip) para que haga el trabajo. Esto se configura en la segunda pestaña:

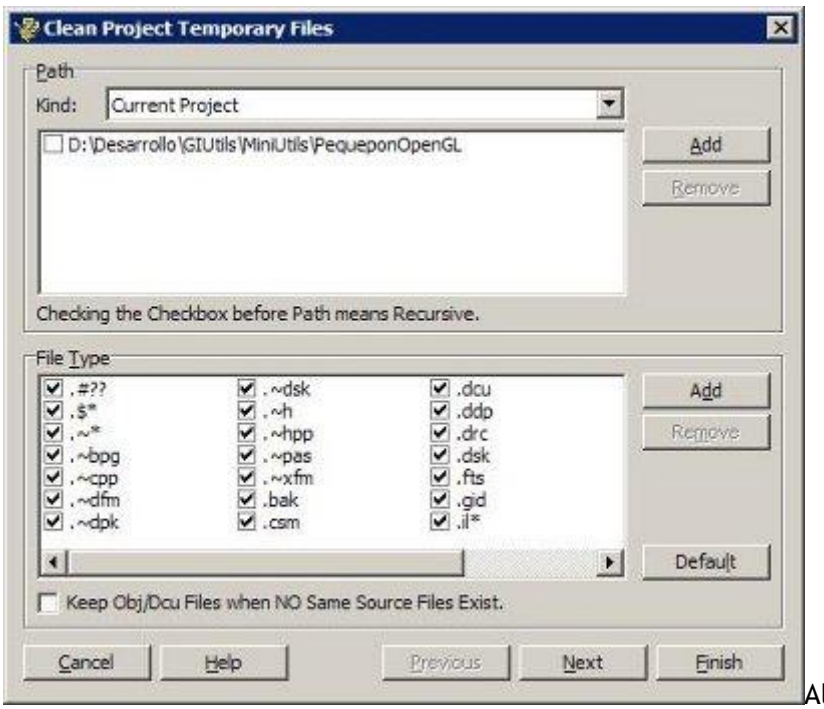


Pero es que además, en la tercera pestaña podemos hacer que después de realizar la copia de seguridad llame a un programa en concreto:

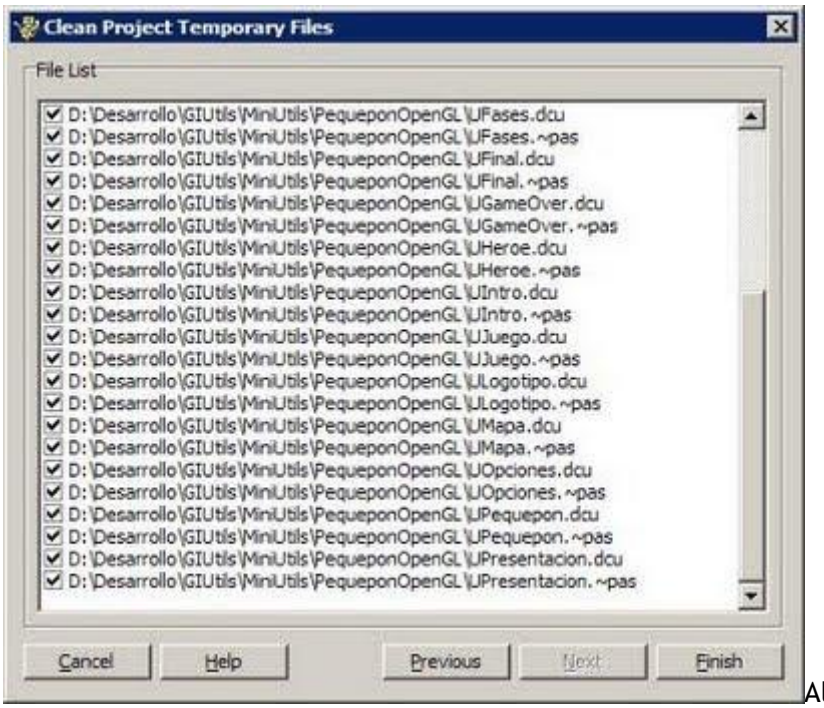


Con esta última opción podemos llamar por ejemplo a otro programa para que conecte por FTP y envíe la copia a un servidor externo, podemos enviarla por correo electrónico, etc.

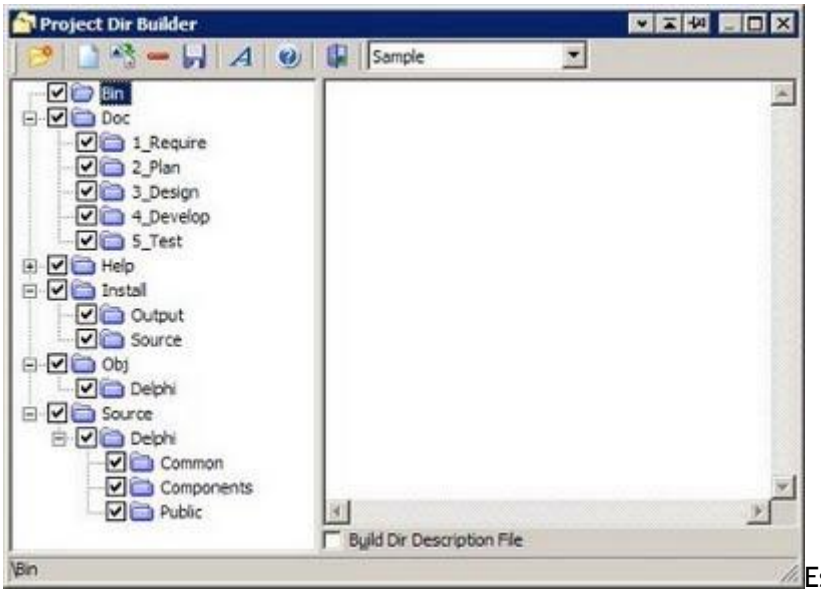
La segunda opción (**Clean Temporaly Files**) se utiliza para limpiar archivos temporales que no vamos a necesitar antes de hacer la copia de seguridad:



Al pulsar sobre el botón **Next** nos mostrará un listado con todos los que ha encontrado:



Al pulsar sobre el botón **Finish** realizará el proceso de limpieza. Y por último tenemos la opción **Project Dir Builder**que nos va a permitir organizar la estructura de directorios de nuestro proyecto:

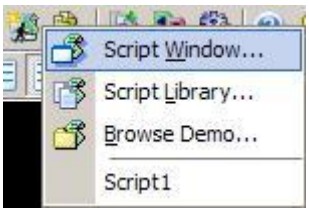


Esto puede ser de utilizar si queremos compartir el código fuente con otros usuarios en Internet subiendo el proyecto a un repositorio como CSV. Aunque en la ayuda no se explica muy bien sobre como utilizarlo, es algo raro.

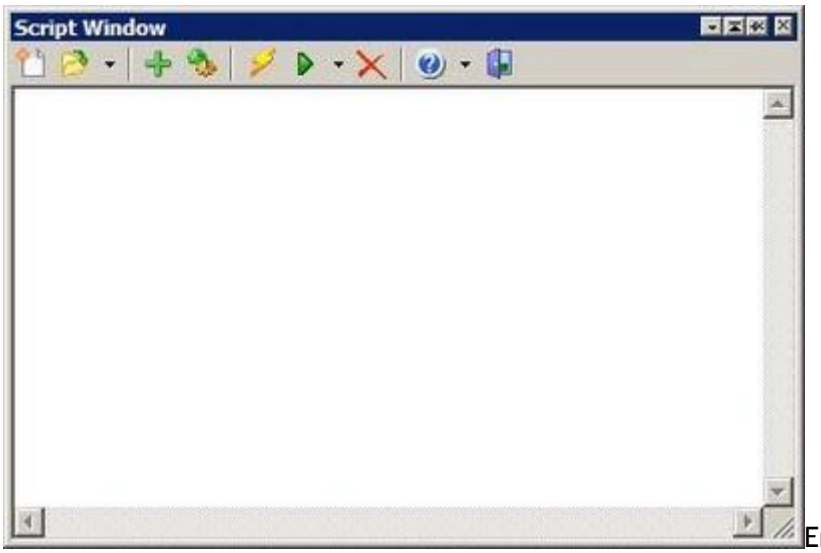
LAS OPCIONES DE LOS SCRIPTS

Luego están las herramientas para crear scripts que se programan utilizando su propia librería. Este es el botón que se utiliza para crear los scripts:

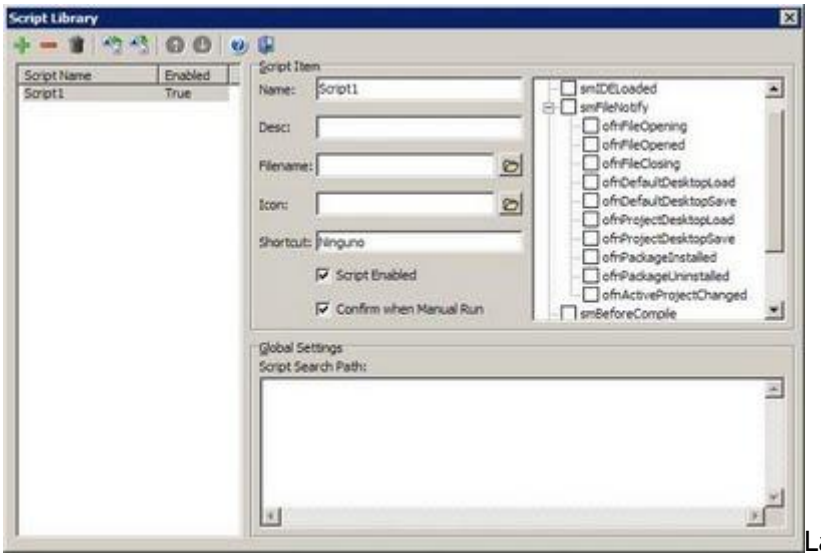

Al pulsarlo tenemos estas tres opciones:



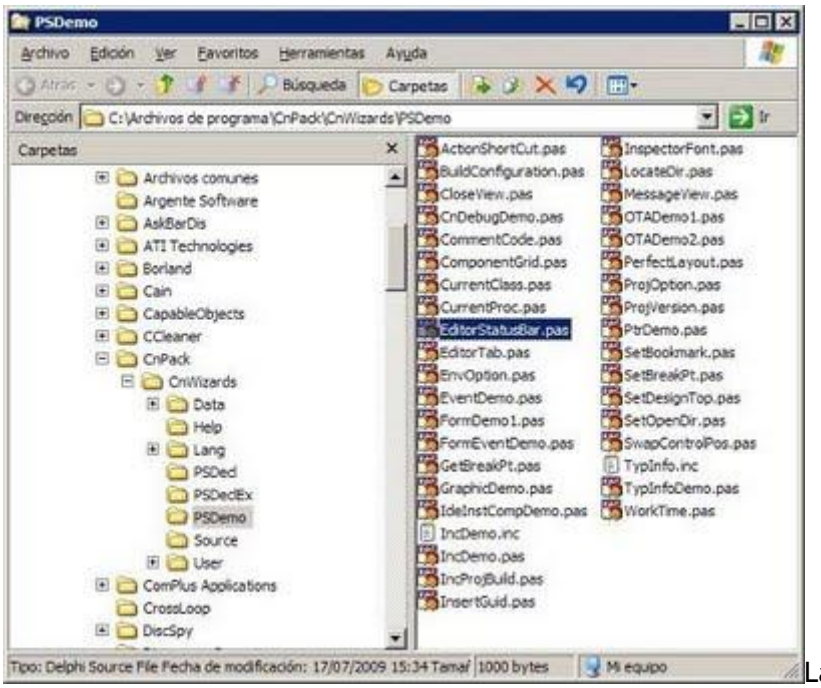
La primera opción abre el compilador de script:



En la segunda opción abrimos la ventana de librerías de scripts para ver los eventos que podemos programar:



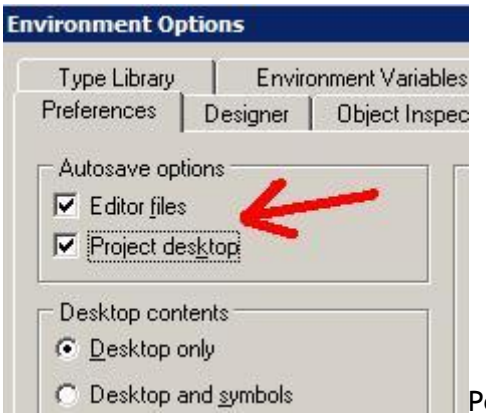
La tercera opción abre el explorador de Windows y nos sitúa en la carpeta donde están los scripts de ejemplo:



Las siguientes opciones que puedan aparecer son los script creados o cargados por nosotros. Este tema de los scripts lo ampliaré más adelante si tengo más tiempo y averiguo como va.

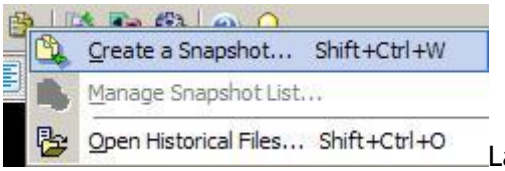
GUARDANDO Y CARGANDO LA CONFIGURACIÓN ACTUAL

El último botón del grupo nos sirve para guardar o cargar el estado del IDE tal como lo dejamos. Esto también lo lleva por ejemplo Delphi 7 con las opciones **Tools** -> **Environment Options** y activando estas dos opciones:

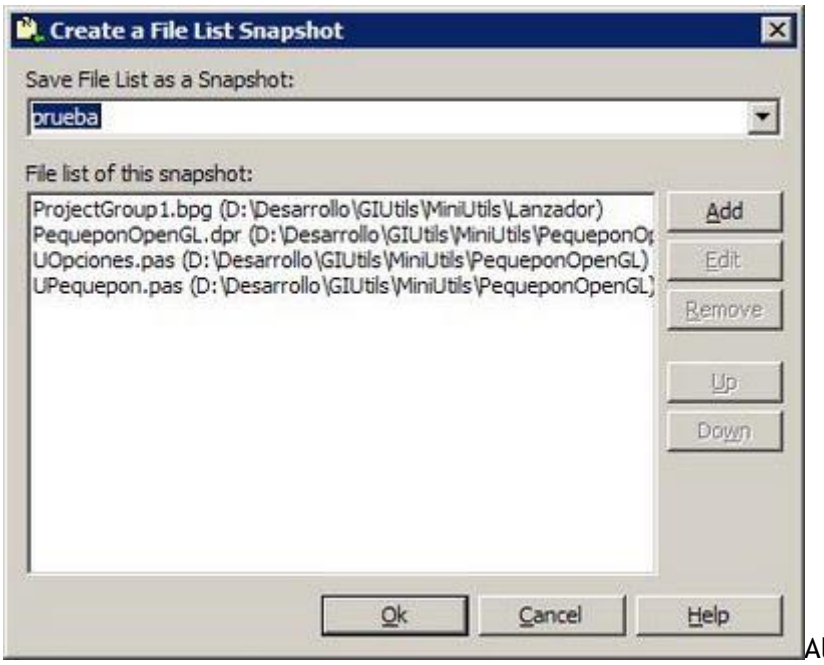


Pero el inconveniente está en que solo podemos guardar un estado por proyecto, mientras que con este botón podemos guardar varios perfiles:

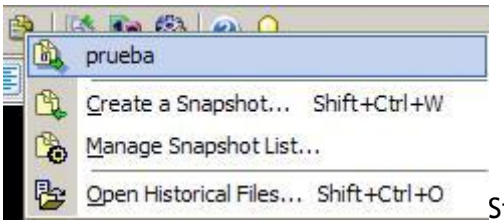

Al pulsarlo tenemos estas tres opciones:



La primera opción sirve para crear un nuevo snapshot. En la ventana que aparece le damos un nombre al primer campo:

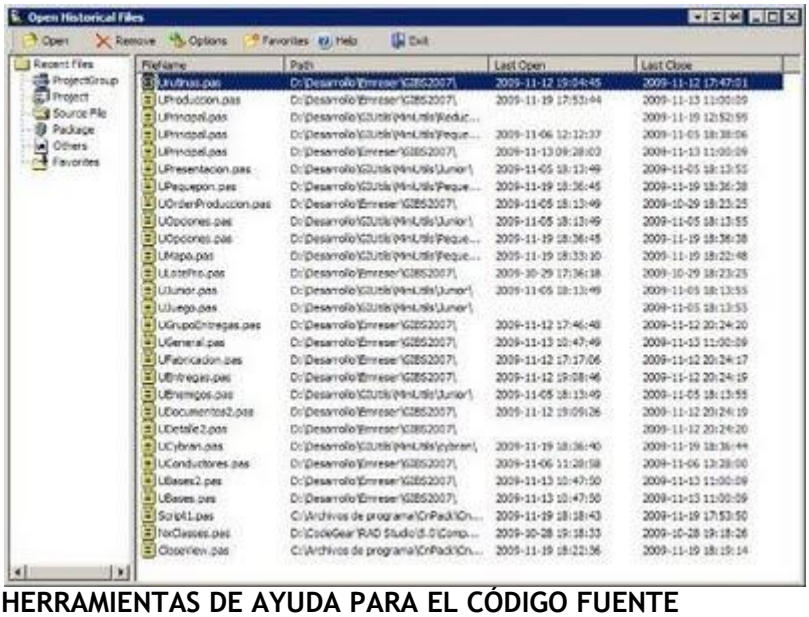


Al pulsar el botón **Ok** veremos que ha memorizado ese snapshot en ese mismo menú:



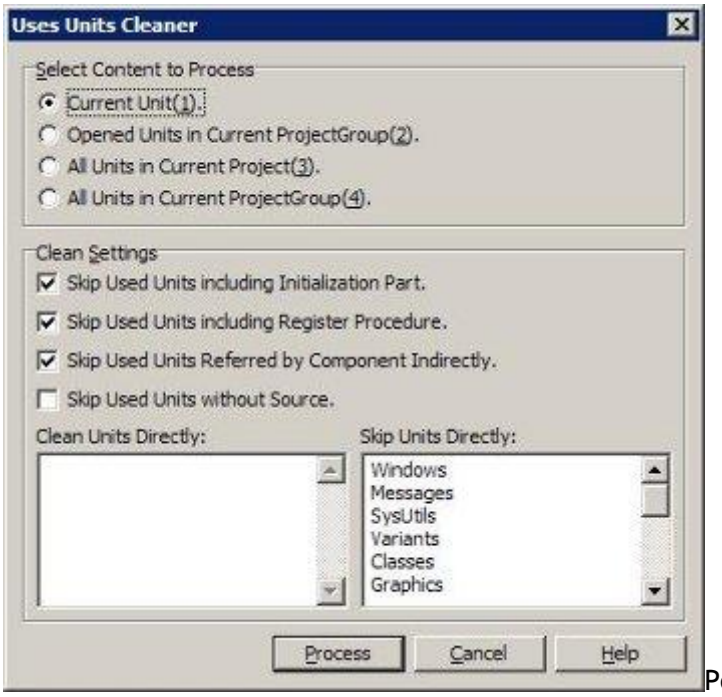
Si nos vamos a otro proyecto y abrimos otras unidades, con sólo seleccionar este snapshot, cerrará el proyecto actual, reabrirá el proyecto donde capturamos el snapshot y abrirá todas las unidades como las dejamos. Es algo muy potente para saltar de un proyecto a otro cuando necesitamos capturar trozos de una unidad de otro proyecto para traerla a la actual.

La opción **Manage Snapshot List** vuelve a abrir la ventana de los snapshots creados por si queremos eliminar o modificar alguno. Y la última opción (**Open Historical Files**) nos muestra un histórico con todas últimas unidades que hemos tocado:



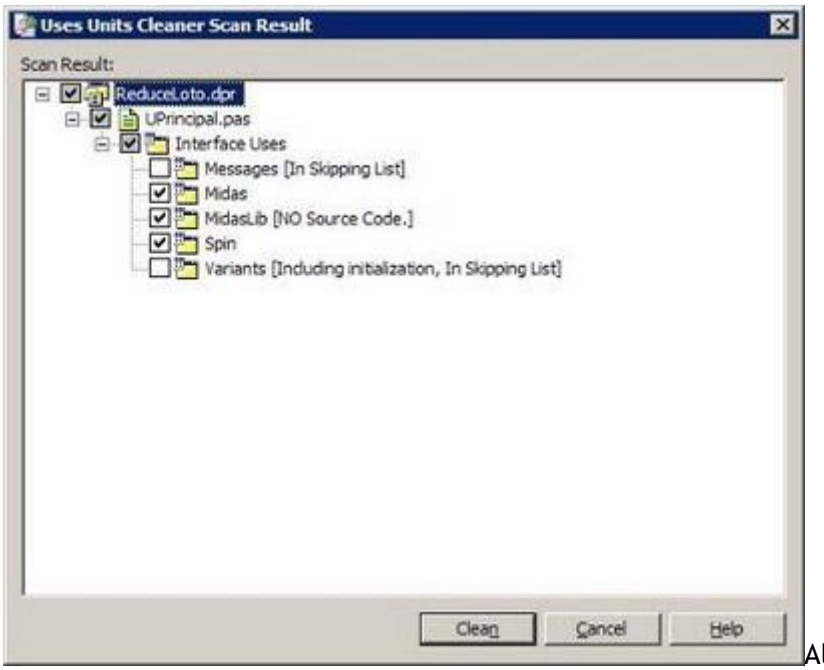
Aquí tenemos el siguiente grupo de tres botones que son de gran ayuda:

 El primer botón es el encargado de borrar las unidades que tenemos incluidas en la sección **uses** pero que ya no utilizamos:



Por ejemplo, si insertamos un componente nuevo en el formulario y guardamos se añadirá su unidad correspondiente en la sección **uses**, pero si lo eliminamos ya no se borra su unidad.

Al pulsar el botón **Process** comenzará a realizar el análisis y nos dirá que unidades tenemos vinculadas que ya no utilizamos:

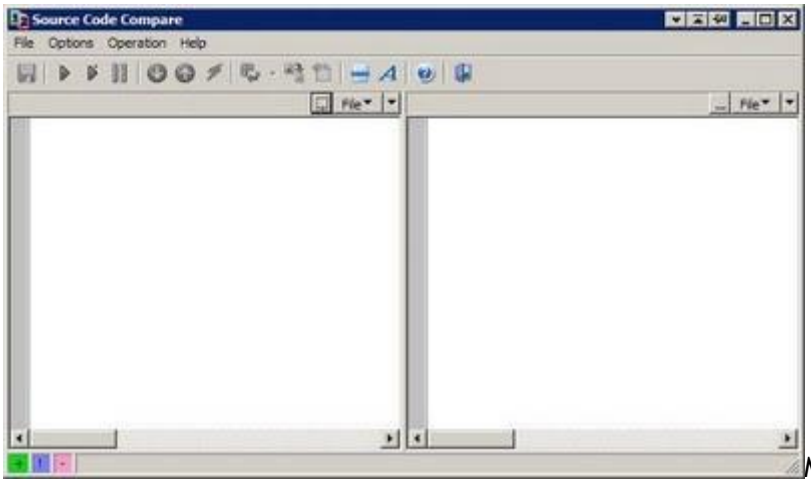


Al pulsar el botón **Clean** eliminará todas las unidades inservibles, acelerando de ese modo la compilación y enlazado de los binarios DCU. Este análisis lo hace por defecto de la unidad actual, pero podemos decirle que lo haga para todas las unidades del proyecto. Si no

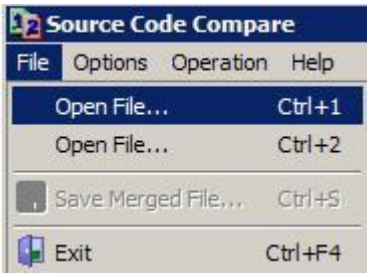
hubiese ninguna unidad inservible diría esto:

Vamos el siguiente botón:

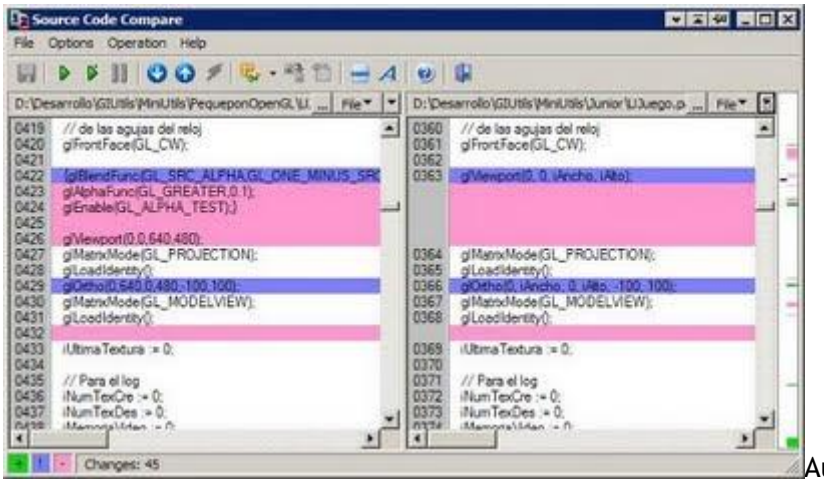
Esta herramienta sirve para comparar dos unidades de código fuente (o cualquier archivo de texto) en busca de sus diferencias. Es similar una la herramienta que siempre utilizo (y es gratuita) llamada WinMerge. Nos muestra esta ventana:



Mediante la opción **File** abrimos un archivo y luego otro:



Una vez tenemos cargados los dos archivos pulsamos el botón **Source code compare** (el play) o bien pulsamos la tecla **F9** y realizará una comparación entre ambas unidades:



Aunque contiene una

opción para mezclar ambos archivos e igualar el código fuente no lo recomiendo porque puede armar un desastre impresionante. Aquí echo de menos el poder editar manualmente ambas unidades para arreglar yo mismo las diferencias (en eso le gana WinMerge).

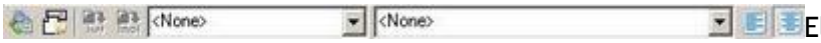
El último botón de este grupo sirve para abrir la ventana de opciones:

 Esta ventana de opciones comenzaremos a verla en el siguiente artículo porque es inmensamente grande:

Al su lado vienen dos botones para la ayuda y el acerca de:

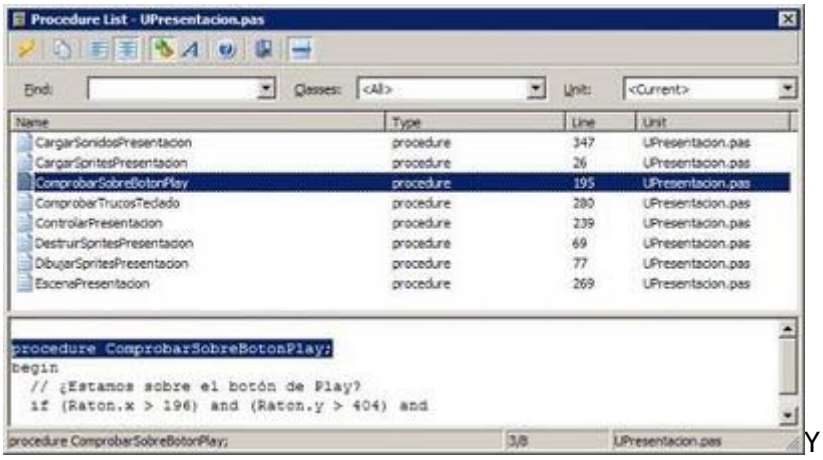
LA BARRA INFERIOR DE ACCESO RÁPIDO

Debajo de toda esta barra que hemos visto tenemos esta otra:



El primer botón sirve para la búsqueda de procedimientos en la unidad donde estamos:

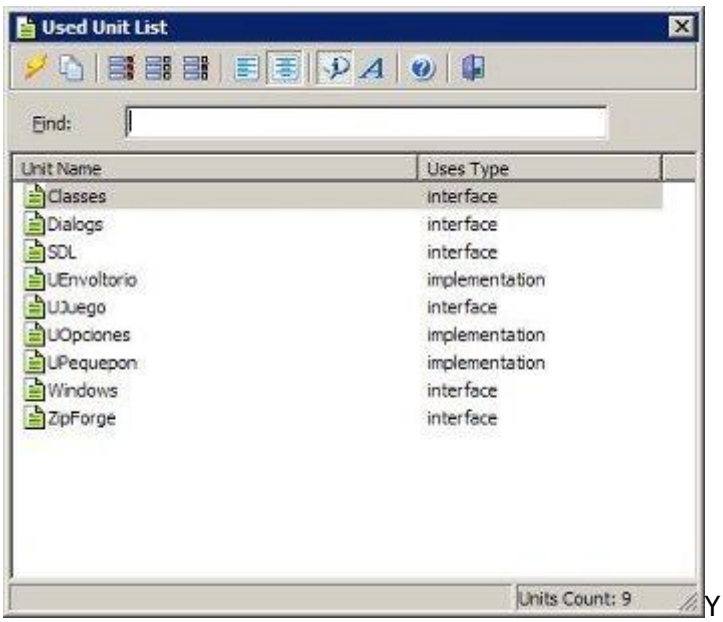
 Aparte de poder buscarlos, también podemos ver las 4 primeras líneas de código del que tenemos seleccionado en la lista:



Y como en todas las ventanas de búsqueda que incorpora cnPack, tenemos un campo de búsqueda para filtrar el listado por palabras clave.

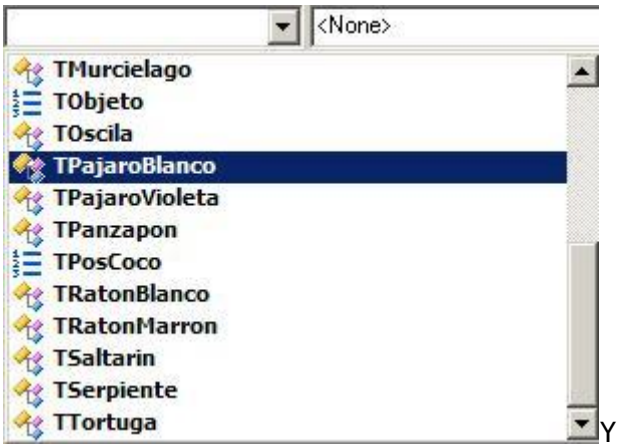
Luego está el botón de búsqueda de unidades:


Que abre la ventana que vimos anteriormente:

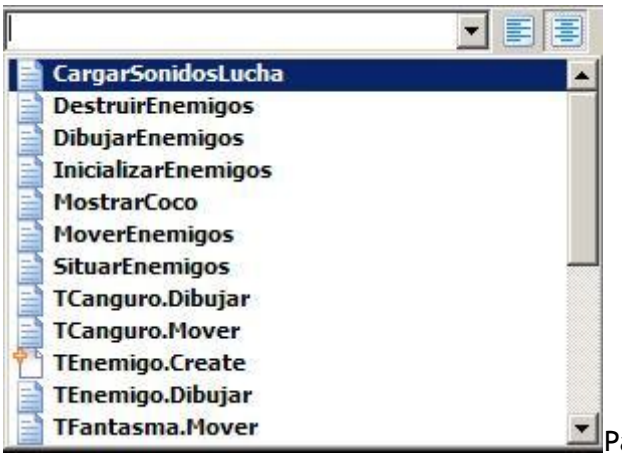


Y los otros dos botones que le siguen sirven para saltar en el código a la interfaz o a la implementación, como ya vimos con anterioridad:

Luego le sigue un desplegable con la lista de clases:



Y otro con la lista de procedimientos:



Para ambos desplegables de búsqueda tenemos estos dos botones que les siguen:


Si tenemos activado el de la izquierda significa que cuando comencemos a escribir para buscar una clase o un procedimiento, solo buscará las que empiecen por la palabra que hemos puesto. En cambio, si activamos la de la derecha buscará todas que la contengan dicha palabra.

Con esto concluimos todo lo referente a las barras de herramientas que tiene el editor de código. En el próximo artículo entraremos a las opciones de cnPack para configurarlo todo.

Pruebas realizadas en Delphi 7 y Delphi 2007.

Publicado por Administrador en 09:37 0 comentarios
Etiquetas: expertos

13 noviembre 2009

El experto cnPack (3)

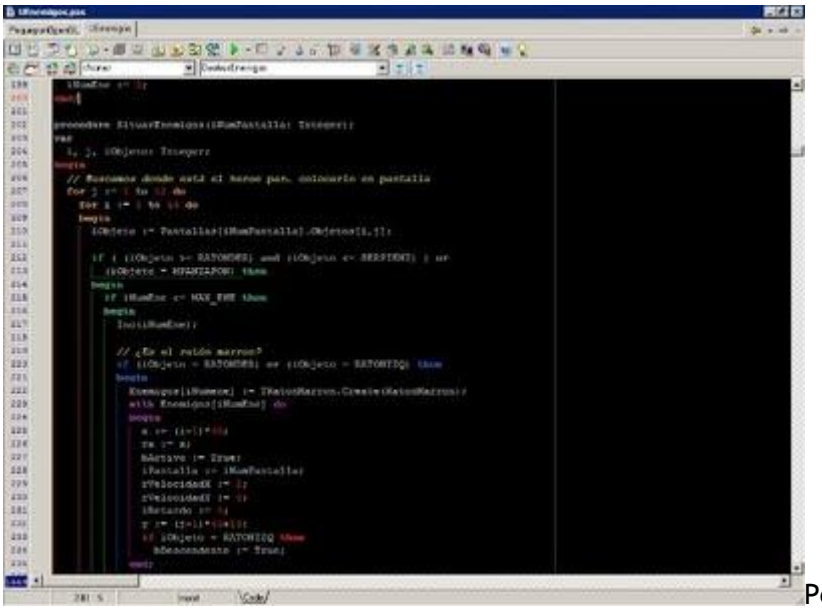
Después de ver la barra de botones que nos permiten editar formularios con mucha más facilidad, vamos a pasar a ver la barra que aparece en la parte superior del editor de código.

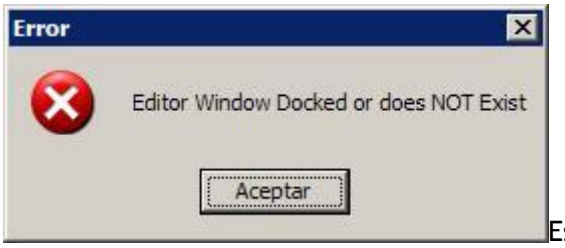
BOTONES PARA LA VISUALIZACIÓN Y EL ASISTENTE

Al igual que hice anteriormente, vamos a comenzar a ver la barra superior de izquierda a derecha por grupos, comenzando con los dos primeros botones:

El primer botón nos sirve para poder editar el código fuente a pantalla completa. Al activarlo nos aparece este mensaje sobre su descripción:

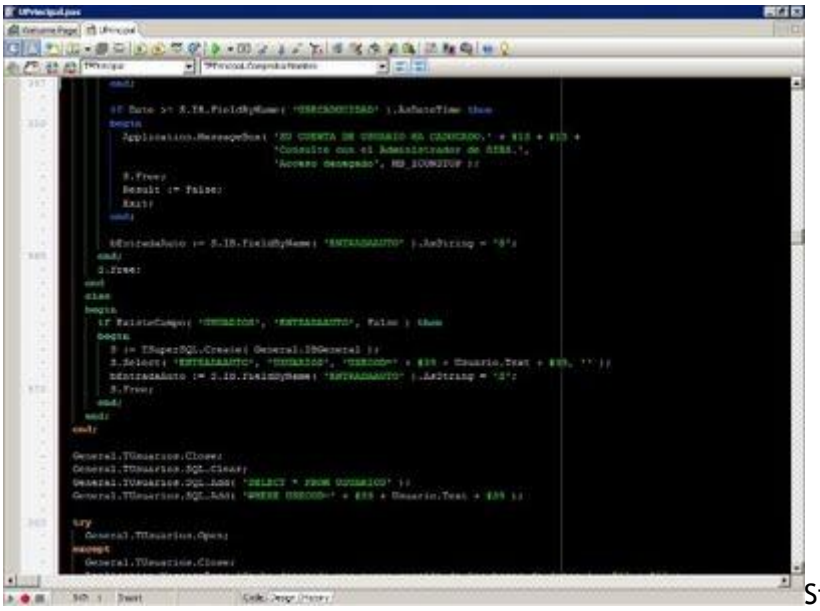
En Delphi 7 se vería así:


Pero en Delphi 2007 no funciona y nos dirá este error:

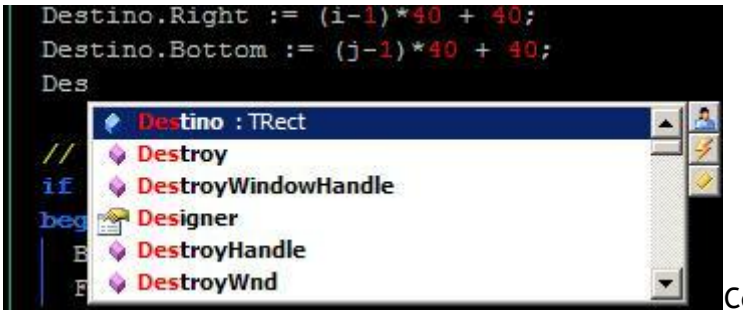


Eso es porque tenemos el IDE en modo **Default Layout**, por lo que demos cambiarlo al modo **Classic Unlocked**cambiando en desplegable que tenemos en la parte superior derecha de Delphi:

Ahora si que podemos activar la edición de código a pantalla completa:


Si el segundo botón está activado aparecerá el asistente de código al estilo Visual Studio:


Este asistente es más potente que el que incorpora Delphi porque con sólo comenzar a escribir ya nos va dando sugerencias sobre el objeto, función o procedimiento que estamos buscando:



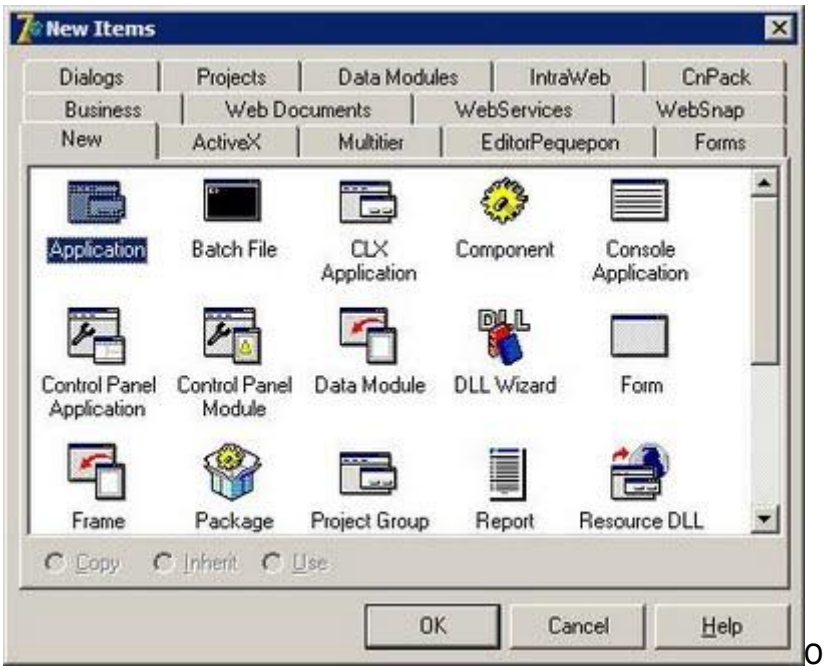
Con el mismo botón podemos activarlo o desactivarlo según nos convenga.

BOTONES PARA LA CARGA Y GRABACIÓN DE UNIDADES

A continuación tenemos otro botón que aparece en Delphi 7 pero que no aparece en Delphi 2007:

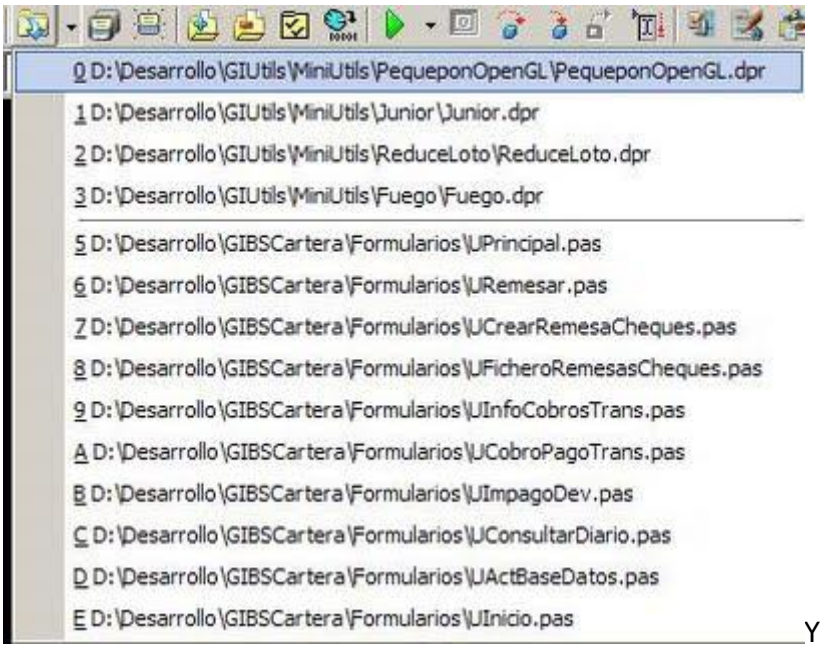

Este botón sirve para crear una nueva unidad, es decir, es el equivalente a seleccionar en el menú superior **File** ->**New** -> **Unit**. A su derecha tenemos el siguiente botón que nos sirve para añadir un nuevo elemento a Delphi:


Es el equivalente a seleccionar **File** -> **New** -> **Other** en Delphi 7:



O en Delphi 2007:

Pasando ya al siguiente grupo, tenemos un botón desplegable para reabrir proyectos anteriores (**File** -> **Reopen**):



Y cerrando el grupo tenemos los botones de **Guadar todo** y **Guardar Como**:

LOS BOTONES DEL PROYECTO

En el siguiente grupo de botones tenemos que los dos primeros se utilizan para añadir archivos al proyecto o eliminarlos:


Después tenemos este otro que muestra las opciones del proyecto:


Y el último construye el proyecto (lo recompila todo):

BOTONES PARA LA EJECUCIÓN Y LA DEPURACIÓN

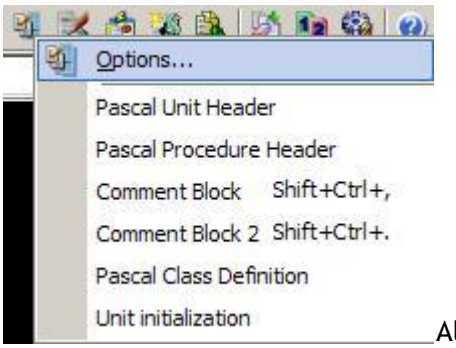
El siguiente grupo de compone de 6 botones que son prácticamente los mismos que tiene Delphi en sus barras de herramientas:


Son los botones de ejecutar, detener, los de depuración, etc. Los han puesto aquí por si trabajamos con el editor de código a pantalla completa.

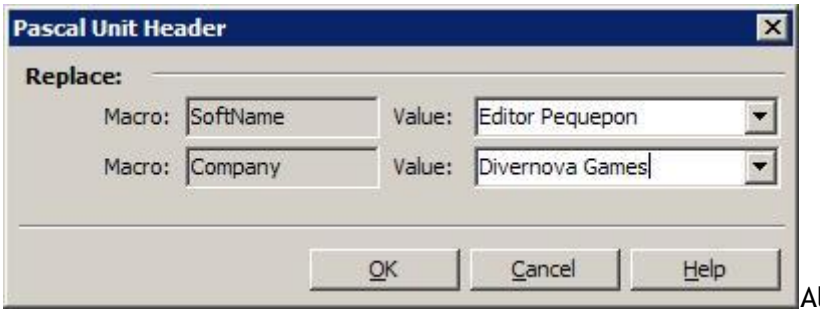
EL BOTÓN DE LOS COMENTARIOS

Aquí es donde comienza la parte interesante. Tenemos un primer grupo con 5 botones:


El primer botón es el encargado de documentar nuestro código fuente con plantillas automáticas en las que tenemos que hacer pocos o casi ningún cambio:

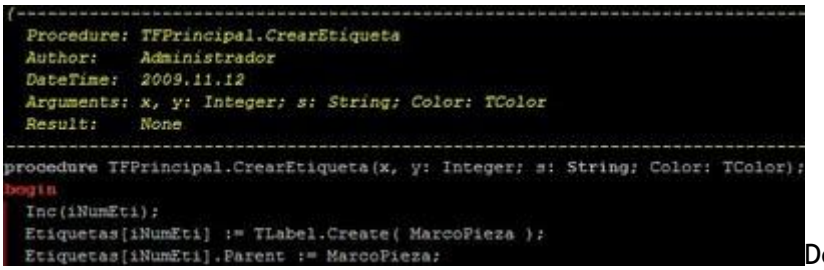


Al pulsar sobre este botón se despliegan una serie de opciones donde cada una de ellas crea un comentario en cierta zona de código. La segunda opción (**Pascal Unit Header**) crea un comentario al inicio de nuestra unidad cuyos datos debemos introducir en esta ventana que aparece:

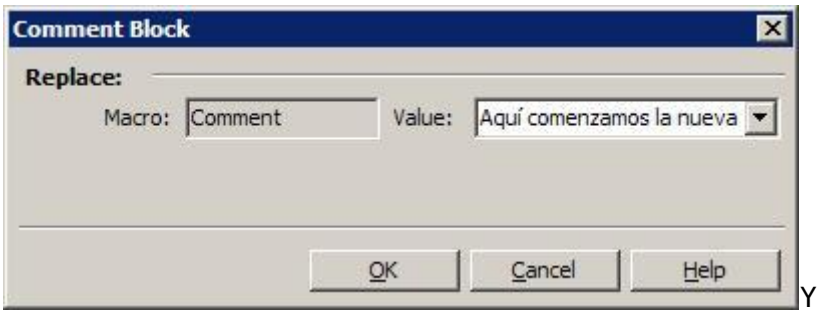


Al pulsar el botón **Ok** nos creará este comentario al principio de la unidad:

Luego tenemos la opción **Pascal Procedure Header** que sirve para comentar nuestros procedimientos. Antes de pulsar sobre esta opción debemos situarnos sobre la misma línea donde está la palabra **procedure** o **function** e insertará el comentario de golpe:



Después le siguen dos opciones para añadir dos bloques de comentarios. La opción **Comment Block** nos muestra un cuadro de diálogo donde debemos añadir el comentario:



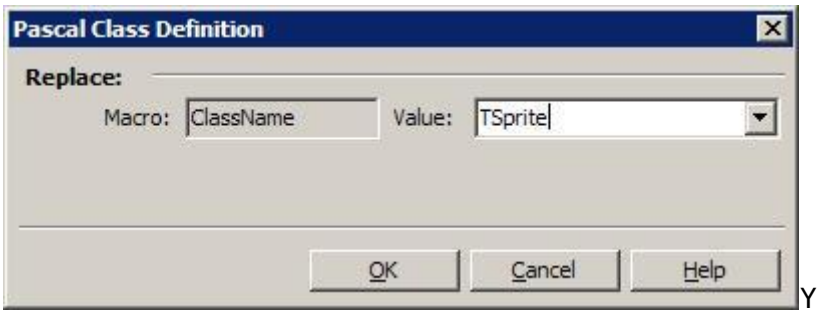
Y mete este comentario justo donde tengamos situado el cursor:

```
//====================================
// Aquí comenzamos la nueva rutina
//====================================
```

Y la opción **Comment Block 2** cambia los separadores por guiones:

```
//------------------------------------
// Aquí comenzamos la nueva rutina
//------------------------------------
```

También podemos crear la definición de una clase seleccionando **Pascal Class Definition** y en esta ventana que aparece escribimos el nombre de la clase:

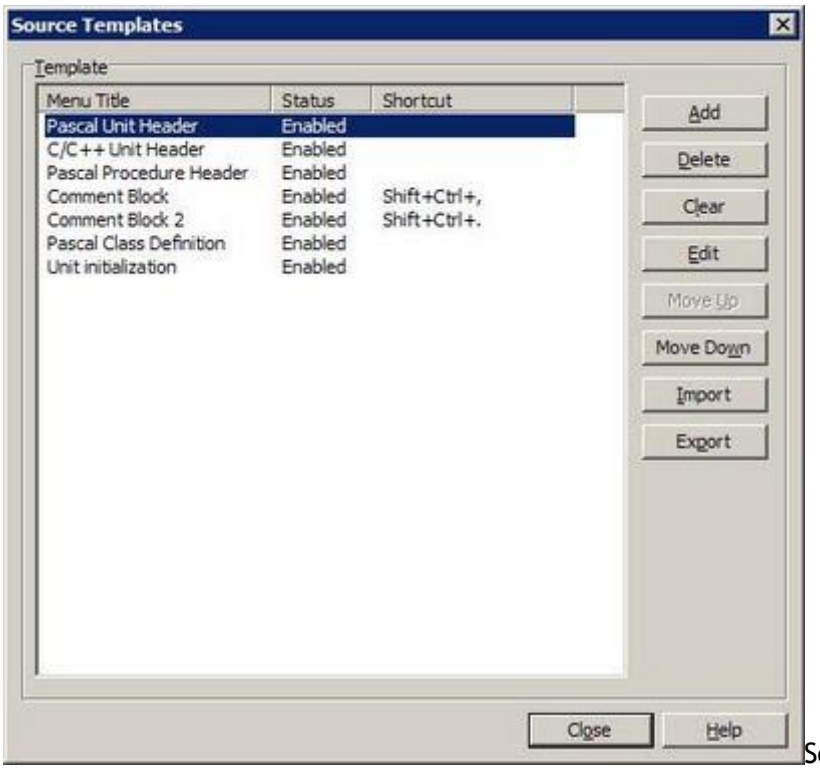


Y nos insertará donde tengamos el cursor todo esto (debería estar en la sección **type**):

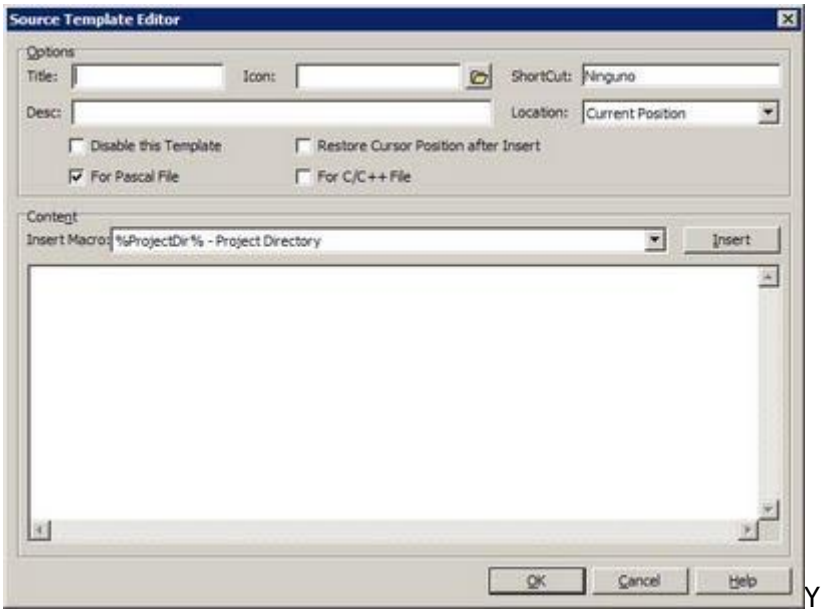
```
{ TSprite }

  TSprite = class(TObject)
  private

  protected

  public
    constructor Create;
    destructor Destroy; override;
  published

  end;
```

Todo lo que sea ganar tiempo esta muy bien. A la última opción **(Unit inicialization)** no le veo mucha utilidad ya que inserta esto:

```
initialization

finalization
```
Y eso ya lo hace Delphi cada vez que creamos una nueva unidad. Lo mejor es la primera opción (**Options**) donde vienen definidas las plantillas que hemos visto:



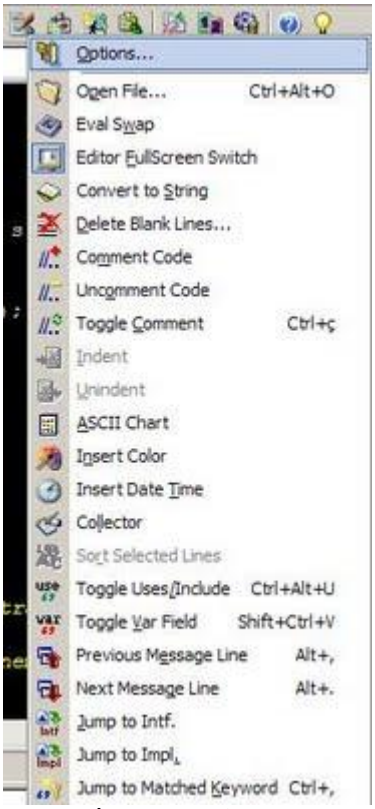
Sólo tenemos que pulsar el botón **Add** e insertar la plantilla que nos venga en gana:



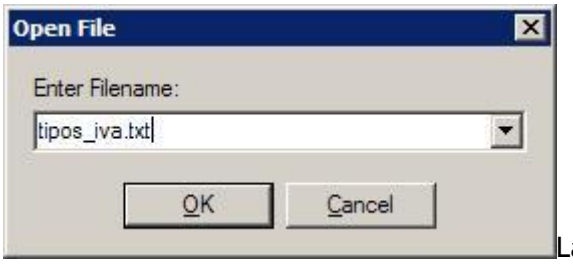
Y aunque Delphi 2007 incorpora esto mismo de las plantillas, yo veo a **cnPack** mucho más fácil de usar.

EL BOTÓN DE MODIFICACIÓN DE CÓDIGO

El siguiente botón tiene también una cantidad inmensa de opciones:



La opción **OpenFile** se utiliza para abrir un archivo que esté situado en la misma carpeta que nuestro proyecto y volcará su contenido en una nueva pestaña de código:



La siguiente opción (**Eval swap**) permite intercambiar el contenido de lo que hay a la izquierda y derecha del símbolo igual. Por ejemplo, si tenemos este trozo de código:

```
Etiquetas[iNumEti].Parent := MarcoPieza;
Etiquetas[iNumEti].Left := x;
Etiquetas[iNumEti].Top := y;
Etiquetas[iNumEti].Caption := s;
Etiquetas[iNumEti].Font.Color := Color;
```

Al seleccionarlo y elegir la opción **Eval Swap** invertirá la igualación:

```
MarcoPieza := Etiquetas[iNumEti].Parent;
x := Etiquetas[iNumEti].Left;
y := Etiquetas[iNumEti].Top;
s := Etiquetas[iNumEti].Caption;
Color := Etiquetas[iNumEti].Font.Color;
```

Es algo muy útil para copiar datos entre tablas, registros, clases y viceversa.

La opción **Editor Fullscreen Switch** lo único que hace es pasar el editor a pantalla

completa o viceversa, como el botón que vimos anteriormente.

La opción **Convert to String** lo que hace es poner entre comillas la cadena que seleccionemos:

```
procedure TFPrincipal.NuevaPantallaClick(Sender:
begin
  ETituloPantalla.Caption := Nueva pantalla;
```

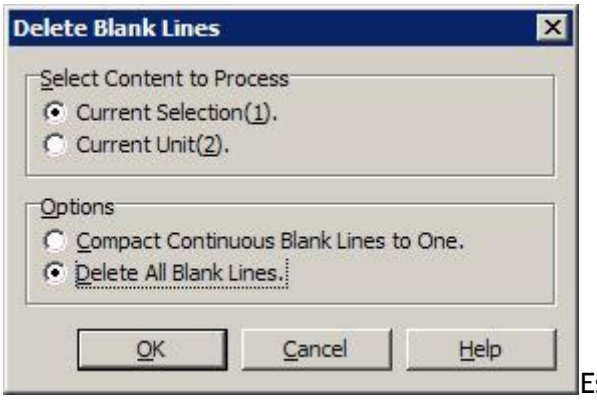
Quedaría así:

```
procedure TFPrincipal.NuevaPantallaClick(Sender:
begin
  ETituloPantalla.Caption := 'Nueva pantalla';
```

Mediante la opción **Delete Blank Lines** podemos coger un trozo de código y eliminar los espacios entre líneas:

```
begin
  // Primero dibujamos el fondo
  Origen.Top := 0;
  Origen.Left := 0;
  Origen.Right := 640;
  Origen.Bottom := 480;

  Destino.Top := 0;
  Destino.Left := 0;
  Destino.Right := 640;
  Destino.Bottom := 480;

  CopyMode := cmSrcCopy;
  CopyRect( Destino, ImagenFondo.Canvas, Origen );
```

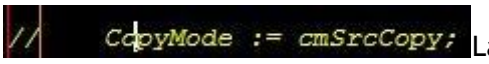
Siempre que seleccionemos en esta ventana la opción **Delete All Blank Lines**:



Esto le dará más vida a nuestra tecla SUPR:

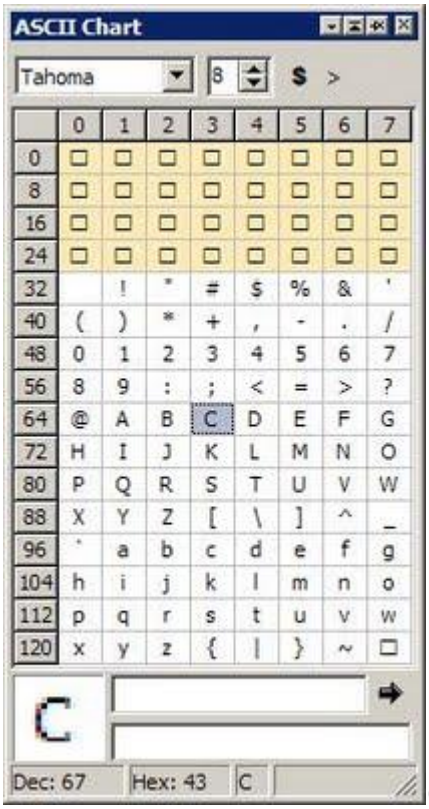
```
begin
  // Primero dibujamos el fondo
  Origen.Top := 0;
  Origen.Left := 0;
  Origen.Right := 640;
  Origen.Bottom := 480;
  Destino.Top := 0;
  Destino.Left := 0;
  Destino.Right := 640;
  Destino.Bottom := 480;
  CopyMode := cmSrcCopy;
  CopyRect( Destino, ImagenFondo.Canvas, Origen );
  CopyMode := cmSrcCopy;
  CopyRect( Destino, ImagenFondo.Canvas, Origen );
```

Luego vienen opciones más sencillas como **Comment code**, **Uncomment code** o **Toggle comment** que convierten la línea donde tenemos el cursor en una línea de comentario o bien lo vuelven a quitar:

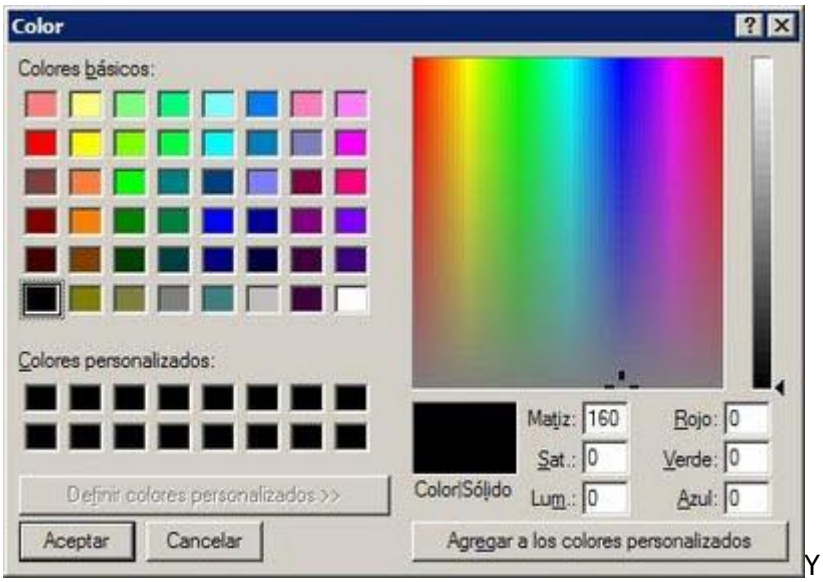


Las opciones **Ident** y **Unident** cogen el código seleccionado y lo meten dos espacios hacia dentro o hacia fuera respectivamente. Esto no tiene mucha utilidad ya que para ello tenemos las combinaciones de teclas de funciones que trae el mismo Delphi: **CTRL + K + I** y **CTRL + K + U**.

La opción **ASCII Chart** si viene muy bien para averiguar los números de caracteres:

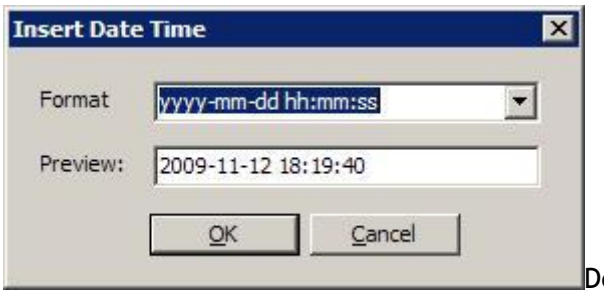


Otra opción que siempre hemos echado en falta en Delphi es saber un número de color específico sin tener que ir a un programa de diseño gráfico. Pues ahora seleccionando **Insert color** primero nos aparecerá la típica ventana para elegir el color:

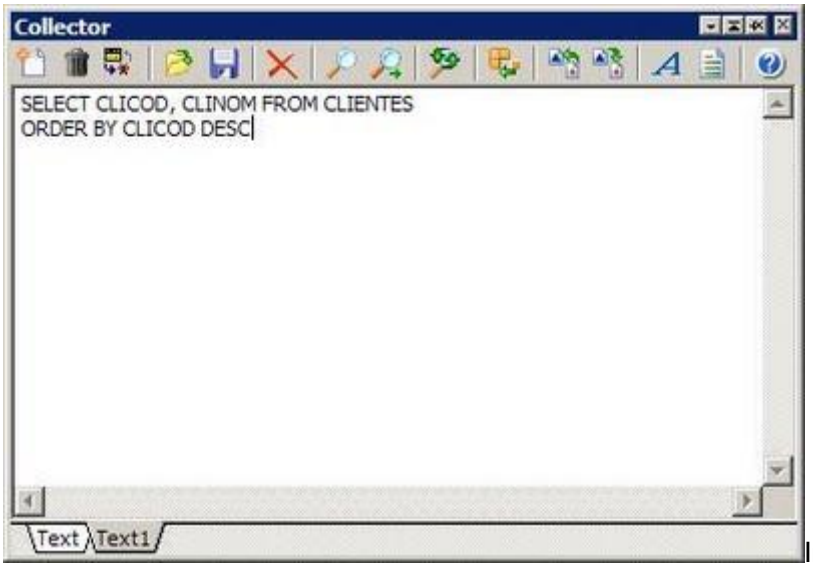


Y luego lo meterá donde tengamos puesto el cursor:

$00D5D500 Igualmente podemos insertar un fecha y una hora con la opción **Insert Date Time**:



Después viene la opción **Collector** que es un pequeño bloc de notas donde podemos cargar o guardar cualquier información importante para que la tengamos accesible en cualquier momento:



Incluso podemos crear varias pestañas donde guardar la información clasificada. Si cerramos esta ventana y la volvemos a abrir seguirá ahí la información escrita.

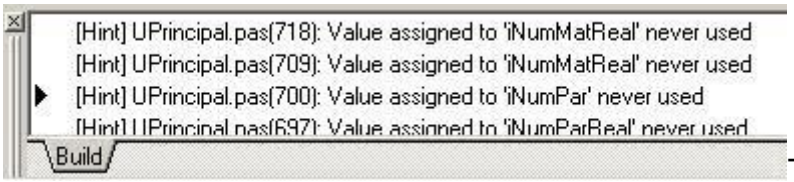
También está la opción de poder seleccionar varias líneas y ordenarlas alfabéticamente mediante la opción **Sort selected lines**. Viene bien para ordenar todas las variables

definidas de una clase cuando es muy grande y hay que volver a buscarlas.

Luego le sigue la opción **Toggle uses/include** que nos sitúa directamente en la línea donde se encuentra la sección**uses** para poder seguir añadiendo una nueva unidad. Esta opción hay que utilizarla con las combinaciones de teclas CTRL + ALT + U para que sea realmente efectiva. Aunque yo prefiero utilizar la opción de Delphi **File** -> **Use unit**.

La otra opción que le sigue (**Toggle Var Field**) si que viene bien porque nos sitúa en la sección de definición de variables cuando estamos situados dentro de un procedimiento o función. Su combinación de teclas es CTRL + SHIFT + V. Aunque lo mejor sería utilizar las opciones de **Refactoring** de Delphi 2007 para declarar variables automáticamente.

Las opciones **Previous Message Line** y **Next Message Line** se utilizan para retroceder o avanzar a través de la ventana de mensajes de Delphi. Además, salta a la línea donde está el problema:



También tienen sus teclas rápidas: **ALT + . y ALT + ,**

Y después están las opciones para saltar a la interfaz o a la implementación mediante **Jump to Int**, y **Jump to Imp**.

Debido a que estas barras de herramientas que estamos viendo son bastante extensas, continuaré en el siguiente artículo.

Pruebas realizadas en Delphi 7 y Delphi 2007.

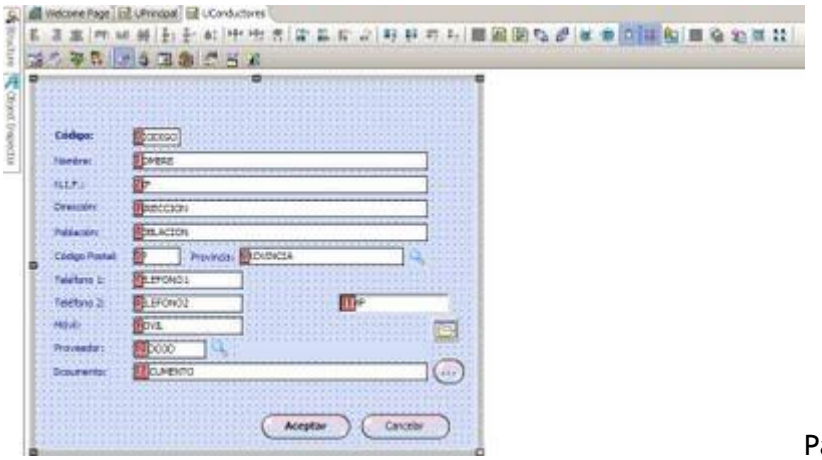
06 noviembre 2009

El experto cnPack (2)

Vamos a entrar en profundidad con todas las características que tiene este fantástico experto para Delphi comenzando con la barra de herramientas que aparece en la parte superior del editor cuando estamos modificando un formulario.

ALINEAR LOS COMPONENTES DEL FORMULARIO

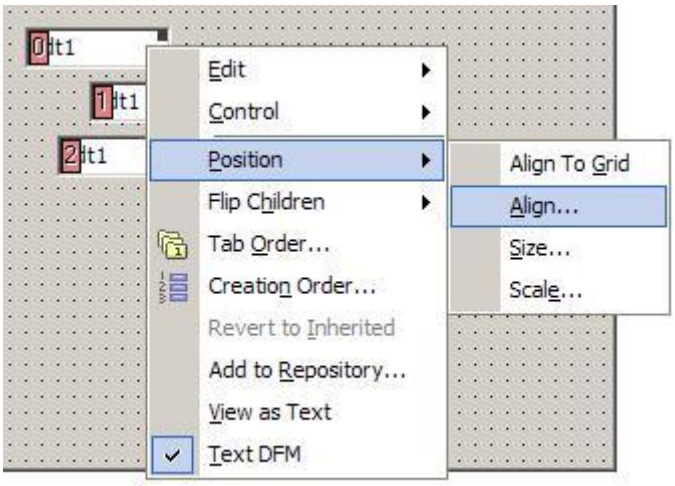
Al editar un formulario en Delphi 2007 nos aparece una barra con tantos botones en la parte superior que debemos esconder los paneles laterales:



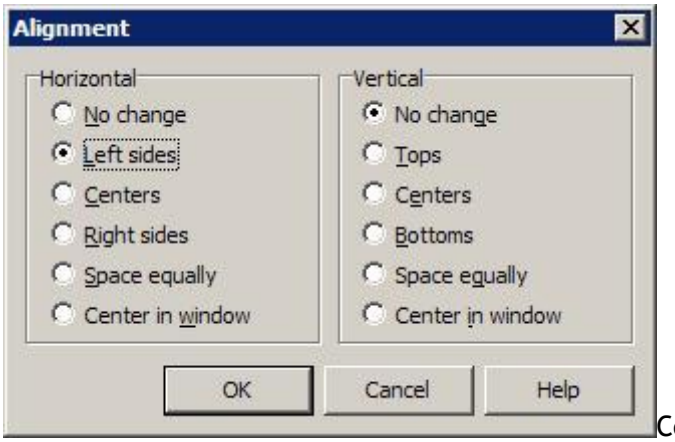
Para verlo detenidamente

hay que hacerlo por grupos de izquierda a derecha. El primer grupo de botones que tenemos a la izquierda es el encargado de alinear dos más componentes gráficos que tengamos seleccionados:

 Estos botones nos evitar tener que utilizar la alineación de componentes de Delphi que resulta más incómoda porque hay que sacarla con el botón derecho del ratón:



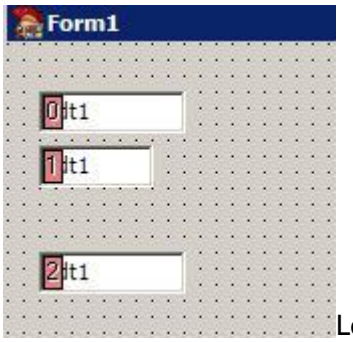
Y para colmo luego hay que utilizar este formulario:



Con estos tres botones que tenemos en la barra de **cnPack** podemos alinear a la izquierda, a la derecha o centrarlos. Los siguientes tres botones hacen exactamente lo mismo pero con la alineación vertical:

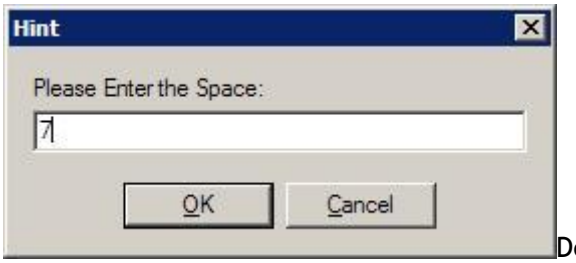
 Los siguientes tres botones son los encargados de espaciar verticalmente los componentes seleccionados manteniendo exactamente igual el espacio entre los mismos:

 Aunque hay una diferencia respecto al IDE de Delphi: el primer botón permite realizar un espaciado de componentes diciéndole nosotros cuantos pixels queremos exactamente de separación entre componentes. Por ejemplo, supongamos que queremos espaciar verticalmente estos componentes con 7 pixels entre ellos:

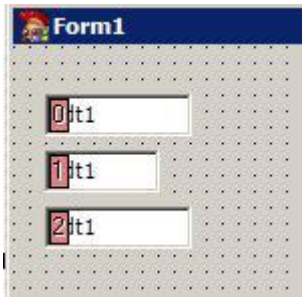


Los seleccionamos y pulsamos este botón:


Entonces se abrirá este cuadro de diálogo preguntándonos cuantos pixels queremos de separación:



Después de pulsar **Ok** cumplirá con su objetivo a la perfección:



Esto es ideal cuando se trabaja con un grupo amplio de programadores y cada uno monta los formularios como quiere. El jefe de proyecto puede obligar a los programadores a que diseñen la interfaz con las mismas medidas.

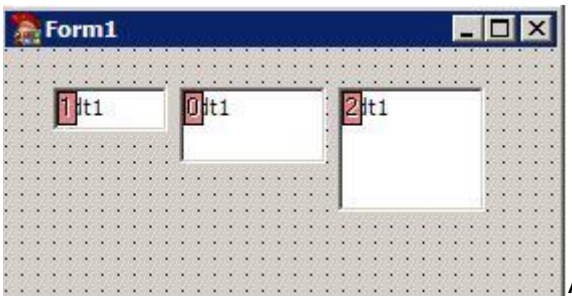
Los otros dos componentes son para que exista el mismo espacio entre todos los seleccionados o para pegarlos todos quitando espacios:


Tenemos exactamente los mismos tres botones pero para la separación horizontal:


REDIMENSIONAR LOS COMPONENTES DEL FORMULARIO

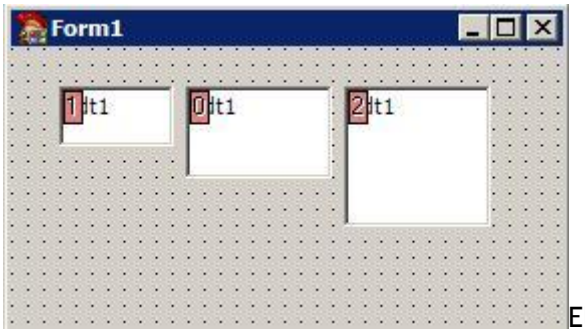
Los siguientes cuatro botones serán los encargados de redimensionar los controles seleccionados según nuestras preferencias:


Los dos primeros botones incrementan o decrementan el tamaño de los componentes en unidades de la rejilla. Por ejemplo, si tenemos estas dimensiones:



Al pulsar este botón:


Incrementará su tamaño vertical por su parte inferior:



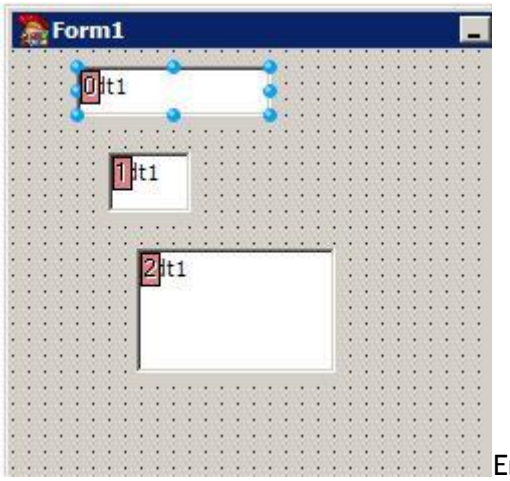
El botón de la lado hace el efecto contrario. Luego está este botón que amplia el tamaño de todos al más grande (verticalmente):


Y el último los disminuye verticalmente al más pequeño:


Los siguientes cuatro botones cumplen el mismo cometido pero horizontalmente:

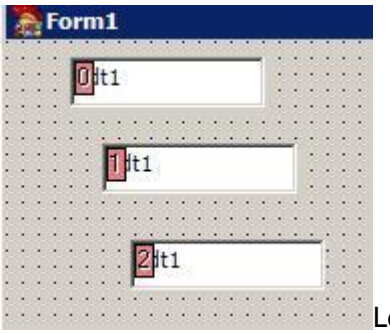

Pasemos al siguiente grupo:


El primer botón es bastante interesante porque permite establecer el tamaño horizontal y vertical de los componentes seleccionados a partir del primero seleccionado. Por ejemplo, vamos a hacer que este los dos componentes de abajo tengan el mismo tamaño que el primero que hemos seleccionado:



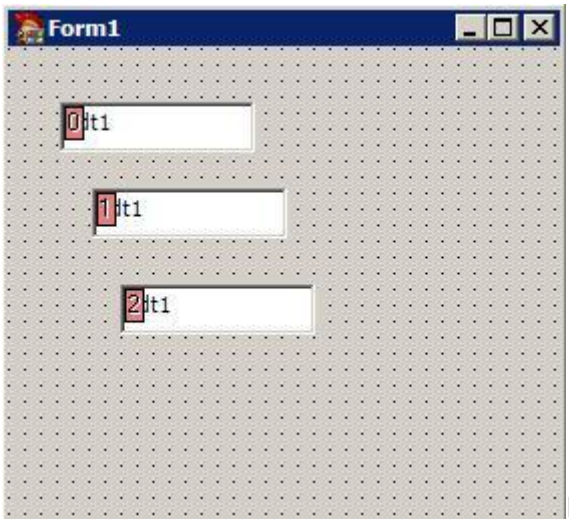
En vez de seleccionar los tres componentes a la vez, los seleccionamos secuencialmente y luego pulsamos el botón:


Y los igualará en tamaño al primer componente:

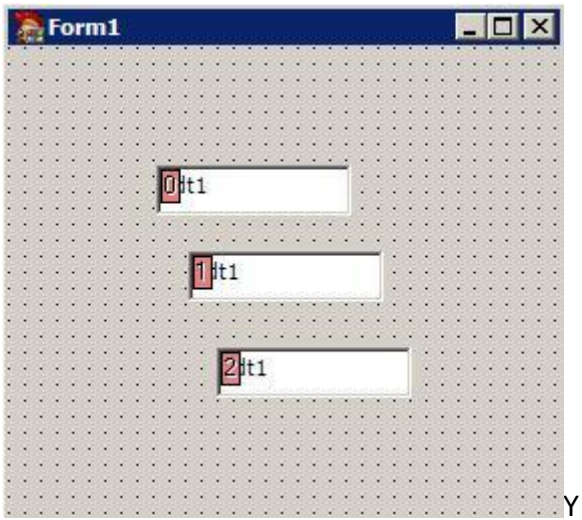


Los siguientes dos botones centran en el formulario todos los componentes seleccionados pero manteniendo su posición original entre ellos:


Si originalmente tenía así el formulario:



Después de seleccionar los tres componentes y pulsar ambos botones lo centrará todo:



Y los dos últimos botones de este grupo se encargan de traer al frente o enviar al fondo los componentes seleccionados:


ALINEAR COMPONENTES RESPECTO A LA REJILLA

Vamos con el siguiente grupo:

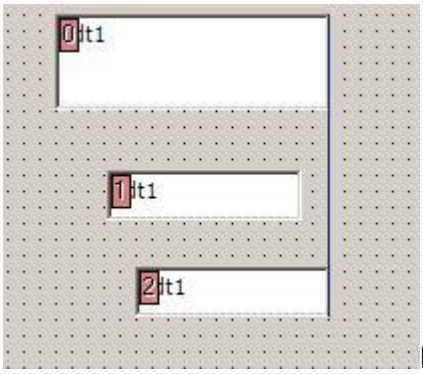

El primer botón se encarga de ajustar la posición de la esquina superior izquierda del componente seleccionado a la cuadrícula de la rejilla:

 Y el segundo ajusta a la rejilla el tamaño horizontal y vertical del componente:

 El siguiente botón también es muy interesante porque funciona en estado apagado o encendido. Cuando esta activado, al redimensionar un componente con el ratón veremos que se ajusta a la cuadrícula:

 Sin embargo, podemos ampliar el tamaño del mismo de píxel en píxel a la vieja usanza con el teclado: tecla SHIFT + CURSORES. Este otro botón activa o desactiva la guía que permite alinear unos componentes respecto a otros:

 Es la guía que incorporan por defecto las últimas versiones de Delphi:



Este botón no está disponible en Delphi 7. Y el último botón de este grupo bloquea los controles del formulario para que no puedan ni moverse ni redimensionarse:

SELECCIONANDO COMPONENTES

Vamos con el último grupo de la barra superior:

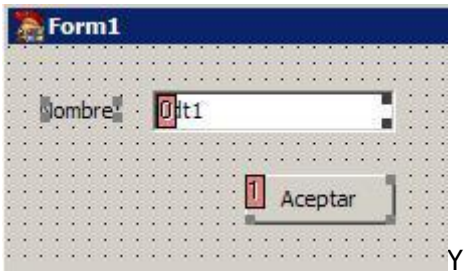
 El primer botón nos sirve para seleccionar el formulario donde estamos situados cuando el foco lo tenemos sobre otro componente:

 Esto viene bastante bien cuando diseñamos nuestra propia piel para el formulario y tenemos tantos controles encima del mismo que no podemos ni hacer clic para poder acceder a sus eventos a través del inspector de objetos. Si el formulario ya se encuentra seleccionado entonces se desactiva este botón.

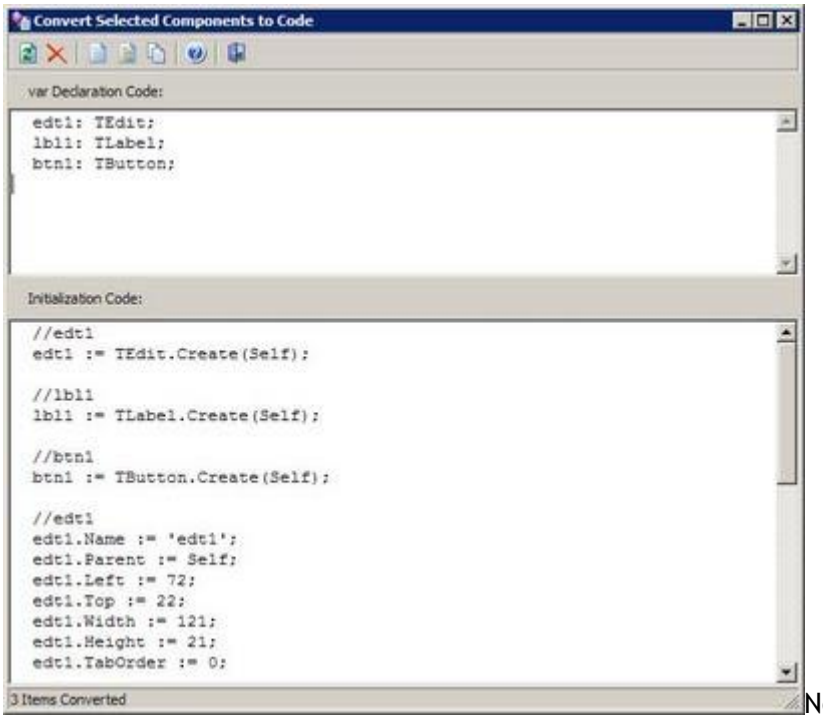
El segundo botón copia los nombres de los componentes (propiedad **Name**) seleccionados al portapapeles:

 El tercer botón si es bastante interesante. Nos devuelve el código fuente de los componentes seleccionados como si los hubiésemos creado por código:

 Por ejemplo, al seleccionar estos componentes:



Y pulsar luego dicho botón aparecerá este formulario:



Nos devuelve el código de creación de los componentes con todas sus propiedades, algo muy útil cuando queremos esconder algún componente del archivo DFM como por ejemplo: registrar el producto, introduzca la clave de licencia, etc. Sólo hay que copiar este código en el evento **OnCreate** del formulario y eliminar los componentes del mismo.

El cuarto botón es bastante curioso:


Hace invisibles en tiempo de diseño los componentes no visuales como por ejemplo un **TIBQuery**:

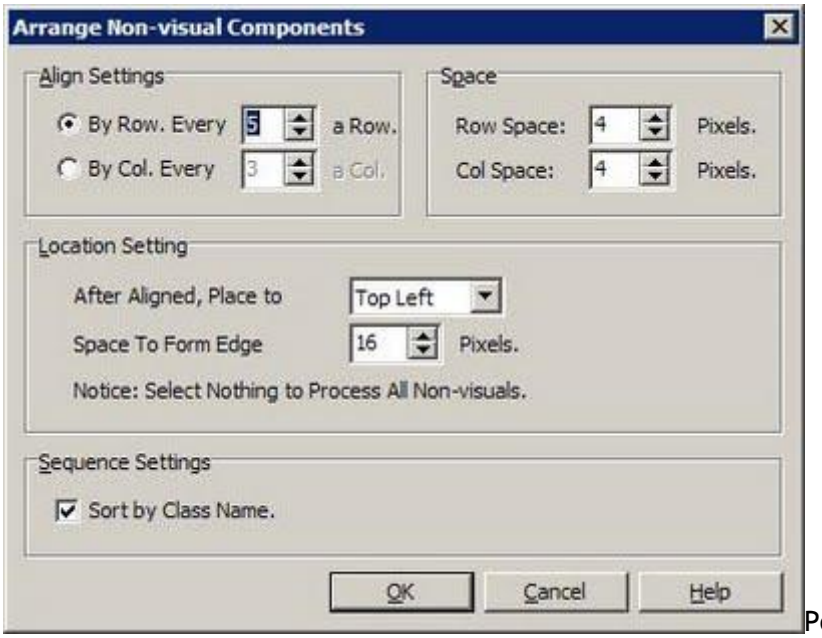
Al pulsar este botón pasa esto:

Para que se vuelva invisible primero hay que seleccionar otro componente. La única utilidad que le veo a esto es cuando queremos mostrar a otros programadores o clientes como va el diseño de nuestro formulario sin que se vean estos componentes. Si volvemos a pulsar este botón se harán visibles de nuevo. Esto no afecta para nada a la aplicación, sólo a la fase de diseño.

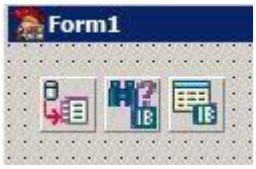
El último botón si que tiene bastante utilidad:


Cuando tenemos varios componentes no visuales desordenados en el formulario:

Podemos reorganizarlos todos pulsando este botón y nos aparecerá un formulario para preguntarnos donde los queremos colocar y la separación entre los mismos:



Por defecto los coloca en la esquina superior izquierda del formulario:



Continuando con nuestro tema de selección de componentes vamos a comenzar con el primer grupo de la barra inferior:


El primer botón activa la selección múltiple:

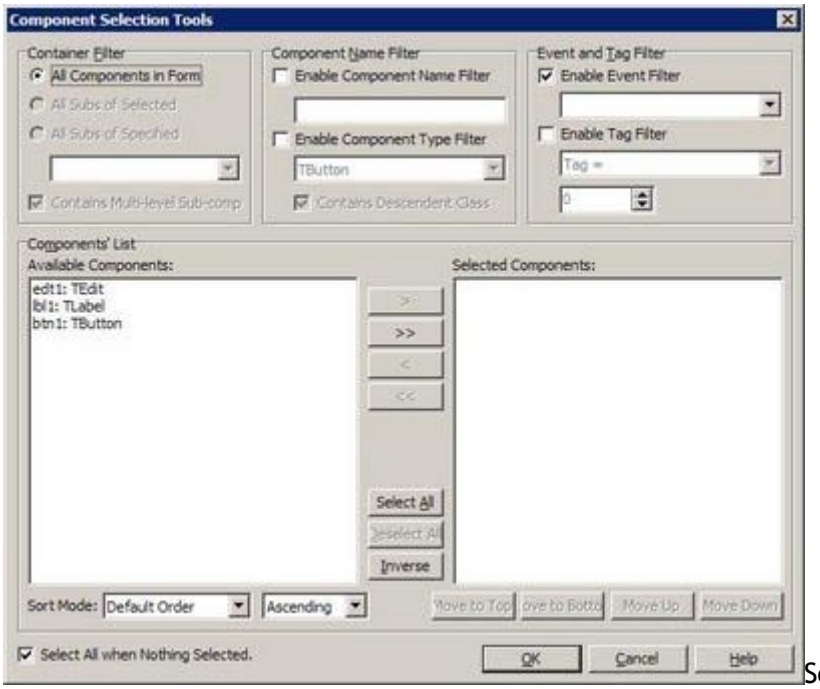

Al pulsarlo nos aparece este mensaje:

Cuya traducción es esta:

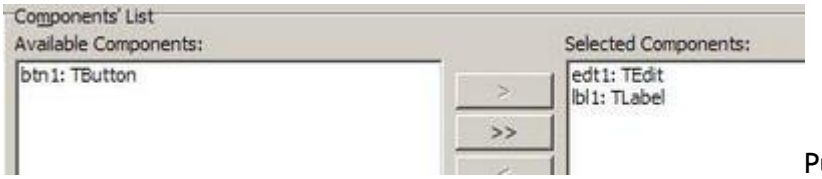
Bienvenido a selector de componentes!

Este asistente puede seleccionar componentes en forma actual por nombre, tipo o otras condiciones a pesar de su visibilidad. Por ejemplo, si queremos cambiar la altura de todos los botones de 25 a 21, podemos seleccionar todos los botones y cambiar su altura una vez. Se admiten varios filtros y clasificación. A veces es útil modificar los pedidos de componentes seleccionados.

Después de pulsar el botón **Continue** nos aparece este formulario:



Solo hay que pasar de la lista de la izquierda a la lista de la derecha los nombres de los componentes que queremos seleccionar:

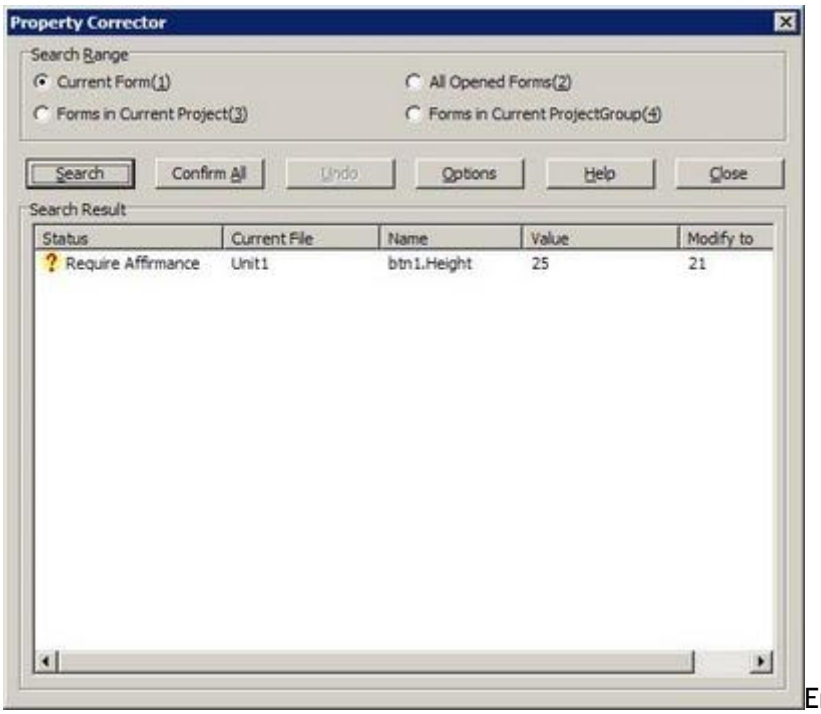


Pulsamos el botón **Ok** y volverá al formulario seleccionando los componentes que hemos elegido. Ya podemos ir

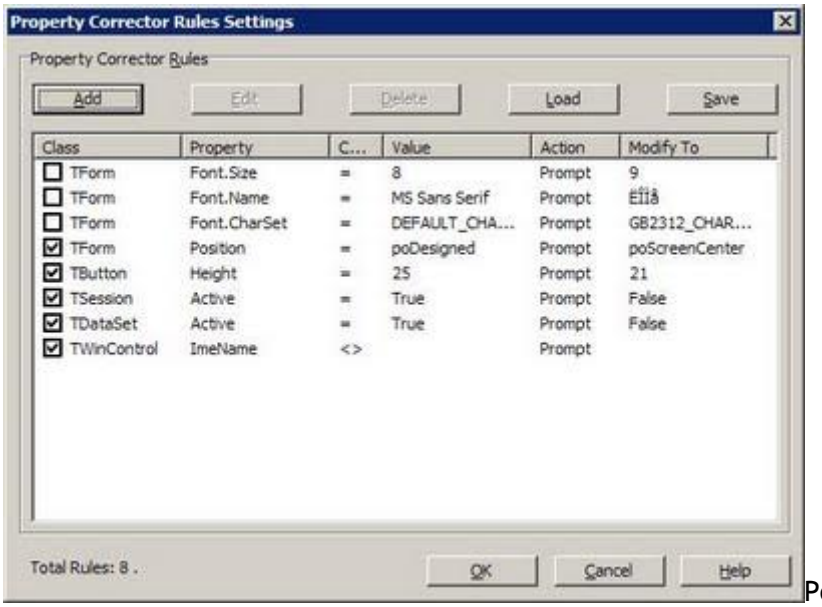
al inspector de objetos a cambiar sus propiedades.

El segundo botón es un corrector de propiedades donde podemos configurar como queremos que ciertas propiedades de los componentes se cumplan:

Un ejemplo que lleva es que todos los botones (**TButton**) del formulario deberían tener una altura de 21 pixels. Al pulsar este botón aparecerá esta ventana en la que debemos pulsar el botón **Search** para buscar componentes que no se ajustan a nuestras reglas:



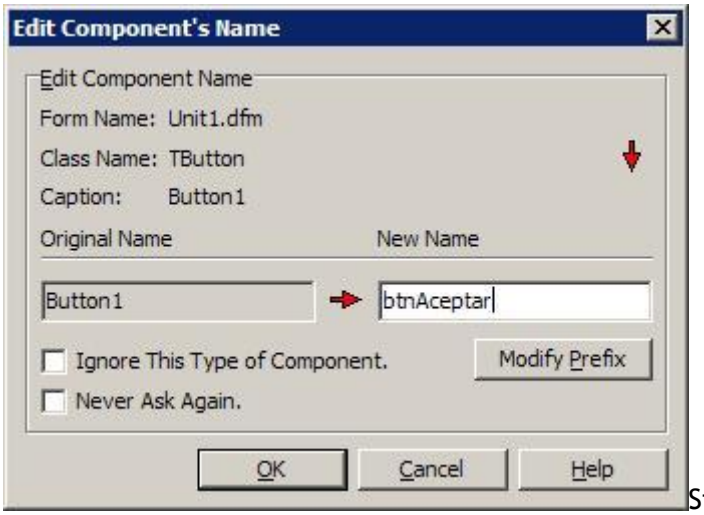
En este caso nos dice que el botón tiene una altura de 25 en vez de 21. Debemos cambiarlo nosotros manualmente en el inspector de objetos (hubiese sido ideal que lo hiciera él por nosotros). ¿Dónde están esas reglas? En el botón**Options**:



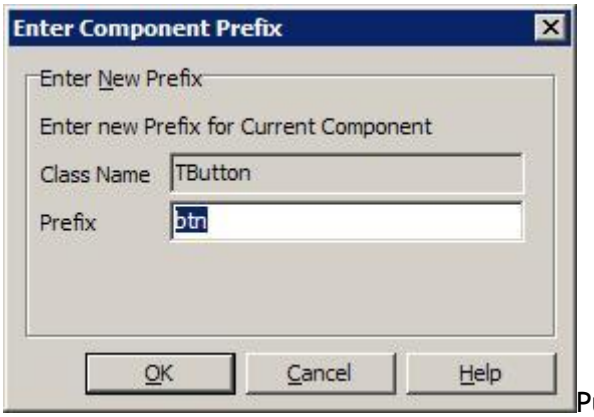
Podemos añadir cualquier componente y establecer el valor de debería tener cierta propiedad. Es otra herramienta ideal para forzar a los programadores a hacer las cosas del mismo modo que el resto del equipo. Al pulsar el botón**Add** debemos rellenar todo esto:

RENOMBAR COMPONENTES

Habréis visto que cuando copiamos o insertamos un componente en el formulario aparece un nuevo asistente que nos indica el prefijo que debemos dar al componente según su clase (notación húngara). Por ejemplo, al insertar un botón **Aceptar**, su nombre debería ser **btnAceptar**:



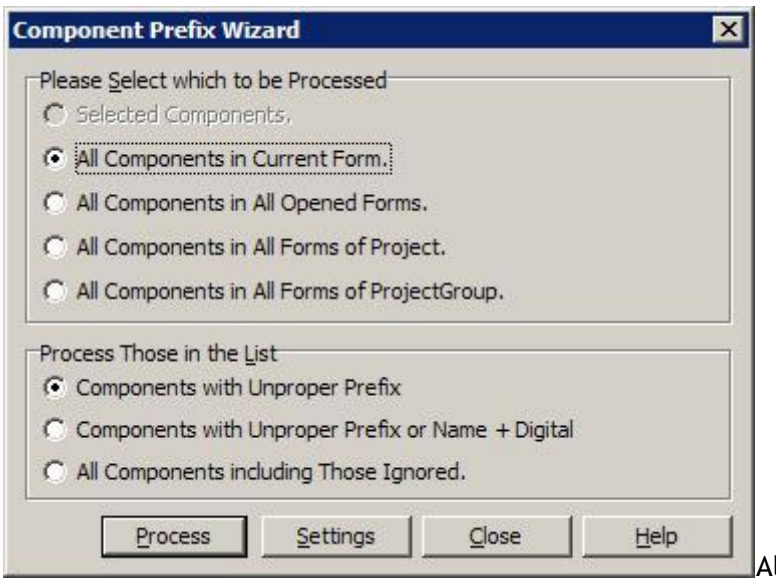
Si en la misma ventana pulsamos el botón **Modify Prefix** podemos elegir otro prefijo para esta clase:



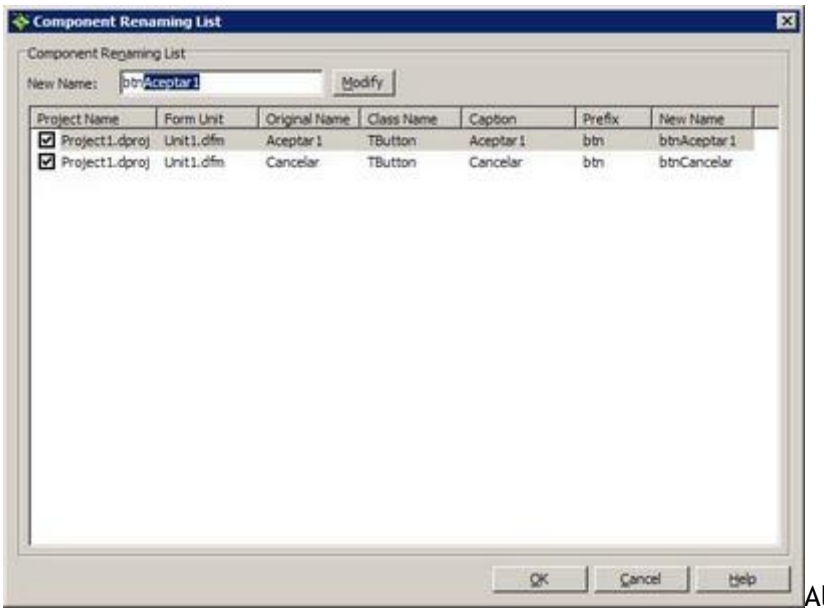
Pues bien, si a partir de ahora queremos utilizar esta metodología y no queremos pegarnos la paliza cambiando el nombre de cada componente podemos utilizar este botón:

 Al pulsarlo nos describe primero para que se utiliza:

Al pulsar el botón **Continue** nos aparece otro cuadro de diálogo donde podemos especificar si queremos cambiar solo este formulario o todos los formularios del proyecto:

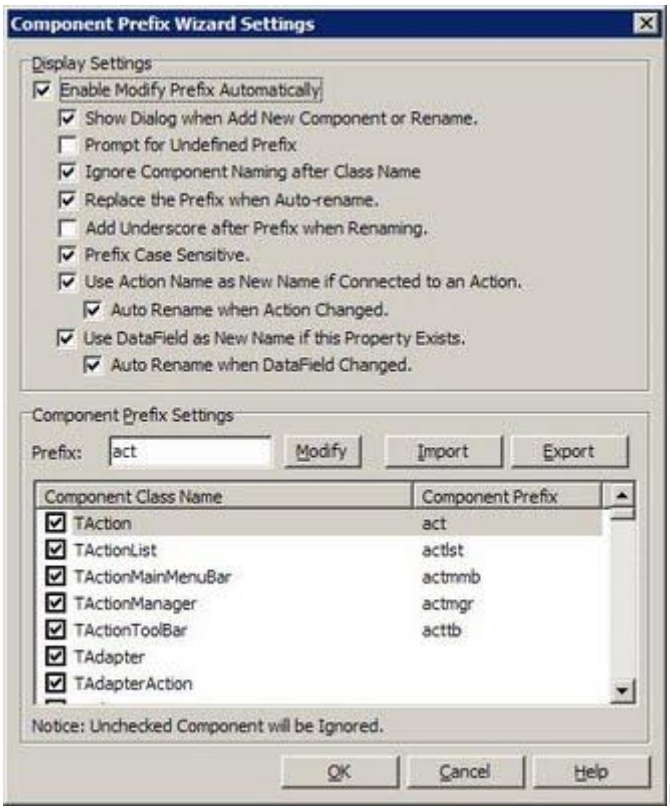


Al pulsar el botón **Process** veremos una lista con los componentes que ha encontrado que no se ajustan a esta normativa y el nuevo nombre que les va a dar:



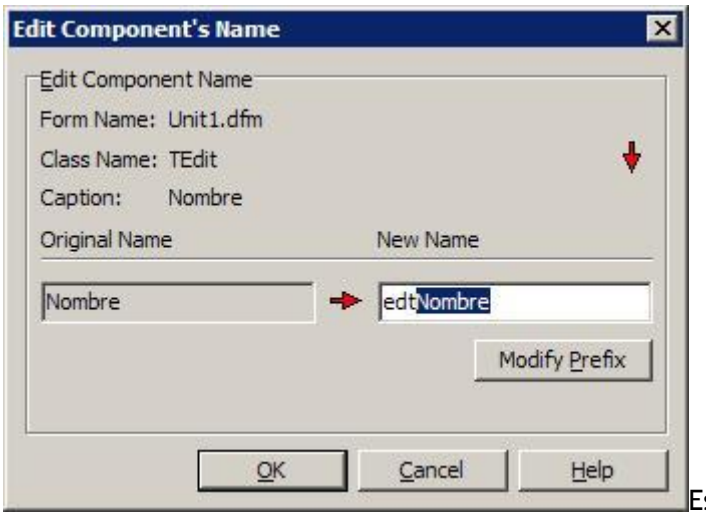
Al pulsar **Ok** hará todo el trabajo por nosotros. Lo único que no me hace gracia de este sistema este también me cambia la propiedad **Caption** de los mismos:

Y he estado mirando sus opciones (**Settings**) pero no he encontrando nada que desactive esto:



El último botón de este grupo se utiliza para renombrar el componente seleccionado sin tener que recurrir al inspector de objetos:


Nos mostrará la ventana que vimos anteriormente:

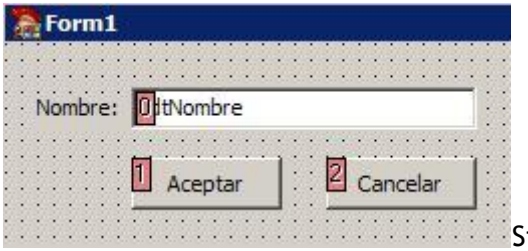


Esto sería para hacerlo uno a uno. Y también nos hace la faena de cambiarnos las propiedades **Caption** o **Text**, según el componente.

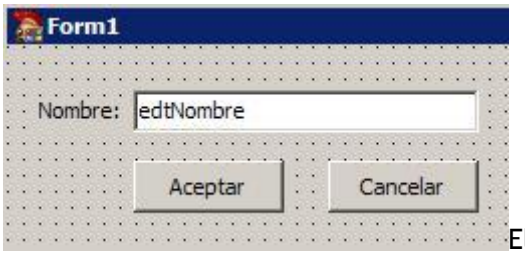
CONFIGURAR LA TABULACIÓN

Vamos con el penúltimo grupo:


El primer botón que aparece seleccionado sirve para activar o desactivar el número de tabulación que aparece en cada componente del formulario:

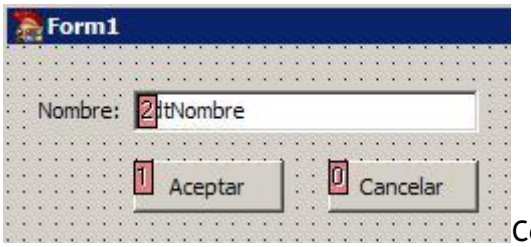


Si lo desactivamos se queda como cualquier Delphi sin **cnPack**:



El siguiente botón es bastante bueno para reordenar el orden de tabulación de los componentes:


Si tenemos el orden de tabulación desordenado:

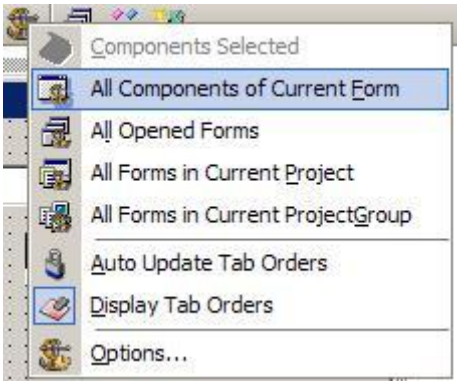


Con sólo pulsar este botón nos quitará mucho trabajo:

Aunque parezca que no funciona al principio, pinchamos sobre cualquier componente y refrescará el formulario. Si este botón lo dejamos activado se encargará automáticamente de hacer la tabulación de izquierda a derecha y de arriba abajo.

El siguiente botón reordena también las tabulaciones pero no se queda activado:

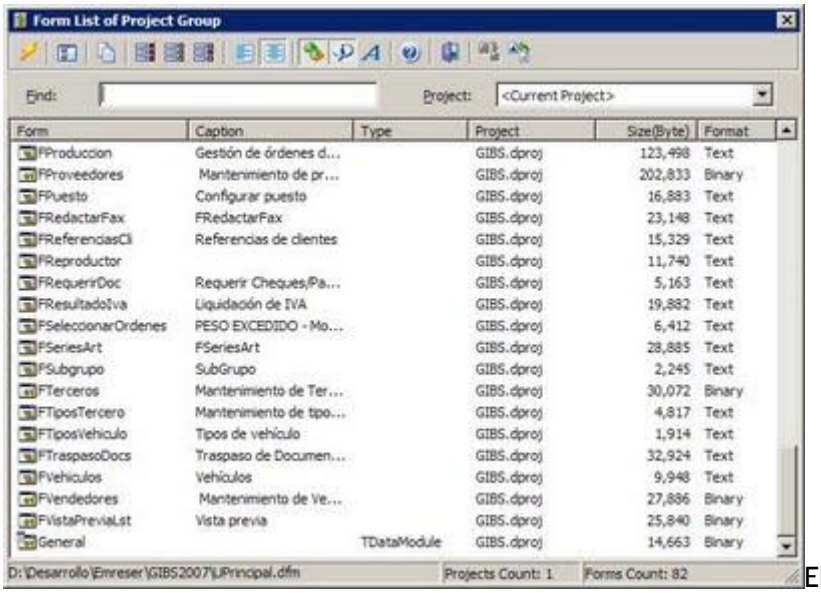

Y con último botón de este grupo podemos arreglar la tabulación te todos los formularios del proyecto:



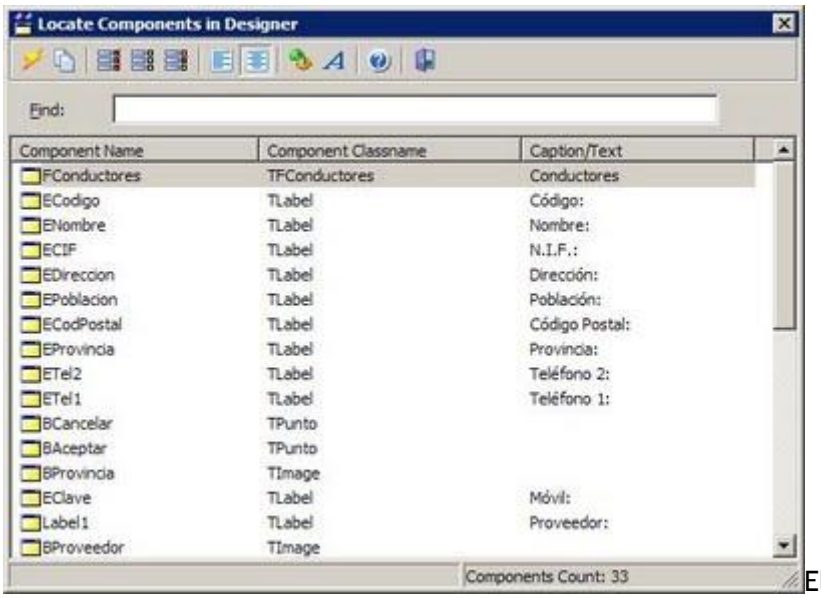
BUSCANDO FORMULARIOS Y COMPONENTES

Por fin llegamos al último grupo:


El primer botón es para la búsqueda de formularios en el proyecto:

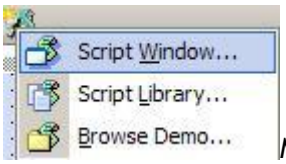


El segundo botón si que es realmente útil para buscar componentes en el formulario actual (sobre todo cuando tenemos más capas que una cebolla):

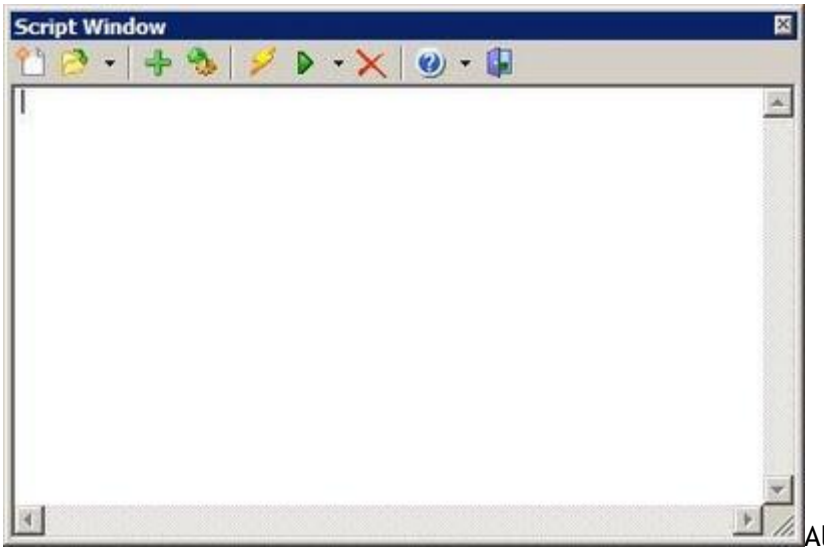


El último botón es el más avanzado de todos:

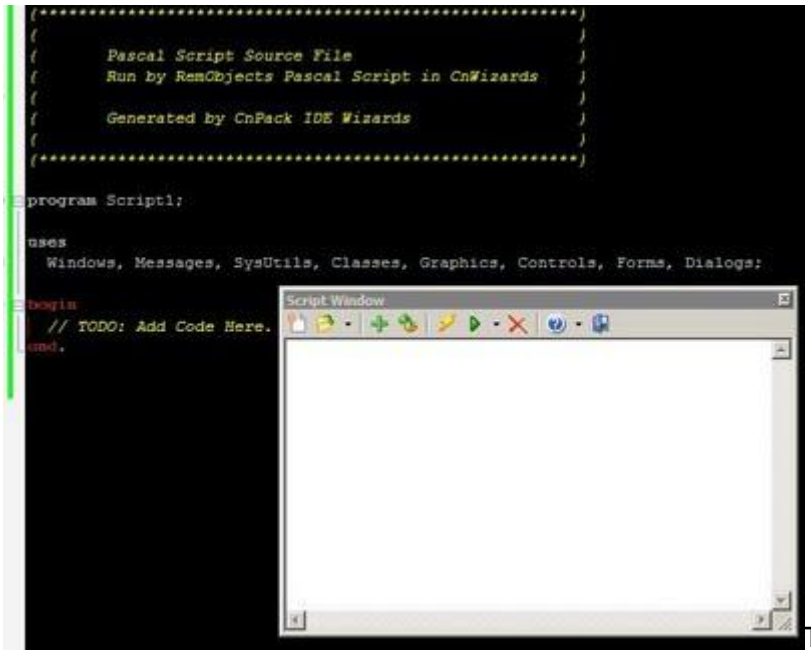

Al pulsarlo aparece un menú contextual con estas tres opciones:



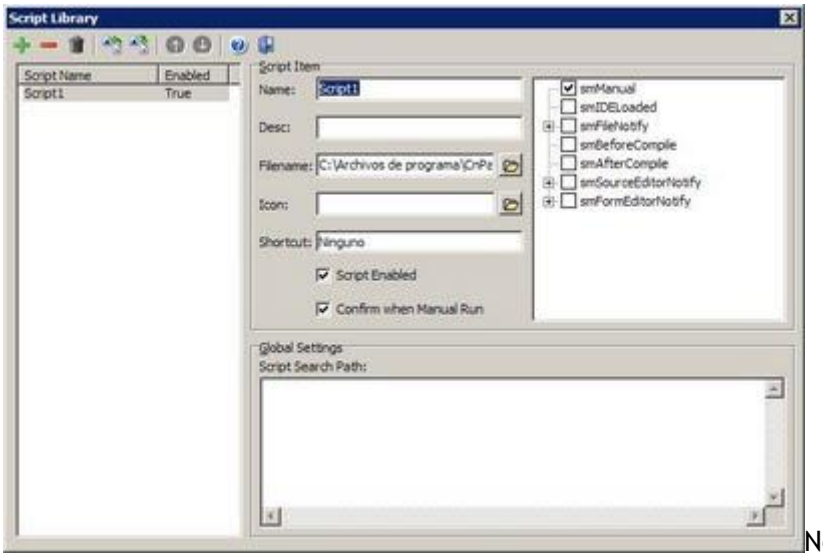
Mediante la opción **ScriptWizard** podemos incrustar dentro de nuestro programa un lenguaje script llamado**RemObjects Pascal Script** que permite modificar el comportamiento de nuestro programa sin tener que compilar cada vez:



Al pulsar el botón **New a script** nos creará una unidad nueva como si se tratase de un programa independiente:



También podemos pulsar el botón + para insertar script predefinidos que lleva dentro:



No voy a hablar sobre como crear estos script porque la ayuda que lleva el mismo es bastante buena y con muchos ejemplos. Quizás si me sobra tiempo, al final de esta serie de artículos pueda abarcar este tema, ya que tiene muy buena pinta. La opción **Browse Demo** nos muestra gran cantidad de ejemplos que han añadido los creadores de este experto.

En el próximo artículo veremos la barra de herramientas que se muestra cuando estamos editando código fuente.

Pruebas realizadas en Delphi 7 y Delphi 2007.

30 octubre 2009

El experto cnPack (1)

Pese a que las últimas versiones de Delphi (sobre todo Delphi 2010) incorporan gran cantidad de características en su IDE, hay que reconocer que algunas de ellas todavía no están a la altura de otros expertos realizados por terceros, como por ejemplo EurekaLog.

cnPack incorpora tal cantidad de ampliaciones al IDE de Delphi que le dan una gran

fortaleza al programador con características que se encuentran en otros entornos como Visual Studio. Y lo mejor de todo: es completamente gratuito y funciona para todas las versiones de Delphi (incluidas de la 7 hacia atrás).

DONDE DESCARGAR CNPACK

Aunque los programadores de este experto son chinos, si entramos en su dominio nos redireccionará automáticamente a una página en inglés:

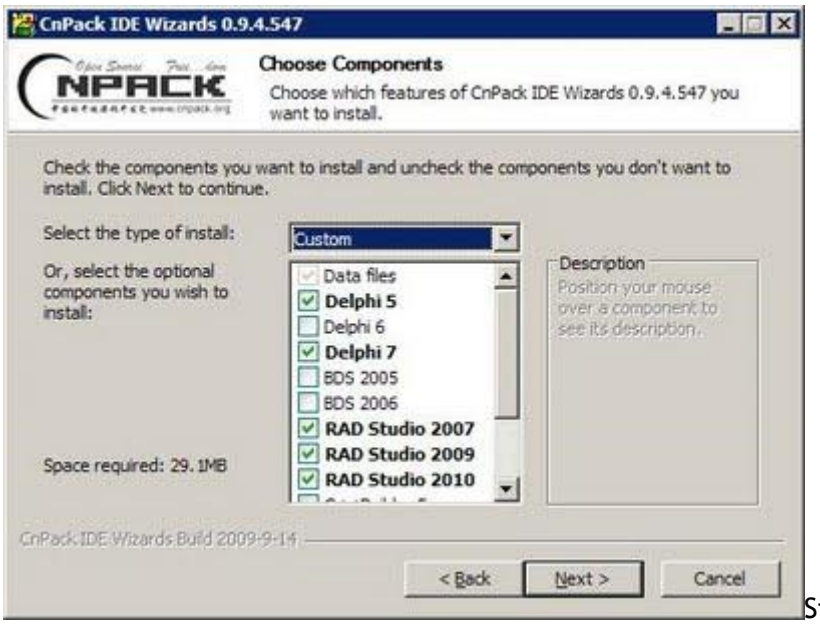
http://www.cnpack.org

He marcado en un círculo rojo la última versión que hay disponible a la fecha de este artículo.

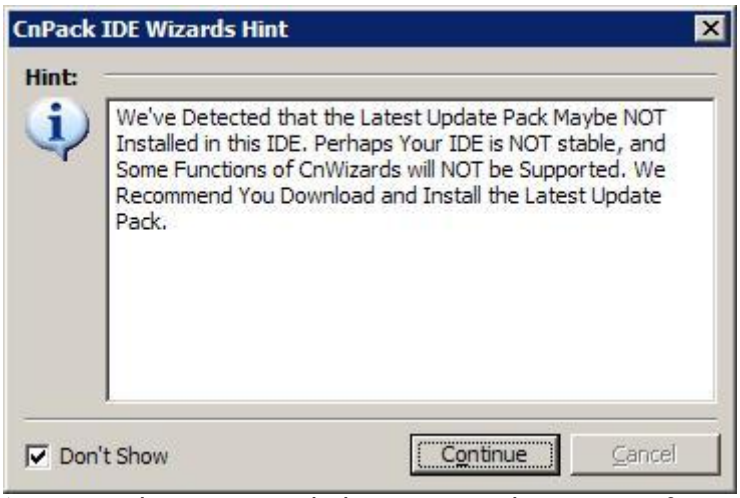
INSTALAR CNPACK

La instalación tiene un tamaño de 10,8 MB y debemos cerrar todas las versiones de Delphi que tengamos abiertas antes de continuar:

En uno de los pasos de la instalación nos marcará en negrita todas las versiones de Delphi que ha encontrado instaladas en nuestro equipo y por defecto estarán todas seleccionadas:



Si la versión que tenéis de Delphi 2007 no tiene instalados los últimos Updates puede que os aparezca un mensaje advirtiendo que el IDE podría ser inestable:

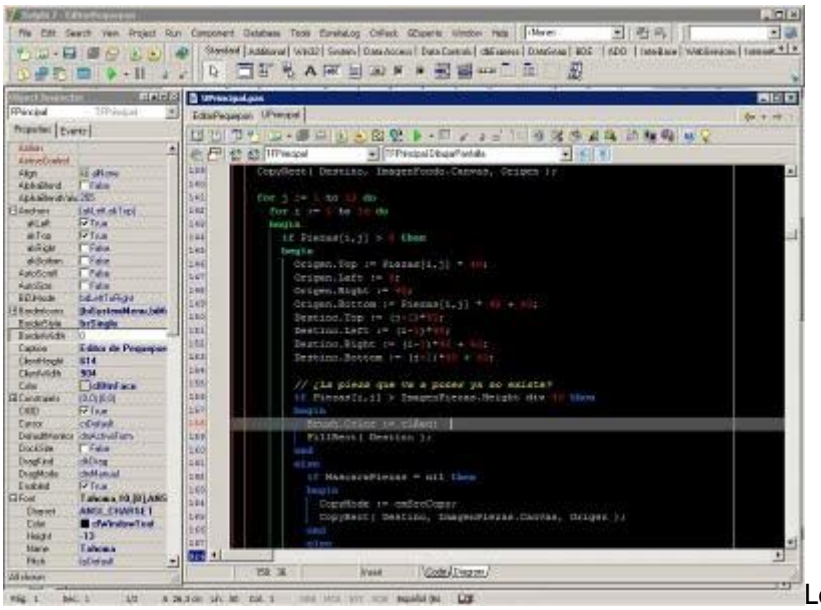


Aunque yo lo tengo instalados estos parches y va perfectamente. Se me ha colgado alguna vez el IDE pero no creo que haya sido precisamente por el **cnPack**. Antes también lo hacía, así que…

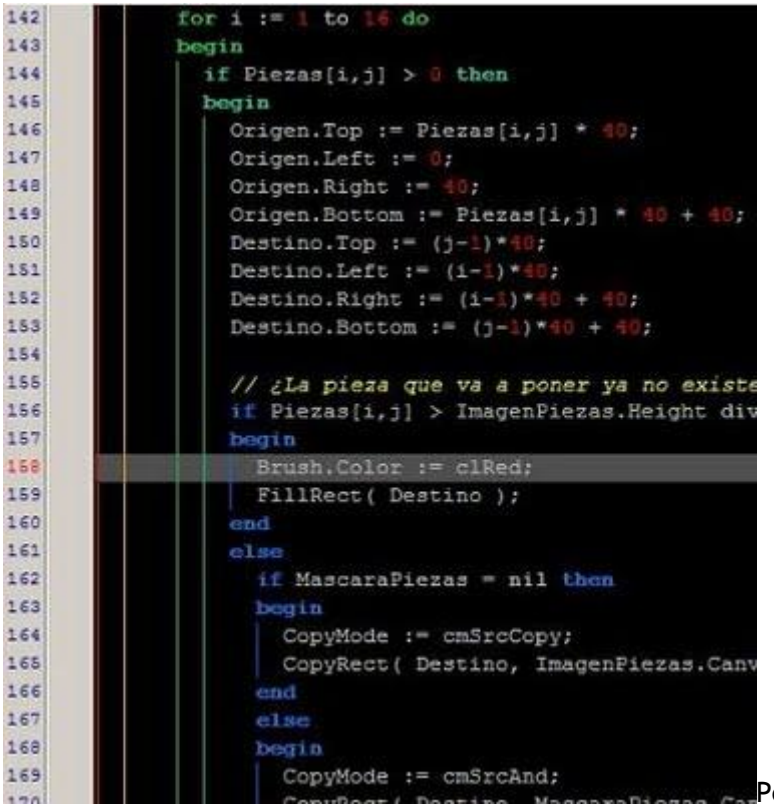
TRABAJANDO CON CNPACK

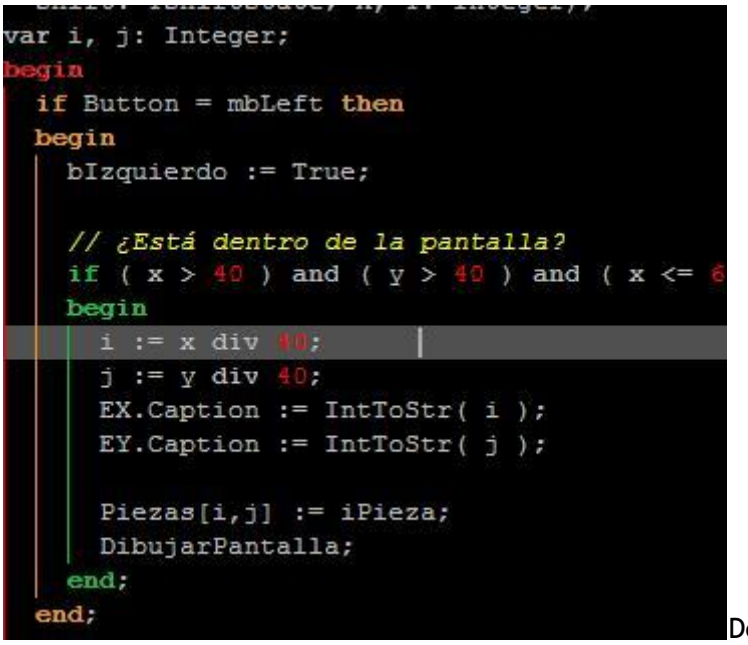
Vamos a ver que aspecto tiene nuestro entorno de trabajo cuando instalamos este experto. Lo veremos desde el punto de vista de Delphi 7 y de Delphi 2007.

Este es el aspecto que tiene Delphi 7 al arrancar con este experto:


Lo primero que se aprecia es el chorro de iconos nuevos encima del editor de código:


Además nos aparece a la izquierda del editor de código algo que no tenían las versiones de Delphi 7 y anteriores, la numeración de líneas:

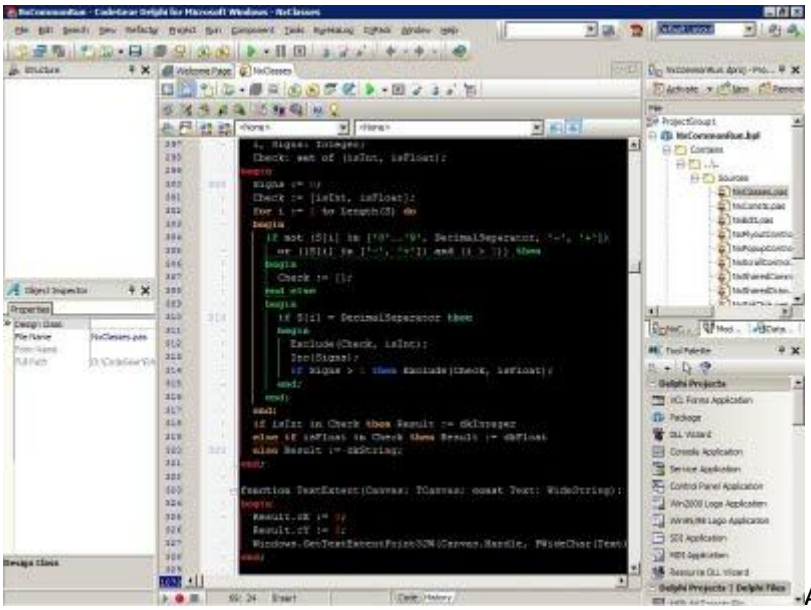

Pero es que si nos fijamos en el código fuente veremos que a añadido un nuevo sistema de coloración de palabras reservadas de modo que según el nivel de identación tienen un color u otro:



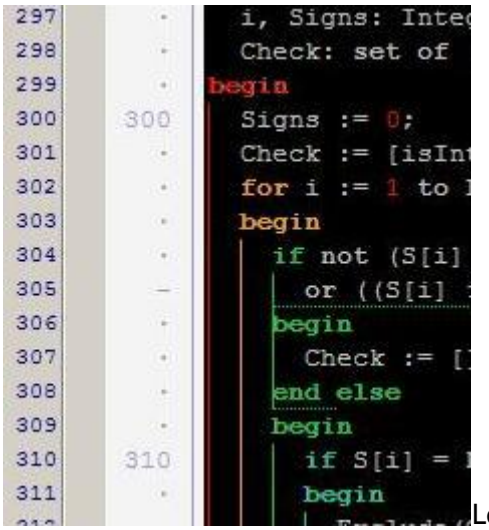
```
var i, j: Integer;
begin
  if Button = mbLeft then
  begin
    bIzquierdo := True;

    // ¿Está dentro de la pantalla?
    if ( x > 40 ) and ( y > 40 ) and ( x <= 6
    begin
      i := x div 40;
      j := y div 40;
      EX.Caption := IntToStr( i );
      EY.Caption := IntToStr( j );

      Piezas[i,j] := iPieza;
      DibujarPantalla;
    end;
  end;
```

De un solo vistazo podemos saber si todos los **begin** cierran correctamente con su correspondiente **end**. También podemos apreciar en la imagen que nos ilumina la línea actual donde estamos situados. Luego veremos más adelante como activar y desactivar todas estas opciones.

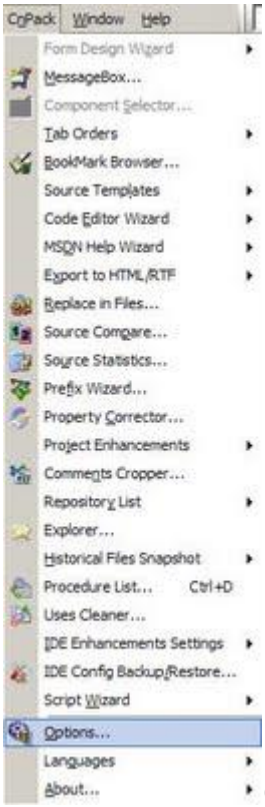
El aspecto que tiene Delphi 2007 a arrancar con este experto es el siguiente:



Aquí se produce la primera contradicción: nos aparecen los números de línea de **cnPack** mas los números de línea que ya lleva el IDE de Delphi:

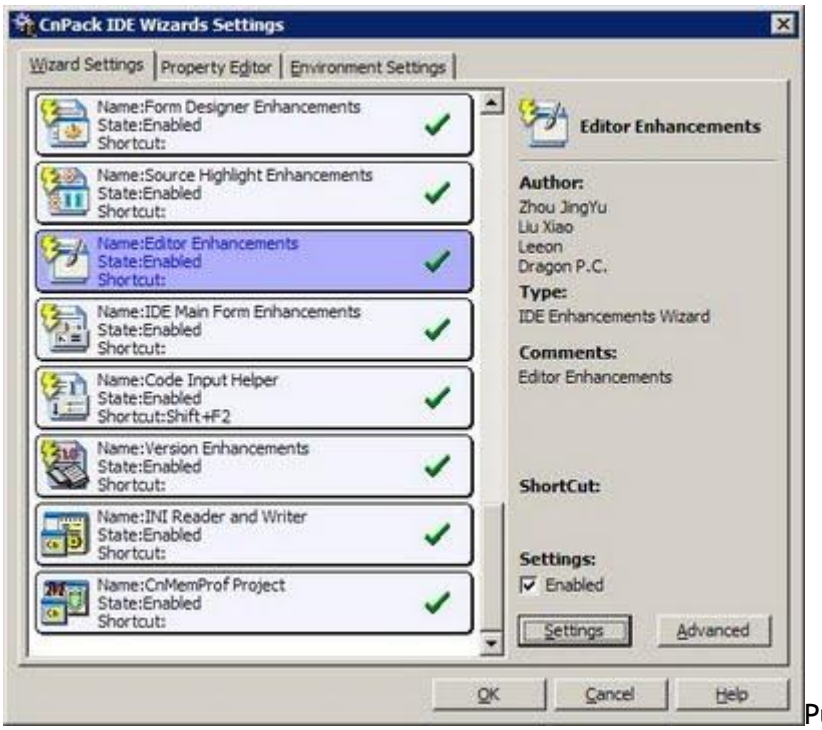


Lo lógico sería desactivar los números de **cnPack** seleccionando en el menú superior **cnPack** -> **Options**:

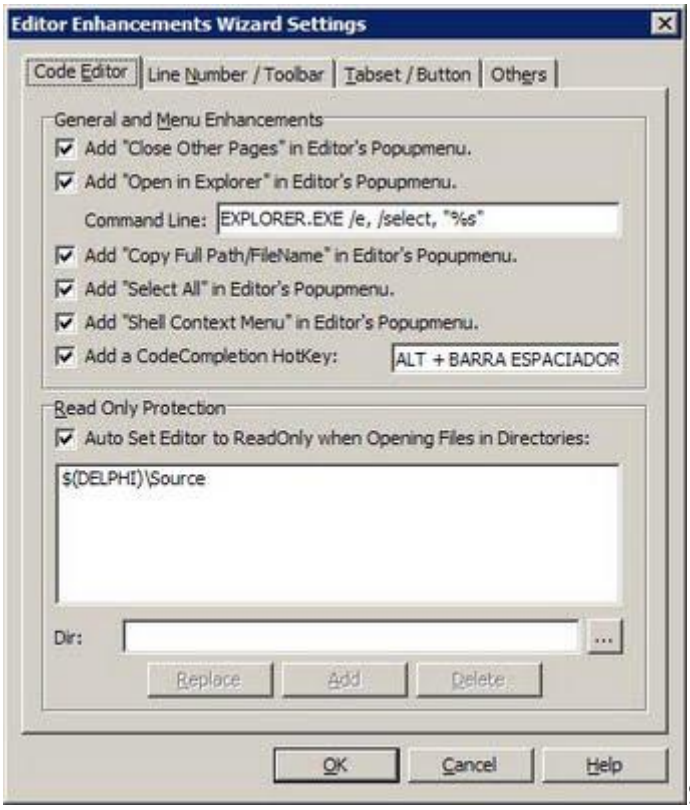


O bien pulsando este botón en la barra de herramientas que hay encima del editor de código:

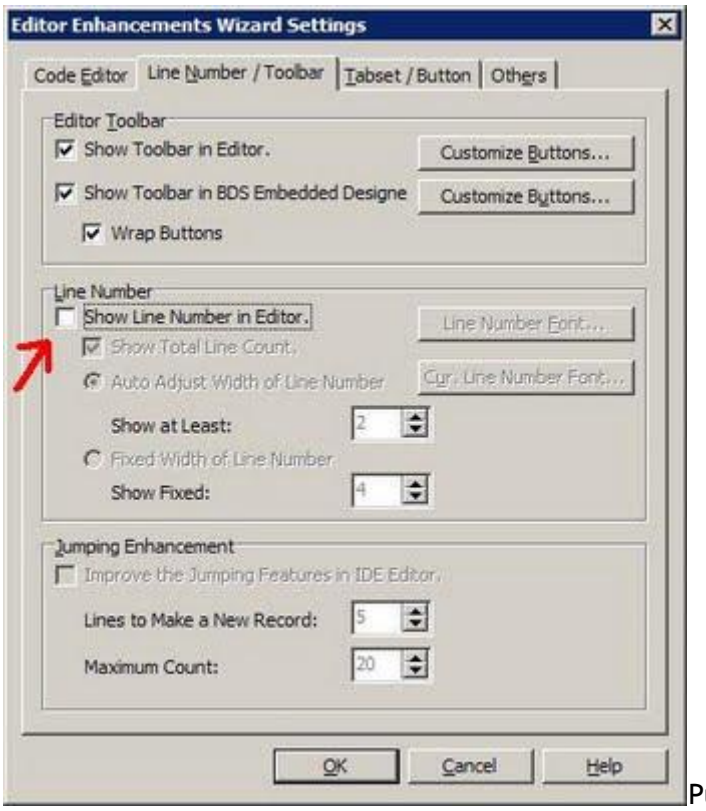

En la ventana que aparece seleccionamos el apartado **Editor Enhacements**:



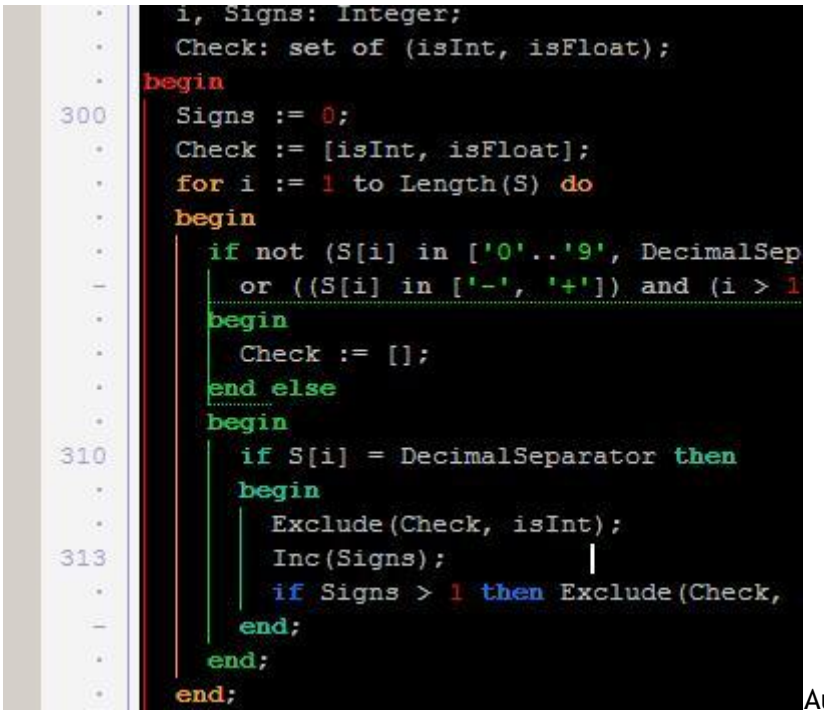
Pulsamos el botón **Settings** que hay a la derecha y aparecerá esta otra ventana:



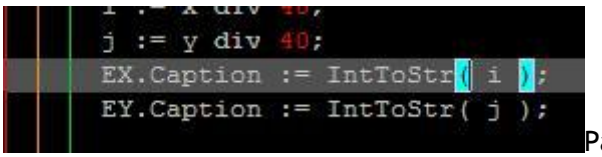
Seleccionamos la pestaña **Line Number / Toolbar** y desactivamos la opción **Show Line Number in Editor**:



Pulsamos **Ok** en esta ventana y en la anterior y ya tenemos Delphi 2007 listo para trabajar:



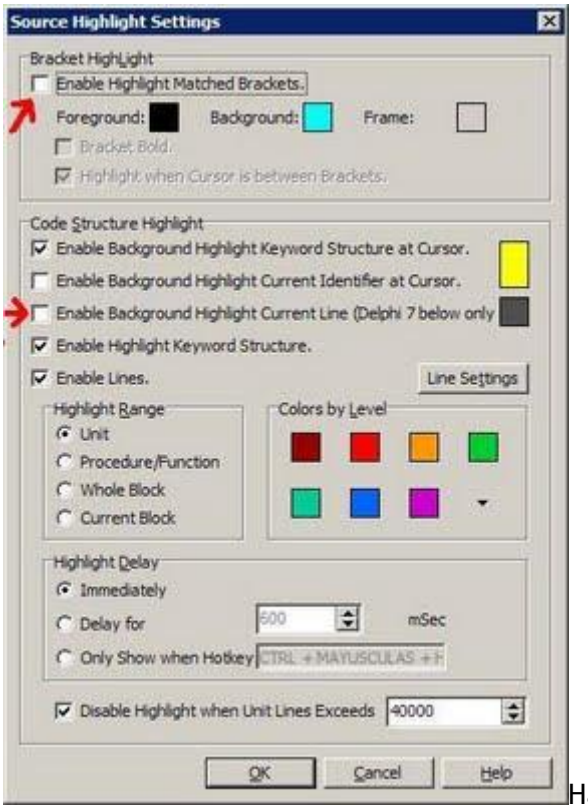
Aunque todavía hay dos cosas que por lo menos a mí nunca me han gustado en las versiones modernas de Delphi: una es el resaltar la línea donde estamos situados y otra es que me ilumine los paréntesis donde tengo situado el cursor (los programadores no somos tan cegatos):



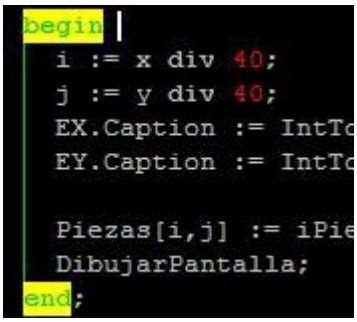
Para quitar ambos volvemos a abrir la

ventana de opciones y seleccionamos **Source Hightlight Enhacements**:

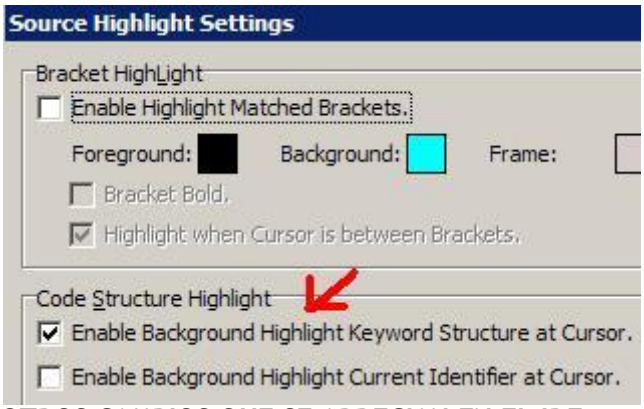
Pulsamos el botón **Settings** y desactivamos estas dos opciones:



Hay otro tipo de iluminación que tiene que enciende las palabras **begin** y **end** cuando nos situamos sobre una de ellas:

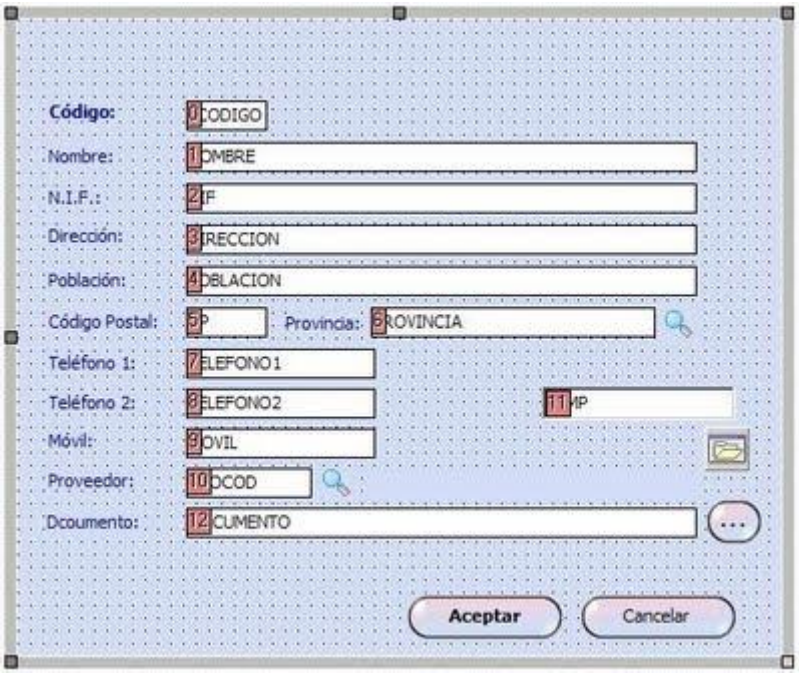


Esta no merece la pena quitarla porque viene muy bien para saber que **end** cierra cada **begin**. De todas formas, si queréis quitarla se encuentra en la ventana de opciones que hemos visto anteriormente:



OTROS CAMBIOS QUE SE APRECIAN EN EL IDE

Si nos ponemos a modificar un formulario veremos que en la esquina superior izquierda de cada componente veremos su número de orden de tabulación:

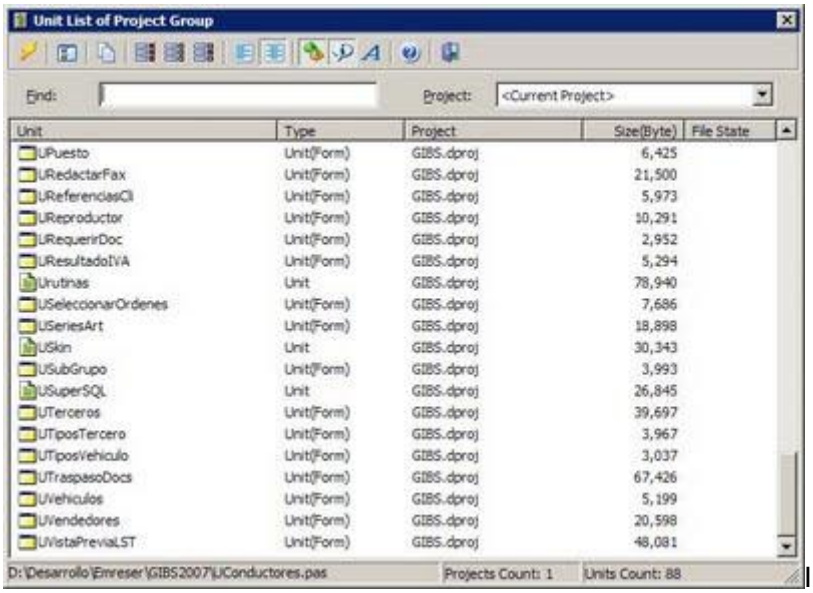


Esto viene muy bien para detectar fallos en la tabulación sin tener que sacar la lista de componentes:

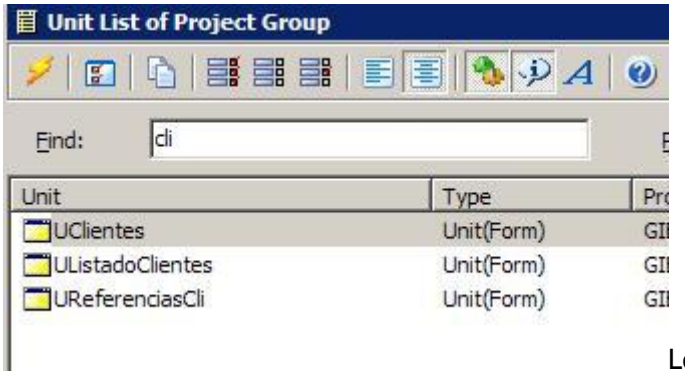
También veremos que se han añadido checkbox a las propiedades booleanas del inspector de objetos:

Si buscamos una unidad pulsando la combinación de teclas **CTRL + F12** o pulsamos este botón:

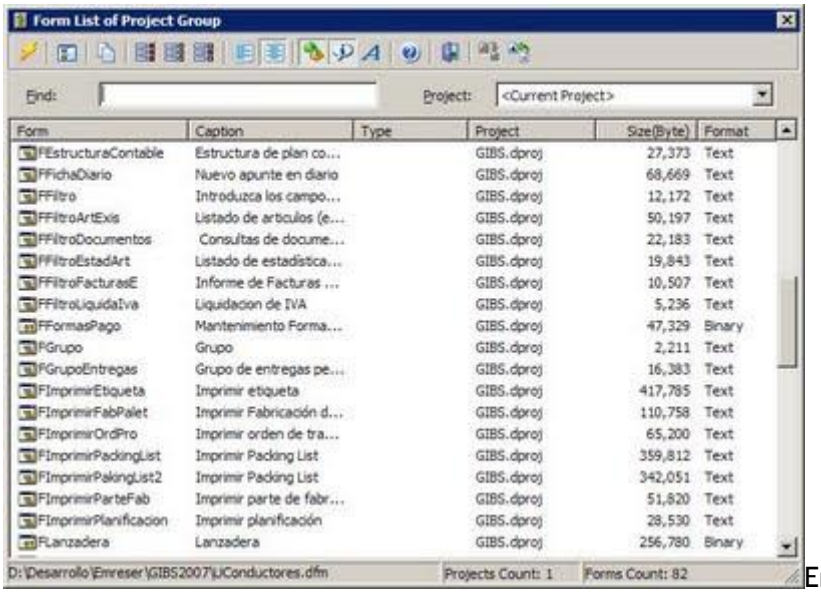

Veremos como cambia la lista de unidades:



Incorpora en la parte de arriba un buscador que nada más comenzar a teclear nos filtrará las unidades que contienen dicha cadena de texto:



Lo mismo sucede si buscamos un formulario pulsando **SHIFT + F12**:



En el próximo artículo entraré mas en detalle entrando en cada una de las características que añade a Delphi y recomendaré aquellas de deberíamos quitar porque son más una molestia que una ayuda.

Pruebas realizadas en Delphi 7 y Delphi 2007.

Publicado por Administrador en 09:59 6 comentarios
Etiquetas: expertos

23 octubre 2009

Mi primer videojuego independiente (y 6)

Voy a terminar esta serie de artículos comentando otras dificultades que encontré en el juego como pueden ser el dibujado de piezas con scroll y la colisión con las mismas.

CARGANDO LAS PIEZAS DEL JUEGO

Como dije anteriormente, la pantalla de 640 x 480 la he dividido en piezas de 40 x 40 pixels lo que me dan un total de 16 piezas horizontales y 12 verticales. Estas piezas las guardo en un gran sprite vertical:

El problema está en que como tengo que meterlo en texturas que no superen los 256 de altura lo he partido en texturas de 64 x 256 (me sobra 24 horizontalmente y 16 verticalmente). En cada textura me caben 6 bloques de 40 x 40.

Entonces realizo el proceso de carga de este modo:

```
procedure CargarSpritesPequepona;
var
  Zip: TZipForge;
  Stream: TMemoryStream;
  Comprimido: PSDL_RWops;
  i, yp: Integer;
begin
  Piezas := TSprite.Create('Piezas', True, 40, 2120, 40,
40);

  // Extraemos las piezas del archivo zip
```

```
  AbrirZip('graficos.zip', Stream, Zip, Comprimido);
  DescomprimirSprite(Stream, Zip, Comprimido, Piezas, True,
40, 40, 'piezas.png');

  // Partimos el sprite en texturas y las subimos a la
memoria de vídeo
  yp := 0;
  for i := 1 to 10 do
  begin
    Piezas.CrearTextura(0, yp, 64, 256, 0, 0, 40, 240);
    Inc(yp, 240);
  end;
  Piezas.LiberarSuperficie;

  Zip.Free;
  Stream.Free;
end;
```

Una vez que le mandamos las texturas a OpenGL podemos liberar la superficie que hemos extraído del archivo zip y lo cerramos.

DIBUJANDO LAS PIEZAS CON SCROLL HORIZONTAL

La función de dibujar en pantalla realiza un barrido horizontal y vertical por toda la pantalla y va dibujando las piezas de 40 x 40 y encima los objetos. A continuación explicaré cada parte:

```
procedure DibujarPantalla;
var
  p, i, j, x, y: Integer;
  bDibujar: Boolean;
begin
  for p := 1 to iNumPan do
  begin
    for j := 1 to 12 do
    begin
      for i := 1 to 16 do
      begin
        x := (i-1)*40+iScroll+(p-1)*640;
        y := (j-1)*40;

        // Comprobamos que la pieza a dibujar esté dentro
de la pantalla

        if (x+40 >= 0) and (x <= 640) then
        begin
          if Pantallas[p].Piezas[i,j] > 0 then
          begin
            Piezas.x := x;
            Piezas.y := y;
            Piezas.iSubY := Pantallas[p].Piezas[i, j];
            Piezas.Dibujar;
```

```
            end;

            // Caso especial para la llave

            if Pantallas[p].Objetos[i,j] = LLAVE then
              DibujarAnimacionLlave(x, y );

            // Dibujamos el resto de objetos
            bDibujar := False;

            // Sólo dibujamos los objetos principales
            if (Pantallas[p].Objetos[i,j] > LLAVE) and
               (Pantallas[p].Objetos[i,j] < RATONDER) then
              bDibujar := True;

            // Sólo dibujamos los objetos a recoger
            if (Pantallas[p].Objetos[i,j] = PANZACOCO) or
               (Pantallas[p].Objetos[i,j] = ESCUDO) or
               (Pantallas[p].Objetos[i,j] = MUELLE) or
               (Pantallas[p].Objetos[i,j] = MUELLE2) or
               (Pantallas[p].Objetos[i,j] = RELOJ) then
              bDibujar := True;

            // El tope de los enemigos no lo sacamos
            if Pantallas[p].Objetos[i,j] = TOPE then
              bDibujar := False;

            if bDibujar then
            begin
              Objetos.x := x;
              Objetos.y := y;
              Objetos.iSubY := Pantallas[p].Objetos[i, j];
              Objetos.Dibujar;
            end;

            // Caso especial para la puerta
            if Pantallas[p].Objetos[i,j] = PUERTA2 then
              DibujarAyuda(x, y);
          end;
        end;
      end;
    end;
end;
```

La primera parte comienzo con un bucle de recorre todas las pantallas y dentro de cada una las piezas horizontal (i) y verticalmente (j):

```
for p := 1 to iNumPan do
begin
  for j := 1 to 12 do
    begin
```

```
      for i := 1 to 16 do       begin
        x := (i-1)*40+iScroll+(p-1)*640;
        y := (j-1)*40;
```

Aquí la parte más importante es la variable **x**, que dependiendo de la pantalla **p** y el desplazamiento de la variable**iScroll** determinaremos que pantalla dibujar y desde donde comenzamos. No conviene dibujar más de lo normal ya que podríamos ralentizar la librería OpenGL.

Después de convertir las coordenadas de las pantallas virtuales a la pantalla de vídeo entonces compruebo si puedo dibujarlas o no:

```
if (x+40 >= 0) and (x <= 640) then
begin
  if Pantallas[p].Piezas[i,j] > 0 then
  begin
    Piezas.x := x;
    Piezas.y := y;
    Piezas.iSubY := Pantallas[p].Piezas[i, j];
    Piezas.Dibujar;
  end;
```

Las pantallas las almaceno en este array bidimensional:

```
Pantallas: array[1..MAX_PAN] of TPantalla;
```

Donde **TPantalla** es un registro que almacena todas las piezas:

```
TPantalla = record
  Piezas, Objetos: array[1..16, 1..12] of byte;
end;
```

Lo que viene a continuación es dibujar los objetos, pero aquí hay que tener cuidado. Los objetos especiales como los topes de los enemigos, las puertas o las llaves tienen su propia rutina de dibujado por lo que las evito mediante la variable **bDibujar**:

```
// Caso especial para la llave

if Pantallas[p].Objetos[i,j] = LLAVE then
  DibujarAnimacionLlave(x, y );

// Dibujamos el resto de objetos
bDibujar := False;

// Sólo dibujamos los objetos principales
if (Pantallas[p].Objetos[i,j] > LLAVE) and
   (Pantallas[p].Objetos[i,j] < RATONDER) then
  bDibujar := True;

  // Sólo dibujamos los objetos a recoger
  if (Pantallas[p].Objetos[i,j] = PANZACOCO) or
     (Pantallas[p].Objetos[i,j] = ESCUDO) or
     (Pantallas[p].Objetos[i,j] = MUELLE) or
```

```
      (Pantallas[p].Objetos[i,j] = MUELLE2) or
      (Pantallas[p].Objetos[i,j] = RELOJ) then
    bDibujar := True;

    // El tope de los enemigos no lo sacamos
    if Pantallas[p].Objetos[i,j] = TOPE then
      bDibujar := False;

    if bDibujar then
    begin
      Objetos.x := x;
      Objetos.y := y;
      Objetos.iSubY := Pantallas[p].Objetos[i, j];
      Objetos.Dibujar;
    end;

    // Caso especial para la puerta
    if Pantallas[p].Objetos[i,j] = PUERTA2 then
      DibujarAyuda(x, y);
```

La variable **iScroll** la inicializo a cero al comenzar el juego:

```
iScroll := 0;
```

Entonces cuando pulso la tecla hacia la derecha incremento el scroll:

```
// Si el heroe pasa a la mitad derecha de la pantalla
// entonces comprobamos si tiene que hacer scroll:
if Heroe.rx+30 > 320 then
    ScrollDerecha;
```

La variable **rx** son las coordenadas del heroe en pantalla en números reales. Antes de moverlo he comprobado que si estamos en la primera pantalla y no pasamos de la mitad de la misma no haga scroll.

Entonces el scroll hacia la derecha decrementa la variable **iScroll** para que las pantallas se vayan desplazando hacia la izquierda mientras nuestro personaje permanece en el centro de la pantalla:

```
procedure ScrollDerecha;
begin
  if Abs(iScroll) < (iNumPan-1)*640 then
    Dec(iScroll, Heroe.iVelocidad);
end;
```

La variable **iNumPan** es el número de pantallas totales que tiene esta fase. La variable **iVelocidad** la he fijado en 3 para realizar el scroll aunque a veces de incrementa cuando estamos andando por una plataforma móvil en su misma dirección: Ahora para movernos a hacia la izquierda comprobamos si no estamos en la última pantalla:

```
if Heroe.rx > 320 then
    ScrollIzquierda
else
```

```
  if iScroll <= 0 then
    ScrollIzquierda;
```

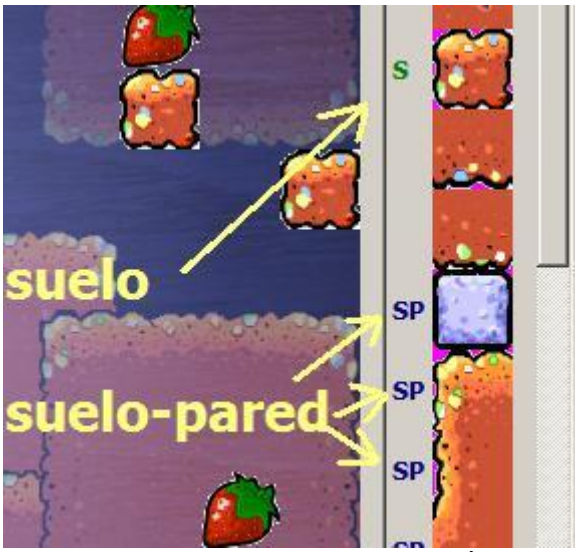
Y entonces muevo todas las pantallas hacia la derecha controlando de que no me pase de la primera pantalla:

```
procedure ScrollIzquierda;
begin
  if (iScroll<0) and (Heroe.Sprite.x+30<320) then
    Inc(iScroll, Heroe.iVelocidad);
end;
```

Básicamente tenemos aquí todo lo que necesitamos para hacer un scroll horizontal y bidireccional. El fallo que cometí fue no haber diseñado un mundo gigante horizontalmente en vez de haberlo partido todo en pantallas. Hubiese sido más fácil así.

CONTROLANDO EL CHOQUE CON PAREDES Y SUELOS

Cuando movemos el personaje por pantalla tenemos que controlar que no choque con los bloques de 40 x 40 que hacen de pared o de suelo. Para ello hice una distinción en el editor para saber los bloques que hacen de suelo y los que hacen de pared:



Con los objetos que hacen de suelo sólo podemos chocar al caer verticalmente pero no horizontalmente. Y con los que hacen de suelo-pared chocamos por todos lados. Lo primero que tenemos que hacer es una función de tal modo que si le pasamos un par de coordenadas nos diga que pieza hay en esa coordenada:

```
function LeerPieza(x,y: Integer): Byte;
var
  xMapa, yMapa, iPantalla: Integer;
begin
  LeerCoordenadasMapa(x, y, xMapa, yMapa, iPantalla);
  if (xMapa>=1) and (xMapa<=16) and (yMapa>=1) and
(yMapa<=12) then
    Result := Pantallas[iPantalla].Piezas[xMapa,yMapa]
  else
    Result := 0;
end;
```

Pero antes llama al procedimiento **LeerCoordenadasMapa** que transforma esas coordenadas a pantalla según el scroll:

```
procedure LeerCoordenadasMapa(x,y: Integer; var xMapa,
yMapa, iPantalla: Integer);
begin
  iPantalla := x div 640+1;
  // ¿La pieza a mirar está en la pantalla de la izquierda?
  if x < 0 then
    x := (iPantalla-1)*640+x;

  // ¿La pieza a mirar está en la pantalla de la derecha?
  if x > 640 then
    x := x-(iPantalla-1)*640;

  xMapa := x div 40+1;
  yMapa := y div 40+1;
end;
```

El peligro de esta función esta en que hay que saber si nos hemos pasado de pantalla y hay que asomarse a la primera columna de la siguiente pantalla.

Una vez sabemos la pieza que hay en un determinado lugar podemos saber si chocamos con ella:

```
function ChocaConSuelo(x,y: Integer): Boolean;
var
  i, iPieza: Integer;
begin
  iPieza := LeerPieza(x,y);
  Result := False;
  for i := 1 to iNumSue do
    if iPieza = Suelos[i] then
      Result := True;
end;
```

El array **Suelos** contiene la numeración de suelos con los que chocamos según lo que hicimos en el editor de pantallas. De la misma forma he operado para comprobar el choque con las paredes, recoger los objetos, etc.

DIBUJANDO LOS NUMEROS DEL MARCADOR

Aunque la librería SDL permite escribir caracteres utilizando las fuentes de Windows, os recomiendo no utilizarlo porque salen las letras en monocromo y pixeladas. Lo mejor es generar los sprites con los caracteres que necesitamos:

Entonces creamos un procedimiento para imprimir dichos números utilizando estos sprites:

```
procedure DibujarNumero(x,y: Integer; iNumero: Integer);
begin
```

```
  if ( iNumero <> 99 ) then
    Exit;

  Numeros.x := x;
  Numeros.y := y;
  Numeros.iSubX := iNumero div 10;
  Numeros.Dibujar;
  Inc(Numeros.x,36);
  Numeros.iSubX := iNumero mod 10;
  Numeros.Dibujar;
end;
```

Como solo hay que imprimir una puntuación de 0 a 99 entonces cojo el primer dígito y lo divido por 10 y del segundo dígito calculo el resto. También lo podíamos haber realizado convirtiendo el número a string y luego recorriendo la cadena de caracteres e imprimiendo un dígito por cada carácter.

FINALIZANDO

Así como he mostrado estos ejemplos podía haber escrito miles de líneas más sobre enemigos, animaciones, etc., pero necesitaría un tiempo que no tengo. Creo que lo más importante de lo que he hablado es la renderización de polígonos con OpenGL.

Actualmente estoy pasando este motor a 1024 x 768 y 60 fotogramas por segundo y para probarlo estoy haciendo un pequeño juego gratuito que ya os mostraré por aquí cuando lo termine.

Después de todo esto que os he soltado, ¿quién se anima a hacer videojuegos? No diréis que no os lo he puesto a huevo.

Pruebas realizadas en Delphi 7.

Publicado por Administrador en 16:36 2 comentarios
Etiquetas: juegos

16 octubre 2009

Mi primer videojuego independiente (5)

Vamos a terminar de ver el resto de la clase **TSprite** con el procedimiento más importante del mismo: el encargado de dibujar los polígonos.

DIBUJANDO POLÍGONOS EN PANTALLA

Primero voy a volcar todo el procedimiento de **Dibujar** y luego explico cada parte:

```
procedure TSprite.Dibujar;
var
  i, j, k: Integer;
  rDespX, rDespY: Real; // Desplazamiento X e Y respecto al
eje de coordenadas
  rEscalaTexX, rEscalaTexY: Real;
  rTexX, rTexY: Real; // Coordenadas de la esquina superior
izquierda de la textura (0,0)
  iSubX2, iSubY2: Integer; // reajustamos los subsprites
según la textura
  iNumSprHor, iNumSprVer: Integer; // N° de sprites que
```

```
caben en la textura horizontal y verticalmente
begin
  // Si no tiene texturas no hacemos nada
  if iNumTex = 0 then
    Exit;

  if bTransparente then
  begin
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  end
  else
    glDisable(GL_BLEND);

  iSubX2 := 0;
  iSubY2 := 0;

  // ¿Tiene subsprites?
  if bSubSprites then
  begin
    if iTexSub > 0 then
      iTexAct := iTexSub
    else
      iTexAct := 1;

    // Entonces calculamos el ancho, alto y posición de
cada subsprite

    iNumSprHor := Textura[1].iAnchoTex div iAnchoSub;

    // ¿El subsprite se sale de la textura actual?
    if (iSubX+1) > iNumSprHor then
    begin
      if iNumSprHor <> 0 then
      begin
        // Calculamos en que textura va a caer nuestro
subsprite
        iTexAct := iSubX div iNumSprHor+1;

        // Calculamos en que subsprite cae dentro de esa
textura
        iSubX2 := iSubX mod iNumSprHor;
      end
      else
      begin
        iTexAct := 1;
        iSubX2 := 0;
      end;
    end
    else
      iSubX2 := iSubX;
```

```
    // Si el subsprite se pasa de la textura pasamos a la
siguiente textura vertical
    // Calculamos cuantos sprites caben verticalmente
    iNumSprVer := Textura[1].iAltoTex div iAltoSub;

    // ¿El subsprite se sale de la textura actual?
    if (iSubY+1) > iNumSprVer then
    begin
      if iNumSprVer <> 0 then
      begin
        // Calculamos en que textura va a caer nuestro
subsprite
        iTexAct := iSubY div iNumSprVer+1;

        // Calculamos en que subsprite cae dentro de esa
textura
        iSubY2 := iSubY mod iNumSprVer;
      end
      else
      begin
        iTexAct := 1;
        iSubY2 := 0;
      end
    end
    else
      iSubY2 := iSubY;
  end;

  // Dibujamos todas las texturas del sprite (o sólo la
actual)

  for i := 1 to iNumTex do
  begin
    // ¿Hay que imprimir sólo una textura?
    if iTexAct > 0 then
      // ¿No es esta textura?
      if i <> iTexAct then
        // Pasamos a la siguiente textura
        Continue;

    // Calculamos desde donde comienza la textura
    rDespX := x+Textura[i].iIncX;
    rDespY := y+Textura[i].iIncY;

    // Dibujamos todos los triángulos del objeto que
contiene cada sprite

    for j := 1 to 2 do
    begin
      glBindTexture(GL_TEXTURE_2D, Textura[i].ID);
```

```
      glBegin(GL_TRIANGLES);
      for k := 1 to 3 do
      begin
        // ¿Tiene subsprites?
        if bSubsprites then
        begin
          rTexX :=
CompToDouble(iSubX2*iAnchoSub)/CompToDouble(Textura[i].iAnc
ho);
          rTexY :=
CompToDouble(iSubY2*iAltoSub)/CompToDouble(Textura[i].iAlto
);
          rEscalaTexX :=
CompToDouble(iAnchoSub)/CompToDouble(Textura[i].iAncho);
          rEscalaTexY :=
CompToDouble(iAltoSub)/CompToDouble(Textura[i].iAlto);

glTexCoord2f(rTexX+Textura[i].Triangulo[j].Vertice[k].u*rEs
calaTexX,

rTexY+Textura[i].Triangulo[j].Vertice[k].v*rEscalaTexY);
        end
        else
        begin
          // Cortamos toda la textura

glTexCoord2f(Textura[i].Triangulo[j].Vertice[k].u,

Textura[i].Triangulo[j].Vertice[k].v);
        end;


glVertex3f(Textura[i].Triangulo[j].Vertice[k].x*rEscalaX+rD
espX,
                  480-
(Textura[i].Triangulo[j].Vertice[k].y*rEscalaY+rDespY), //
la coordenada Y va al revés

Textura[i].Triangulo[j].Vertice[k].z+rProfundidadZ);
      end;

      glEnd();
    end;
  end;

  rProfundidadZ := rProfundidadZ + 0.01;
  glDisable(GL_BLEND);
end;
```

Lo primero que hacemos es declarar estas variables:

```
i, j, k: Integer;
rDespX, rDespY: Real;
rEscalaTexX, rEscalaTexY: Real;
rTexX, rTexY: Real;
iSubX2, iSubY2: Integer;
iNumSprHor, iNumSprVer: Integer;
```

Las variables **i, j, k** las voy a utilizar para recorrer las texturas, los triángulos y los vértices de este sprite.

Luego tenemos las variables **rDesX**, **rDespY** que serán las coordenadas de las esquina superior izquierda de la textura que vamos a capturar dentro del sprite.

Después declaramos **rEscalaTexX**, **rEscalaTexY** para calcular las coordenadas de OpenGL dentro de la textura, que como dije anteriormente van de 0 a 1. Haciendo una regla de tres podemos averiguar su proporción.

Como paso intermedio he declarado las variables **rTexX**, **rTexY**, **iSubX2**, **iSubY2**, **iNumSprHor**, **iNumSprVer** para extraer las texturas de cada subsprite.

Al comenzar el procedimiento lo primero que hago es asegurarme de que el sprite tenga texturas y habilitar la transparencia si el sprite tiene esta propiedad:

```
begin
  // Si no tiene texturas no hacemos nada
  if iNumTex = 0 then
    Exit;

  if bTransparente then
  begin
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  end
  else
    glDisable(GL_BLEND);
```

La transparencia la realizamos con el canal alfa, por lo que es necesario que las texturas seran de 32 bits de color con los componentes (R,G,B,A) siendo **A** el canal alfa. Es importante utilizar un programa de diseño gráfico que soporte este canal (como GIMP).

En el siguiente bloque comprobamos si el sprite tiene subsprites, por lo que extraemos el trozo que nos interesa según las propiedades **iSubX** e **iSubY** del sprite:

```
  // ¿Tiene subsprites?
  if bSubSprites then
  begin
    if iTexSub > 0 then
      iTexAct := iTexSub
    else
      iTexAct := 1;
```

```
    // Entonces calculamos el ancho, alto y posición de
cada subsprite

    iNumSprHor := Textura[1].iAnchoTex div iAnchoSub;

    // ¿El subsprite se sale de la textura actual?
    if (iSubX+1) > iNumSprHor then
    begin
      if iNumSprHor <> 0 then
      begin
        // Calculamos en que textura va a caer nuestro
subsprite
        iTexAct := iSubX div iNumSprHor+1;

        // Calculamos en que subsprite cae dentro de esa
textura
        iSubX2 := iSubX mod iNumSprHor;
      end
      else
      begin
        iTexAct := 1;
        iSubX2 := 0;
      end;
    end
    else
      iSubX2 := iSubX;

    // Si el subsprite se pasa de la textura pasamos a la
siguiente textura vertical
    // Calculamos cuantos sprites caben verticalmente
    iNumSprVer := Textura[1].iAltoTex div iAltoSub;

    // ¿El subsprite se sale de la textura actual?
    if (iSubY+1) > iNumSprVer then
    begin
      if iNumSprVer <> 0 then
      begin
        // Calculamos en que textura va a caer nuestro
subsprite
        iTexAct := iSubY div iNumSprVer+1;

        // Calculamos en que subsprite cae dentro de esa
textura
        iSubY2 := iSubY mod iNumSprVer;
      end
      else
      begin
        iTexAct := 1;
        iSubY2 := 0;
      end
    end
```

```
    else
      iSubY2 := iSubY;
  end;
```

Aquí pueden ocurrir dos casos: que los subsprites no superen el tamaño de la textura o que si la superan entonces me encargo de averiguar en que textura cae el subsprite que necesito.

Ahora viene lo mas gordo. En un primer nivel voy recorriendo todas las texturas de las que se compone este sprite:

```
for i := 1 to iNumTex do
begin
  // ¿Hay que imprimir sólo una textura?
  if iTexAct > 0 then
    // ¿No es esta textura?
    if i <> iTexAct then
      // Pasamos a la siguiente textura
      Continue;

  // Calculamos desde donde comienza la textura
  rDespX := x+Textura[i].iIncX;
  rDespY := y+Textura[i].iIncY;
```

En el segundo nivel pasamos a recorrer los dos triángulos de la textura (si fuera un juego 3D habría que expandir este bucle según el número de polígonos del objeto 3D):

```
  for j := 1 to 2 do
  begin
    glBindTexture(GL_TEXTURE_2D, Textura[i].ID);
    glBegin(GL_TRIANGLES);
```

La función **glBindTexture** de dice a OpenGL que vamos a dibujar texturas 2D y le pasamos el identificador de la textura y luego mediante **glBegin(GL_TRIANGLES)** le indicamos que vamos a dibujar polígonos de tres vértices.

En el tercer y último nivel recorremos los tres vértices de cada triángulo y dependiendo si tiene subsprites o no recorto la textura a su medida:

```
for k := 1 to 3 do
  begin
    // ¿Tiene subsprites?
    if bSubsprites then
    begin
      rTexX :=
CompToDouble(iSubX2*iAnchoSub)/CompToDouble(Textura[i].iAnc
ho);
      rTexY :=
CompToDouble(iSubY2*iAltoSub)/CompToDouble(Textura[i].iAlto
);
      rEscalaTexX :=
CompToDouble(iAnchoSub)/CompToDouble(Textura[i].iAncho);
      rEscalaTexY :=
```

```
CompToDouble(iAltoSub)/CompToDouble(Textura[i].iAlto);

glTexCoord2f(rTexX+Textura[i].Triangulo[j].Vertice[k].u*rEs
calaTexX,

rTexY+Textura[i].Triangulo[j].Vertice[k].v*rEscalaTexY);
    end
    else
    begin
      // Cortamos toda la textura
      glTexCoord2f(Textura[i].Triangulo[j].Vertice[k].u,
                   Textura[i].Triangulo[j].Vertice[k].v);
    end;
```

Y esta función es la que dibuja realmente los polígonos en pantalla:

```
glVertex3f(Textura[i].Triangulo[j].Vertice[k].x*rEscalaX+rD
espX,
            480-
(Textura[i].Triangulo[j].Vertice[k].y*rEscalaY+rDespY), //
la coordenada Y va al revés

Textura[i].Triangulo[j].Vertice[k].z+rProfundidadZ);
```

Como las coordenadas en OpenGL van en coordenadas ortogonales he tenido que invertir la coordenada **Y** para ajustarla a coordenadas de pantalla.

Para finalizar le indicamos que hemos terminado de dibujar triángulos y desactivo el efecto **GL_BLEND**.

```
     end;

     glEnd();
   end;
 end;

 rProfundidadZ := rProfundidadZ + 0.01;
 glDisable(GL_BLEND);
end;
```

La variable **rProfundidadZ** va incrementandose poco a poco para determinar el orden de los sprites en pantalla mediante el buffer de profundidad Z. Este buffer lo inicializo antes de comenzar a dibujar sprites, como veremos más adelante. Esto evita la superposición de polígonos en pantalla (y parpadeos).

OTROS PROCEDIMIENTOS DEL SPRITE

He mantenido el procedimiento **DibujarEn** por si necesitamos copiar el contenido de una superficie en otra antes de subir la textura a la memoria de vídeo:

```
procedure TSprite.DibujarEn( SuperficieDestino:
PSDL_Surface );
```

```
var
  Origen, Destino: TSDL_Rect;
begin
  // ¿Está visible el sprite?
  if bVisible then
    // ¿Tiene subsprites?
    if bSubSprites then
    begin
      Origen.x := iSubX*iAnchoSub;
      Origen.y := iSubY*iAltoSub;
      Origen.w := iAnchoSub;
      Origen.h := iAltoSub;
      Destino.x := x;
      Destino.y := y;
      Destino.w := iAnchoSub;
      Destino.h := iAltoSub;

      // Dibujamos el subsprite seleccionado
      if SDL_BlitSurface( Superficie, @Origen,
SuperficieDestino, @Destino ) < 0 then
      begin
        ShowMessage( 'Error al dibujar el sprite "' +
sNombre + '" en pantalla.' );
        bSalir := True;
        Exit;
      end;
    end
    else
    begin
      Origen.x := 0;
      Origen.y := 0;
      Origen.w := iAncho;
      Origen.h := iAlto;
      Destino.x := x;
      Destino.y := y;
      Destino.w := iAncho;
      Destino.h := iAlto;
      // Lo dibujamos entero
      if SDL_BlitSurface( Superficie, @Origen,
SuperficieDestino, @Destino ) < 0 then
      begin
        ShowMessage( 'Error al dibujar el sprite "' +
sNombre + '" en pantalla.' );
        bSalir := True;
        Exit;
      end;
    end;
end;
```

Luego está el procedimiento **Crear** que crea una superficie vacía dentro de un sprite que no viene de ningún archivo de disco:

```
procedure TSprite.Crear;
begin
 // Creamos una superficie vacía
 Superficie := SDL_CreateRGBSurface( SDL_HWSURFACE, iAncho,
iAlto, 32, 0, 0, 0, 0 );
end;
```

Se puede crear un sprite vacío en el que luego se va escribiendo la puntuación de los marcadores antes de llevarlos a pantalla.

DIBUJANDO SPRITES EN PANTALLA

Con este sistema es como dibujo todos los sprites del juego en pantalla:

```
procedure DibujarSprites;
begin
  rProfundidadZ := 0.0;

  case iEscena of
    ESC_LOGOTIPO: DibujarSpritesLogotipo;
    ESC_PRESENTA: DibujarSpritesPresentacion;
    ESC_OPCIONES: DibujarSpritesOpciones;
    ESC_MAPA: DibujarSpritesMapa;
    ESC_JUEGO: DibujarSpritesJuego;
    ESC_GAMEOVER: DibujarSpritesGameOver;
    ESC_CASTILLO: DibujarCastillo;
    ESC_PEQUEPONA: DibujarPequepona;
    ESC_INTRO: DibujarIntro;
  end;
end;
```

Inicializo primero el buffer de profundidad y luego determino que tengo de dibujar. Para hacer el juego lo más modular posible lo he dividido todo en escenas, de modo que llamo a cada escena según corresponda. Este sería el procedimiento dentro del juego:

```
procedure DibujarSpritesJuego;
begin
  Fondo.Dibujar;
  DibujarPantalla;
  DibujarPlataformas;
  DibujarEnemigos;
  Heroe.Dibujar;
  DibujarFresa;
  DibujarVida;
  DibujarLlave;
  DibujarCoco;
  Estrellas;

  if bPausa then
    DibujarPausa;
end;
```

De este modo vamos descomponiendo el dibujado de sprites en forma de árbol respetando la prioridad de cada sprite.

DIBUJANDO SUPERSPRITES

¿Cómo podemos dibujar toda la pantalla mediante texturas de 256x256? Pues troceándola en texturas de potencia de 2. Por ejemplo, este sería el proceso de carga de la pantalla de presentación:

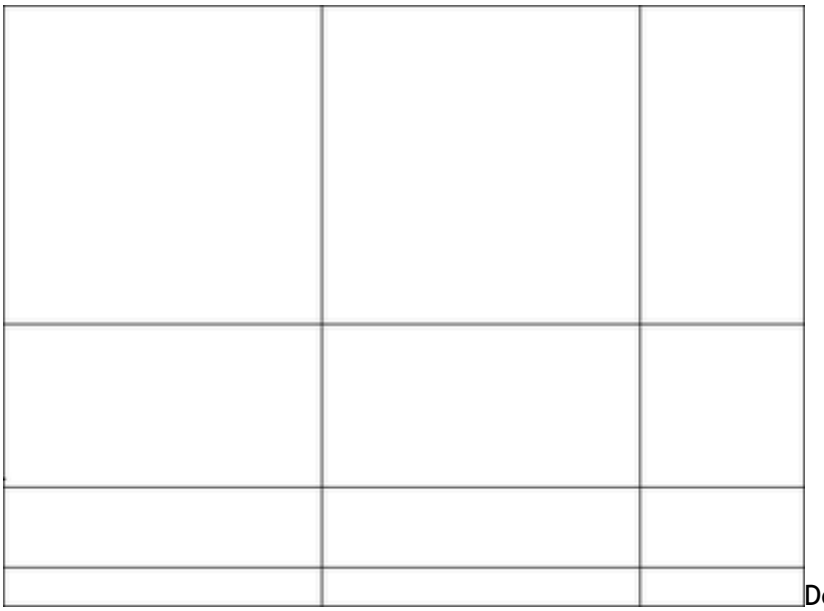
```
procedure CargarSpritesPresentacion;
var
  Zip: TZipForge;
  Stream: TMemoryStream;
  Comprimido: PSDL_RWops;
begin
  // Creamos los sprites que van a cargar los datos
comprimidos

  Presentacion := TSprite.Create('Presentacion', False,
640, 480, 0, 0);

  AbrirZip('gfcs\presentacion.gfcs', Stream, Zip,
Comprimido);
  DescomprimirSprite(Stream, Zip, Comprimido, Presentacion,
False, 0, 0, 'presentacion.png');

  CrearPantalla640x480(Presentacion);

  Zip.Free;
  Stream.Free;
end;
```

Por si acaso había que hacer más pantallas de 640 x 480 me hice un prodecimiento general que partía toda la pantalla y las enviaba a la memoria de vídeo:

```
procedure CrearPantalla640x480(Sprite: TSprite);
begin
  // Fila de arriba
  Sprite.CrearTextura(  0,   0, 256, 256,   0,   0, 256,
256);
  Sprite.CrearTextura(256,   0, 256, 256, 256,   0, 256,
256);
  Sprite.CrearTextura(512,   0, 128, 256, 512,   0, 128,
256);

  // Fila central
  Sprite.CrearTextura(  0, 256, 256, 128,   0, 256, 256,
256);
  Sprite.CrearTextura(256, 256, 256, 128, 256, 256, 256,
256);
  Sprite.CrearTextura(512, 256, 128, 128, 512, 256, 128,
```

```
256);

  // Penúltima fila de abajo
  Sprite.CrearTextura(  0, 384, 256,  64,   0, 384, 256,
64);
  Sprite.CrearTextura(256, 384, 256,  64, 256, 384, 256,
64);
  Sprite.CrearTextura(512, 384, 128,  64, 512, 384, 256,
64);

  // Última fila de abajo
  Sprite.CrearTextura(  0, 448, 256,  32,   0, 448, 256,
32);
  Sprite.CrearTextura(256, 448, 256,  32, 256, 448, 256,
32);
  Sprite.CrearTextura(512, 448, 128,  32, 512, 448, 256,
32);

  Sprite.LiberarSuperficie;
end;
```

Esa sería la pantalla troceada:



Después para dibujarla sólo tengo que hacer esto:

```
Presentacion.Dibujar;
```

Y ya se encargan mis rutinas de dibujar las 12 texturas del sprite. Es lo que yo llamo un supersprite. En el siguiente artículo comentaré como resolví el tema del scroll horizontal así como otros problemas similares (el héroe, los enemigos, etc.).

Pruebas realizadas con Delphi 7.

Publicado por Administrador en 09:41 0 comentarios
Etiquetas: juegos

09 octubre 2009

Mi primer videojuego independiente (4)

Después de preparar toda la artillería que necesitamos para dibujar sprites en OpenGL vamos a implementar por fin la clase **TSprite**. Vamos a ver como se carga la textura de un sprite desde un archivo comprimido con zip.

CARGANDO SPRITES

Esta clase es similar a la que hice en cursos anteriores pero ahora comento lo que he añadido nuevo respecto a OpenGL:

```
TSprite = class
  sNombre: string;                    // Nombre del archivo
del sprite
  rx, ry: Real;                       // Coordenadas del
sprite en coma flotante
  x, y: Integer;                      // Coordenadas del
sprite en pantalla
  iAncho, iAlto: Integer;             // Altura y anchura
del sprite
  bVisible, bTransparente: Boolean;
  bSubsprites: Boolean;               // ¿Tiene subsprites?
  iNumSubX, iNumSubY: Integer;        // Nº de subsprites a
lo ancho y a lo alto
  iAnchoSub, iAltoSub: Integer;       // Ancho y alto del
subsprite
  iSubX, iSubY: Integer;              // Coordenada del
subsprite dentro del sprite
  Superficie: PSDL_Surface;
  rEscalaX, rEscalaY: Real;           // Escala de
ampliación X e Y

  Textura: array[1..MAX_TEXTURAS] of TTextura; // Texturas
disponibles para este sprite
  iNumTex: Integer;                   // Nº de texturas
  iTexAct: Integer;                   // Si es mayor de
cero sólo se imprime esta   textura (sin subsprites)
  iTexSub: Integer;                   // Si es mayor de
cero sólo se imprime esta textura (con subsprites)

  constructor Create(sNombreSprite: String; bSubsprites:
Boolean;
    iAncho, iAlto, iAnchoSub, iAltoSub: Integer);
  destructor Destroy; override;
  procedure CargarSuperficieComprimida(Comprimido:
PSDL_RWops);
  procedure Dibujar;
  procedure DibujarEn(SuperficieDestino: PSDL_Surface);
  procedure Crear;
```

```
  procedure CrearTextura(xInicial, yInicial, iAncho, iAlto,
    iIncrementoX, iIncrementoY, iAnchoTex, iAltoTex:
Integer );
  procedure LiberarSuperficie;
end;
```

Las primeras variables contienen un nombre único que le asigno cada sprite, sus dimensiones y las coordenadas en pantalla en números enteros y en números reales:

```
sNombre: string;                    // Nombre del archivo
del sprite
rx, ry: Real;                       // Coordenadas del
sprite en coma flotante
x, y: Integer;                      // Coordenadas del
sprite en pantalla
iAncho, iAlto: Integer;             // Altura y anchura del
sprite
```

Después tenemos tres variables booleanas para saber si el sprite esta visible, si es transparente o tiene subsprites:

```
bVisible, bTransparente: Boolean;
bSubsprites: Boolean;               // ¿Tiene subsprites?
```

¿Qué es un subsprite? Pues un sprite dentro de otro sprite:

 Como la clase **TSprite** la utilizo para todo, lo mismo nos toca dibujar toda la pantalla con un solo sprite compuesto de muchas texturas de 256 x256 (lo que yo llamo un supersprite) que lo mismo sólo tenemos que dibujar una parte del sprite (subsprite), como en el caso de la tortuga que aprovecho la misma textura para poner todas las posiciones.

En el caso de que nuestro sprite tenga subsprites entonces debemos darle las dimensiones de los mismos:

```
iNumSubX, iNumSubY: Integer;        // N° de subsprites a
lo ancho y a lo alto
iAnchoSub, iAltoSub: Integer;       // Ancho y alto del
subsprite
iSubX, iSubY: Integer;              // Coordenada del
subsprite dentro del sprite
```

Las variables **iSubX** e **iSubY** las utilizo para saber que subsprite voy a imprimir antes de llamar al procedimiento**Dibujar**. Después tenemos la superficie donde vamos a cargar la textura del sprite antes de subirla a la memoria de vídeo:

```
Superficie: PSDL_Surface;
rEscalaX, rEscalaY: Real;           // Escala de ampliación
X e Y
```

Las variables **rEscalaX** y **rEscalaY** las utilizo para aumentar o disminuir el tamaño del

sprite. Esto lo hice para aumentar el tamaño de los botones del menú principal:

Aquí añado algunas novedades pertenecientes a OpenGL:

```
Textura: array[1..MAX_TEXTURAS] of TTextura; // Texturas
disponibles para este sprite
iNumTex: Integer;                    // N° de texturas
iTexAct: Integer;                    // Si es mayor de cero
sólo se imprime esta textura (sin subsprites)
iTexSub: Integer;                    // Si es mayor de cero
sólo se imprime esta textura (con subsprites)
```

Creo un array de texturas donde le pongo por ejemplo un máximo de 100 texturas por sprite:

```
const
  MAX_TEXTURAS = 100; // N° máximo de texturas por sprite
```

Y las variables **iTexAct** y **iTexSub** las utilizo para sprites que son más grandes de 256 x 256 cuando tengo que seleccionar que textura voy a imprimir. Ya lo veremos en el siguiente artículo cuando muestre las rutinas de impresión de polígonos.

El constructor del sprite inicializa todas estas variables así como las que le pasamos como parámetro para crear el sprite:

```
constructor TSprite.Create(sNombreSprite: String;
bSubsprites: Boolean;
  iAncho, iAlto, iAnchoSub, iAltoSub: Integer);
var
  i: Integer;
begin
  sNombre := sNombreSprite;
  Superficie := nil;
  bVisible := True;
  bTransparente := True;
  rx := 0;
  ry := 0;
  x := 0;
  y := 0;
  Self.iAncho := iAncho;
```

```
  Self.iAlto := iAlto;
  Self.bSubsprites := bSubSprites;
  iNumSubX := 0;
  iNumSubY := 0;
  Self.iAnchoSub := iAnchoSub;
  Self.iAltoSub := iAltoSub;
  iSubX := 0;
  iSubY := 0;
  rEscalaX := 1;
  rEscalaY := 1;
  iNumTex := 0;
  rProfundidadZ := 0;
  iTexSub := 0;

  // Inicializamos el array de texturas
  for i := 1 to MAX_TEXTURAS do
    Textura[i] := nil;
end;
```

Por ejemplo, para cargar el fondo de pantalla creo este sprite:

```
Fondo := TSprite.Create('Fondo', False, 640, 480, 0, 0);
```

En cambio, para el pájaro violeta necesito crear subsprites:

 PajaroVioleta := TSprite.Create('PajaroVioleta', True, 180, 56, 45, 28);

O puede haber sprites como la pausa donde todo es una sola textura:

 Pausa := TSprite.Create('Pausa', False, 303, 292, 0, 0);

El destructor de esta clase debe eliminar todas las texturas y objetos que haya creados en memoria:

```
destructor TSprite.Destroy;
var
  i: Integer;
begin
  LiberarSuperficie;

  // Destruimos las texturas
  for i := 1 to MAX_TEXTURAS do
    if Textura[i] <> nil then
      FreeAndNil(Textura[i]);

  iNumTex := 0;
  inherited;
end;
```

El procedimiento **LiberarSuperficie** elimina la superficie de OpenGL que una vez que subimos a memoria de vídeo ya no necesitamos:

```
procedure TSprite.LiberarSuperficie;
begin
  // Destruimos la superficie cargada
  if Superficie <> nil then
  begin
    SDL_FreeSurface(Superficie);
    Superficie := nil;
  end;
end;
```

CARGANDO IMÁGENES COMPRIMIDAS CON ZIP

Aquí entramos en una parte importante de esta clase. Para proteger nuestros gráficos vamos a cargar las texturas PNG que están comprimidas en un archico ZIP con contraseña. De este modo evitamos que otra gente utilice nuestros gráficos para hacer otros juegos cutres. Esta sería la rutina de carga:

```
procedure TSprite.CargarSuperficieComprimida(Comprimido:
PSDL_RWops);
begin
  if Superficie <> nil then
  begin
    SDL_FreeSurface(Superficie);
    Superficie := nil;
  end;

  Superficie := IMG_Load_RW(Comprimido, 1);

  if Superficie = nil then
  begin
    ShowMessage('Error al cargar el archivo comprimido ' +
sNombre + '.');
    Exit;
```

```
  end;

  iAncho := Superficie.w;
  iAlto := Superficie.h;

  // Fijamos el negro como color transparente
  if bTransparente then
    SDL_SetColorKey(Superficie, SDL_SRCCOLORKEY or
SDL_RLEACCEL,
      SDL_MapRGBA(Pantalla.format, 255, 0, 255, 1));
end;
```

El parámetro **Comprimido** es del tipo **PSDL_RWops**, un tipo especial definido en la librería SDL para cargar gráficos desde un buffer de memoria. Luego os mostraré como lo creo al descomprimir el archivo ZIP.

Al principio del procedimiento compruebo si ya la hemos cargado anteriormente para liberarla:

```
if Superficie <> nil then
begin
  SDL_FreeSurface( Superficie );
  Superficie := nil;
end;
```

A continuación cargo la superficie y almaceno el ancho y alto del sprite:

```
Superficie := IMG_Load_RW( Comprimido, 1 );

if Superficie = nil then
begin
  ShowMessage( 'Error al cargar el archivo comprimido ' +
sNombre + '.' );
  Exit;
end;

iAncho := Superficie.w;
iAlto := Superficie.h;
```

Y en el caso de que sea transparente le asigno como color transparente el rosa o el canal alfa:

```
// Fijamos el negro como color transparente
if bTransparente then
  SDL_SetColorKey( Superficie, SDL_SRCCOLORKEY or
SDL_RLEACCEL,
    SDL_MapRGBA( Pantalla.format, 255, 0, 255, 1 ) );
```

Para cargar los sprites hago esto:

```
var
  Zip: TZipForge;
```

```
  Stream: TMemoryStream;
  Comprimido: PSDL_RWops;
  PajaroVioleta: TSprite;
begin
  AbrirZip('graficos.zip', Stream, Zip, Comprimido);
  DescomprimirSprite(Stream, Zip, Comprimido,
PajaroVioleta, True, 45, 28,
    'pajaro_violeta.png');
  ...
```

Para ello creé un procedimiento para abrir archivos Zip utilizando el componente ZipForge:

```
procedure AbrirZip(sArchivo: string; var Stream:
TMemoryStream;
  var Zip: TZipForge; var Comprimido: PSDL_RWops);
begin
  Stream := TMemoryStream.Create;
  Zip := TZipForge.Create(nil);
  Zip.Password := 'miclave';
  Zip.FileName := sRuta+sArchivo;
  Zip.TempDir := sRuta;
  zip.BaseDir := sruta;
  Zip.OpenArchive(fmOpenRead);
  Comprimido := nil;
end;
```

Y una vez abierto voy descomprimiendo texturas:

```
procedure DescomprimirSprite(Stream: TMemoryStream; Zip:
TZipForge;
  Comprimido: PSDL_RWops; Sprite: TSprite; bSubSprites:
Boolean;
  iAncho, iAlto: Integer; sNombre: string);
begin
  Stream.Clear;
  Zip.ExtractToStream(sNombre, Stream);
  Comprimido := SDL_RWFromMem(Stream.Memory, Stream.Size);
  Sprite.CargarSuperficieComprimida(Comprimido);
  Sprite.bSubsprites := bSubSprites;

  if bSubSprites then
  begin
    Sprite.iAnchoSub := iAncho;
    Sprite.iAltoSub := iAlto;
  end;
end;
```

Este procedimiento va extrayendo del archivo zip cada textura y la carga en la superficie del sprite. Entonces después de descomprimir la textura del archivo zip me la llevo de la superficie SDL a la memoria de vídeo:

```
var
  Zip: TZipForge;
  Stream: TMemoryStream;
  Comprimido: PSDL_RWops;
  PajaroVioleta: TSprite;
begin
  AbrirZip('graficos.zip', Stream, Zip, Comprimido);
  DescomprimirSprite(Stream, Zip, Comprimido,
PajaroVioleta, True, 45, 28,
    'pajaro_violeta.png');
  Sprite.CrearTextura(0, 0, 256, 64, 0, 0, 180, 56);
  Sprite.LiberarSuperficie;
  Zip.Free;
  Stream.Free;
end;
```

El procedimiento de crear la textura sube mediante OpenGL la textura a la memoria de vídeo:

```
procedure TSprite.CrearTextura(xInicial, yInicial, iAncho,
iAlto,
  iIncrementoX, iIncrementoY, iAnchoTex, iAltoTex:
Integer);
begin
  // Creamos la textura a partir de esta imagen
  if iNumTex < MAX_TEXTURAS then
  begin
    Inc(iNumTex);
    Textura[iNumTex] := TTextura.Create(Superficie,
xInicial, yInicial,
      iAncho, iAlto, iIncrementoX, iIncrementoY, iAnchoTex,
iAltoTex,
      iAnchoSub, iAltoSub, bSubsprites);
    Textura[iNumTex].sNombre := sNombre;
  end;
end;
```

Después la podemos liberar así como el archivo ZIP y el stream que hemos utilizado como intermediario. No os podéis imaginar las explosiones que me dio Delphi antes de conseguirlo. En el próximo artículo veremos como dibujar el sprite en pantalla según si es un sprite normal, un subsprite o un supersprite.

Pruebas realizadas en Delphi 7.

Publicado por Administrador en 09:36 4 comentarios
Etiquetas: juegos

02 octubre 2009

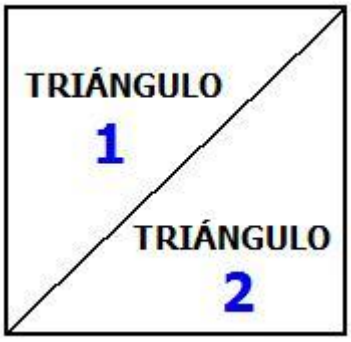
Mi primer videojuego independiente (3)

Después de poner en marcha la librería OpenGL vamos a ver todo lo que necesitamos antes de implementar la clase**TSprite**. Necesitamos preparar nuestra librería **UJuego.pas** con la capacidad de dibujar polígonos.

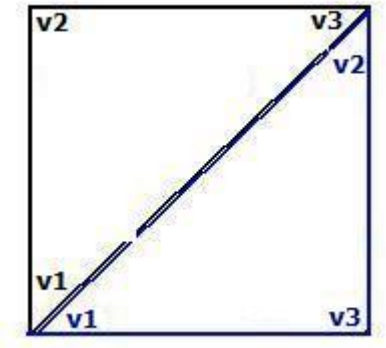
LOS VERTICES

Dibujar sprites en con aceleración en 3D no suele ser tan fácil como copiar una imagen de un sitio a otro. En OpenGL debemos dibujar polígonos proporcionando tanto las coordenadas de sus vértices como las de sus texturas.

Los polígonos pueden ser cuadrados o triángulares, aunque cada GPU divide los cuadrados en triángulos ya que se procesan más rápidos internamente. Como a mí siempre me ha gustado hacer las cosas al nivel más bajo posible, entonces lo que hago es utilizar dos triángulos por cada sprite:



A cada triángulo hay que darle sus coordenadas en el sentido de las agujas del reloj, tal como elegimos anteriormente en las opciones de inicialización de OpenGL. Por lo tanto, por cada triángulo tenemos que proporcionar los tres vértices:



En el ejemplo de la imagen superior, suponiendo que nuestro sprite este situado en la esquina superior izquierda de la pantalla y que tenga un algo y alto de 100 pixels entonces las coordenadas del primer triángulo serían:

```
V1(x=  0, y=100, z=0)
V2(x=  0, y=  0, z=0)
V3(x=100, y=  0, z=0)
```

Supuestamente nunca llegaremos a utilizar la coordenada de profundidad Z porque estamos haciendo un juego 2D. Luego veremos más adelante como la utilizo mediante el Z-Buffer para ordenar la impresión de sprites. Y estas serían las coordenadas para el segundo triángulo:

```
V1(x=  0, y=100, z=0)
V2(x=100, y=  0, z=0)
V3(x=100, y=100, z=0)
```

Pero es que aquí no acaba la cosa. Sólo le hemos dicho donde queremos dibujar el polígono en pantalla pero es que además hay que mapear una textura encima del triángulo. Es como si a cada triángulo le supiésemos una pegatina encima (la textura):


Por ello, cada vértice tiene dos sistemas de coordenadas, una para dibujarlo **(x,y,z)** y otras para la textura **(u, v)**. La coordenada **u** es la horizontal y **v** la vertical. Equivalen a unas coordenadas **x, y** dentro de la textura. Estas coordenadas vienen en números reales y van desde 0 a 1. Para nuestro sprite estas serían las coordendas **u,v** de la textura:

```
V1(u=0, v=1)
V2(u=0, v=0)
V3(u=1, v=0)
```

Y para el segundo polígono:

```
V1(u=0,v=1)
V2(u=1,v=0)
V3(u=1,v=1)
```

Es prácticamente lo mismo que las coordenadas **x, y** pero a escala 0 a 1. Si hiciésemos un juego en 3D ya se encargarían los programas como 3D Studio de darnos las coordenadas **(x,y,z,u,v)** por cada vértice.

Entonces, en el momento justo donde llevamos los polígonos a pantalla debemos primero darle a OpenGL las coordenadas de la textura y después las coordenadas del triángulo. Ya lo veremos más adelante cuando lancemos los polígonos a pantalla.

Pues bien, todo este rollo que os he soltado es para definir la clase **TVertice** que contiene las coordenadas en pantalla y en la textura:

```
TVertice = class
  x, y, z: GLfloat;                                    //
Coordenadas
  u, v: GLfloat;                                       //
Coordenadas de la textura

  constructor Create; overload;                        //
constructor por defecto
  constructor Create(x, y, z, u, v: GLfloat); overload; //
constructor con parámetros
end;
```

He definido un par de constructores. El básico:

```
constructor TVertice.Create;
begin
  x := 0;
  y := 0;
  z := 0;
  u := 0;
  v := 0;
end;
```

Y otro por si queremos crear el vértice y pasarle directamente las coordenadas:

```
constructor TVertice.Create(x, y, z, u, v: GLfloat);
begin
  Self.x := x;
  Self.y := y;
  Self.z := z;
  Self.u := u;
  Self.v := v;
end;
```

El tipo **GLfloat** viene definido por defecto por OpenGL aunque también podemos utilizar los tipos **Real** o **Double**. Lo mejor es utilizar todo lo posible los tipos de datos de OpenGL por si hay que pasar nuestro motor a otros lenguajes como C/C++, C#, Python o Ruby, ya que tanto la librería SDL como la OpenGL son multiplataforma y multilenguaje.

LOS TRIÁNGULOS

Lo siguiente es definir la clase **TTriangulo** que va a contener los 3 vértices del polígono y un identificador para la textura:

```
TTriangulo = class
  Vertice: array[1..3] of TVertice;              // Los 3
```

```
vertices del triángulo
  IdTextura: GLuint;                                  // N° de
textura asignada a este triángulo

  constructor Create; overload;                       //
constructor por defecto
  constructor Create(x1, y1, z1, u1, v1,
                     x2, y2, z2, u2, v2,
                     x3, y3, z3, u3, v3: GLFloat );
overload;  // constructor con parámetros
  destructor Destroy; override;
end;
```

Antes de comenzar a renderizar polígonos en pantalla debemos subir todas las texturas a la memoria de vídeo. Cada vez que subimos una textura, la librería OpenGL nos proporciona un identificador de la textura (1, 2, 3, …). Necesitamos guardarlo en **IdTextura** para más adelante.

Para esta clase he creado dos constructores y un destructor:

```
constructor TTriangulo.Create;
var i: Integer;
begin
  // Creamos los tres vértices
  for i := 1 to 3 do
    Vertice[i] := TVertice.Create;
end;

constructor TTriangulo.Create(x1, y1, z1, u1, v1,
                              x2, y2, z2, u2, v2, x3,
                              y3, z3, u3, v3: GLFloat);
begin
  // Creamos los tres vértices según sus parámetros
  Vertice[1] := TVertice.Create(x1, y1, z1, u1, v1);
  Vertice[2] := TVertice.Create(x2, y2, z2, u2, v2);
  Vertice[3] := TVertice.Create(x3, y3, z3, u3, v3);
end;

destructor TTriangulo.Destroy;
begin
  Vertice[1].Free;
  Vertice[2].Free;
  Vertice[3].Free;
end;
```

LAS TEXTURAS

La clase encargada de almacenar los datos de la textura para cada par de triángulos es la siguiente:

```
TTextura = class
  sNombre: string;                                    //
```

```
Nombre del sprite
  iAncho, iAlto: Integer;                                  //
Ancho y alto de la textura
  iIncX, iIncY: Integer;                                   //
Incremento de X, Y respecto al sprite
  iAnchoTex, iAltoTex: Integer;                            //
Alto y ancho de la textura que vamos a recortar

  Triangulo: array[1..2] of TTriangulo;                    // Los
dos triángulos del sprites
  ID: GLuint;                                              //
Identificador de la textura

  constructor Create(Origen: PSDL_Surface; xInicial,
yInicial, iAncho, iAlto,
    iIncrementoX, iIncrementoY, iAnchoTex, iAltoTex,
iAnchoSub,
    iAltoSub: Integer; bSubsprites: Boolean); overload;
  destructor Destroy; override;
end;
```

Definimos las siguientes variables:

sNombre: nombre del sprite asociado a esta textura.

iAncho, **iAlto**: Alto y ancho de la textura.

iIncX, **iIncY**: Son las coordenadas desde donde capturamos la textura (por efecto en la esquina superior izquierda (0,0)). Cuando un sprite es más grande de 256x256 necesito crear varias texturas que simulen un supersprite. Por ello necesito estas variables para saber cual es la esquina superior izquierda donde comienza la textura dentro del sprite.

iAnchoTex, **iAltoTex**: Ancho y largo real del sprite dentro de la textura. Como en OpenGL sólo podemos crear texturas con un ancho y alto que sea potencia de dos entonces guardo también en estas variables el ancho y largo real del sprite. Puede ser que el sprite sea de 96x78 pero necesite crear para el mismo una textura de 128x128.

Después tenemos dos triángulos definidos dentro de la textura y su identificador:

```
Triangulo: array[1..2] of TTriangulo;                  // Los
dos triángulos del sprites
ID: GLuint;                                            //
Identificador de la textura
```

El constructor que he creado para la textura es algo complejo, porque no sólo crea los triángulos que vamos a utilizar sino que también carga la textura desde la superficie que le pasamos como parámetro. En vez de cargar la textura desde un archivo PNG la cargo desde un archivo ZIP comprimido. La razón no es otra que la de proteger nuestros gráficos PNG para que nadie pueda manipularlos.

Lo que hago es comprimir todos los gráficos que necesito en archivos ZIP con contraseña de manera que sólo el programador sabe la clave de los mismos. De este modo mantenemos la propiedad intelectual del trabajo realizado por el diseñador gráfico, evitando a sí que cualquier pardillo coja nuestros gráficos y haga un juego web con flash

en cuatro días llevándose el mérito.

Es algo más complejo de lo normal pero merece la pena. Veamos primero todo el procedimiento y luego analicemos sus partes:

```
constructor TTextura.Create(Origen: PSDL_Surface; xInicial,
yInicial, iAncho,
  iAlto, iIncrementoX, iIncrementoY, iAnchoTex, iAltoTex,
  iAnchoSub, iAltoSub: Integer;   bSubsprites: Boolean);
var
  Temp: PSDL_Surface;
  Org, Des: TSDL_Rect;
  iLongitudX, iLongitudY: Integer;
begin
  // Guardamos el ancho y alto de la textura que vamos a
crear
  Self.iAncho := iAncho;
  Self.iAlto := iAlto;
  iIncX := iIncrementoX;
  iIncY := iIncrementoY;
  Self.iAnchoTex := iAnchoTex;
  Self.iAltoTex := iAltoTex;

  // Definimos las coordenadas de los dos triángulos de la
textura
  if bSubsprites then
  begin
    iLongitudX := iAnchoSub;
    iLongitudY := iAltoSub;
  end
  else
  begin
    iLongitudX := iAncho;
    iLongitudY := iAlto;
  end;

  Triangulo[1] := TTriangulo.Create(       0.0, iLongitudY,
0.0, 0.0, 1.0,
                                           0.0,        0.0,
0.0, 0.0, 0.0,
                                  iLongitudX,        0.0,
0.0, 1.0, 0.0);

  Triangulo[2] := TTriangulo.Create(       0.0, iLongitudY,
0.0,  0.0, 1.0,
                                  iLongitudX,        0.0,
0.0,  1.0, 0.0,
                                  iLongitudX, iLongitudY,
0.0,  1.0, 1.0);

  // Creamos una nueva superficie con canal alpha
```

```
  if SDL_BYTEORDER = SDL_BIG_ENDIAN then
    Temp := SDL_CreateRGBSurface(SDL_SRCALPHA, iAncho,
iAlto, 32, $FF000000,
                                 $00FF0000, $0000FF00,
$000000FF)
  else
    Temp := SDL_CreateRGBSurface(SDL_SRCALPHA, iAncho,
iAlto, 32, $000000FF,
                                $0000FF00, $00FF0000,
$FF000000);

  // Activamos el canal alpha en la superficie origen antes
de copiarla
  SDL_SetAlpha(Origen, 0, 0);

  // Copiamos la superficie origen a la superficie temporal
  Org.x := xInicial;
  Org.y := yInicial;

  // Evitamos que copie un trozo más ancho que el de la
textura
  if Origen.w > iAncho then
    Org.w := iAncho
  else
    Org.w := Origen.w;

  // Evitamos que copie un trozo más ancho que el de la
textura
  if Origen.h > iAlto then
    Org.h := iAlto
  else
    Org.h := Origen.h;

  Des.x := 0;
  Des.y := 0;
  Des.w := Org.w;
  Des.h := Org.h;
  SDL_BlitSurface(Origen, @Org, Temp, @Des);

  // Le pedimos a OpenGL que nos asigne un identificador de
la textura
  glGenTextures(1, ID);

  // Enviamos la textura a la superficie de vídeo con
OpenGL
  glBindTexture(GL_TEXTURE_2D, ID);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR); // Filtro de
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR); // suavizado
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
```

```
GL_CLAMP); // Para que no se
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP); // repita la textura
  glTexImage2D(GL_TEXTURE_2D, 0, 4, iAncho, iAlto, 0,
GL_RGBA,
    GL_UNSIGNED_BYTE, Temp.pixels);

  // Liberamos la superficie temporal
  SDL_FreeSurface(Temp);
end;
```

En primer lugar defino una supercifie, dos rectángulos y las longitudes horizontales y verticales de la textura:

```
var
  Temp: PSDL_Surface;
  Org, Des: TSDL_Rect;
  iLongitudX, iLongitudY: Integer;
```

Luego guardamos todo lo que nos pasan como parámetro en las variables de la clase:

```
Self.iAncho := iAncho;
Self.iAlto := iAlto;
iIncX := iIncrementoX;
iIncY := iIncrementoY;
Self.iAnchoTex := iAnchoTex;
Self.iAltoTex := iAltoTex;
```

Determinamos el ancho y alto de la textura según si vamos a dibujar todo el sprite o una parte del mismo (subsprites):

```
if bSubsprites then
begin
  iLongitudX := iAnchoSub; // ancho subsprite
  iLongitudY := iAltoSub;  // alto subsprite
end
else
begin
  iLongitudX := iAncho;    // ancho sprite
  iLongitudY := iAlto;     // alto sprite
end;
```

Ahora creamos los dos tríangulos dentro de nuestra textura:

```
Triangulo[1] := TTriangulo.Create(       0.0, iLongitudY,
0.0, 0.0, 1.0,
                                         0.0,        0.0,
0.0, 0.0, 0.0,
                                  iLongitudX,        0.0,
0.0, 1.0, 0.0);

Triangulo[2] := TTriangulo.Create(       0.0, iLongitudY,
```

```
0.0,  0.0, 1.0,
                                    iLongitudX,        0.0,
0.0,  1.0, 0.0,
                                    iLongitudX, iLongitudY,
0.0,  1.0, 1.0);
```

Mientras las coordenadas de los vértices vayan en el sentido de las agujas del reloj, da igual su orden. Creamos una nueva superficie activando el canal alfa:

```
if SDL_BYTEORDER = SDL_BIG_ENDIAN then
  Temp := SDL_CreateRGBSurface(SDL_SRCALPHA, iAncho, iAlto,
32, $FF000000,
                               $00FF0000, $0000FF00,
$000000FF)
else
  Temp := SDL_CreateRGBSurface(SDL_SRCALPHA, iAncho, iAlto,
32, $000000FF,
                               $0000FF00, $00FF0000,
$FF000000);

SDL_SetAlpha(Origen, 0, 0);
```

Después copiamos a la superficie creada el trozo de sprite que vamos a aplicar como textura:

```
Org.x := xInicial;
Org.y := yInicial;

// Evitamos que copie un trozo más ancho que el de la
textura
if Origen.w > iAncho then
  Org.w := iAncho
else
  Org.w := Origen.w;

// Evitamos que copie un trozo más ancho que el de la
textura
if Origen.h > iAlto then
  Org.h := iAlto
else
  Org.h := Origen.h;

Des.x := 0;
Des.y := 0;
Des.w := Org.w;
Des.h := Org.h;
SDL_BlitSurface(Origen, @Org, Temp, @Des);
```

Por último, subimos la textura a la memoria de vídeo pidiéndole a OpenGL que nos proporcione un identificador para la misma y liberamos la superficie temporal:

```
glGenTextures(1, ID);
```

```
// Enviamos la textura a la superficie de vídeo con OpenGL
glBindTexture(GL_TEXTURE_2D, ID);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR); // Filtro de
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR); // suavizado
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
GL_CLAMP); // Para que no se
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
GL_CLAMP); // repita la textura
glTexImage2D(GL_TEXTURE_2D, 0, 4, iAncho, iAlto, 0,
GL_RGBA,
 GL_UNSIGNED_BYTE, Temp.pixels);

// Liberamos la superficie temporal
SDL_FreeSurface(Temp);
```

Sólo queda eliminar la textura en su destructor para no dejar nada en la memoria de vídeo:

```
destructor TTextura.Destroy;
var Textura: GLuint;
begin
  Textura := ID;
  glDeleteTextures(ID,Textura);
  Triangulo[2].Free;
  Triangulo[1].Free;
  inherited;
end;
```

En el próximo artículo entraremos con la clase **TSprite** y veremos como se carga desde un archivo ZIP nuestras texturas en archivos PNG utilizando el componente ZipForge.

Pruebas realizadas en Delphi 7.

Publicado por Administrador en 09:43 2 comentarios
Etiquetas: juegos

25 septiembre 2009

Mi primer videojuego independiente (2)

Una vez que hemos visto por encima las dificultades más importantes que se me dieron con el editor, vamos a pasar a ver las partes más importantes del juego. Sobre todo voy a

centrarme en el apartado gráfico, ya que lo que se refiere a joystick, sonido, etc. no cambia casi nada respecto lo que escribí en los anteriores artículos referentes a la SDL.

EL NÚCLEO DEL JUEGO

El núcleo principal del juego no difiere demasiado respecto a lo escribí anteriormente. Como no se utiliza en casi nada las librerías de Delphi lo que hay que hacer es crear un nuevo proyecto, eliminar el formulario principal y escribir este código directamente en el archivo **DPR**:

```
begin
  InicializarSDL('Pequepon');
  CargarOpciones;
  ModoVideo(640, 480, 32, bPantallaCompleta);
  Teclado := TTeclado.Create;
  Temporizador := TTemporizador.Create;
  Joystick := TJoystick.Create;
  Raton := TRaton.Create;
  ControlSonido := TControlSonido.Create;
  InicializarJuego;

  while not bSalir do
  begin
    Temporizador.Actualizar;

    if Temporizador.Activado then
    begin
      Teclado.Leer;
      Joystick.Leer;
      Raton.Leer;
      ControlarEventos;
      ComenzarRender;
      DibujarSprites;
      FinalizarRender;
      Temporizador.Incrementar;
    end
    else
      Temporizador.Esperar;
  end;

  Demo.Free;
  DestruirSprites;
  FinalizarJuego;
  Raton.Free;
  Joystick.Free;
  ControlSonido.Free;
  Temporizador.Free;
  Teclado.Free;
  FinalizarSDL;
end.
```

Podemos dividir el núcleo en tres bloques principales:

1º **Inicialización**: cambiamos el modo de vídeo y creamos los objetos que necesitamos a lo largo del juego:

```
InicializarSDL('Pequepon');
CargarOpciones;
ModoVideo(640, 480, 32, bPantallaCompleta);
Teclado := TTeclado.Create;
Temporizador := TTemporizador.Create;
Joystick := TJoystick.Create;
Raton := TRaton.Create;
ControlSonido := TControlSonido.Create;
InicializarJuego;
```

La variable booleana **bPantallaCompleta** recoge de las opciones si estamos en modo ventana o en pantalla completa.

2º **Bucle infinito principal**: Por este bucle pasará todo el juego 40 veces por segundo. No terminará hasta que la variable booleana **bSalir** este activada:

```
while not bSalir do
begin
  Temporizador.Actualizar;

  if Temporizador.Activado then
  begin
    Teclado.Leer;
    Joystick.Leer;
    Raton.Leer;
    ControlarEventos;
    ComenzarRender;
    DibujarSprites;
    FinalizarRender;
    Temporizador.Incrementar;
  end
  else
    Temporizador.Esperar;
end;
```

3º **Finalización y destrucción de objetos**: Nos deshacemos de todos los objetos que hay en memoria y finalizamos el modo de video para volver al escritorio de Windows:

```
Demo.Free;
DestruirSprites;
FinalizarJuego;
Raton.Free;
Joystick.Free;
ControlSonido.Free;
Temporizador.Free;
Teclado.Free;
FinalizarSDL;
```

Comencemos viendo las rutinas de inicialización más importantes.

INICIALIZANDO LA LIBRERÍA SDL

Todas las rutinas de mi motor 2D las introduje dentro de una unidad llamada **UJuego.pas**. De este modo, si tenemos que hacer otro juego sólo hay que importar esta unidad y tenemos medio trabajo hecho.

Veamos como arrancar la librería SDL con el procedimiento **InicializarSDL**:

```
procedure InicializarSDL(sTitulo: String);
var i: Integer;
begin
  // Inicializamos la librería SDL
  if SDL_Init(SDL_INIT_VIDEO or SDL_DOUBLEBUF or
SDL_INIT_JOYSTICK) < 0 then
  begin
    ShowMessage('Error al inicializar la librería SDL.');
    SDL_Quit;
    bSalir := True;
    Exit;
  end;

  SDL_WM_SetCaption(PChar(sTitulo), nil);
  bSalir := False;
  sRuta := ExtractFilePath(ParamStr(0));

  // Inicializamos los sprites
  iNumSpr := 0;
  for i := 1 to MAX_SPRITES do
    Sprite[i] := nil;
end;
```

Probamos a cambiar el modo de vídeo activando el doble buffer y el joystick. Si funciona entonces le ponemos el título a la ventana (por si ejecutamos el juego en modo ventana), memorizamos en la variable **sRuta** donde esta nuestro ejecutable (para cargar luego los sprites) e inicializamos los sprites que están declarados en este array global:

```
var
  Sprite: array[1..MAX_SPRITES] of TSprite;
```

El número máximo de sprites que fijé que pueden estar a la vez cargados fue de 50:

```
const
  MAX_SPRITES  = 50;
```

Aunque nunca llegué a gastarlos del todo. Ya veremos la clase **TSprite** más adelante.

CAMBIANDO EL MODO DE VÍDEO

Como vamos a trabajar con OpenGL, aquí ya hay cambios importantes respecto a la rutinas de SDL tradicionales:

```
procedure ModoVideo(iAncho, iAlto, iProfundidadColor:
Integer; bPantallaCompleta: Boolean);
begin
  SetEnvironmentVariable('SDL_VIDEO_CENTERED', '1');

  SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 32);
  SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);

  // Activamos la sincronización vertical
  SDL_GL_SetAttribute(SDL_GL_SWAP_CONTROL, 1);

  // Pasamos a modo de video de la ventana especificada
  if bPantallaCompleta then
    Pantalla := SDL_SetVideoMode(iAncho, iAlto, 32,
SDL_FULLSCREEN or SDL_OPENGL)
  else
    Pantalla := SDL_SetVideoMode(iAncho, iAlto, 32,
SDL_OPENGL);

  if Pantalla = nil then
  begin
    ShowMessage( 'Error al cambiar de modo de video.' );
    SDL_Quit;
    Exit;
  end;

  InicializarOpenGL(iAncho, iAlto);
end;
```

Vamos a analizar este procedimiento porque es la base de todo. Antes de hacer nada le decimos a la SDL que en el caso de que ejecutemos el juego en modo ventana me la centre en el escritorio. Esto lo hacemos creando una variable de entorno:

```
SetEnvironmentVariable('SDL_VIDEO_CENTERED', '1');
```

Después le decimos a la librería OpenGL que vamos a renderizar polígonos con 32 bits de color y que active el doble buffer:

```
SDL_GL_SetAttribute(SDL_GL_DEPTH_SIZE, 32);
SDL_GL_SetAttribute(SDL_GL_DOUBLEBUFFER, 1);
```

Y aquí tenemos la razón de porque pase todo el juego a OpenGL:

```
// Activamos la sincronización vertical
SDL_GL_SetAttribute(SDL_GL_SWAP_CONTROL, 1);
```

Luego activamos el modo de vídeo en modo ventana o pantalla completa según lo que nos pasen como parámetro:

```
if bPantallaCompleta then
  Pantalla := SDL_SetVideoMode(iAncho, iAlto, 32,
```

```
SDL_FULLSCREEN or SDL_OPENGL)
else
  Pantalla := SDL_SetVideoMode(iAncho, iAlto, 32,
SDL_OPENGL);

if Pantalla = nil then
begin
  ShowMessage('Error al cambiar de modo de video.');
  SDL_Quit;
  Exit;
end;
```

La variable **Pantalla** es una superficie de SDL donde vamos a renderizar todos los polígonos:

```
var
  Pantalla: PSDL_Surface; // Pantalla principal de vídeo
```

PREPARANDO LA ESCENA OPENGL

Al final del procedimiento **ModoVideo** llamamos a otro procedimiento llamado **InicializarOpelGL** que inicializa la escena donde vamos a dibujar los polígonos:

```
procedure InicializarOpenGL(iAncho, iAlto: Integer);
var
  rRatio: Real;
begin
  rRatio := CompToDouble(iAncho) / CompToDouble(iAlto);

  // Suavizamos los márgenes de los polígonos
  glShadeModel(GL_SMOOTH);

  // Ocultamos las caras opuestas
  glCullFace(GL_BACK);
  glEnable(GL_CULL_FACE);

  // Ponemos el negro como color de fondo
  glClearColor(0, 0, 0, 0);

  // Configuramos el zbuffer
  glClearDepth(1);

  // Establecemos la perspectiva de visión
  gluPerspective(60, rRatio, 1.0, 1024.0);

  // Habilitamos el mapeado de texturas
  glEnable(GL_TEXTURE_2D);

  // Activamos el z-buffer
  glEnable(GL_DEPTH_TEST);
  glDepthMask(TRUE);
```

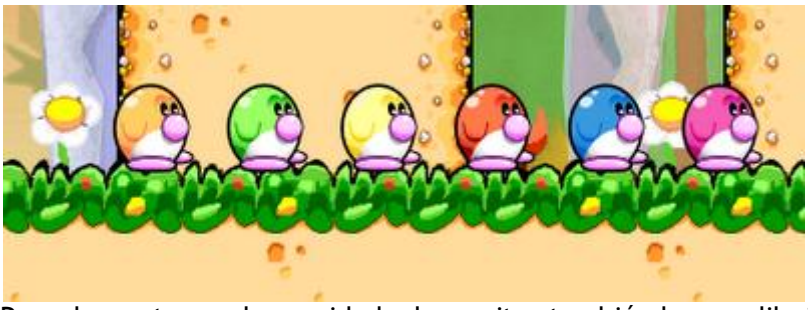
```
  // Sólo se dibujarán aquellas caras cuyos vértices se
hallen en sentido
  // de las agujas del reloj
  glFrontFace(GL_CW);

  glViewport(0,0,640,480);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(0,640,0,480,-100,100);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
end;
```

Lo primero que hacemos en este procedimiento es calcular el ratio de perspectiva de la cámara respecto a al ancho y alto de la superficie:

```
var
  rRatio: Real;
begin
  rRatio := CompToDouble(iAncho) / CompToDouble(iAlto);
```

Luego configuramos la impresión de polígonos para que suavice los bordes dentados que aparecen en los márgenes de cada polígono. Esto es importante porque entre esta función y el suavizado utilizando el canal alfa da un aspecto a los sprites de dibujos animados:

```
glShadeModel(GL_SMOOTH);
```



Para dar un toque de suavidad a los sprites también hemos dibujado el contorno de los mismos de color negro pero dando una pequeña semitransparencia para que se mezcle con el fondo. Aunque el juego esté solo a 640 x 480 parece que está a más resolución.

Ahora le decimos que vamos a ocultar los polígonos con las caras opuestas. Esto sobre todo tiene más sentido en juegos 3D:

```
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
```

Le estamos diciendo que sólo queremos que imprima los polígonos que estén mirando a la cámara según el orden que demos a los vértices que veremos más adelante. Imaginaos un cubo en 3D:


Si os dais cuenta, sólo se ven tres caras a la vez. Las otras tres caras siempre quedan ocultas. Entonces, ¿para que imprimirlas? Los responsables de OpenGL ya pensaron en esta situación y crearon esta opción para que sólo se impriman los polígonos que están de cara a la cámara. Como nuestro juego es en 2D siempre se van a ver todos. Pero es bueno activarlo por si alguna vez metemos una rotación.

Le indicamos que el fondo de la pantalla va a ser de color negro:

```
glClearColor(0, 0, 0, 0);
```

El cuarto componente es el canal alfa (la transparencia). Luego configuramos la profundidad del Z-Buffer:

```
glClearDepth(1);
```

El Z-Buffer en este caso es muy importante. Determina el orden de superposición de polígonos. Si no lo activamos lo mismo aparece un enemigo delante del personaje que detrás, incluso podría provocar parpadeos si se cruzan dos polígonos que están en la misma coordenada Z. Ya veremos esto detenidamente con los sprites.

Fijamos la perspectiva de la cámara:

```
gluPerspective(60, rRatio, 1.0, 1024.0);
```

Esta perspectiva determina el enfoque de la cámara respecto a la escena. Podemos acercarla o alejarla a nuestro antojo. En este caso la he ajustado exactamente a la pantalla. Ahora activamos la carga de texturas:

```
glEnable(GL_TEXTURE_2D);
```

Activamos el buffer Z:

```
glEnable(GL_DEPTH_TEST);
glDepthMask(TRUE);
```

La siguiente función le indica el sentido de los vértices de los polígonos (en el sentido de las agujas del reloj):

```
glFrontFace(GL_CW);
```

Todos los polígonos que cuyo órden de los vértices esté al contrario de las agujas del reloj no se imprimirán.

A continuación fijamos el ancho y alto de la proyección en pantalla así como el sistema de coordenadas:

```
glViewport(0, 0, iAncho, iAlto);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
```

```
glOrtho(0,640,0,480,-100,100);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
```

La librería OpenGL es tan flexible que permite establecer que rango van a tener nuestras coordenadas. En este caso van a ser de 0 a 640 horizontalmente y de 0 a 480 verticalmente. Originalmente el eje de coordenadas está en el centro de la pantalla lo que es un coñazo a menos que vayamos a hacer juegos en 3D con coordenadas de polígonos que vengan de programas como 3D Studio, Maya, etc.

Aquí la función más importante es **glOrtho** que pasa el sistema de coordenadas de 3D a un sistema ortogonal 2D. Si lo hubiésemos dejado en 3D daría la sensación de que las piezas se resquebrajan en las orillas de la pantalla y se quedan bien en el centro. Por ello quitamos la perspectiva tridimensional. Con este sistema, independientemente de la coordenada Z de los polígonos siempre se verá igual ante la cámara, sin profundidad.

Esto ya lo aclararé cuando llegue a la rutina de impresión de polígonos. En el próximo artículo veremos la clase**TSprite** y como lo convertí todo a OpenGL.

Pruebas realizadas en Delphi 7.

18 septiembre 2009

Mi primer videojuego independiente (1)

Durante una serie de artículos os voy a explicar mi pequeña experiencia en lo que se refiere a la programación de mi primer videojuego independiente y comercial así como su venta y distribución. Sobre todo voy a orientarme en la programación en Delphi.

Lo que voy a explicar viene a ser una extensión de esta serie de artículos que escribí hace tiempo sobre programación con SDL:

Programar videojuegos con la librería SDL (1)
Programar videojuegos con la librería SDL (2)
Programar videojuegos con la librería SDL (3)
Programar videojuegos con la librería SDL (4)
Programar videojuegos con la librería SDL (5)
Programar videojuegos con la librería SDL (6)
Programar videojuegos con la librería SDL (7)
Programar videojuegos con la librería SDL (8)
Programar videojuegos con la librería SDL (9)
Programar videojuegos con la librería SDL (10)
Programar videojuegos con la librería SDL (11)

Para los que crean que con Delphi sólo se pueden crear aplicaciones de gestión y utilidades les voy a demostrar que están muy equivocados. Trabajando a tiempo parcial y durante 7 meses al final pudimos hacer entre dos personas (programador y diseñador gráfico) el videojuego Pequepon Adventures que he metido en mi página web deDivernova Games.

Y todo con Delphi 7 y programando a pelo en Win32. Ni siquiera llego a utilizar el objeto **Application**. Podéis descargar la demo de aquí:

http://www.divernovagames.com/pequeponadventures.html

Si hubiese trabajado 8 horas al día de Lunes a Viernes sin matarme, realmente hubiera tardado unos 3 meses en realizarlo. Lo que más que costó fue el diseño de pantallas, ya que el juego tiene 30 niveles con más o menos 10 pantallas cada uno, lo que dan un total de 300 pantallas.


Aunque pueda parecer sencillo al principio, diseñar pantallas es desesperante. Para hacer un videojuego como dios manda creo que harían falta por lo menos cuatro personas (programador, diseñador de niveles, diseñador gráfico y músico).

PARA APRENDER HAY QUE EQUIVOCARSE

Cuando uno es novato en estos temas lo primero que hace es tirarse al toro sin planificar nada y teniendo las ideas más o menos claras pero sin un objetivo concreto. Durante muchos años he intentado hacer videojuegos primero en lenguaje ensamblador programando directamente los gráficos a la SVGA mediante puertos, luego en C/C++ con

las librerías Allegro y SDL para terminar programando el Delphi con SDL y OpenGL. Eso sin contar con los DIV Games Studio, Fenix, GameMaker y demás hierbas.

Pero lo que pasa siempre es que empiezas el juego con ilusión y conforme van pasando los meses y va incrementándose la dificultad al final abandonas creyendo que el lenguaje o la librería gráfica que estas utilizando no son lo suficientemente buenos y comienzas a programar en otros lenguajes más potentes.

Pero el problema no esta en los lenguajes, está en nosotros mismos. Hay que comenzar con un proyecto pequeño y abarcable a corto plazo, da igual que sea cutre y pequeño, lo importante es aprender a terminarlo en un tiempo estimado y sin errores.

Podéis comenzar con algo como un Tetris, un juego de objetos ocultos o una pequeña aventura gráfica con un plazo no superior a tres meses. Una vez terminado el primer juego tendremos la moral suficiente para comenzar el siguiente con más calidad y contenidos.

Tampoco creo lo que dicen muchos que para crear un buen proyecto hay que estar motivado e ilusionado. Con el tiempo la ilusión se acaba y terminas abandonando. Creo que hay que tener una meta clara: ganar dinero. Lo demás son tonterías. Tu modelo de

negocio puede basarse en hacer algo gratuito que descargue mucha gente y vivir de la publicidad, o bien cobrar por copia (arriesgándonos siempre con el pirateo).

EL DESARROLLO DE PEQUEPON ADVENTURES

Uno no se explica como un juego tan pequeño puede dar tantos quebraderos de cabeza. Y es por una sencilla razón: la falta de experiencia. Ya puedes leerte 100 libros de programación en los mejores lenguajes o en las librerías OpenGL o DirectX, pero hasta que no te pones a programar no te puedes ni imaginar los problemas que van a salir.

Después de tirarme 6 meses programando todo el videojuego, me puse a probarlo en otros equipos y me llevé un chasco impresionante. La librería SDL en modo 2D (sin utilizar aceleración gráfica) funciona muy bien cuando las pantallas son estáticas, pero si realizamos un scroll entonces el retrazo vertical de algunas tarjetas gráficas llega a crear unos cortes tan feos y parpadeantes que pueden matar a un epiléctico.

Me ofusqué buscando por Internet como activar la sincronización vertical en este modo de vídeo pero no encontré absolutamente nada. Todo el mundo decía que sólo funciona cuando la SDL dibuja polígonos OpenGL.

Así que después de una semana maldiciendo mi suerte me propuse dedicarme un mes más a cambiar todo el motor gráfico a OpenGL con las siguientes ventajas y dificultades:

- Todas las rutinas de dibujado de sprites hay que rehacerlas de nuevo.

- Hay que dibujar con polígonos con un ancho y alto que sea potencia de 2 (32x32, 64x64, 128x256, 32x128, etc.).

- Los polígonos no pueden superar un máximo de 256x256. Si se puede pero no todas las tarjetas lo soportan, por lo que si queréis que vuestro videojuego funcione en el mayor número de ordenadores posible no hay que pasarse de ese rango. Supongo que las últimas versiones de DirectX se pasarán esto por el forro.

- Las texturas de los polígonos hay que enviarlas todas de una vez a la memoria de la tarjeta de vídeo antes de comenzar a dibujar. No podemos subir o eliminar texturas en tiempo real porque el desastre puede ser impresionante.

- Por fin podemos activar el retrazo vertical.

- No es necesario crear pantallas temporales ni doble o triple buffer. OpenGL se encarga de todo.

- Aunque OpenGL puede dibujar polígonos de todo tipo yo prefiero partirlo todo en triángulos para que la tarjeta gráfica no tenga que complicarse la vida, aunque las últimas GPU ya ha hacen de todo. Por tanto, cada sprite tiene dos polígonos (triángulos).

- Al dibujar sprites por hardware el consumo de recursos es mínimo. Si ejecutaís **Pequepon Adventure**s en modo ventana (ver Opciones) y abrís el **Administrador de tareas** de Windows veréis que a veces el juego llega a consumir entre un 0 y un 5% de procesador y todo a 40 frames por segundo. Esto es importantísimo en portátiles para que dure más la batería.

El resultado fue menos traumático de lo que me creía ya que aproveché el código fuente que tenía en C++ de hace años cuando intenté hacer videojuegos 3D en OpenGL con mi propio motor 3D tipo Quake, pero nunca lo terminé por lo que he hablado: el mucho abarca poco aprieta.

 Así que en estos artículos pondré fragmentos de código referentes al este nuevo motor 2D pero con aceleración gráfica OpenGL. Como comprenderéis no voy a dar todo el código fuente del juego por dos razones: es un juego comercial y que son 9.700 líneas de código. Pero sí hablaré de las partes que he considerado más difíciles.

EL EDITOR DE PANTALLAS

Programar un juego no sólo es hacer un motor 2D o 3D. Tenemos que crear herramientas externas que guarden la información que necesitamos. En mi caso fue el editor de pantallas. El juego esta programado a una resolución de 640 x 480. Lo que hice fue partir la pantalla en piezas (tiles) de 40 x 40 pixels, lo que sale un total de 16 piezas horizontales y 12 verticales:

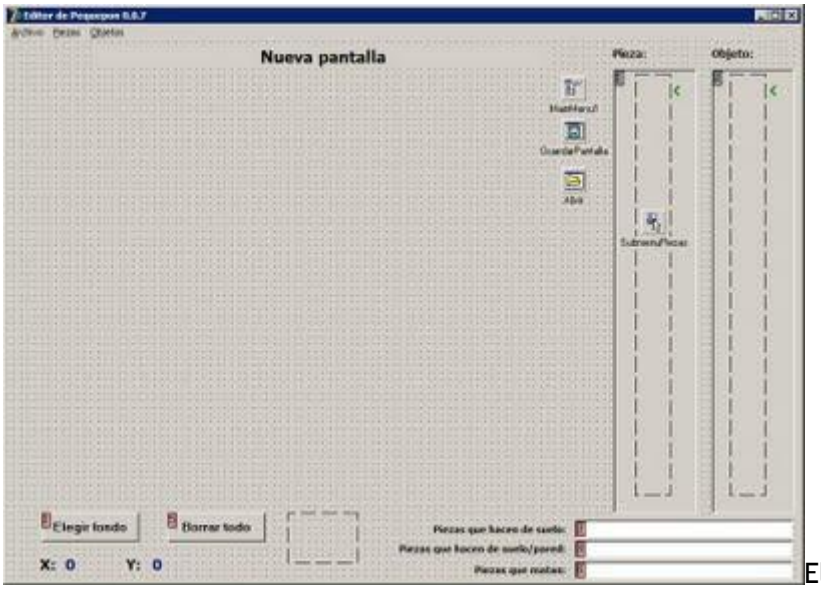
Aquí cometí mi primer error. Al principio pensamos en hacer un pequeño videojuego gratuito tipo Snowy con una sola pantalla por nivel. Pero vimos que el juego se hacía muy corto. O hacíamos las piezas más pequeñas o le metíamos más pantallas por la derecha haciendo el Pequepon saltara a la siguiente pantalla cuando llegara a la parte derecha de la pantalla.

Pero entonces se me cruzaron los cables e intenté hacer un scroll. Al principio me salió lento y horrible. Pero entonces comencé a optimizar el motor y llegué a hacer un scroll suave y aceptable que se mueve de 3 en 3 pixels.

El error cometido fue el tener que dibujar entre dos pantallas. Cuando estas entre la primera y la segunda pantalla tienes que dibujar un trozo de cada, con la dificultad que conlleva. Cuando lleguemos a la parte de mi motor 2D os diré como resolví el problema.

Lo que debería haber hecho es un mundo gigante con un buen array bidimensional que abarque todas las pantallas de ese nivel. Ya he aprendido la lección para el siguiente juego. Pero volvamos al editor.

El editor es una aplicación normal de Delphi que contiene un solo formulario:



El juego se dibuja a tres capas:

1º capa: pantalla de fondo.

2º capa: las piezas con las que choca pequepon (suelos, paredes, bloques, etc.).

3º capa: los objetos que recogemos (fresas, llave, esfera, puerta, etc.).

En la parte derecha del editor tenemos una columna para las piezas y otra para los objetos. Como hay más piezas y objetos de lo que pueden caber en pantalla, lo que hice fue meter los bitmaps de las piezas y los objetos dentro de un componente **TScrollBox** para poder llegar a todas moviendo la barra de desplazamiento vertical.

Si selecciono una pieza o un objeto coloco una flecha verde a su lado. Luego cuando nos vamos a la pantalla si pinchamos con el botón izquierdo del ratón dibujamos piezas y si lo hacemos con el botón derecho del ratón ponemos objetos. La primera pieza y el primer objeto los he dejado transparentes.

¿Por qué el fondo de las piezas son de color rosa? Pues porque es el color que utilizo de máscara para dibujarlas. No me interesaba el negro porque tanto los personajes como los objetos tienen el borde negro.

Entonces en el evento **OnMouseDown** del formulario compruebo primero si estamos dentro de la pantalla y comenzamos a dibujar:

```
procedure TFPrincipal.FormMouseDown(Sender: TObject;
Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var i, j: Integer;
begin
  if Button = mbLeft then
  begin
    bIzquierdo := True;

    // ¿Está dentro de la pantalla?
    if ( x > 40 ) and ( y > 40 ) and ( x <= 680 ) and ( y
<= 520 ) then
    begin
      i := x div 40;
      j := y div 40;
      EX.Caption := IntToStr( i );
      EY.Caption := IntToStr( j );

      Piezas[i,j] := iPieza;
      DibujarPantalla;
    end;
  end;

  if Button = mbRight then
  begin
    bDerecho := True;

    // ¿Está dentro de la pantalla?
    if ( x > 40 ) and ( y > 40 ) and ( x <= 680 ) and ( y
```

```
<= 520 ) then
    begin
      i := x div 40;
      j := y div 40;
      EX.Caption := IntToStr( i );
      EY.Caption := IntToStr( j );

      Objetos[i,j] := iObjeto;
      DibujarPantalla;
    end;
  end;
end;
```

Las piezas y los objetos son dos array bidimensionales de bytes, por lo tanto solo podemos poner un máximo de 255 piezas distintas:

```
var
  Piezas, Objetos: array[1..16,1..12] of byte;
```

Esto no me preocupa porque por cada mundo vuelvo a cargar piezas nuevas:

En cambio, los objetos son iguales para todos los mundos. Aún así sólo he gastado 77 objetos de los 255 que tengo disponibles. Si necesitáis más piezas pues hacéis un array de **DWord**. La procedimiento de **DibujarPantalla** es este:

```
procedure TFPrincipal.DibujarPantalla;
var i, j: Integer;
    Origen, Destino: TRect;
begin
  // Lo dibujamos todo el el buffer

  with Buffer.Canvas do
  begin
    // Primero dibujamos el fondo
    Origen.Top := 0;
    Origen.Left := 0;
```

```
    Origen.Right := 640;
    Origen.Bottom := 480;
    Destino.Top := 0;
    Destino.Left := 0;
    Destino.Right := 640;
    Destino.Bottom := 480;
    CopyMode := cmSrcCopy;
    CopyRect( Destino, ImagenFondo.Canvas, Origen );

    for j := 1 to 12 do
      for i := 1 to 16 do
      begin
        if Piezas[i,j] > 0 then
        begin
          Origen.Top := Piezas[i,j] * 40;
          Origen.Left := 0;
          Origen.Right := 40;
          Origen.Bottom := Piezas[i,j] * 40 + 40;
          Destino.Top := (j-1)*40;
          Destino.Left := (i-1)*40;
          Destino.Right := (i-1)*40 + 40;
          Destino.Bottom := (j-1)*40 + 40;

          // ¿La pieza que va a poner ya no existe?
          if Piezas[i,j] > ImagenPiezas.Height div 40 then
          begin
            Brush.Color := clRed;
            FillRect( Destino );
          end
          else
            if MascaraPiezas = nil then
            begin
              CopyMode := cmSrcCopy;
              CopyRect( Destino, ImagenPiezas.Canvas,
Origen );
            end
            else
            begin
              CopyMode := cmSrcAnd;
              CopyRect( Destino, MascaraPiezas.Canvas,
Origen );
              CopyMode := cmSrcPaint;
              CopyRect( Destino, MascaraPiezas2.Canvas,
Origen );
            end;
        end;

        if Objetos[i,j] > 0 then
        begin
          Origen.Top := Objetos[i,j] * 40;
          Origen.Left := 0;
```

```
          Origen.Right := 40;
          Origen.Bottom := Objetos[i,j] * 40 + 40;
          Destino.Top := (j-1)*40;
          Destino.Left := (i-1)*40;
          Destino.Right := (i-1)*40 + 40;
          Destino.Bottom := (j-1)*40 + 40;

          // ¿El objeto que va a poner ya no existe?
          if Objetos[i,j] > ImagenObjetos.Height div 40
then
          begin
            Brush.Color := clYellow;
            FillRect( Destino );
          end
          else
            if MascaraObjetos = nil then
            begin
              CopyMode := cmSrcCopy;
              CopyRect( Destino, ImagenObjetos.Canvas,
Origen );
            end
            else
            begin
              CopyMode := cmSrcAnd;
              CopyRect( Destino, MascaraObjetos.Canvas,
Origen );
              CopyMode := cmSrcPaint;
              CopyRect( Destino, MascaraObjetos2.Canvas,
Origen );
              //CopyRect( Destino, ImagenObjetos.Canvas,
Origen );
            end;
        end;
      end;
  end;

  // copiamos el buffer a pantalla
  with Canvas do
  begin
    Origen.Top := 0;
    Origen.Left := 0;
    Origen.Right := 640;
    Origen.Bottom := 480;
    Destino.Top := 40;
    Destino.Left := 40;
    Destino.Right := 680;
    Destino.Bottom := 520;
    CopyMode := cmSrcCopy;
    CopyRect( Destino, Buffer.Canvas, Origen );
  end;
end;
```

Dibujar con el canvas de Delphi es un auténtico coñazo. Tenemos primero que cargar los sprites, crear una máscara y luego dibujar la máscara y después los sprites. Para ello tuve que declarar primero todas estas imágenes:

```
var
  MascaraPiezas, MascaraPiezas2, Buffer: TImage;
  MascaraObjetos, MascaraObjetos2: TImage;
```

Y para crear la máscara recorro todos los pixels de la imagen y si el color es rosa entonces invierto los pixels o los elimino:

```
procedure TFPrincipal.CrearMascaraPiezas;
var i, j: Integer;
begin
  // Creamos la máscara de color blanco con el contorno del
sprite
  MascaraPiezas := TImage.Create( nil );
  MascaraPiezas.Width := ImagenPiezas.Width;
  MascaraPiezas.Height := ImagenPiezas.Height;

  MascaraPiezas2 := TImage.Create( nil );
  MascaraPiezas2.Width := ImagenPiezas.Width;
  MascaraPiezas2.Height := ImagenPiezas.Height;

  for j := 0 to ImagenPiezas.Height - 1 do
    for i := 0 to ImagenPiezas.Width - 1 do
    begin
      if ImagenPiezas.Canvas.Pixels[i,j] = $FF00FF then
      begin
        MascaraPiezas.Canvas.Pixels[i,j] := clWhite;
        MascaraPiezas2.Canvas.Pixels[i,j] := clBlack;
      end
      else
      begin
        MascaraPiezas.Canvas.Pixels[i,j] := clBlack;
        MascaraPiezas2.Canvas.Pixels[i,j] :=
ImagenPiezas.Canvas.Pixels[i,j];
      end;
    end;
end;
```

Luego hay que acordarse de destruirlo todo al cerrar el formulario:

```
procedure TFPrincipal.FormDestroy(Sender: TObject);
begin
  Suelos.Free;
  Paredes.Free;
  Matan.Free;
  Buffer.Free;
  MascaraObjetos.Free;
  MascaraObjetos2.Free;
```

```
  MascaraPiezas.Free;
  MascaraPiezas2.Free;
end;
```

Tanto para el videojuego como para este editor utilicé el experto EurekaLog como mi compañero inseparable. Cuando seleccionamos **Archivo** -> **Abrir** cargo la pantalla:

```
procedure TFPrincipal.AbrirPClick(Sender: TObject);
begin
  Abrir.InitialDir := sRutaDat;
  Abrir.Filter := 'Pantalla (*.pan)|*.pan';

  if Abrir.Execute then
  begin
    sArchivoPantalla := Abrir.FileName;
    CargarPantalla( sArchivoPantalla );
    ETituloPantalla.Caption := ExtractFileName (
Abrir.FileName );
  end;
end;
```

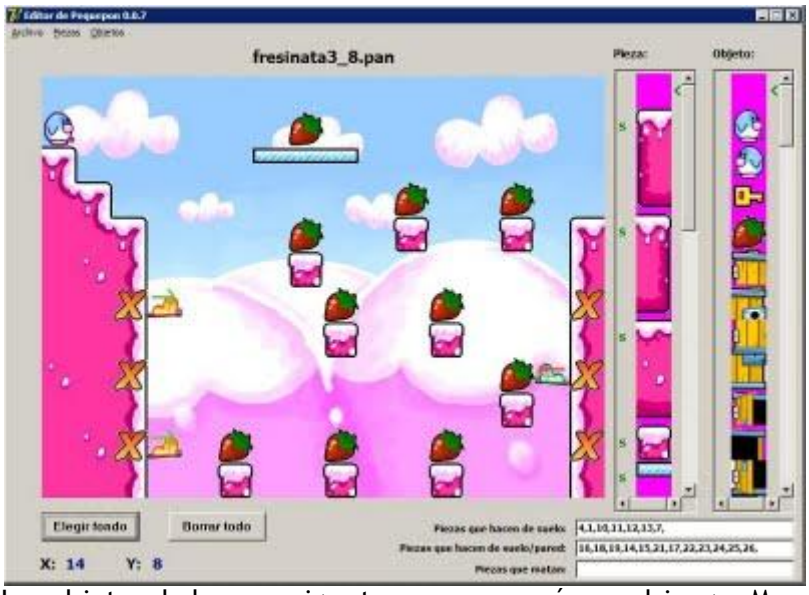
La rutina de cargar pantalla debe cargar el fondo que tiene esta pantalla, las piezas, los objetos, la posición del heroe, los enemigos, etc.:

```
procedure TFPrincipal.CargarPantalla( sArchivo: String );
var
  F: File of byte;
  i, j, iNumSue, iNumPar, iNumMat, iNumSueReal,
iNumParReal, iNumMatReal: Integer;
  b: Byte;
begin
  AssignFile( F, sArchivo );
  Reset( F );

  // Cargamos las piezas
  for j := 1 to 12 do
    for i := 1 to 16 do
      Read( F, Piezas[i,j] );

  // Cargamos los objetos
  for j := 1 to 12 do
    for i := 1 to 16 do
      Read( F, Objetos[i,j] );

  // Leemos el nombre de el archivo de piezas
  sArchivoPiezas := '';
  for i := 1 to 30 do
  begin
    Read( F, b );
    if b <> 32 then
      sArchivoPiezas := sArchivoPiezas + Chr( b );
  end;
```

```
CargarPiezas( sRutaGfcs + sArchivoPiezas + '.bmp' );

// Leemos el nombre de el archivo de fondo
sArchivoFondo := '';
for i := 1 to 30 do
begin
  Read( F, b );
  if b <> 32 then
    sArchivoFondo := sArchivoFondo + Chr( b );
end;
CargarFondo( sRutaGfcs + sArchivoFondo + '.bmp' );

// Leemos el nombre de el archivo de piezas
sArchivoObjetos := '';
for i := 1 to 30 do
begin
  Read( F, b );
  if b <> 32 then
    sArchivoObjetos := sArchivoObjetos + Chr( b );
end;
CargarObjetos( sRutaGfcs + sArchivoObjetos + '.bmp' );

Suelos.Clear;
Paredes.Clear;
Matan.Clear;

if not Eof(F) then
begin
  // Leemos el número de suelos
  Read( F, b );
  iNumSue := b;

  // Leemos el número de paredes
  Read( F, b );
  iNumPar := b;

  // Leemos los suelos
  iNumSueReal := 0;
  for i := 1 to iNumSue do
  begin
    if not Eof(F) then
      Read( F, b );

    if b < ImagenPiezas.Height div 40 then
    begin
      Suelos.Add( IntToStr( b ) );
      Inc(iNumSueReal);
    end;
  end;
  iNumSue := iNumSueReal;
```

```
    // Leemos las paredes
    iNumParReal := 0;
    for i := 1 to iNumPar do
    begin
      if not Eof(F) then
        Read( F, b );

      if b < ImagenPiezas.Height div 40 then
      begin
        Paredes.Add( IntToStr( b ) );
        Inc(iNumParReal);
      end;
    end;
    iNumPar := iNumParReal;

    // Leemos el número que matan
    if not Eof(F) then
    begin
      Read( F, b );
      iNumMat := b;

      // Leemos los que matan
      iNumMatReal := 0;
      for i := 1 to iNumMat do
      begin
        if not Eof(F) then
          Read( F, b );

        if b < ImagenPiezas.Height div 40 then
        begin
          Matan.Add(IntToStr(b));
          Inc(iNumMatReal);
        end;
      end;

      iNumMat := iNumMatReal;
    end;
  end;

  CloseFile( F );
  sArchivo := Abrir.FileName;
  DibujarPantalla;
  MostrarSuelosParedes;
end;
```

Los objetos también los utilizo para colocar los enemigos en pantalla y saber que recorrido van a tener. Para delimitar los movimientos de los enemigos cree un objeto especial (la X) que se ve en el editor pero no en el juego. Cuando un enemigo encuentra este objeto da la vuelta:



Los objetos de los enemigos tampoco se verán en el juego. Me valen para saber donde comienza cada enemigo y el movimiento que va a hacer. Para hacer el editor más cómodo también hice que si dejamos pulsado los botones izquierdo o derecho del ratón podemos dibujar una columna o fila de bloques sin tener que hacer clic cada vez. Esto se hace con el evento **OnMouseMove**:

```
procedure TFPrincipal.FormMouseMove(Sender: TObject; Shift:
TShiftState; X,
  Y: Integer);
var i, j: Integer;
begin
  // ¿Está dentro de la pantalla?
  if ( x > 40 ) and ( y > 40 ) and ( x <= 680 ) and ( y <=
520 ) then
  begin
    i := x div 40;
    j := y div 40;

    if ( i < 1 ) or ( i > 16 ) or ( j < 1 ) or ( j > 12 )
then
      Exit;

    EX.Caption := IntToStr( i );
    EY.Caption := IntToStr( j );

    // ¿Está pulsado el botón izquierdo del ratón?
    if bIzquierdo then
    begin
      Piezas[i,j] := iPieza;
      DibujarPantalla;
    end;

    // ¿Está pulsado el botón izquierdo del ratón?
    if bDerecho then
    begin
      Objetos[i,j] := iObjeto;
```

```
      DibujarPantalla;
    end;
  end;
end;
```

Pero como tenemos que saber si hemos dejado pulsado el botón izquierdo o derecho del ratón entonces creé dos variables globales:

```
var
  bIzquierdo: Boolean; // ¿Está pulsado el botón izquierdo
del ratón?
  bDerecho: Boolean;   // ¿Está pulsado el botón derecho
del ratón?
```

Y en los eventos **OnMouseDown** que hemos visto antes y en el evento **OnMouseUp** controlo cuando el usuario los ha apretado o soltado:

```
procedure TFPrincipal.FormMouseUp( Sender: TObject; Button:
TMouseButton;
  Shift: TShiftState; X, Y: Integer );
begin
  if Button = mbLeft then
    bIzquierdo := False;

  if Button = mbRight then
    bDerecho := False;
end;
```

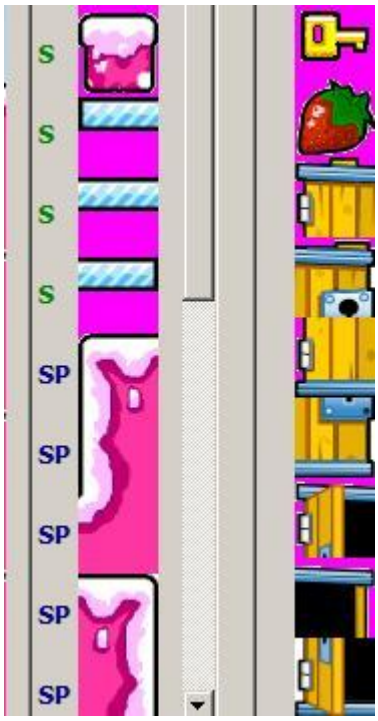
Lo que más me costó al principio fue la rutina de guardar pantalla ya que hay que hay que guardar el nombre del bitmap de fondo, las piezas, los objetos (y enemigos) así como saber con que piezas choca el personaje y los enemigos:

```
procedure TFPrincipal.GuardaPantalla;
var
  F: file of byte;
  i, j: Integer;
  Espacio, b: Byte;
begin
  if sArchivoPiezas = '' then
  begin
    Application.MessageBox( 'Debe seleccionar el archivo de
las piezas.',
      'Atención', MB_ICONEXCLAMATION );
    Exit;
  end;

  if sArchivoFondo = '' then
  begin
    Application.MessageBox( 'Debe seleccionar la imagen de
fondo.',
      'Atención', MB_ICONEXCLAMATION );
    Exit;
```

```
  end;

  if sArchivoObjetos = '' then
  begin
    Application.MessageBox( 'Debe seleccionar el archivo de
los objetos.',
      'Atención', MB_ICONEXCLAMATION );
    Exit;
  end;

  // Guardamos las piezas y objetos de la pantalla
  AssignFile( F, sArchivoPantalla );
  Rewrite( F );
  Espacio := 32;

  // Piezas
  for j := 1 to 12 do
    for i := 1 to 16 do
      Write( F, Piezas[i,j] );

  // Objetos
  for j := 1 to 12 do
    for i := 1 to 16 do
      Write( F, Objetos[i,j] );

  // Guardamos el nombre del archivo de las piezas
  for i := 1 to 30 do
    if i <= Length( sArchivoPiezas ) then
      Write( F, Byte( sArchivoPiezas[i] ) )
    else
      Write( F, Espacio ); // espacio en blanco

  // Guardamos el nombre del archivo de fondo
  for i := 1 to 30 do
    if i <= Length( sArchivoFondo ) then
      Write( F, Byte( sArchivoFondo[i] ) )
    else
      Write( F, Espacio ); // espacio en blanco

  // Guardamos el nombre del archivo de los objetos
  for i := 1 to 30 do
    if i <= Length( sArchivoObjetos ) then
      Write( F, Byte( sArchivoObjetos[i] ) )
    else
      Write( F, Espacio ); // espacio en blanco

  // Guardamos el número de suelos
  b := Suelos.Count;
  Write(F, b);

  // Guardamos el número de paredes
```

```
  b := Paredes.Count;
  Write(F, b);

  // Guardamos los suelos
  for i := 0 to Suelos.Count-1 do
  begin
    b := StrToInt(Suelos[i]);
    Write(F, b);
  end;

  // Guardamos las paredes
  for i := 0 to Paredes.Count-1 do
  begin
    b := StrToInt(Paredes[i]);
    Write(F, b);
  end;

  // Guardamos el número que matan
  b := Matan.Count;
  Write(F, b);

  // Guardamos los que matan
  for i := 0 to Matan.Count-1 do
  begin
    b := StrToInt(Matan[i]);
    Write(F, b);
  end;

  CloseFile( F );
end;
```

Si os fijáis en las columnas de las piezas y los objetos a la izquierda guardo si cada pieza

es libre o es suelo o pared: 

Estoy es muy importante para controlar si nuestro personaje va a chocar sólo verticalmente al caer o choca por todos lados, por ejemplo con el bloque de piedra. Aunque mi editor lleva mucho más código creo que he comentado las partes más importantes. Crear un buen editor de niveles es fundamental para evitar programar demasiado. En el próximo artículo comenzaré a explicar el núcleo del juego y el motor 2D creado con SDL + OpenGL.

Pruebas realizadas en Delphi 7.

Publicado por Administrador en 10:36 11 comentarios
Etiquetas: juegos

11 septiembre 2009

El componente mxProtector (y 3)

Siguiendo con este fantástico componente destinado a proteger nuestras aplicaciones vamos a ver como activar un programa con una contraseña en concreto para un solo cliente. Ideal para programas a medida.

ACTIVACIÓN DE PROGRAMAS POR CLAVE DE ACCESO

El método es tan simple como no poder utilizar el programa o restringir el uso del mismo hasta que el usuario no introduzca una clave de acceso inventada por el programador y con la posibilidad de instalarlo en cualquier PC.

Para este ejemplo he creado un formulario con estos componentes:

Al arrancar el programa nos pedirá directamente la clave de acceso en el caso de que no esté registrada. Si nos

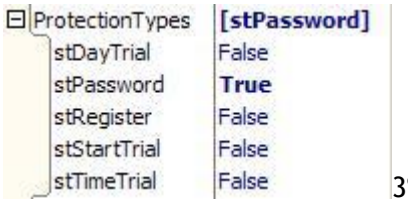
equivocamos al introducirla el programa entrará en modo Demo mostrando el mensaje **Programa sin registrar**.

Tenemos la oportunidad de pulsar el botón **Activar** para introducir la clave de nuevo o bien desactivarlo por si queremos restringir de nuevo el uso del mismo a otros usuarios del mismo PC.

Comencemos configurando el componente **mxProtector** de este modo:

1º En la propiedad **Options** activamos la opción **poPasswordOnce**:

2º Activamos la opción **stPassword** en la tabla en la propiedad **ProtectionTypes**:



3º En el campo password escribimos nuestra contraseña:

Al pulsar la tecla Intro veremos que la contraseña se encripta en otro formato:

Ya no se si esta será la clave encriptada o su hash. A nosotros nos da lo mismo mientras un hacker no pueda adivinarla por ingeniería inversa.

4º En el evento **OnGetPassword** debemos preguntarle al usuario por la contraseña:

```
procedure TFPrincipal.mxProtectorGetPassword(Sender:
TObject;
var Password: string);
begin
  Password := InputBox('Introduzca la clave de activación',
'Clave:', '');
end;
```

5º El evento **OnValidPassword** se ejecutará si el usuario ha acertado la clave, por lo que deshabilitamos el botón**Activar** y habilitamos el botón **Desactivar**:

```
procedure TFPrincipal.mxProtectorValidPassword(Sender:
TObject);
begin
  EMensaje.Caption := 'Programa registrado';
  BActivar.Enabled := False;
  BDesactivar.Enabled := True;
end;
```

6º Y en el caso de que la clave sea incorrecta hacemos lo contrario dentro del evento **OnWrongPassword**:

```
procedure TFPrincipal.mxProtectorWrongPassword(Sender:
TObject;
  WrongPassword: string);
begin
  EMensaje.Caption := 'Programa sin registrar';
  BActivar.Enabled := True;
  BDesactivar.Enabled := False;
  Application.MessageBox('Clave de activación incorrecta',
    'Consulte con su proveedor', MB_ICONSTOP);
end;
```

7º Para finalizar tenemos que pedir la clave al pulsar el botón **Activar**:

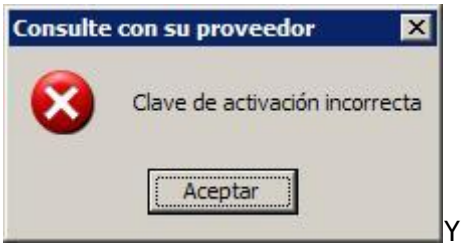
```
procedure TFPrincipal.BActivarClick(Sender: TObject);
begin
  mxProtector.CheckPassword;
end;
```

8º Y eliminarla al pulsar el botón **Desactivar**:

```
procedure TFPrincipal.BDesactivarClick(Sender: TObject);
begin
  mxProtector.Reset;
  Application.MessageBox('Deberá introducir la clave de
activación de nuevo',
    'Programa desactivado', MB_ICONINFORMATION);
  BActivar.Enabled := True;
  BDesactivar.Enabled := False;
end;
```

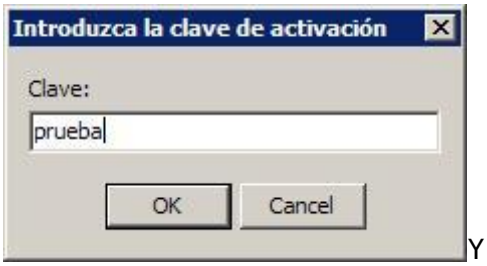
Vamos a probarlo. Al ejecutarlo, como no está registrado lo primero que hará es pedirnos la clave:

Si nos equivocamos mostrará el mensaje de error:



Y el programa aparece en modo demo:

Pulsamos de nuevo el botón **Activar**, introducimos la clave correcta:



Y el programa quedará registrado:

Y aunque cerremos el programa y volvamos a abrirlo, **mxProtector** comprobará de nuevo si la contraseña se ha activado, por lo que no tenemos que implementar ningún código al inicio de la aplicación.

Luego podemos pulsar el botón **Desactivar** para eliminar la licencia en ese equipo:

Este sería uno de los métodos más básicos de venta de software a medida. Cuando te paguen le mandas la clave y activas todas las funcionalidades del programa demo.

Si nos fijamos en los proyectos de demostración que lleva este componente en su directorio demo veremos que se pueden hacer muchas más combinaciones entre proteger con contraseña, limitar el programa por número de ejecuciones, limitar por número de días o por número de serie incluyendo una clave única de hardware.

Con esto finalizo la serie de artículos dedicada al componente **mxProtector**. Mi propósito era contemplar por encima todos los tipos de protección que abarca así su uso a nivel de principiante sin complicaciones.

Por los ejemplos que he visto, se pueden activar varias propiedades a la vez para que el programa sea a la vez una versión demo por tiempo y una vez que el usuario pague entonces se activa según un número de serie y su contraseña, como los programas Shareware profesionales que hay en el mercado.

Lástima que los creadores de este componente vayan a cerrar la página en Diciembre de este año. Espero que dejen por ahí en algún repositorio el código fuente de este maravilloso componente por si alguien se anima a seguirlo.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 09:38 2 comentarios
Etiquetas: seguridad

02 septiembre 2009

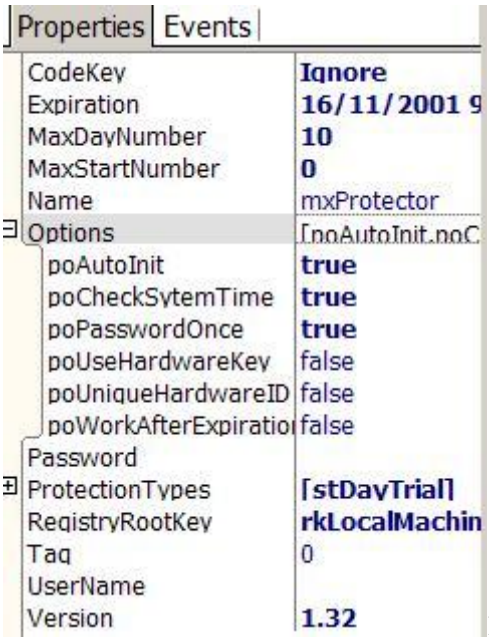
El componente mxProtector (2)

Vamos a continuar viendo otros modos de protección de este componente como pueden ser la limitación por número de días y por medio de licencias mediante un generador de claves.

PROTECCIÓN POR NÚMERO DE DÍAS TRANSCURRIDOS

Debemos configurar el componente **mxProtector** de este modo:

1º Dentro de su propiedad **Options** activamos las opciones **poAutoInit**, **poCheckSystemTime** y **poPasswordOnce**:



2º En su propiedad **Protection Types** debemos

activar **stDayTrial**:

3º También debemos poner en **MaxDayNumber** el número de días que damos de plazo, por ejemplo **30**.

3º En su evento **OnDayTrial** podemos el código que controla el paso de cada día:

```
procedure TFPrincipal.mxProtectorDayTrial(Sender: TObject;
  DaysRemained: Integer);
begin
  if DaysRemained = 1 then
    ENumDias.Caption := 'Sólo te queda un día'
  else
    ENumDias.Caption := Format('Te quedan %d días.',
[DaysRemained]);

  BReiniciar.Enabled := False;
end;
```

4º En su evento **OnExpiration** va el código cuando ya no quedan más días:

```
procedure TFPrincipal.mxProtectorExpiration(Sender:
TObject);
begin
  ENumDias.Caption := 'Le quedan 0 días. Su licencia ha
expirado';
  BReiniciar.Enabled := True;
end;
```

5º Y en su evento **OnInvalidSystemTime** podemos el mensaje que muestra que el usuario ha intentado mover hacia atrás la fecha de Windows para estirar la licencia:

```
procedure TFPrincipal.mxProtectorInvalidSystemTime(Sender:
TObject);
begin
  Application.MessageBox('La hora de tu sistema no es
correcta.',
    'Su licencia ha expirado', MB_ICONSTOP);
end;
```

6º Al pulsar el botón **Reiniciar** volvemos a darle otros 30 días:

```
procedure TFPrincipal.BReiniciarClick(Sender: TObject);
begin
  mxProtector.Reset;
end;
```

Y al igual que vimos en el ejemplo anterior de control por número de ejecuciones,

podemos implementar los
métodos **OnGetString**, **OnPutString**, **OnGetBoolean** y **OnPutBoolean** para que guarde el estado en la clave de registro que queramos o en un archivo INI, binario, etc. Esto es válido para todos los métodos de protección que hemos visto así como los siguientes que voy a comentar.

PROTECCIÓN POR REGISTRO DE NÚMERO DE SERIE

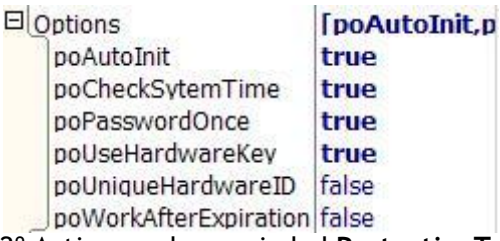
Esta es otra de las protecciones que más se suelen utilizar para distribuir programas con licencia Shareware. Según el nombre del usuario, un ID que haga único al PC y un número de serie podemos proteger cada licencia para que sólo se ejecute en un equipo.

Lo primero que vamos a hacer es el programa que va a registrar el usuario según un número de serie que nos dará un generador de claves que vamos a crear más adelante.

El programa va a tener este formulario:

Veamos como configurar el componente **mxProtector** igual que hemos visto anteriormente:

1º En la propiedad **Options** debemos activar las
opciones **poAutoInit**, **poCheckSystemTime**, **poPasswordOnce** y**poUseHardwareKey**:



2º Activar en la propiedad **ProtectionTypes** el valor **stRegister**:

3º En el evento **OnGetSerialNumber** le pasamos al componente el nombre del usuario y el número de serie generado:

```
procedure TFPrincipal.mxProtectorGetSerialNumber(Sender:
```

```
TObject;
var UserName, SerialNumber: string);
begin
  UserName := Usuario.Text;
  SerialNumber := NumSerie.Text;
end;
```

4º En el evento **OnInvalidSerialNumber** mostramos el mensaje de error en caso de que sea incorrecto:

```
procedure
TFPrincipal.mxProtectorInvalidSerialNumber(Sender:
TObject);
begin
  Application.MessageBox('N° de serie incorrecto',
    'Consulte con el proveedor', MB_ICONSTOP);
end;
```

5º En el evento **OnUnknowHardware** debemos mostrar un mensaje en caso de que no podamos obtener la ID del PC:

```
procedure TFPrincipal.mxProtectorUnknownHardware(Sender:
TObject);
begin
  Application.MessageBox('El hardware de este equipo es
incompatible con este software.',
    'Consulte con el proveedor', MB_ICONSTOP);
end;
```

6º Al pulsar el botón **Registrar** activamos el producto:

```
procedure TFPrincipal.BRegistrarClick(Sender: TObject);
begin
  mxProtector.Registration;
  ComprobarRegistro;

  if mxProtector.IsRegistered Then
  begin
    Application.MessageBox('Gracias por comprar el
producto',
      'Registro realizado', MB_ICONINFORMATION);
  end;
end;
```

El procedimiento de comprobar el registro es el siguiente:

```
procedure TFPrincipal.ComprobarRegistro;
begin
  if mxProtector.IsRegistered then
  begin
    Caption := 'Programa registrado';
    BRegistrar.Enabled := False;
```

```
    BDesinstalar.Enabled := True;
  end
  else
  begin
    Caption := 'Programa no registrado';
    BRegistrar.Enabled := True;
    BDesinstalar.Enabled := False;
  end;
end;
```

Activamos o desactivamos los botones de registrar o desinstalar así como el título del formulario según este registrado el programa o no.

7º A pulsar el botón **Desinstalar** desinstalamos la clave de registro:

```
procedure TFPrincipal.BDesinstalarClick(Sender: TObject);
begin
  mxProtector.Reset;
  Application.MessageBox('Ya puede desinstalar del
producto',
    'Registro cancelado', MB_ICONINFORMATION);
  ComprobarRegistro;
end;
```

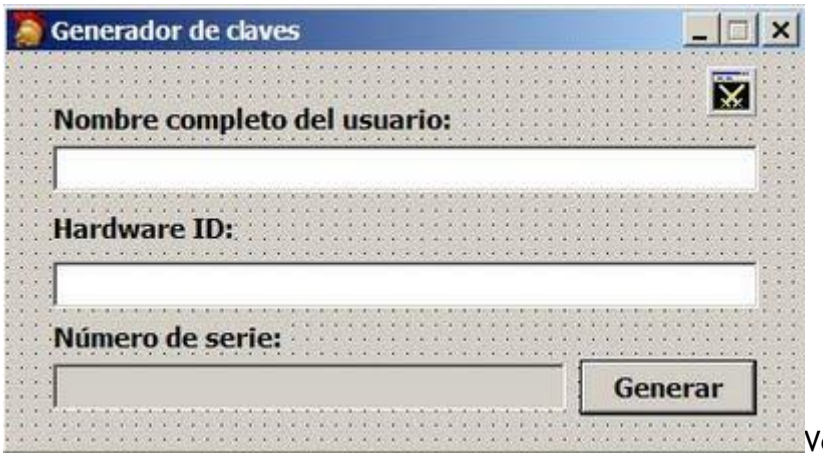
8º Por último, tenemos que comprobar en el evento **OnCreate** del formulario si hemos registrado el programa y obtener ID del PC:

```
procedure TFPrincipal.FormCreate(Sender: TObject);
begin
  ID.Text := mxProtector.GetHardwareID;
  ComprobarRegistro;
end;
```

CREANDO EL GENERADOR DE CLAVES

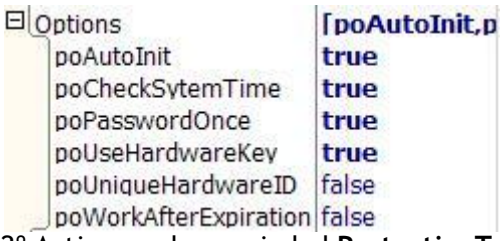
Ahora vamos a hacer un programa que genere números de serie según el nombre del usuario y un ID único del PC (número de serie del disco duro, de la tarjeta de Windows, etc.).

El programa va a tener este formulario:



Veamos como configurar el componente **mxProtector** para este programa:

1º En la propiedad **Options** debemos activar las opciones **poAutoInit**, **poCheckSystemTime**, **poPasswordOnce** y**poUseHardwareKey**:



2º Activar en la propiedad **ProtectionTypes** el valor **stRegister**:

3º En el evento **OnGetHardwareID** nos encargamos de pasarle al componente el ID del equipo:

```
procedure TFPrincipal.mxProtectorGetHardwareID(Sender:
TObject;
var HardwareID: string);
begin
  HardwareID := ID.Text;
end;
```

4º El botón **Generar** le pedimos al componente que nos genere un número de clave según el usuario y el ID del PC:

```
procedure TFPrincipal.BGenerarClick(Sender: TObject);
begin
  NumSerie.Text :=
mxProtector.GenerateSerialNumber(Usuario.Text);
end;
```

Vamos a probar ambos programas. Abrimos primero la aplicación:

Copiamos la clave que nos ha dado el programa, abrimos el generador de claves y escribimos nuestro nombre completo y el ID generado por el programa. Después pulsamos el botón **Generar** y nos dará la clave de registro:

Ahora copiamos la clave de registro, la llevamos al programa, introducimos nuestro nombre y pulsamos el botón**Registrar**:

Si cerramos la aplicación y volvemos a abrirla veremos que el programa ya está registrado. Podemos quitar la licencia pulsando el botón **Desinstalar** que dejará el programa como al principio.

Aunque parezca algo complicado, si tuviésemos que hacer todo esto a mano habría que

escribir mucho más código. En el próximo artículo seguiremos viendo otros modos de protección.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 13:08 14 comentarios 
Etiquetas: seguridad

31 julio 2009

El componente mxProtector (1)

Un asunto importante que no debemos descuidar en la distribución de nuestros programas es la protección de los mismos frente a las copias piratas. Para ello hay infinidad de herramientas que permiten proteger el software por número de ejecuciones, por fecha límite, activación por número de serie, etc.

Si no queremos complicarnos mucho la vida hay herramientas que permiten proteger un ejecutable una vez compilado como pueden ser Armadillo, ASProtect, ExeCryptor, Enigma Protector, IntelliProtector, etc. Pero casi todas ellas (sobre todo las más buenas) son comerciales y ninguna es perfecta. Un buen crackeador puede reventar cualquiera de ellas utilizando ingeniería inversa mediante desensambladores y dumpeadores de memoria avanzados.

Si os interesa investigar sobre seguridad informática hay una página web en la que también tiene código fuente de Delphi dedicada a este tema:

http://indetectables.net

Hablan sobre todo de cómo crear troyanos, virus, protectores de archivos EXE o como protegerse contra los mismos. Sobre todo hay que fijarse en el foro.

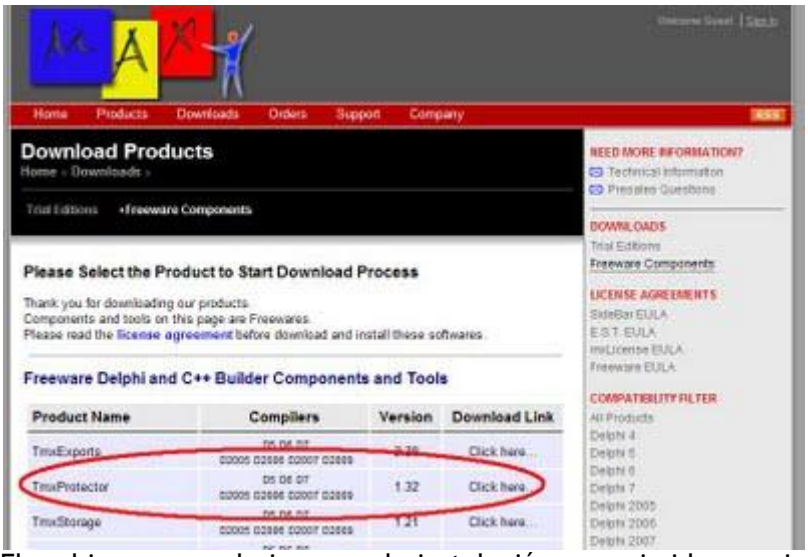
DESCARGANDO EL COMPONENTE

Lo que si podemos hacer es crear una protección a nuestro programa que sin ser muy compleja por lo menos evite que cualquier principiante pueda copiarlo. Para ello vamos a ver el componente **mxProtect** que cumple este cometido a la perfección. La versión actual es la 1.32 (a la fecha de escribir este artículo).

Este componente lo podemos encontrar en esta página web:

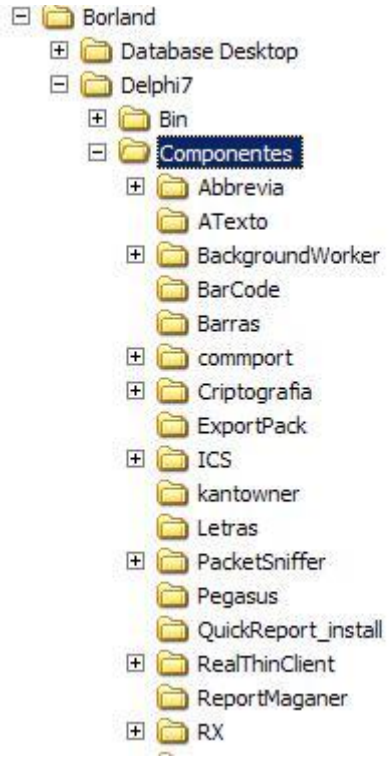
http://www.maxcomponents.net/

Tiene algunos componentes comerciales y otros con licencia freeware. Es este caso, el componente **mxProtect** es freeware y lo podemos descargar seleccionando en su página web el apartado **Downloads** -> **Freeware Components**:



El archivo que nos bajamos es la instalación comprimida con zip que tiene un tamaño de 355 KB. Es la típica instalación de siempre donde nos pedirá donde instalar el componente (por defecto en Archivos de programa):

Yo tengo por costumbre instalar los componentes en una carpeta debajo de cada versión de Delphi, por ejemplo:



De este modo, si vamos a utilizar el componente en dos versiones distintas de Delphi (en mi caso Delphi 7 y Delphi 2007) lo mejor es que cada una tenga su copia del componente para que se compilen por separado, es decir, instalamos por ejemplo el componente **mxProtector** dentro de la carpeta:

D:\RadStudio2007\Componentes\mxProtector

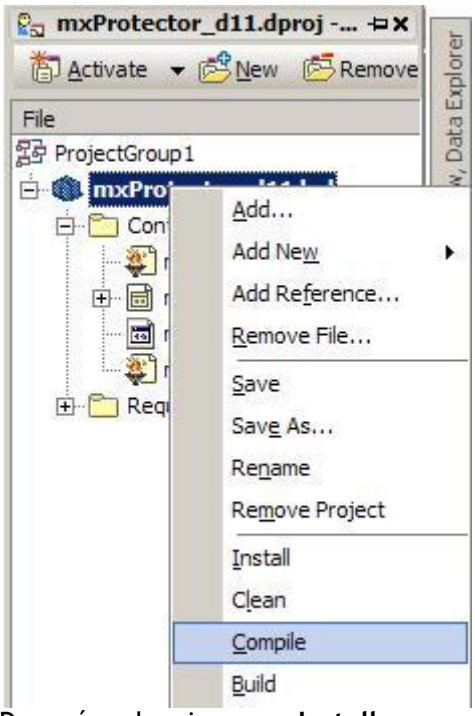
Y luego le sacamos una copia a mano a la carpeta:

D:\Delphi7\Componentes\mxProtector

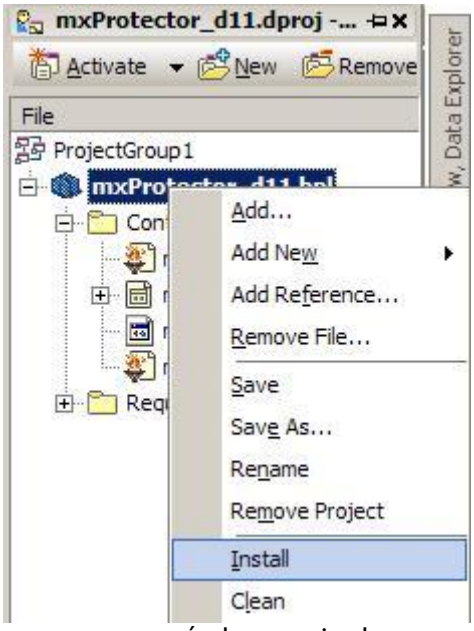
De este modo, cada versión del compilador no estropeará el componente de la otra.

INSTALANDO EL COMPONENTE

Una vez que lo tenemos instalado debemos añadirlo a Delphi 2007 abriendo el paquete **mxProtector_11.dpk**. En caso de que sea **Delphi 2009** seria **mxProtector_12.dpk**. En la ventana del **Project Manager** pinchamos el archivo**mxProtector_11.bpl** con el botón derecho del ratón y seleccionamos **Compile**:



Después seleccionamos **Install**:



y nos aparecerá el mensaje de que acaba de instalar el componente:

Al abrir o crear un nuevo proyecto veremos ese componente en la paleta:

Para Delphi 7 sería prácticamente lo mismo.

PROTEGER UN PROGRAMA POR EL NÚMERO DE EJECUCIONES

Cada vez que se ejecute el programa restará una vez al contador de número de ejecuciones en ese PC. Vamos a ver un ejemplo de cómo crear un programa que permita ejecutarse 3 veces.

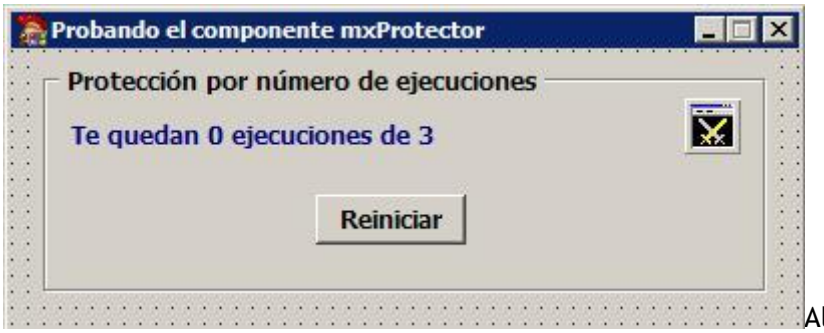
Antes tengo que aclarar una cosa. Este componente tiene dos modos de ejecución: o él se encarga de guardar (dios sabe donde) el número de ejecuciones o nosotros nos encargamos de decirle donde tiene que guardar estos datos (en un archivo INI, en un archivo binario, en el registro del sistema, etc.).

Yo prefiero esta segunda opción, ya que aunque es más pesada de implementar nos da más libertad a la hora de proteger nuestro software. Este será el método que yo voy a utilizar, concretamente lo voy a guardar en el registro del sistema dentro de la clave:

\HKEY_LOCAL_MACHINE\MiEmpresa\MiPrograma

Por lo demás le dejamos al componente **mxProtector** que guarde en esa clave lo que tenga que guardar.

Para probar este ejemplo voy a comenzar un nuevo proyecto que tenga este sencillo formulario:



Al componente de la clase **TMXProtector** lo he llamado **mxProtector** y a la etiqueta que va a mostrar el número de ejecuciones que nos quedan la he llamado **ENumEje**.

Para ello vamos a realizar estas acciones:

1º Ponemos la propiedad **MaxStartNumber** del componente **mxProtector** a **3**.

2º En el mismo componente activamos sólo el valor **stStartTrial** de su propiedad **ProtectionType**. Esto hará que se al arrancar nuestro programa se ejecute el

evento **OnStartTrial**.

2º En el evento **OnStartTrial** podemos el código que se va a ejecutar por primera vez:

```
procedure TFPrincipal.mxProtectorStartTrial(Sender:
TObject;
  StartsRemained: Integer);
begin
  ENumEje.Caption := Format('Te quedan %d ejecuciones de
%d',
    [StartsRemained, mxProtector.MaxStartNumber]);
end;
```

Este código se ejecutará automáticamente al inicio del programa, ya que está activada la opción **poAutoInit** dentro de la propiedad **Options**.

2º Al pulsar el botón **Reiniciar** ponemos a cero en número de veces que hemos ejecutado el programa:

```
mxProtector.Reset;
```

Podemos hacer que el componente se encargue de guardar ese número por si mismo o bien nos encargamos nosotros de todo el proceso (lo mejor). Como vamos a guardar en número de ejecuciones en el registro el sistema entonces haremos que en el evento **OnReset** elimine la clave del registro:

```
procedure TFPrincipal.mxProtectorReset(Sender: TObject; var
Handled: Boolean);
var
  Reg: TRegistry;
begin
  Handled := True;
  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_LOCAL_MACHINE;
  Reg.DeleteKey('\Software\MiEmpresa\MiPrograma');
  Reg.Free;
end;
```

Si ponemos a **True** la variable **Handled** le estamos diciendo al componente **mxProtector** que nosotros nos encargamos de guardar el contador de ejecuciones. Si la ponemos a false se encarga él. Lo que no sé es donde la guarda. Lo he estado monitorizando con el programa RegMon en el registro del sistema y no he encontrado donde lo mete.

3º En el evento **OnExpiration** debemos añadir el código de lo que queramos que haga cuando se finalicen el número de ejecuciones (se acabe la versión trial):

```
procedure TFPrincipal.mxProtectorExpiration(Sender:
TObject);
begin
  ENumEje.Caption := 'Ya no te quedan más ejecuciones';
end;
```

Si por ejemplo estamos haciendo un programa de facturación podíamos cambiar o

eliminar la contraseña de acceso al motor de bases de datos para que no pueda volver a funcionar el programa.

Este tipo de protección hace que este componente cargue y grabe una variable tipo booleana y otra de texto. Nosotros nos vamos a encargar de guardar y recoger estas variables. Así que vamos a reprogramar los eventos que necesitamos:

4º En el evento **OnGetBoolean** leemos la variable booleana que **mxProtector** nos pida:

```
procedure TFPrincipal.mxProtectorGetBoolean(Sender:
TObject; var APath,
  AKey: string; var AResult, Handled: Boolean);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_LOCAL_MACHINE;
  try
    if Reg.OpenKey('\Software\MiEmpresa\MiPrograma', True)
then
      if Reg.ValueExists(AKey) then
        AResult := Reg.ReadBool(AKey);

    Handled := True;
    Reg.CloseKey;
  finally
    Reg.Free;
  end;
end;
```

Al igual que antes, le ponemos la variable **Handled** a **True** para decirle al componente que nos encargamos del asunto.

5º Lo mismo hacemos para cargar una variable de tipo string en el evento **OnGetString**:

```
procedure TFPrincipal.mxProtectorGetString(Sender: TObject;
var APath, AKey,
  AResult: string; var Handled: Boolean);
var
  Reg: TRegistry;
begin
  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_LOCAL_MACHINE;
  try
    if Reg.OpenKey('\Software\MiEmpresa\MiPrograma', True)
then
      if Reg.ValueExists(AKey) then
        AResult := Reg.ReadString(AKey);

    Handled := True;
    Reg.CloseKey;
  finally
```

```
    Reg.Free;
  end;
end;
```

6º Ahora hacemos lo mismo para guardar una variable booleana en el evento **OnPutBoolean**:

```
procedure TFPrincipal.mxProtectorPutBoolean(Sender:
TObject; var APath,
  AKey: string; var ASavedData, Handled: Boolean);
var
  Reg: TRegistry;
begin
  Handled := True;
  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_LOCAL_MACHINE;
  if Reg.OpenKey('\Software\MiEmpresa\MiPrograma', True)
then
  begin
    Reg.WriteBool(AKey, ASavedData);
    Reg.CloseKey;
  end;
  Reg.Free;
end;
```

7º Y los mismo para una variable string en el evento **OnPutString**:

```
procedure TFPrincipal.mxProtectorPutString(Sender: TObject;
var APath, AKey,
  ASavedData: string; var Handled: Boolean);
var
  Reg: TRegistry;
begin
  Handled := True;
  Reg := TRegistry.Create;
  Reg.RootKey := HKEY_LOCAL_MACHINE;
  if Reg.OpenKey('\Software\MiEmpresa\MiPrograma', True)
then
  begin
    Reg.WriteString(AKey, ASavedData);
    Reg.CloseKey;
  end;
  Reg.Free;
end;
```

8º Por último sólo nos queda reprogramar los eventos **OnCodeData** y **OnDecodeData**:

```
procedure TFPrincipal.mxProtectorCodeData(Sender: TObject;
var ACode: string);
begin
  ACode := ACode;
end;
```

```
procedure TFPrincipal.mxProtectorDeCodeData(Sender:
TObject; var ACode: string);
begin
  ACode := ACode;
end;
```

Ahora mismo no tienen ningún tipo de codificación. Lo mismo que leen o cargan del registro del sistema es lo que va a parar al componente. Aquí podíamos ampliar la funcionalidad utilizando alguna función de encriptación y desencriptación como suele hacerse comúnmente con las funciones booleanas XOR, aunque esto se sale de los objetivos de este artículo.

Después de todo este rollo que os he metido, vamos a ejecutar el programa para ver que guarda en el registro:

Cuando se ejecuta el programa automáticamente nos resta el número de ejecuciones (eso lo hace sólo el componente) y nos guarda esto en el registro (hacer clic para ampliar):

Cerramos el programa y volvemos a abrirlo y nos queda una ejecución:

Y en el registro vemos que sólo cambia el valor **S2**:

Ejecutamos por última vez el programa y se acaban el nº de ejecuciones:

Y el estado del valor **S2** vuelve a cambiar:

El contenido del registro puede cambiar dependiendo del PC, de la fecha y hora del sistema o de algún patrón interno del componente mxProtector. Nosotros no tenemos que preocuparnos por eso. Lo más que podemos hacer es modificar los eventos **OnCodeDate** y **OnDecodeDate** para despistar aun más a los crackers.

También podíamos encriptar el ejecutable con algún compresor de archivos EXE tipo UPX para evitar la ingenieria inversa. Aún así, ningún sistema de protección es perfecto, pero por lo menos da algo más de seguridad frente a los crackeadores novatos.

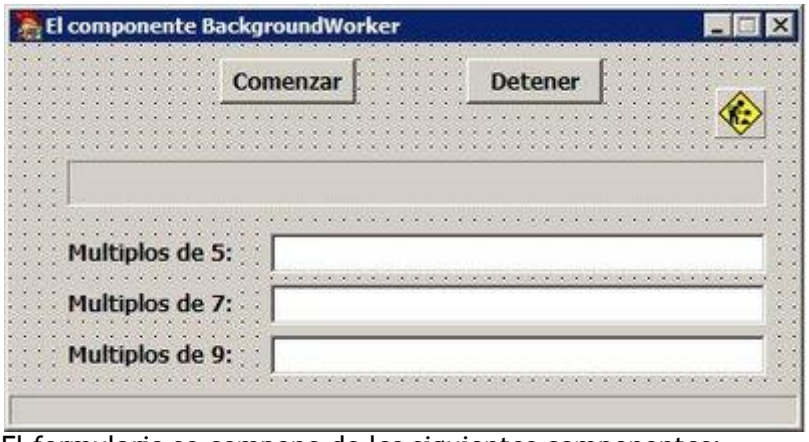
En el siguiente artículo seguiremos viendo los otros métodos de protección que incorpora este componente.

Pruebas realizadas en RAD Studio 2007.

17 julio 2009

El componente BackwroundWorker (y 2)

Vamos a ver un ejemplo de cómo utilizar este componente con un hilo de ejecución que recorra todos los números del 1 al 100 y nos calcule los múltiplos de 5, 7 y 9. Este sería el formulario principal de nuestro proyecto:



El formulario se compone de los siguientes componentes:

- 2 botones de la clase **TButton** llamados **BComenzar** y **BCancelar** para iniciar o detener el hilo de ejecución.

- 1 componente **TBackgroundWorker** llamado **BackgroundWorker**.

- 1 barra de progreso de la clase **TProgressBar** llamada **Progreso** y con su propiedad **Smooth** a **True**.

- 3 etiquetas (**TLabel**) y 3 casillas (**TEdit**)
llamadas **Multiplos5**, **Multiplos7** y **Multiplos9** donde se irán alojando los múltiplos encontrados de cada uno.

- 1 barra de estado de la clase **TStatusBar** llamada **Estado** con su propiedad **SimplePanel** a **True**.

Comencemos a meter código:

1º Al pulsar el botón Comenzar ponemos en marcha el hilo:

```
procedure TForm1.BComenzarClick(Sender: TObject);
begin
  BackgroundWorker.Execute;
end;
```

2º Al pulsar el botón Cancelar le pedimos al hilo que se detenga:

```
procedure TForm1.BDetenerClick(Sender: TObject);
begin
  BackgroundWorker.Cancel;
end;
```

3º En el evento **OnWork** del componente **BackgroundWorker** recorremos los 100 números:

```
procedure TForm1.BackgroundWorkerWork(Worker:
TBackgroundWorker);
var
  i: Integer;
begin
  for i := 1 to 100 do
  begin
    // ¿Hay que cancelar el proceso?
    if Worker.CancellationPending then
    begin
      // Le indicamos al hilo que lo cancelamos
      Worker.AcceptCancellation;
      Exit;
    end;

    // ¿Es múltiplo de 5?
    if i mod 5 = 0 then
      Worker.ReportFeedback(5, i);

    // ¿Es múltiplo de 7?
    if i mod 7 = 0 then
      Worker.ReportFeedback(7, i);

    // ¿Es múltiplo de 9?
```

```
    if i mod 9 = 0 then
      Worker.ReportFeedback(9, i);

    // Esperamos 50 milisegundos
    Sleep(50);

    // Incrementamos la barra de progreso
    Worker.ReportProgress(i);
  end;
end;
```

Al principio del bucle controlamos si nos piden abandonar el hilo de ejecución:

```
// ¿Hay que cancelar el proceso?
if Worker.CancellationPending then
begin
  // Le indicamos al hilo que lo cancelamos
  Worker.AcceptCancellation;
  Exit;
end;
```

Después comprobamos los múltiplos de cada número y en el caso de que así sea envío provoco un evento**OnWorkFeedBack** para notificar el número que he encontrado:

```
// ¿Es múltiplo de 5?
if i mod 5 = 0 then
  Worker.ReportFeedback(5, i);

// ¿Es múltiplo de 7?
if i mod 7 = 0 then
  Worker.ReportFeedback(7, i);

// ¿Es múltiplo de 9?
if i mod 9 = 0 then
  Worker.ReportFeedback(9, i);
```

Y por último provoco un pequeño retardo de 50 milisegundos (para ir viendo el progreso) y provoco un evento**OnWorkProgress** para notificar el incremento en la barra de progreso:

```
// Esperamos 50 milisegundos
Sleep(50);

// Incrementamos la barra de progreso
Worker.ReportProgress(i);
```

Sigamos...

4º En el evento **OnWorkFeedBack** recogemos el mensaje que nos manda el hilo para ir guardando en cada casilla los múltiplos de cada número:

```
procedure TForm1.BackgroundWorkerWorkFeedback(Worker:
```

```
TBackgroundWorker;
FeedbackID, FeedbackValue: Integer);
begin
  case FeedbackID of
    5: Multiplos5.Text := Multiplos5.Text +
IntToStr(FeedbackValue) + ',';
    7: Multiplos7.Text := Multiplos7.Text +
IntToStr(FeedbackValue) + ',';
    9: Multiplos9.Text := Multiplos9.Text +
IntToStr(FeedbackValue) + ',';
  end;
end;
```

5º En el evento **OnWorkProgress** actualizamos la barra de progreso y la barra de estado en pantalla:

```
procedure TForm1.BackgroundWorkerWorkProgress(Worker:
TBackgroundWorker;
PercentDone: Integer);
begin
  Progreso.Position := PercentDone;
  Estado.SimpleText := 'Procesando... ' +
IntToStr(PercentDone) + '%';
end;
```

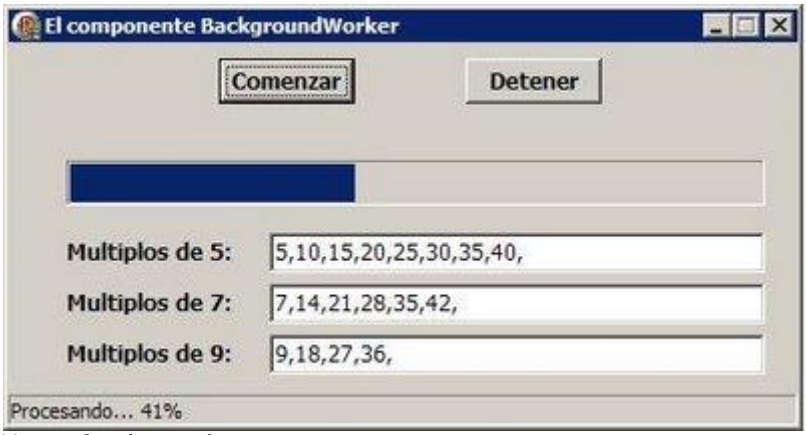
6º En el evento **OnWorkComplete** le decimos al usuario en la barra de progreso que hemos terminado:

```
procedure TForm1.BackgroundWorkerWorkComplete(Worker:
TBackgroundWorker;
Cancelled: Boolean);
begin
  Estado.SimpleText := 'Proceso finalizado';
end;
```

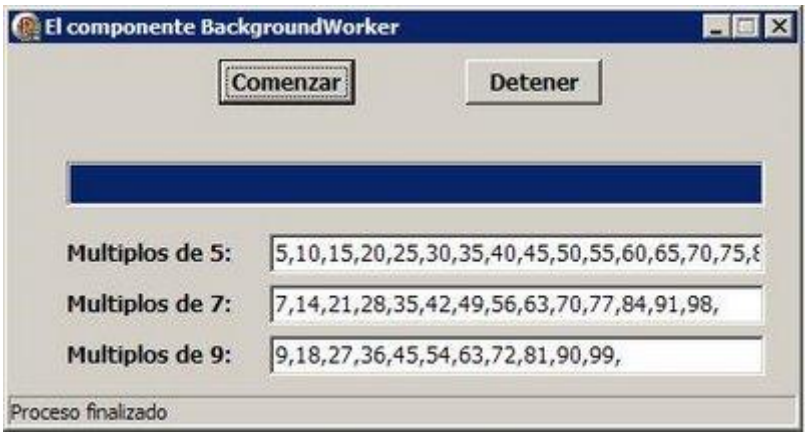
7º Por si acaso al usuario le da por cerrar el formulario mientras está el hilo en marcha tenemos que cancelarlo y esperar a que termine (al igual que hacen los programas como Emule o BitTorrent cuando lo cerramos y espera a liberar memoria y cerrar las conexiones):

```
procedure TForm1.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
  // ¿Esta el hilo en marcha?
  if BackgroundWorker.IsWorking then
  begin
    // Le indicamos que se detenga
    BackgroundWorker.Cancel;
    // Esperamos a que se detenga
    BackgroundWorker.WaitFor;
  end;
end;
```

Al ejecutar el programa irá rellenando las casillas correspondientes:



Hasta finalizar el proceso:



Como podemos apreciar, este componente nos facilita mucho la labor con los hilos de ejecución respecto a la clase**TThread**. Por las pruebas que he realizado en otros programas más complejos (facturación) es bastante sencillo de manejar y muy estable.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 09:25 2 comentarios
Etiquetas: componentes, sistema

03 julio 2009

El componente BackgroundWorker (1)

Aunque haya finalizado el tema referente a los hilos de ejecución con la clase **TThread** no está demás hablar de otros componentes no oficiales de Delphi que también están relacionados con los hijos de ejecución.

En este caso el componente me lo recomendó un lector de este blog (gracias a Jorge Abel) y me puse a echarle un vistazo a ver que tal. Este componente puede descargarse gratuitamente de esta página web (licencia Freeware):

http://www.delphiarea.com/products/

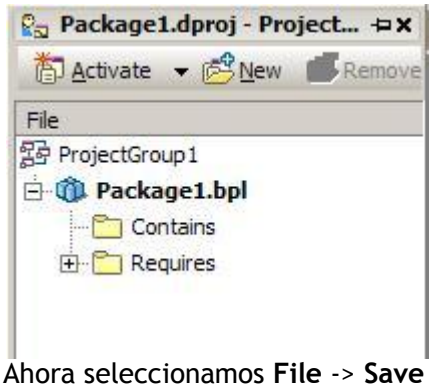
Concretamente en la sección de componentes que viene más abajo:

Al igual que la clase **TThread**, este componente permite crear un hilo de ejecución paralelo al hilo primario de nuestra aplicación pero de una manera sencilla y sin tener que heredad de clases.

INSTALAR EL COMPONENTE

Como es un componente que va suelto (sin paquete) vamos a proceder a crear un paquete para el sólo. Para ello creamos una carpeta llamada **DephiArea** donde vamos a descomprimir los archivos del componente:

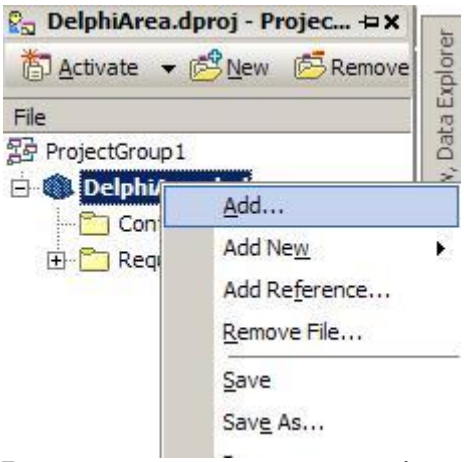
Ahora nos vamos a Delphi y seleccionamos **File** -> **New** -> **Package – Delphi for Win32**. Aparecerá esto a la derecha en la ventana del proyecto:



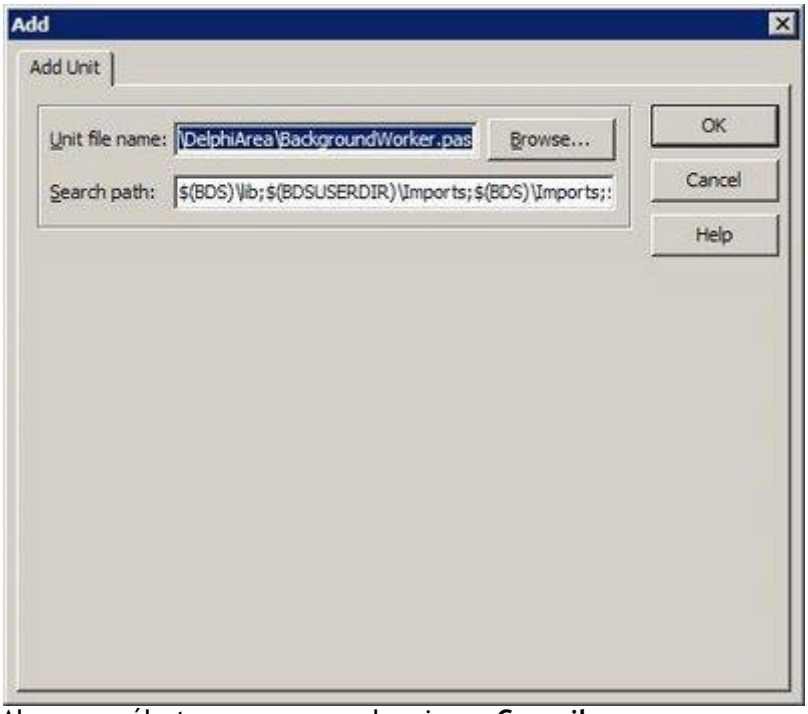
Ahora seleccionamos **File** -> **Save all** y nos vamos a la carpeta donde hemos descomprimido el componente, por ejemplo:

D:\CodeGear\RAD Studio\5.0\Componentes\DelphiArea

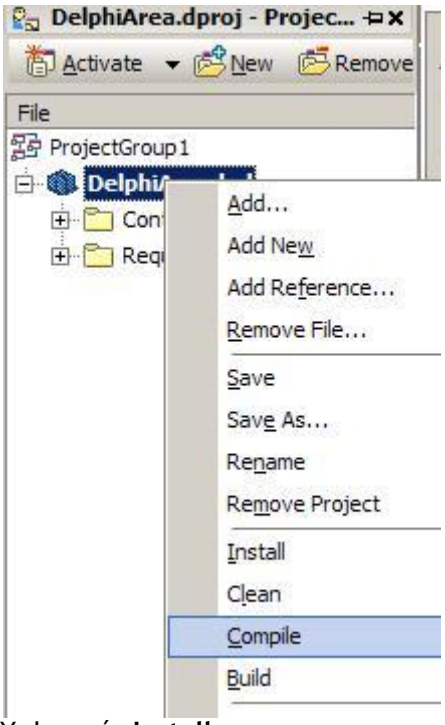
Y guardamos el proyecto con el nombre **DelphiArea.dproj**. Volvemos a la ventana del proyecto y pinchamos el nombre del mismo con el botón derecho del ratón y seleccionamos **Add**:



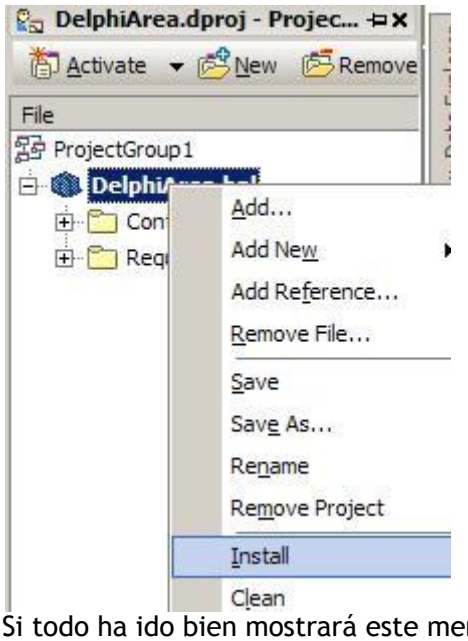
En esta ventana que aparece pulsamos el botón **Browse** y elegimos el archivo **BackgroundWorker.pas**:



Ahora ya sólo tenemos que seleccionar **Compile**:



Y después **Install**:



Si todo ha ido bien mostrará este mensaje:

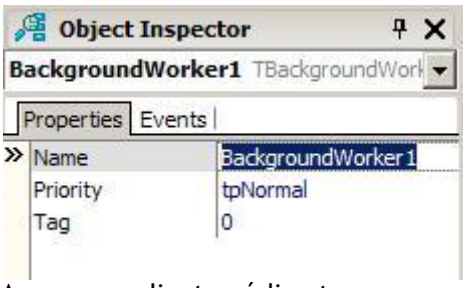
Si creamos un nuevo proyecto y nos fijamos en la paleta de componentes veremos el nuevo componente que hemos instalado:

Ya estamos listos para comenzar a trabajar con el mismo.

LAS CARACTERÍSTICAS DEL COMPONENTE BACKGROUNDWORKER

Insertamos el componente en el formulario donde queremos lanzar el hilo de ejecución:

Si nos fijamos en el inspector de objetos veremos que no se han estresado añadiendo propiedades:



Aunque mediante código tenemos acceso a estas variables:

CancellationPending: esta bandera booleana es utilizada para indicarnos que debemos abandonar el bucle principal del hilo (el evento **OnWork** que veremos más adelante). Por defecto está a **False**. Se pondrá a **True** cuando llamemos al método **Cancel**.

IsCancelled: variable booleana que nos dice si el hilo ha sido cancelado.

IsWorking: indica si el hilo se sigue ejecutando con normalidad.

ThreadID: es un número entero con el identificador que Windows ha asignado a nuestro hilo. Podemos utilizar esta propiedad para llamar a otras funciones de la API de Windows relacionadas con los hilos y procesos.

Y para modificar su comportamiento tenemos estos métodos:

procedure Execute;

Comienza la ejecución del hilo.

procedure Cancel;

Solicita al hilo que detenga la ejecución, es decir, pone la variable **CancellationPending** a **True** para que seamos nosotros los que abandonemos el bucle cerrado del evento **OnWork**.

procedure WaitFor;

Espera a que termine la ejecución del hilo.

procedure ReportProgress(PercentDone: Integer);

A este método sólo podemos llamarlo dentro del evento **OnWork**. Lo que hace es realizar una llamada al evento**OnWorkProgress** para que situemos en el mismo lo que vamos a

hacer para informar al hilo primario del porcentaje de trabajo que hemos realizado.

procedure ReportProgressWait(PercentDone: Integer);

Igual que el procedimiento anterior pero espera a que termine el código que hemos colocado dentro de**OnWorkProgress**.

procedure ReportFeedback(FeedbackID, FeedbackValue: Integer);

Al igual que los dos procedimientos anteriores, sólo podemos llamar al mismo dentro del evento **OnWork**. Realiza una llamada al evento **OnWorkFeedBack** para que podamos enviar mensajes al hilo primario.

procedure ReportFeedbackWait(FeedbackID, FeedbackValue: Integer);

Lo mismo que el procedimiento anterior pero espera a que termine la ejecución del código que hemos puesto en el evento **OnWorkFeedBack**.

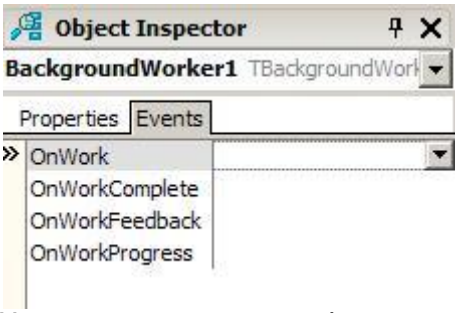
procedure Synchronice(Method: ThreadMethod);

Sólo podemos llamar a este procedimiento dentro del evento **OnWork**. Le pasamos como parámetro el método del hilo principal (VCL) para que lo ejecute por nosotros, por ejemplo, para mostrar información en pantalla o incrementar una barra de progreso.

procedure AcceptCancellation;

Sólo podemos llamar a este procedimiento dentro del evento **OnWork**. Debemos llamar a este procedimiento cuando dentro del evento **OnWork** hemos visto que la variable **CancellationPending** es **True** y entonces abandonamos el bucle principal y llamamos a este procedimiento para informar a hilo que hemos aceptado salir del mismo.

Estos son los eventos que incorpora:



Veamos para que sirve cada evento:

OnWork: Este método equivale al procedimiento **Execute** de la clase **TThread**, de hecho, se ejecuta cuando llamamos al procedimiento **Execute**. Es aquí donde debemos introducir el bucle continuo que va a hacer el trabajo pesado. A cada ciclo del bucle debemos comprobar el valor de la variable **CancellationPending** para saber si debemos terminarlo. Una vez abandonemos el bucle debemos llamar al método **AcceptCancellation** que pondrá la variable **IsCancelled** a **True**.

OnWorkComplete: este evento se ejecutará cuando termine la ejecución del código que hemos puesto en el evento **OnWork** ya sea porque ha terminado su ejecución o porque lo hemos cancelado. Aquí podemos poner el código encargado de liberar los objetos creados o cerrar archivos abiertos.

OnWorkProgress: se ejecutará el código que introducimos a este evento cuando desde el

evento **OnWork** hemos hecho una llamada a los procedimientos **ReportProgress** o **ReportProgressWait**. Desde aquí podemos informar al hilo primario de sobre cómo vamos.

OnWorkFeedBack: se ejecuta cuando nuestro hilo envía mensajes al hilo primario mediante los procedimientos**ReportFeedBack** o **ReportFeedBackWait**.

COMO INSTALARLO EN DELPHI 7

Para instalar este componente en Delphi 7 hay que seguir estos pasos:

1º Seleccionamos en el menú superior **Install** -> **Component**.

2º En la ventana que aparece seleccionamos la pestaña **Into new Package**.

3º Pulsamos el botón **Browse** del campo **Unit file name** y seleccionamos el archivo **BackgroundWorker.pas**.

4º Pulsamos el botón **Browse** del campo **Package file name** y escribimos **DelphiArea.dpk**.

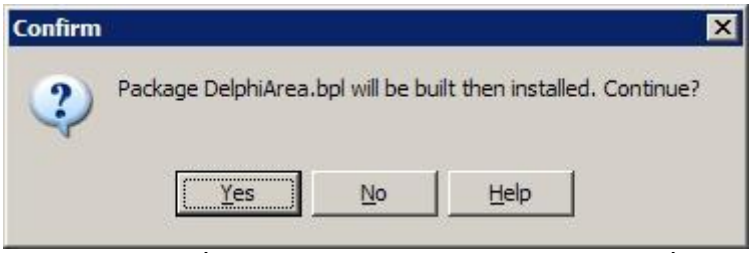
5º En el campo **Package Description** escribimos **Delphi Area**.

Suponiendo que el componente lo tengo descomprimido en esta carpeta:

D:\Borland\Delphi7\Componentes\BackgroundWorker

Quedaría la ventana de este modo:

Pulsamos el botón **Ok** y aparecerá esta mensaje:



Pulsamos el botón **Yes** y si todo ha ido bien aparecerá este mensaje:

Con esto ya tiene que aparecer arriba en la paleta de componentes:

En el próximo artículo veremos un ejemplo de cómo crear un hilo de ejecución utilizando este componente.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 09:52 3 comentarios
Etiquetas: componentes, sistema

19 junio 2009

Los Hilos de Ejecución (y 4)

Hoy voy a terminar de hablar de otras cuestiones relacionadas con los hilos de ejecución como pueden ser las variables de tipo **threadvar** y las secciones críticas.

ACCESO A LA MISMA VARIABLE POR MULTIPLES HILOS

Antes de ver como utilizar una variable **threadvar** veamos un problema que se puede plantear cuando varios hilos intentan acceder a una variable global sin utilizar el método **synchronize**.

Siguiendo con nuestro ejemplo de los tres hilos que incrementan una barra de progreso cada uno, supongamos que quiero que cada barra llegue de 0 a 100 y que cuando termine voy a hacer que termine el hilo.

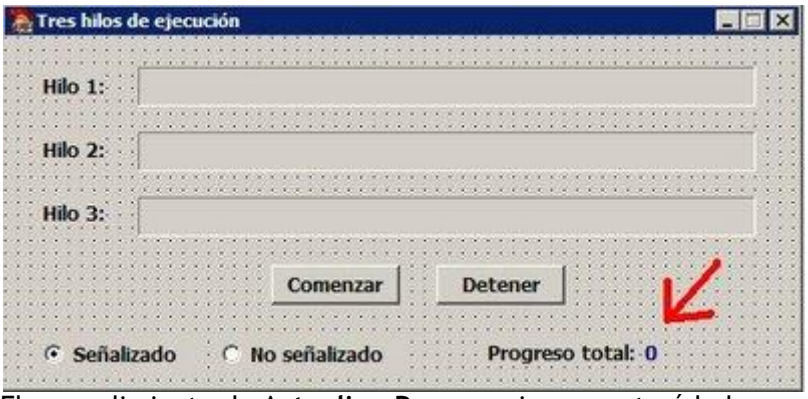
Después voy a crear una variable global llamada **iContador**:

```
var
  iContador: Integer;
```

Cuando un hilo incremente la barra de progreso entonces incrementará también esta variable global **iContador**:

```
procedure THilo.Execute;
begin
  inherited;
  FreeOnTerminate := True;
  while not Terminated do
  begin
    Inc(iContador);
    Synchronize(ActualizarProgreso);
    Sleep(100);
  end;
end;
```

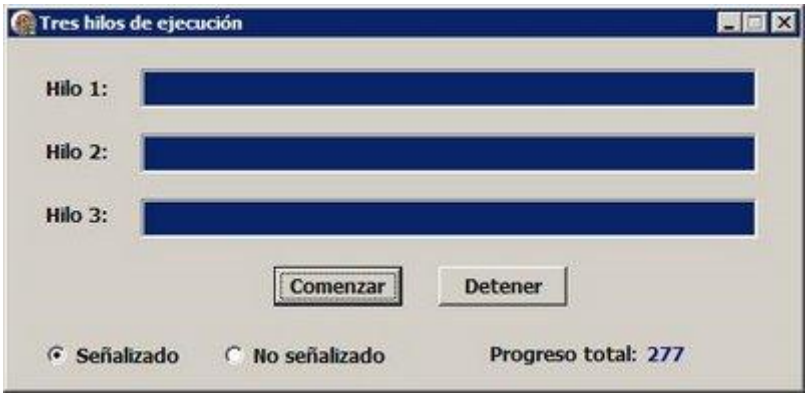
También he añadido al formulario una etiqueta llamada **ETotal** que mostrará por pantalla el contenido de la variable**iContador**:



El procedimiento de **ActualizarProgreso** incrementará la barra de progreso y mostrará el contador del formulario:

```
procedure THilo.ActualizarProgreso;
begin
  Progreso.StepIt;
  FTresHilos.ETotal.Caption := IntToStr(iContador);

  if Progreso.Position = 100 then
  Terminate;
end;
```

Supuestamente, si tres hilos de ejecución incrementan cada barra de 0 a 100 entonces cuando terminen de ejecutarse el contador tendrá el valor 300. Pero no es así:



Me ha salido 277 pero lo mismo puede dar 289 que 291. Como cada hilo accede a la variable global **iContador**simultáneamente lo mismo la incrementa después de otro hilo que machaca el incremento del hilo anterior.

Ya vimos que esto puede solucionarse incluyendo la sentencia **Inc(iContador)** dentro del procedimiento**ActualizarProgreso**, de modo que mediante la sentencia **Synchronize** sólo el hilo primario podrá incrementar esta variable. Esto tiene un inconveniente y es que se forma un cuello de botella en los hilos de ejecución porque cada hilo tiene que esperar a que el hilo primario incremente la variable.

Veamos si se puede solucionar mediante variables **threadvar**.

LAS VARIABLES THREADVAR

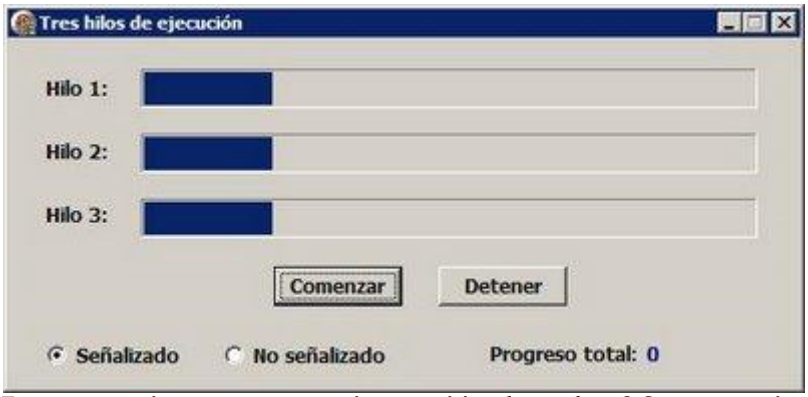
Las variables de tipo **threadvar** son declaradas globalmente en nuestro programa para que puedan ser leídas por uno o más hilos simultáneamente pero no pueden ser modificadas por los mismos. Me explico.

Cuando un hilo lee de una variable global **threadvar**, si intenta modificarla sólo modificará una copia de la misma, no la original, ya que solo puede ser modificada por el hilo primario. Delphi creará automáticamente una copia de la variable **threadvar** para cada hilo (como si fuera una variable privada dentro del objeto que hereda de **Thread**).

Una variable **threadvar** se declara igual que una variable global:

```
implementation

threadvar
  iContador: Integer;
```

Si volvemos a ejecutar el programa veremos que **iContador** nunca de mueve:



Entonces, ¿de que nos sirve la variable **threadvar**? Su cometido es crear una variable donde sólo el hilo primario la pueda incrementar pero que a la hora de ser leía por un hilo secundario siempre tenga el mismo valor.

Para solucionar este problema lo mejor es que cada clase tenga su propio contador y que luego en el formulario principal creemos un temporizador que muestre la suma de los contadores de cada hilo y de este modo no se crean cuellos de botella ni es necesario utilizar **synchronize**.

Una utilidad que se le puede dar a este tipo de variables es cuando el hilo primario debe suministrar información crítica en tiempo real a los hijos secundarios y sobre todo cuando queremos que ningún hilo lea un valor distinto de otro por el simple hecho que lo ha ejecutado después.

SECCIONES CRÍTICAS

Anteriormente vimos como los objetos **mutex** podían controlar que cuando se ejecute cierto código dentro de un hilo de ejecución, los demás hilos tienen que esperarse a que termine.

Esto también puede crearse mediante secciones críticas (**CrititalSection**). Una sección crítica puede ser útil para evitar que varios hilos intenten enviar o recibir simultáneamente información de un dispositivo (monotarea). Naturalmente esto solo tiene sentido si hay dos o más instancias del mismo hilo, sino es absurdo.

Aprovechando el caso anterior de los tres hilos incrementando la barra de progreso, imaginemos que cada vez que un hilo intenta incrementa su barra de progreso, tiene que enviar su progreso por un puerto serie a un dispositivo. Aquí vamos a suponer que ese puerto serie a un objeto **TStringList** (que bien podía ser por ejemplo un**TFileStream**).

Primero creamos una variable global llamada **Lista**:

```
var
  Lista: TStringList;
```

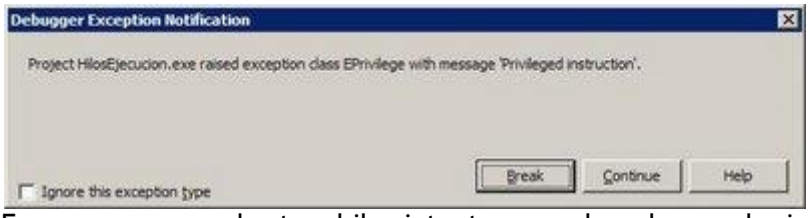
Después la creo cuando pulso el botón **Comenzar**:

```
procedure TFTresHilos.BComenzarClick(Sender: TObject);
begin
  Lista := TStringList.Create;
  Progreso1.Position := 0;
  Progreso2.Position := 0;
  Progreso3.Position := 0;
  ...
```

Y en el procedimiento **Execute** envío la posición de su barra de progreso al **StringList**:

```
procedure THilo.Execute;
begin
  inherited;
  FreeOnTerminate := True;
  while not Terminated do
  begin
    Synchronize(ActualizarProgreso);
    Lista.Add(IntToStr(Progreso.Position));
    Sleep(100);
  end;
end;
```

Pero al ejecutar el programa puede ocurrir esto:



Eso ocurre porque los tres hilos intentan acceder a la vez al mismo objeto **StringList**. Si bien podíamos solucionar esto utilizando **Synchronize** volvemos al problema de que cada hilo pierde su independencia respecto al hilo principal.

Lo que vamos a hacer es que si un hilo está enviando algo al objeto **StringList** los otros hilos no pueden enviarlo, pero si seguir su normal ejecución. Esto se soluciona creando lo que se llama una sección crítica que aísla el momento en que un hilo hace esto:

```
Lista.Add(IntToStr(Progreso.Position));
```

Para crear una sección crítica primero tenemos que declarar esta variable global:

```
var
  Lista: TStringList;
  SeccionCritica: TRTLCriticalSection;
```

Al pulsar el botón **Comenzar** inicializamos la sección crítica:

```
procedure TFTresHilos.BComenzarClick(Sender: TObject);
begin
  Lista := TStringList.Create;
  InitializeCriticalSection(SeccionCritica);
  Progreso1.Position := 0;
  Progreso2.Position := 0;
  Progreso3.Position := 0;
  ...
```

Acordándonos que hay que liberarla al pulsar el botón **Detener**:

```
procedure TFTresHilos.BDetenerClick(Sender: TObject);
begin
  Hilo1.Terminate;
  Hilo2.Terminate;
  Hilo3.Terminate;
  DeleteCriticalSection(SeccionCritica);
end;
```

Después hay que modificar el procedimiento **Execute** para introducir nuestra instrucción peligrosa dentro de la sección crítica:

```
procedure THilo.Execute;
begin
  inherited;
  FreeOnTerminate := True;
  while not Terminated do
  begin
    Synchronize(ActualizarProgreso);
    EnterCriticalSection(SeccionCritica);
    Lista.Add(IntToStr(Progreso.Position));
    LeaveCriticalSection(SeccionCritica);
    Sleep(100);
  end;
end;
```

Todo lo que se ejecute dentro de **EnterCritialSection** y **LeaveCriticalSection** solo será ejecutado a la vez por hilo, evitando así la concurrencia. Con esto solucionamos el problema sin tener que recurrir al hilo primario.

Aunque he abarcando bastantes temas respecto a los hilos de ejecución todavía quedan muchas cosas que entran en la zona de la API de Windows y que se salen de los objetivos de este artículo. Si encuentro algo más interesante lo publicaré en artículos independientes.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 12:18 2 comentarios
Etiquetas: sistema

05 junio 2009

Los Hilos de Ejecución (3)

En la tercera parte de este artículo vamos a ver como establecer la prioridad en los hilos de ejecución así como controlar su comportamiento mediante objetos **Event** y **Mutex**.

LA PRIORIDAD EN LA EJECUCIÓN DE LOS HILOS

Aparte de poder controlar nosotros mismos la velocidad de hilo dentro del procedimiento **Execute** utilizando la función **Sleep**, **GetTickCount** o **TimeGetTime** también podemos establecer la prioridad que Windows le va a dar a nuestro hilo respecto al resto de aplicaciones.

Esto se hace utilizando la propiedad **Priority** que puede contener estos valores:

```
type
  TThreadPriority = (
    tpIdle,          // El hilo sólo se ejecuta cuando el
procesador está desocupado
    tpLowest,        // Prioridad más baja
    tpLower,         // Prioridad baja.
    tpNormal,        // Prioridad normal
    tpHigher,        // Prioridad alta
    tpHighest,       // Prioridad muy alta
    tpTimeCritical   // Obliga a ejecutarlo en tiempo real
    );
```

No deberíamos abusar de las prioridades **tpHigher**, **tpHighest** y **tpTimeCritical** a menos que sea estrictamente necesario, ya que podían restar velocidad a otras tareas críticas de Windows (como nuestro querido Emule).

Vamos a ver un ejemplo en el que voy a crear tres hilos de ejecución que actualizarán cada uno una barra de progreso cada 100 milisegundos:

Esta sería la definición del hilo:

```
THilo = class(TThread)
  Progreso: TProgressBar;
  procedure Execute; override;
  procedure ActualizarProgreso;
end;
```

Y su implementación:

```
{ THilo }
```

```
procedure THilo.ActualizarProgreso;
begin
  Progreso.StepIt;
end;

procedure THilo.Execute;
begin
  inherited;
  FreeOnTerminate := True;
  while not Terminated do
  begin
    Synchronize(ActualizarProgreso);
    Sleep(100);
  end;
end;
```

Su única misión es incrementar la barra de progreso que le toca y esperar 100 milisegundos. Al pulsar el botón**Comenzar** hacemos esto:

```
procedure TFTresHilos.BComenzarClick(Sender: TObject);
begin
  Progreso1.Position := 0;
  Progreso2.Position := 0;
  Progreso3.Position := 0;
  Hilo1 := THilo.Create(True);
  Hilo2 := THilo.Create(True);
  Hilo3 := THilo.Create(True);
  Hilo1.Progreso := Progreso1;
  Hilo2.Progreso := Progreso2;
  Hilo3.Progreso := Progreso3;
  Hilo1.Resume;
  Hilo2.Resume;
  Hilo3.Resume;
end;
```

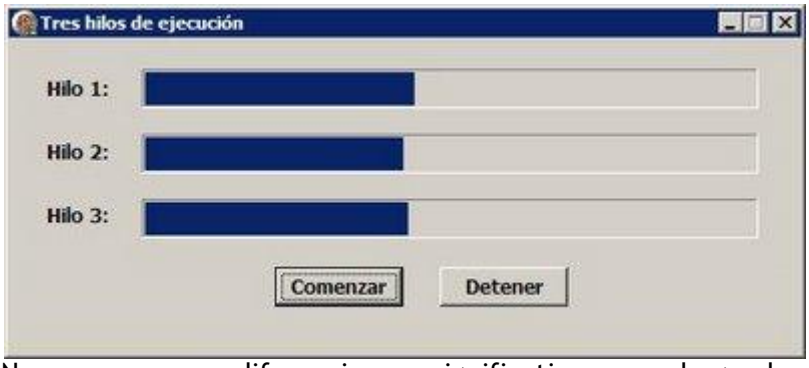
Al ejecutarlo los tres hilos van exactamente iguales:

Pero ahora supongamos que hago esto antes de ejecutarlos:

```
Hilo1.Priority := tpTimeCritical;
```

```
Hilo2.Priority := tpLowest;
Hilo3.Priority := tpHighest;
```

Al ejecutarlo nos ponemos a navegar con Firefox o nuestro navegador preferido en páginas que den caña al procesador y al observar nuestros hilos veremos que se han desincronizado según su prioridad:



No es que sea una diferencia muy significativa pero a largo plazo se nota como Windows para repartiendo el trabajo. Esto puede venir muy bien para llamar a rutinas que procesen datos en segundo plano (**tpIdle**) o bien enviar comandos a máquinas conectadas al PC por puerto serie, paralelo o USB que necesiten ir como reloj suizo (**tpTimeCritical**).

CONTROLANDO LOS HILOS CON EVENTOS DE WINDOWS

Los hilos de ejecución también permiten comenzar su ejecución o detenerse dependiendo de un evento externo que o bien puede ser controlado por el hilo primario de nuestra aplicación o bien por otro hilo.

Los eventos son objetos definidos en la API de Windows que permiten tener dos estados: señalizados y no señalizados.

Para crear un evento se utiliza esta función de la API de Windows:

```
CreateEvent(
  lpEventAttributes, // atributos de seguridad
  bManualReset,      // si está a True significa que
nosotros nos encargamos de señalizarlo
  bInitialState,     // Señalizado o no señalizado
  lpName             // Nombre el evento
  ): THandle;
```

Los atributos de seguridad vienen definidos en la API de Windows (en C) del siguiente modo:

```
typedef struct _SECURITY_ATTRIBUTES {
  DWORD  nLength;
  LPVOID lpSecurityDescriptor;
  BOOL   bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Si le ponemos **nil** le asignará los atributos por defecto que tenemos como usuario en el sistema operativo Windows. Este parámetro sólo sirve para los sistemas operativos Windows NT, XP y Vista. En los Windows 9X no hará nada.

¿Para que podemos utilizar los eventos? Pues podemos crear uno que le diga a los hilos cuando deben comenzar, detenerse o esperar cierto tiempo. Hay que procurar que el nombre del evento (**lpName**) sea original para que no colisione con otro nombre de otra aplicación. Para averiguar si un evento ha producido un error podemos llamar a la función:

```
function GetLastError: DWord;
```

Si devuelve un cero es que todo ha ido bien. Los objetos **Event** también pueden ser creados sin nombre, enviando como último parámetro el valor **nil**.

Para crear un evento voy a crear una variable global en la implementación con el **Handle** del evento que vamos a crear:

```
implementation

var
  Evento: THandle;
```

Antes de crear los hilos y ejecutarlos creo el evento señalizado:

```
procedure TFTresHilos.BComenzarClick(Sender: TObject);
begin
  Evento := CreateEvent(nil, True, True, 'MiEvento');
  ...
```

Entonces modifico el procedimiento **Execute** para que haga cada ciclo si el evento está señalizado y si no es así, que espere indefinidamente:

```
procedure THilo.Execute;
begin
  inherited;
  FreeOnTerminate := True;
  while not Terminated do
  begin
    Synchronize(ActualizarProgreso);
    Sleep(100);
    WaitForSingleObject(Evento, INFINITE);
  end;
end;
```

La función **WaitForSingleObject** espera a que el evento que le pasamos como primer parámetro esté señalizado para seguir, en caso contrario seguirá esperando indefinidadamente (**INFINITE**). También podíamos haber hecho que esperara un segundo:

```
WaitForSingleObject(Evento, 1000);
```

Para señalizar o no los eventos he añadido al formulario dos componentes **RadioButton**:

Para señalizarlo llamo a la función **SetEvent**:

```
procedure TFTresHilos.SenalizadoClick(Sender: TObject);
begin
  SetEvent(Evento);
end;
```

Y para no señalizarlo utilizo **PulseEvent**:

```
procedure TFTresHilos.NoSenalizadoClick(Sender: TObject);
begin
  PulseEvent(Evento);
end;
```

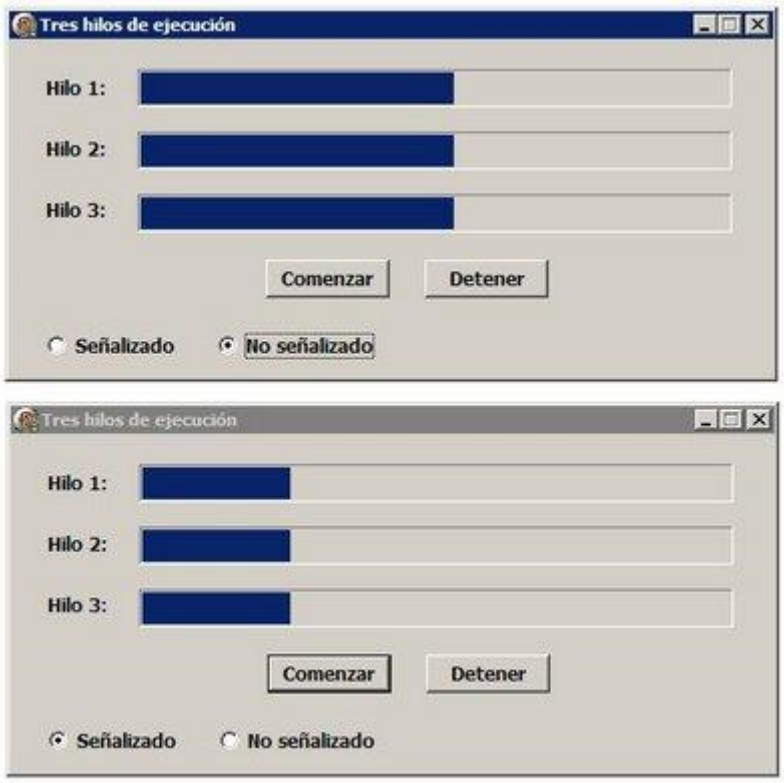
Tenemos dos funciones para no señalizar un evento:

PulseEvent: no señaliza el evento inmediatamente.

ResetEvent: no señaliza el evento la próxima vez que pase por **WaitForSingleObject**.

Y lo mejor de todo esto que no sólo podemos activar y desactivar hilos de ejecución dentro de la misma aplicación sino que además podemos hacerlo entre distintas aplicaciones que se están ejecutando a la vez.

En mi ejemplo he abierto dos instancias de la misma aplicación y no he señalizado en la primera un evento, con lo que se han detenido las dos:



Esto nos da mucha potencia para sincronizar distintas aplicaciones simultáneamente.

LOS OBJETOS MUTEX

La API de Windows también nos trae otro tipo de objetos llamados **Mutex**, también conocidos como semáforos binarios que funcionan de manera similar a los eventos pero que además permite asignarles un dueño. La misión de los objetos **Mutex** es evitar que varios hilos accedan a la vez a los mismos recursos, al estilo del procedimiento**Synchronize**.

Al contrario de los objetos **Event** donde todos los hilos podían esperar o no al evento en cuestión, con los objetos**Mutex** podemos hacer que solo uno de los hilos pueda trabajar a la vez y que los otros se esperen hasta nuevo aviso. Lo veremos más claro con un ejemplo.

Para crear un objeto Mutex utilizamos esta función:

```
function CreateMutex(lpMutexAttributes, bInitialOwer,
lpName);
```

Al igual que con los eventos, el parámetro **lpMutexAttributes** sirve para establecer los atributos de seguridad. El parámetro **bInitialOwer** que se encarga de decir si el hilo que llama a **CreateMutex** es el propietario de este **Mutex**o por el contrario si queremos que el primer hilo que espere al **Mutex** es el dueño del mismo.

En el ejemplo que vimos anteriormente con las tres barras de progreso, supongamos que cada vez que se incrementa la barra no queremos que las otras barran lo hagan también. Esto puede ser muy útil cuando hay varios hilos que intentan acceder al mismo recurso (grabar en CD-ROM, enviar señales por un puerto, etc.).

Al contrario del ejemplo de los eventos que me interesaba que se activaran o desactivaran todos a la vez, aquí con los **Mutex** me interesa que mientras a un hilo le

toca trabajar, los otros hilos tienen que esperarse a que termine (100 ms).

Al igual que hice con el objeto **Event** voy a crear una variable global con el handle del **Mutex**:

```
implementation

var
  Mutex: THandle;
```

Ahora creo el **Mutex** al pulsar el botón **Comenzar**:

```
procedure TFTresHilos.BComenzarClick(Sender: TObject);
begin
  Mutex := CreateMutex(nil, True, 'Mutex1');
  Progreso1.Position := 0;
  Progreso2.Position := 0;
  Progreso3.Position := 0;
  Hilo1 := THilo.Create(True);
  Hilo2 := THilo.Create(True);
  Hilo3 := THilo.Create(True);
  Hilo1.Progreso := Progreso1;
  Hilo2.Progreso := Progreso2;
  Hilo3.Progreso := Progreso3;
  Hilo1.Resume;
  Hilo2.Resume;
  Hilo3.Resume;
  ReleaseMutex(Mutex);
end;
```

Para que un hilo no se lance antes que otro cuando ejecuto **Resume**, lo que hago es que no suelto el **Mutex** hasta que los tres hilos están en ejecución (como en una carrera).

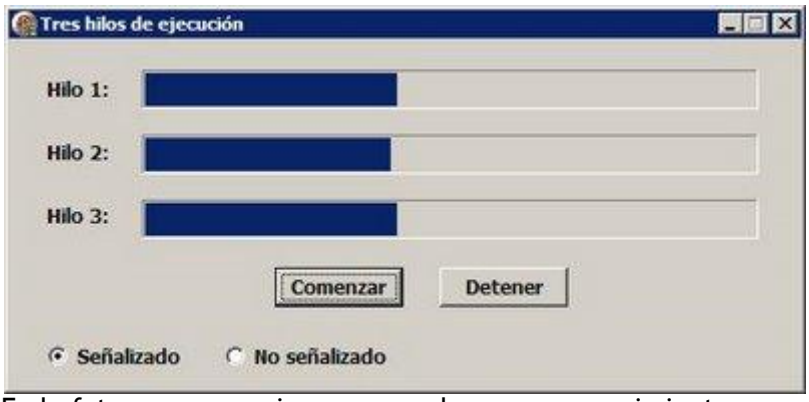
Después solo tengo que ir ejecutando cada uno haciendo esperar a los demás hasta que termine:

```
procedure THilo.Execute;
begin
  inherited;
  FreeOnTerminate := True;
  while not Terminated do
  begin
    WaitForSingleObject(Mutex, INFINITE); // captura el
mutex (los demás a esperar)
    Synchronize(ActualizarProgreso);
    Sleep(100);
    ReleaseMutex(Mutex); // mutex liberado hasta que lo
capture otro hilo
  end;
end;
```

Entre las líneas **WaitForSingleObject** y **ReleaseMutex**, lo demás hilos quedan parados hasta que termine. De este modo creamos un cuello de botella que impide que varios

hilos accedan simultáneamente a los mismos recursos.

Al ejecutarlo este es el resultado:



En la foto no se aprecia pero cuando se ve en movimiento vemos que van escalonados, de trozo en trozo en vez de píxel a píxel.

También podíamos haber creado dos objetos **Mutex** que hagan que un hilo no comience a ejecutarse hasta que otro hilo le active su **Mutex** correspondiente. Las combinaciones que se pueden hacer con **Mutex** y **Event** son infinitas.

En el próximo artículo veremos como crear variables de tipo **ThreadVar** y otros asuntos interesantes.

Pruebas realizadas en RAD Studio 2007.

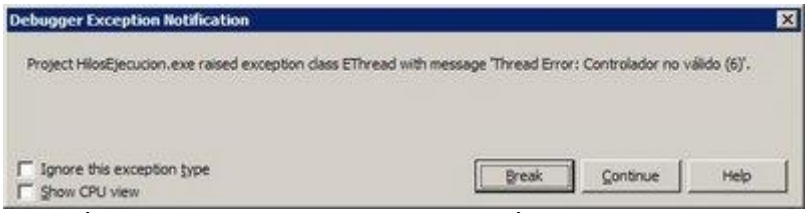
Publicado por Administrador en 18:54 4 comentarios
Etiquetas: sistema

22 mayo 2009

Los Hilos de Ejecución (2)

En el artículo anterior vimos un sencillo ejemplo de cómo crear un contador utilizando un hilo de ejecución, pero hay un pequeño inconveniente que tenemos que arreglar.

Dentro de un hilo de ejecución podemos actualizar la pantalla (los formularios y componentes) siempre y cuando la actualización sea muy esporádica y el trabajo a realizar sea muy rápido (entrar y salir). Pero si durante el evento**Execute** de un hilo intentamos dibujar o actualizar los componentes VCL más de lo normal, puede provocar una excepción de este tipo:



También suele ocurrir si dos hilos ejecutándose en paralelo intentan actualizar el mismo componente VCL.

Esto se debe a que la librería VCL no ha sido diseñada para trabajar con problemas de concurrencia en múltiples hilos de ejecución.

SINCRONIZANDO CÓDIGO ENTRE HILOS

Para evitar este problema vamos a utilizar el procedimiento **Synchronice** que permite

que un hilo llame a un procedimiento definido dentro del mismo pero en el contexto del hilo primario, es decir, el hilo principal de nuestro programa ejecuta el procedimiento que el hilo le manda como parámetro utilizando **Synchronice**. Es como si el hilo le dijera al hilo principal: ejecútame esto que a mi me da la risa.

De este modo ya podemos manipular los componentes VCL sin preocupación.

Modificamos la definición de la clase:

```
type
  TContador = class(TThread)
    dwTiempo: DWord;
    iSegundos: Integer;
    Etiqueta: TLabel;
    constructor Create; reintroduce; overload;
    procedure Execute; override;
    procedure ActualizarPantalla;
  end;
```

Y la implementación de **ActualizarPantalla** es la siguiente:

```
procedure TContador.ActualizarPantalla;
begin
  Etiqueta.Caption := IntToStr(iSegundos);
end;
```

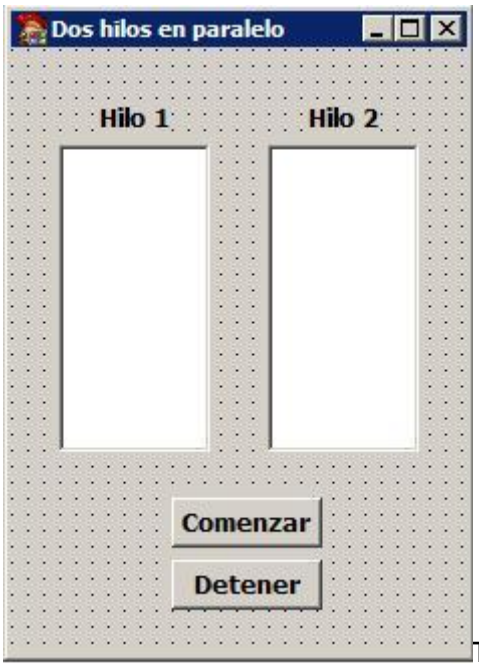
Entonces sólo hay que sincronizar el hilo con la VCL en el procedimiento **Execute**:

```
procedure TContador.Execute;
begin
  inherited;
  OnTerminate := Terminar;

  // Contamos hasta 10 segundos
  while (iSegundos < 10) and not Terminated do
  begin
    // ¿Han pasado 1000 milisegundos?
    if GetTickCount - dwTiempo > 1000 then
    begin
      // Incrementamos el contador de segundos y
      // actualizamos la etiqueta
      Inc(iSegundos);
      Synchronize(ActualizarPantalla);
      dwTiempo := GetTickCount;
    end;
  end;
end;
```

Llamamos al procedimiento **ActualizarPantalla** utilizando **Synchronize**. Y no sólo podemos utilizar este método para sincronizarnos con los componentes VCL sino que también con variables globales.

Ahora vamos otro caso de dos hilos incrementando paralelamente una misma variable

que hace de contador. Tenemos este formulario:



Tenemos dos listas (**TListBox**) que nos van a mostrar el valor de la variable global **i** que van a incrementar dos hilos a la vez cuya clase es la siguiente:

```
type
  THilo = class(TThread)
    Lista: TListBox;
    procedure Execute; override;
    procedure MostrarContador;
  end;
```

Y esta sería su implementación:

```
{ THilo }

procedure THilo.Execute;
begin
  inherited;
  FreeOnTerminate := True;
  while not Terminated do
  begin
    i := i + 1;
    Synchronize(MostrarContador);
    Sleep(1000);
  end;
end;
```

El procedimiento **Execute** incrementa la variable entera global **i**, la muestra por pantalla en su lista correspondiente y espera un segundo. Luego tenemos el procedimiento que muestra el valor de la variable **i** según el hilo:

```
procedure THilo.MostrarContador;
begin
```

```
  Lista.Items.Add('i='+IntToStr(i));
end;
```

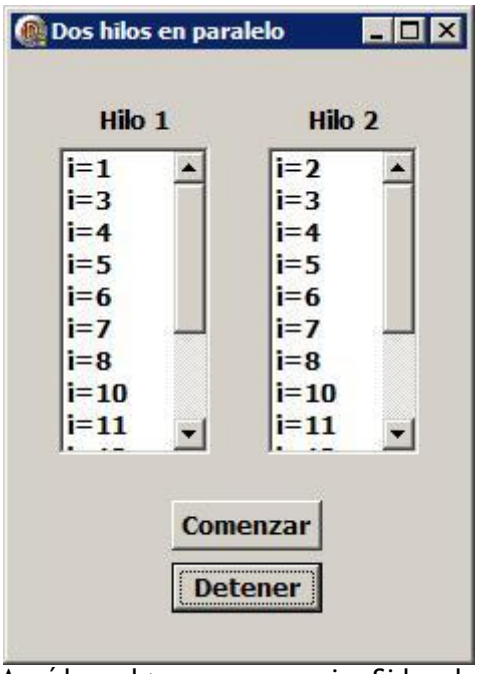
Cuando pulsemos el botón **Comenzar**, ponemos el contador **i** a cero, creamos los dos hilos, les asignamos su lista correspondiente y los ponemos en marcha:

```
procedure TFDosHilos.BComenzarClick(Sender: TObject);
begin
  i := 0;
  Hilo1 := THilo.Create(False);
  Hilo2 := THilo.Create(False);
  Hilo1.Lista := Lista1;
  Hilo2.Lista := Lista2;
  Hilo1.Resume;
  Hilo2.Resume;
end;
```

En cualquier momento podemos detener la ejecución de ambos hilos pulsando el botón **Detener**:

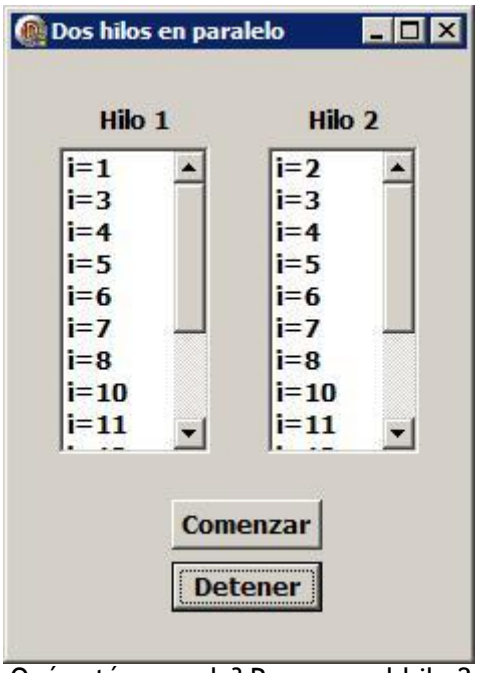
```
procedure TFDosHilos.BDetenerClick(Sender: TObject);
begin
  Hilo1.Terminate;
  Hilo2.Terminate;
end;
```

Al ejecutarlo conforme está ahora mismo este sería el resultado:



Aquí hay algo que no encaja. Si los dos hilos incrementan una vez la variable **i** sólo deberían verse números pares o todo caso el valor de un hilo debería ser distinto al del otro, pero eso es lo que pasa cuando no van sincronizados los incrementos de la variable **i**.

Sin tocar el código fuente, vuelvo a ejecutar el programa y me da este otro resultado:



¿Qué está pasando? Pues que el hilo 2 puede incrementar la variable **i** justo después de que la incremente el hilo 1 o bien pueden hacerlo los dos a la vez, provocando un retraso en el contador.

Esto podemos solucionarlo sincronizando el incremento de la variable **i** dentro de un nuevo procedimiento que podemos añadir a la clase **THilo**:

```
THilo = class(TThread)
  Lista: TListBox;
  procedure Execute; override;
  procedure MostrarContador;
  procedure IncrementarContador;
end;
```

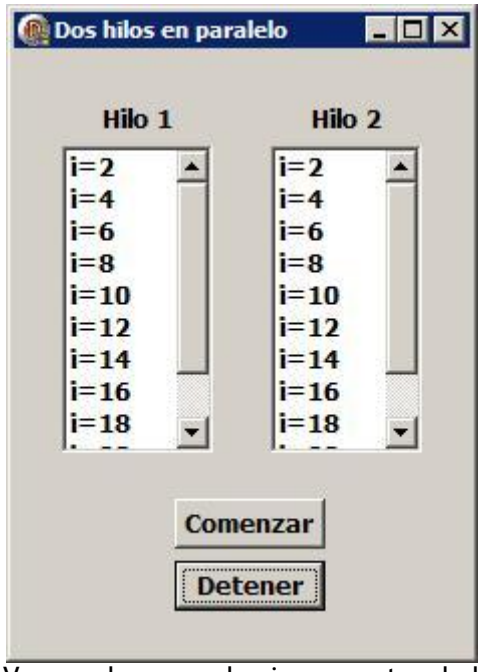
El procedimiento **IncrementarContador** sólo hace esto:

```
procedure THilo.IncrementarContador;
begin
  i := i + 1;
end;
```

Luego hacemos que el procedimiento **Execute** sincronice el incremento de la variable **i**:

```
procedure THilo.Execute;
begin
  inherited;
  FreeOnTerminate := True;
  while not Terminated do
  begin
    Synchronize(IncrementarContador);
    Synchronize(MostrarContador);
    Sleep(1000);
  end;
end;
```

Este sería el resultado al ejecutarlo:



Vemos ahora que los incrementos de la variable **i** son de dos en dos y evitamos que un hilo estropee el incremento de otro. Aunque esto no soluciona a la perfección la sincronización ya que si por ejemplo tenemos otra aplicación (como el navegador Firefox) que ralentiza el funcionamiento de Windows, podría provocar que se relentice uno de los hilos respecto al otro y provoque de nuevo la descoordinación en los incrementos. Eso veremos como solucionarlo en el siguiente artículo mediante semáforos y eventos.

PARAR, REANUDAR, DETENER O ESPERAR LA EJECUCIÓN DEL HILO

Siguiendo con el primer ejemplo (la clase **TContador**), una vez el hilo está en marcha podemos suspenderlo (manteniendo su estado) utilizando el método **Suspend**:

```
Contador.Suspend;
```

Para que continúe sólo hay que volver a llamar al procedimiento **Resume**:

```
Contador.Resume;
```

En cualquier momento podemos saber si está suspendido el hilo mediante la propiedad:

```
if contador.Suspended then
  ...
```

Si queremos detener definitivamente la ejecución del hilo entonces deberíamos añadirle al bucle principal de nuestro procedimiento **Execute** esta comprobación:

```
// Contamos hasta 10 segundos
while (iSegundos < 10) and not Terminated do
begin
 ...
```

Entonces podíamos añadir al formulario un botón llamado **Detener** que al pulsarlo haga esto:

```
Contador.Terminate;
```

Lo que hace realmente el procedimiento **Terminate** no es terminar el hilo de ejecución, sino poner la propiedad**Terminated** a **True** para que nosotros salgamos lo antes posible del bucle infinito en el que está sumergido nuestro hilo. Lo que nunca hay que intentar es hacer esto para terminar un hilo:

```
Contador.Free;
```

Lo que es terminar, seguro que termina, pero la explosión la tenemos asegurada. Para cerrar el hilo inmediatamente podemos llamar a una función de la API de Windows llamada **TerminateThread**:

```
TerminateThread(Contador.Handle, 0);
```

El primer parámetro es el **Handle** del hilo que queremos detener y el según parámetro es el código de salida que puede ser leído posteriormente por la función **GetExitCodeThread**. Recomiendo no utilizar la función**TerminateThread** a menos que sea estrictamente necesario. Es mejor utilizar el procedimiento **Terminate** y procurar salir limpiamente del bucle del hilo cerrando los asuntos que tengamos a medio (cerrando ficheros abiertos, liberando memoria de objetos creados, etc.).

Por otro lado tenemos el procedimiento **DoTerminate** que no termina el hilo de ejecución, sino que provoca el evento **OnTerminate** en el cual podemos colocar todo lo que necesitamos para terminar nuestro hilo. Por ejemplo, podemos poner al comienzo de nuestro procedimiento **Execute**:

```
procedure TContador.Execute;
begin
  inherited;
  OnTerminate := Terminar;
  ...
```

Y luego definimos dentro de nuestra clase **TContador** el evento **Terminar**:

```
procedure TContador.Terminar(Sender: TObject);
begin
  // Escribimos aquí el código que necesitamos
  // para liberar los recursos del hilo y luego
  // le mandamos una señal para que termine el bucle
principal
  Terminate;
end;
```

También podemos hacer que el hilo principal de la aplicación espere a que termine la ejecución de un hilo antes de seguir ejecutando código. Esto se consigue con el procedimiento **WaitFor**:

```
Contador.WaitFor;
```

Aunque esto puede ser algo peligroso porque no permite que Windows procese los mensajes de nuestra ventana. Podría utilizarse por ejemplo para ejecutar un programa externo (de MSDOS) y esperar a que termine su ejecución.

En el siguiente artículo veremos como podemos controlar las esperas entre el hilo principal y los hilos secundarios para que se coordinen en sus trabajos.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 12:53 1 comentarios
Etiquetas: sistema

08 mayo 2009

Los Hilos de Ejecución (1)

Hace tiempo escribí un pequeño post sobre como crear hilos de ejecución:

Cómo crear un hilo de ejecución

Pero debido a que últimamente los procesadores multinúcleo van tomando más relevancia y que algunos lectores de este blog han pedido que me extienda en el tema, voy a meterme de lleno en el apasionante y peligroso mundo de las aplicaciones multihilo.

Cuando manejamos Windows creemos que es un sistema operativo robusto que puede ejecutar gran cantidad de tareas a la vez. Eso es lo que parece externamente, pero realmente sólo tiene dos hilos de ejecución en los últimos procesadores de doble núcleo. Y en procesadores Pentium 4 e inferiores sólo contiene un solo hilo de ejecución (dos mediante el disimulado HyperThreading que tantos problemas nos dio con Interbase 6).

Entonces, ¿Cómo puede ejecutar varias tareas a la vez? (Emule, Bittorrent, MSN, reproductor multimedia, etc.). Primero tenemos que ver lo que significa un proceso.

LOS PROCESOS

Un proceso (un programa EXE o un servicio de Windows) contiene como mínimo un hilo, llamado hilo primario. Si queremos que nuestro proceso tenga más hilos, se lo tenemos que pedir al sistema operativo. Estos nuevos hilos pueden ejecutarse paralelamente al hilo primario y ser independientes.

Un proceso se compone principalmente de estas tres áreas:

AREA DE CÓDIGO: Es una zona de memoria de sólo lectura donde está compilado nuestro programa en código objeto (binario).

HEAP (MONTÓN): Es en esta zona de memoria de lectura/escritura donde se suelen guardar las variables globales que creamos así como las instancias de los objetos que se van creando en memoria.

PILA: Aquí se guardan temporalmente los parámetros que se pasan entre funciones así como las direcciones de llamada y retorno de procedimientos y funciones. Todos los datos que se introducen en la pila tienen que salir mediante el método LIFO (lo último que entra es lo primero que tiene que salir) porque si no provoca un desbordamiento del puntero del procesador que hace que retorne a otra zona de memoria provocando el error Access Violation 000000 o FFFFFF, es decir, intenta salirse del segmento de memoria de nuestra aplicación.

Estas tres zonas se encuentran herméticamente separadas las unas de las otras y si por

error un comando de nuestro programa intenta acceder fuera del heap o de la pila provocará el conocido mensaje que tanto nos gusta: Access Violation. Suele ocurrir al intentar acceder a un objeto que no ha sido creado en el heap (dirección 000000 = nil), etc.

El conjunto de estas tres zonas de memoria es el proceso. Ahora bien, si creamos un nuevo hilo de ejecución dentro del proceso, éste tendrá su propia pila, aunque compartirá la misma zona de datos (heap).

EL PROCESO MULTITAREA

Windows realiza la multitarea cediendo un pequeño tiempo de procesador a cada proceso, de modo que en un solo segundo pueden ejecutarse ciertos de procesos simultáneos. Para los viejos roqueros que estudiamos ensamblador vimos que antes de cambiar de tarea Windows guarda el estado de todos los registros en la pila:

```
PUSH EAX
PUSH EBX
...
```

Para luego restaurarlos y seguir su marcha:

```
...
POP EBX
POP EAX
```

No está de más adquirir unos pequeños conocimientos de ensamblados de x86 para conocer a fondo las tripas de la máquina.

Cuando Windows para el control de un proceso, memoriza el estado de los registros del procesador y la pila y pasa al siguiente proceso hasta que finalice su tiempo. Realmente, este cambio de procesos no lo realiza Windows, sino el mismo procesador que trae esta característica por hardware (desde el 80386. Hasta el procesador 80286 tenía este modo multitarea aunque sólo en 16 bits).

Pero lo que vamos a ver es este artículo es un cambio de ejecución entre hilos del mismo proceso, algo que consume muchos menos recursos que el cambio entre procesos.

Por ejemplo, los navegadores webs modernos actuales como Firefox o Chrome ejecutan cada web dentro del mismo proceso pero en hilos diferentes (incluso Chrome lo hace en distintos procesos), de modo que si una página web se queda colgada no afecta a otra situada en la siguiente pestaña (lo que no se es si llamarán al sistema de hilos de Windows y tendrán su propio núcleo duro de ejecución, su propio sistema multitarea).

Pero programar hilos de ejecución no está exento de problemas. ¿Qué pasaría si dos hilos acceden a la vez a la misma variable de memoria? O peor aún, ¿Y si intentan escribir a la vez en una unidad de CD-ROM? Las excepciones que pueden ocurrir pueden ser catastróficas, aunque afortunadamente a partir de Windows NT y XP se controlan muy bien todos estos problemas de concurrencia.

Lo que si es realmente difícil es intentar depurar dos hilos de ejecución que utilizan los mismos recursos del proceso. De ahí a que sólo hay que recurrir a los hilos en casos estrictamente necesarios.

LA CLASE TTHREAD

Toda la complejidad de los hilos de ejecución que se programan en la API de Windows quedan encapsulados en Delphi en esta sencilla clase. Para crear un hilo de ejecución basta heredar de la clase **TThread** y sobrecargar el método **Execute**:

```
type
  THilo = class(TThread)
    procedure Execute; override;
  end;
```

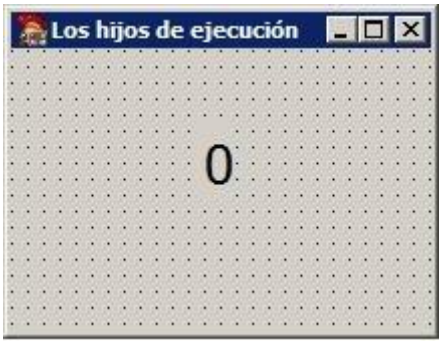
En la implementación del procedimiento **Execute** es donde hay que introducir el código que queremos ejecutar cuando arranque nuestro hilo de ejecución:

```
{ THilo }

procedure THilo.Execute;
begin
  inherited;

end;
```

Veamos un ejemplo sencillo de un hilo de ejecución con un contador de 10 segundos.

CREANDO UN CONTADOR DE SEGUNDOS

Voy a crear un nuevo proyecto con un formulario en el que sólo va a haber una etiqueta (**TLabel**) llamada**EContador** con el contador de segundos:



A este formulario lo he llamado **FPrincipal**. En la interfaz del mismo voy a definir la clase **TContador** que va a heredar de un hilo **TThread**:

```
type
  TContador = class(TThread)
    dwTiempo: DWord;
    iSegundos: Integer;
    Etiqueta: TLabel;
    constructor Create; reintroduce; overload;
    procedure Execute; override;
  end;
```

He sobrecargado el constructor para que inicialice mis contadores de tiempo y segundos:

```
constructor TContador.Create;
begin
  inherited Create(True); // llamamos al constructor del
```

```
padre (TThread)
  dwTiempo := GetTickCount;
  iSegundos := 0;
end;
```

La función **GetTickCount** nos devuelve un número entero que representa el número de milisegundos que han pasado desde que encendimos nuestro PC. Si queréis más precisión podéis utilizar la función **TimeGetTime**declarada en la unidad **MMSystem** (sobre todo si vais a programar videojuegos).

Siguiendo con la implementación de nuestra clase **TContador**, la variable **dwTiempo** la voy a utilizar para controlar el número de milisegundos que van pasando desde que ejecutamos el hilo. Y la variable **iSegundos** es un contador de segundos de 0 a 10. También he añadido una referencia a una **Etiqueta** de tipo **TLabel** que se la daremos al crear una instancia del hilo en el evento **OnCreate** del formulario principal:

```
procedure TFPrincipal.FormCreate(Sender: TObject);
begin
  Contador := TContador.Create(True);
  Contador.Etiqueta := EContador;
  Contador.FreeOnTerminate := True;
  Contador.Resume;
end;
```

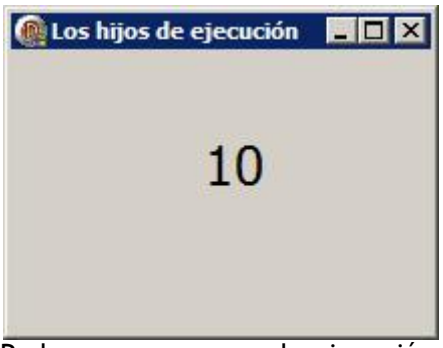
Después de crear el hilo le pasamos la etiqueta que tiene que actualizar y le decimos mediante la propiedad**FreeOnTerminate** que se libere de memoria automáticamente al terminar la ejecución del hilo.

Después llamamos al método **Resume** que lo que hace internamente es ejecutar el procedimiento **Execute** de la clase **TThread** que tendrá este código:

```
procedure TContador.Execute;
begin
  inherited;

  // Contamos hasta 10 segundos
  while iSegundos < 10 do
    // ¿Han pasado 1000 milisegundos?
    if GetTickCount - dwTiempo > 1000 then
    begin
      // Incrementamos el contador de segundos y
actualizamos la etiqueta
      Inc(iSegundos);
      Etiqueta.Caption := IntToStr(iSegundos);
      dwTiempo := GetTickCount;
    end;
end;
```

Creamos un bucle cerrado que va contando de 1000 en 1000 milisegundos e incrementando el contador de segundos. Cuando se salga de este bucle se terminará la ejecución del hilo automáticamente. También se podía haber utilizado procedimiento **Sleep**, pero nunca me ha gustado mucho esta función porque miestras se está ejecutando no podemos hacer absolutamente nada.

Eso ocurrirá al ejecutar el programa y cuando llegue el contador a 10:



Podemos ver como va la ejecución del hilo en la ventana inferior de Delphi si estamos ejecutando el programa en modo depuración:

Como puede verse en las líneas en rojo, el hilo que hemos creado tiene el ID 3736 que no tiene nada que ver con el hilo primario:

En el siguiente artículo seguiremos profundizando un poco más en los hilos de ejecución a través de otros ejemplos.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 10:00 8 comentarios
Etiquetas: sistema

17 abril 2009

Crea tu propio editor de informes (y V)

Voy a cerrar esta serie de artículos dedicada a la creación de nuestro propio editor de informes añadiendo la posibilidad de guardar y cargar los informes que hemos diseñado.

Hay diversas maneras de poder hacerlo: en bases de datos, en archivos de texto, en archivos binarios (con records) y también en XML.

Voy a elegir el lenguaje XML porque es más flexible para futuras ampliaciones.

GUARDANDO TODO EL INFORME EN XML

Para poder guardar y cargar informes necesitamos añadir en el formulario principal el componente de la clase**TXMLDocument** que se encuentra en la sección **Internet**:

Le vamos a poner de nombre XML para simplificar y ponemos su propiedad **Active** a **True**. También vamos a necesitar el componente **TSaveDialog** que se encuentra en la

sección **Dialogs**. A este componente lo he llamado**GuardarXML**.

Vamos a ver en partes como sería guardar todo el documento en XML. Lo primero que hago es comprobar si hay algún informe abierto (una ventana hija MDI de la clase **TFInforme**) y si no es así saco una advertencia:

```
procedure TFPrincipal.GuardarComoClick(Sender: TObject);
var
  NInforme, NConfig, NBanda, NComponente: IXMLNode; //
nodos del XML
  FBuscaInforme: TFInforme;
  i, j: Integer;
  Banda: TQRBand;
  Etiqueta: TQRLabel;
  Figura: TQRShape;
  Campo: TQRDBText;
begin
  if ActiveMDIChild is TFInforme then
    FBuscaInforme := ActiveMDIChild as TFInforme
  else
  begin
    Application.MessageBox('No hay ningún informe para
guardar.',
      'Proceso cancelado', MB_ICONSTOP);
    Exit;
  end;
```

Después limpio todo el componente XML (por si grabamos por segunda vez) y creo un primer nodo llamado **Informe**del que van a colgar todos los demás. También guardo un nodo de configuración donde meto la conexión a la base de datos y la tabla:

```
XML.ChildNodes.Clear;

// Creamos el nodo principal
NInforme := XML.AddChild('Informe');

// Añadimos el nodo de configuración
NConfig := NInforme.AddChild('Configuracion');
NConfig.Attributes['BaseDatos'] :=
FBuscaInforme.BaseDatos.DatabaseName;
NConfig.Attributes['Tabla'] :=
FBuscaInforme.Tabla.TableName;
```

Ahora viene la parte más pesada de este procedimiento. Tengo que recorrer todos los componentes del informe en busca de una banda y cuando la encuentra la grabo en el XML:

```
// Recorremos todos los componentes del informe y los vamos
guardando
for i := 0 to FBuscaInforme.Informe.ComponentCount-1 do
begin
  // ¿Hemos encontrado una banda?
```

```
  if FBuscaInforme.Informe.Components[i] is TQRBand then
  begin
    Banda := FBuscaInforme.Informe.Components[i] as
TQRBand;
    NBanda := NInforme.AddChild('Banda');
    NBanda.Text := Banda.Name;
    NBanda.Attributes['Alto'] := Banda.Height;
    NBanda.Attributes['Tipo'] := Ord(Banda.BandType);
```

Después vuelvo a recorrer con otro bucle todos los componentes pertenecientes a esa banda y los voy guardando:

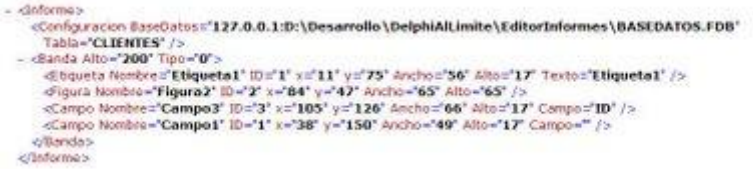
```
   // Recorremos todos los componentes de la banda
   for j := 0 to Banda.ComponentCount-1 do
   begin
     // ¿Es una etiqueta?
     if Banda.Components[j] is TQRLabel then
     begin
       Etiqueta := Banda.Components[j] as TQRLabel;
       NComponente := NBanda.AddChild('Etiqueta');
       NComponente.Attributes['Nombre'] := Etiqueta.Name;
       NComponente.Attributes['ID'] := Etiqueta.Tag;
       NComponente.Attributes['x'] := Etiqueta.Left;
       NComponente.Attributes['y'] := Etiqueta.Top;
       NComponente.Attributes['Ancho'] := Etiqueta.Width;
       NComponente.Attributes['Alto'] := Etiqueta.Height;
       NComponente.Attributes['Texto'] := Etiqueta.Caption;
     end;

     // ¿Es una figura?
     if Banda.Components[j] is TQRShape then
     begin
       Figura := Banda.Components[j] as TQRShape;
       NComponente := NBanda.AddChild('Figura');
       NComponente.Attributes['Nombre'] := Figura.Name;
       NComponente.Attributes['ID'] := Figura.Tag;
       NComponente.Attributes['x'] := Figura.Left;
       NComponente.Attributes['y'] := Figura.Top;
       NComponente.Attributes['Ancho'] := Figura.Width;
       NComponente.Attributes['Alto'] := Figura.Height;
     end;

     // ¿Es un campo?
     if Banda.Components[j] is TQRDBText then
     begin
       Campo := Banda.Components[j] as TQRDBText;
       NComponente := NBanda.AddChild('Campo');
       NComponente.Attributes['Nombre'] := Campo.Name;
       NComponente.Attributes['ID'] := Campo.Tag;
       NComponente.Attributes['x'] := Campo.Left;
       NComponente.Attributes['y'] := Campo.Top;
```

```
        NComponente.Attributes['Ancho'] := Campo.Width;
        NComponente.Attributes['Alto'] := Campo.Height;
        NComponente.Attributes['Campo'] := Campo.DataField;
      end;
    end;
  end;
end;
```

Cuando termine vuelco todo el XML a un archivo preguntándole el nombre al usuario:

```
if GuardarXML.Execute then
  XML.SaveToFile(GuardarXML.FileName);
```

Así quedaría después de guardarlo:



Como puede apreciarse en la imagen (hacer clic para ampliar), la libertad que nos da un archivo XML para guardar una información jerárquica es mucho más flexible que con cualquier otro tipo de archivo.

CARGARDO EL INFORME DESDE UN ARCHIVO XML

Para cargar el informe vamos a necesitar el componente **TOpenDialog** en el formulario principal y que llamaremos**AbrirXML**. Veamos paso a paso el proceso de carga.

Abrimos el documento XML y comenzamos a recorrer todos los nodos en busca del archivo de configuración:

```
procedure TFPrincipal.AbrirClick(Sender: TObject);
var
  FNuevoInforme: TFInforme;
  i, j: Integer;
  NBanda, NComponente: IXMLNode; // nodos del XML
  Etiqueta: TQRLabel;
  Figura: TQRShape;
  Campo: TQRDBText;
begin
  if not AbrirXML.Execute then
    Exit;

  XML.LoadFromFile(AbrirXML.FileName);

  // Creamos un nuevo informe
  FNuevoInforme := TFInforme.Create(Self);

  // Recorremos todos los nodos del XML
  for i := 0 to XML.ChildNodes[0].ChildNodes.Count-1 do
  begin
    // ¿Es el archivo de configuración?
```

```
    if XML.ChildNodes[0].ChildNodes[i].NodeName =
'Configuracion' then
    begin
      FNuevoInforme.BaseDatos.DatabaseName :=

XML.ChildNodes[0].ChildNodes[i].Attributes['BaseDatos'];

      FNuevoInforme.Tabla.TableName :=

XML.ChildNodes[0].ChildNodes[i].Attributes['Tabla'];
    end;
```

He creado el formulario hijo **FNuevoInforme** de la clase **TFInforme** para ir creando en tiempo real los componentes que vamos leyendo del XML. Como podéis ver en el código, parto directamente del primer nodo XML.ChildNodes[0] ya que se con seguridad que el primero nodo se llama **Informe** y que todo lo demás cuelga del mismo.

Después vamos en busca de la banda y la creamos:

```
  // ¿Es una banda?
  if XML.ChildNodes[0].ChildNodes[i].NodeName = 'Banda'
then
  begin
    NBanda := XML.ChildNodes[0].ChildNodes[i];
    FNuevoInforme.BandaActual :=
TQRBand.Create(FNuevoInforme.Informe);

    with FNuevoInforme.BandaActual do
    begin
      Parent := FNuevoInforme.Informe;
      Height := NBanda.Attributes['Alto'];
      BandType := NBanda.Attributes['Tipo'];
    end;
```

Luego voy recorriendo todos los componentes de la banda y los voy metiendo en el informe con sus respectivas propiedades:

```
   // Recorremos todos sus componentes
   for j := 0 to NBanda.ChildNodes.Count-1 do
   begin
     // ¿Es una etiqueta?
     if NBanda.ChildNodes[j].NodeName = 'Etiqueta' then
     begin
       NComponente := NBanda.ChildNodes[j];
       Etiqueta :=
TQRLabel.Create(FNuevoInforme.BandaActual);
       Etiqueta.Parent := FNuevoInforme.BandaActual;
       Etiqueta.Name := NComponente.Attributes['Nombre'];
       Etiqueta.Tag := NComponente.Attributes['ID'];
       Etiqueta.Left := NComponente.Attributes['x'];
       Etiqueta.Top := NComponente.Attributes['y'];
       Etiqueta.Width := NComponente.Attributes['Ancho'];
```

```
        Etiqueta.Height := NComponente.Attributes['Alto'];
        Etiqueta.Caption := NComponente.Attributes['Texto'];
      end;

      // ¿Es una figura?
      if NBanda.ChildNodes[j].NodeName = 'Figura' then
      begin
        NComponente := NBanda.ChildNodes[j];
        Figura :=
TQRShape.Create(FNuevoInforme.BandaActual);
        Figura.Parent := FNuevoInforme.BandaActual;
        Figura.Name := NComponente.Attributes['Nombre'];
        Figura.Tag := NComponente.Attributes['ID'];
        Figura.Left := NComponente.Attributes['x'];
        Figura.Top := NComponente.Attributes['y'];
        Figura.Width := NComponente.Attributes['Ancho'];
        Figura.Height := NComponente.Attributes['Alto'];
      end;

      // ¿Es una campo?
      if NBanda.ChildNodes[j].NodeName = 'Campo' then
      begin
        NComponente := NBanda.ChildNodes[j];
        Campo :=
TQRDBText.Create(FNuevoInforme.BandaActual);
        Campo.Autosize := False;
        Campo.Parent := FNuevoInforme.BandaActual;
        Campo.Tag := NComponente.Attributes['ID'];
        Campo.DataField := NComponente.Attributes['Campo'];
        Campo.Left := NComponente.Attributes['x'];
        Campo.Top := NComponente.Attributes['y'];
        Campo.Width := NComponente.Attributes['Ancho'];
        Campo.Height := NComponente.Attributes['Alto'];
        Campo.Name := NComponente.Attributes['Nombre'];

        if Campo.DataField <> '' then
          Campo.Caption := Campo.DataField
        else
          Campo.Caption := NComponente.Attributes['Nombre'];

        Campo.DataSet := FNuevoInforme.Tabla;
      end;
    end;
  end;
end;
```

Por último, seleccionamos la banda actual:

```
FNuevoInforme.SeleccionarBandaActual;
```

Al final tiene que quedarse igual que el informe original:

Me gustaría haber dejado el editor algo más pulido con características tales como la opción de **Guardar** (creando un procedimiento común para **Guardar** y **Guardar como**), poder seleccionar el tipo de banda, o añadir otros componentes de QuickReport. Pero por falta de tiempo no ha podido ser.

De todas formas, creo que con este material podéis ver lo duro que tiene que ser crear un editor de informes profesional y si alguien se anima a continuar a partir de aquí pues mucha suerte y paciencia. Lo que más se aprende con estos temas es a manipular componentes VCL en tiempo real y a crear nuestro propio editor visual.

DESCARGAR EL PROYECTO

Rapidshare:

http://rapidshare.com/files/223134561/EditorInformes5Final_DelphiAlLimite.zip.html

Megaupload:

http://www.megaupload.com/?d=3MUEMUUD

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 09:43 1 comentarios
Etiquetas: impresión

03 abril 2009

Crea tu propio editor de informes (IV)

En el artículo de hoy voy a añadir al editor la funcionalidad de añadir conectarse a una base de datos Interbase/Firebird para añadir los campos de las tablas.

AÑADIENDO MÁS COMPONENTES AL INFORME

Para poder conectar con la base de datos necesitamos añadir tres componentes al formulario **FInforme**:

1º Un componente **TIBDatabase** que se encuentra en la sección **Interbase** y que vamos a

llamar **BaseDatos**:

Modificamos su propiedad **LoginPrompt** a **False** y no hay que olvidarse de darle el usuario **SYSDBA** y la contraseña**masterkey** al hacer doble clic sobre el mismo.

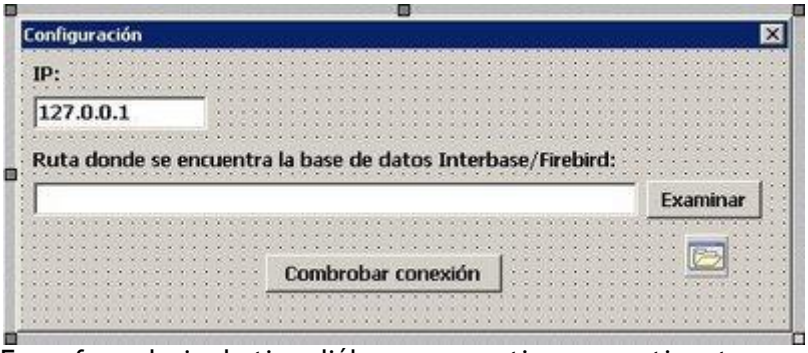
2º Un componente **TIBTransaction** que también se encuentra en **Interbase** y le ponemos el nombre **Transaccion**:

También debemos vincular la transacción a la base de datos mediante la propiedad **DefaultTransaction**.

3º Y un componente de la clase **TIBTable** para traernos los campos de la tabla al hacer la vista previa.

CREANDO EL FORMULARIO DE CONFIGURACION

Creamos un nuevo formulario llamado **FConfiguracion** con lo siguiente:



Es un formulario de tipo diálogo que contiene una etiqueta, un componente **TEdit** llamado **Ruta** y un botón llamado**BExaminar** que al pulsarlo utiliza al componente **TOpenDialog** para elegir una base de datos local:

```
procedure TFConfiguracion.BExaminarClick(Sender: TObject);
begin
  Abrir.Filter := 'Interbase/Firebird|*.gdb;*.fdb';

  if Abrir.Execute then
    Ruta.Text := Abrir.FileName;
end;
```

Si la base de datos fuera remota el botón **Examinar** no sirve de nada ya que la ruta sería una unidad real del servidor.

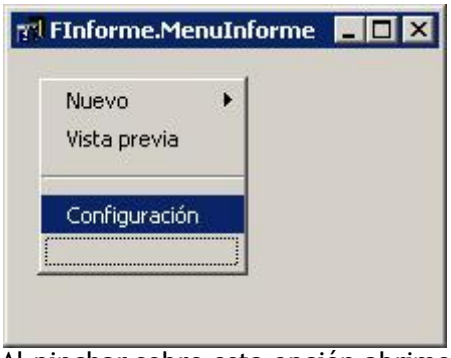
Para poder realizar un test de la conexión lo que hacemos es pasarle a este formulario la base de datos con la que vamos a conectar. Para ello creamos esta variable pública:

```
public
  { Public declarations }
  BaseDatos: TIBDatabase;
```

Para poder compilar este componente debemos añadir a la sección uses la unidad **IBDatabase**. Entonces en el botón de comprobar la conexión hacemos esto:

```
procedure TFConfiguracion.BConectarClick(Sender: TObject);
begin
  if BaseDatos.Connected then
    BaseDatos.Close;

  BaseDatos.DatabaseName := IP.Text + ':' + Ruta.Text;

  try
    BaseDatos.Open;
  except
    raise;
  end;

  Application.MessageBox( 'Conexión realizada
correctamente.',
    'Conectado', MB_ICONINFORMATION );
end;
```

Ahora nos vamos al formulario **FInforme** y añadimos una nueva opción al menú contextual:



Al pinchar sobre esta opción abrimos el formulario de configuración:

```
procedure TFInforme.ConfiguracionClick(Sender: TObject);
begin
  Application.CreateForm( TFConfiguracion, FConfiguracion
);
  FConfiguracion.BaseDatos := BaseDatos;
  FConfiguracion.ShowModal;
end;
```

Una vez conectados a la base de datos ya podemos crear añadir campos de bases de datos **TQRDBText**.

AÑADIENDO LOS COMPONENTES TDBTEXT

Aquí no voy a enrollarme mucho ya que el modo insertar los campos **TQRDBText** va a ser el mismo que he utilizado para los componentes **TQRLabel** o **TQRShape**.

Lo primero que he hecho es añadir la opción **Campo** a nuestro menú contextual:

Cuya implementación es la siguiente:

```
procedure TFInforme.CampoClick(Sender: TObject);
begin
  if BandaActual = nil then
  begin
    Application.MessageBox( 'Debe crear una banda.',
      'Acceso denegado', MB_ICONSTOP );
    Exit;
  end;

  QuitarSeleccionados;
  with TQRDBText.Create(BandaActual) do
  begin
    Parent := BandaActual;
    Left := 10;
    Top := 10;
    Inc(UltimoID);
    Tag := UltimoID;
    Caption := 'Campo' + IntToStr(UltimoID);
    Name := Caption;
  end;
end;
```

Después he modificado el procedimiento **SeleccionarComponente**:

```
procedure TFInforme.SeleccionarComponente;

...

// ¿Ha pinchado una etiqueta, una figura o un campo?
if (Seleccionado is TQRLabel) or (Seleccionado is TQRShape)
or
  (Seleccionado is TQRDBText) then

...
```

Luego he modificado el procedimiento **MoverRatonPulsado**:

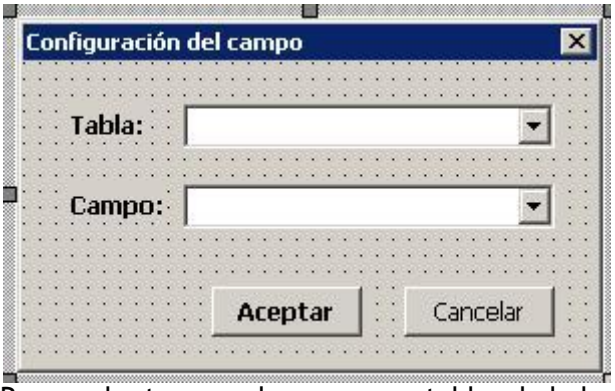
```
procedure TFInforme.MoverRatonPulsado;
```

```
...

// ¿Estamos moviendo una etiqueta, una figura o un campo?
if (Seleccionado is TQRLabel) or (Seleccionado is TQRShape)
or
  (Seleccionado is TQRDBText) then
begin

...
```

También he ampliado los procedimientos **ComprobarSeleccionComponentes, MoverOtrosSeleccionados**,**AmpliarComponenes**, etc., es decir, en todo lo referente a la selección y movimiento de componentes (ya lo veréis en el proyecto).

CONFIGURANDO LA TABLA Y EL CAMPO

Una vez insertados los campos en el informe debemos dar la posibilidad al usuario de especificar el nombre del campo y de la tabla al que pertenece. Para ello voy a crear un nuevo formulario llamado **FCampo** con lo siguiente:



Para poder traernos los campos y tablas de la base de datos le he creado estas tres variables públicas:

```
public
  { Public declarations }
  Campo: TQRDBText;
  Tabla: TIBTable;
  BaseDatos: TIBDatabase;
```

Estas variables nos las va a pasar el formulario **FInforme** cuando el usuario haga clic sobre un campo con la tecla**SHIFT** pulsada. He intentado asignarle un menú contextual a los objeto **TQRDBText** pero como no tienen la propiedad **Popup** no hay manera.

Así que en el procedimiento **SeleccionarComponente** compruebo si el usuario mantienen el botón **SHIFT** pulsado para abrir el formulario **FCampo**:

```
if (Seleccionado is TQRDBText) and
  (HiWord(GetAsyncKeyState(VK_LSHIFT)) <> 0) then
  ConfigurarCampo(Seleccionado as TQRDBText);
```

El procedimiento **ConfigurarCampo** le envía al formulario **FCampo** lo que necesita para trabajar:

```
procedure TFInforme.ConfigurarCampo(Campo: TQRDBText);
begin
  // si no está conectado con la base de datos no hacemos
nada
  if not BaseDatos.Connected then
  begin
    Application.MessageBox('Conecte primero con la base de
datos.',
      'Acceso denegado', MB_ICONSTOP);
    Exit;
  end;

  Application.CreateForm(TFCampo, FCampo);
  FCampo.BaseDatos := BaseDatos;
  FCampo.Campo := Campo;
  FCampo.Tabla := Tabla;
  FCampo.ShowModal;
end;
```

También comprueba si previamente hemos conectado con la base de datos. Dentro del formulario **FCampo** utilizo el evento **OnShow** para pedirle a la base de datos el nombre de las tablas:

```
procedure TFCampo.FormShow(Sender: TObject);
begin
  // Nos traemos el nombre de todas las tablas
  BaseDatos.GetTableNames(SelecTabla.Items, False);
end;
```

Cuando el usuario seleccione una tabla activando con el ComboBox **SelecTabla** el evento **OnChange** entonces vuelo a pedirle a la base de datos el nombre de los campos de la tabla seleccionada:

```
procedure TFCampo.SelecTablaChange(Sender: TObject);
begin
  // Si no ha seleccionado ninguna tabla no hacemos nada
  if SelecTabla.ItemIndex = -1 then
    Exit;

  // Le pedimos a la base de datos que nos de los campos de
la tabla
  BaseDatos.GetFieldNames(SelecTabla.Text,
SelecCampo.Items);
end;
```

Una vez pulsamos el botón **Aceptar** le envío al objeto **TQRDBText** el campo y la tabla a la que pertenece:

```
procedure TFCampo.BAceptarClick(Sender: TObject);
begin
  if SelecTabla.ItemIndex = -1 then
```

```
  begin
    Application.MessageBox('Debe seleccionar una tabla.',
      'Atención', MB_ICONSTOP);
    Exit;
  end;

  if SelecCampo.ItemIndex = -1 then
  begin
    Application.MessageBox('Debe seleccionar un campo.',
      'Atención', MB_ICONSTOP);
    Exit;
  end;

  Campo.DataField := SelecCampo.Text;
  Tabla.TableName := SelecTabla.Text;
  Campo.DataSet := Tabla;
  Close;
end;
```

MODIFICANDO LA VISTA PREVIA

Cuando hagamos una vista previa del documento entonces debemos abrir la tabla de la base de datos antes de imprimir:

```
procedure TFInforme.VistaPreviaClick(Sender: TObject);
begin
  // ¿Ha elegido el usuario una tabla?
  if Tabla.TableName <> '' then
  begin
    // Entonces se la asignamos al informe
    Informe.DataSet := Tabla;
    Tabla.Close;
    Tabla.Open;
  end;

  Informe.Preview;
end;
```

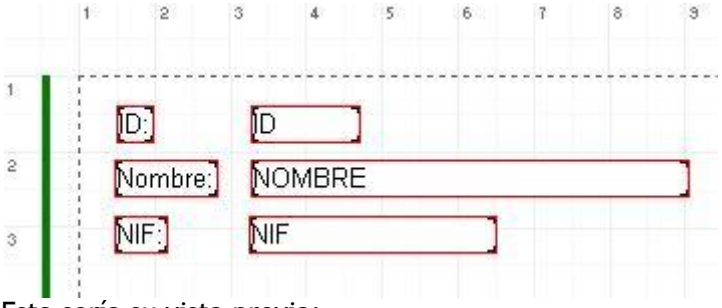
CAMBIAR EL NOMBRE DE LAS ETIQUETAS

Vamos a hacer una pequeña modificación más para poder cambiar el nombre de las etiquetas. Para ello volvemos a modificar el procedimiento **SeleccionarComponente** para que nos pida el nombre de la etiqueta al pincharla con el botón **SHIFT** pulsado:

```
if (Seleccionado is TQRLabel) and
(HiWord(GetAsyncKeyState(VK_LSHIFT)) <> 0) then
begin
  EtiSel := Seleccionado as TQRLabel;
  sNombre := InputBox('Cambiar nombre etiqueta', 'Nombre:',
'');
  if sNombre <> '' then
```

```
    EtiSel.Caption := sNombre;
end;
```

Con todo esto que he montado se puede hacer un pequeño informe a una banda con la ficha de un cliente:



Esta sería su vista previa:

EL PROYECTO

Aquí va el proyecto comprimido en RapidShare y Megaupload:

http://rapidshare.com/files/216858597/EditorInformes4_DelphiAlLimite.rar.html

http://www.megaupload.com/?d=AKDT45NN

También he incluido una base de datos creada con Firebird 2.0 con la tabla de CLIENTES. En el próximo artículo voy a finalizar esta serie dedicada a nuestro propio editor de informes añadiendo la posibilidad de grabar y cargar plantillas de informes.

Pruebas realizadas en RAD Studio 2007.

13 marzo 2009

Crea tu propio editor de informes (III)

Hoy vamos a perfeccionar la función de mover varios componentes seleccionados a la vez. También veremos como añadir figuras gráficas al informe y como cambiar sus dimensiones.

Al igual que en artículos anteriores daré al final del mismo el enlace para descargar todo el proyecto.

MOVIENDO VARIOS COMPONENTES CON EL RATÓN

Una vez hemos conseguido mover dos o más componentes a la vez mediante el teclado

tenemos que hacer lo mismo con el ratón, ya que si tenemos tres etiquetas seleccionadas e intentamos moverlas (cogiendo una) veremos que sólo se mueve esta última.

Para ello hay que modificar la primera parte del procedimiento **MoverRatonPulsado**:

```
procedure TFInforme.MoverRatonPulsado;
var
  dx, dy, xAnterior, yAnterior: Integer;
  Seleccion: TShape;
begin
  // ¿Ha movido el ratón estándo el botón izquierdo
pulsado?
  if ( ( x <> Cursor.X ) or ( y <> Cursor.Y ) ) and (
Seleccionado <> nil ) then
  begin
    // ¿Estamos moviendo una etiqueta o una banda?
    if (Seleccionado is TQRLabel) or (Seleccionado is
TQRShape) then
    begin
      // Movemos el componente seleccionado
      dx := Cursor.X - x;
      dy := Cursor.Y - y;
      xAnterior := Seleccionado.Left;
      yAnterior := Seleccionado.Top;
      Seleccionado.Left := xInicial + dx;
      Seleccionado.Top := yInicial + dy;

      // Movemos la selección
      Seleccion := BuscarSeleccion(Seleccionado.Tag);
      if Seleccion <> nil then
      begin
        Seleccion.Left := xInicial+dx-1;
        Seleccion.Top := yInicial+dy-1;
      end;

      MoverOtrosSeleccionados(Seleccionado.Tag,
Seleccionado.Left,
        Seleccionado.Top, xAnterior, yAnterior);
    end;

...
```

Sólo he añadido tres cosas:

1º He añadido al principio del procedimiento las variables **xAnterior** e **yAnterior** que van a encargarse de guardar la coordenada original de la etiqueta que estoy movimendo.

2º Antes de mover la etiqueta guardo sus coordenadas originales:

```
  xAnterior := Seleccionado.Left;
  yAnterior := Seleccionado.Top;
```

```
  Seleccionado.Left := xInicial + dx;
  Seleccionado.Top := yInicial + dy;
```

3º Llamo al procedimiento **MoverOtrosSeleccionados** que se encargará de arrastrar el resto de etiquetas seleccionadas:

```
  MoverOtrosSeleccionados(Seleccionado.Tag,
Seleccionado.Left,
    Seleccionado.Top, xAnterior, yAnterior);
```

Le paso como parámetro el **ID** de la etiqueta actual (para excluirla), la nueva posición y la anterior. Este sería el procedimiento al que llamamos:

```
procedure TFInforme.MoverOtrosSeleccionados( ID, x, y,
xAnterior, yAnterior: Integer );
var
  i, dx2, dy2: Integer;
  Etiqueta: TQRLabel;
  Seleccion: TShape;
begin
  for i := 0 to BandaActual.ComponentCount - 1 do
  begin
    // ¿Es una etiqueta?
    if (BandaActual.Components[i] is TQRLabel) then
    begin
      Etiqueta := BandaActual.Components[i] as TQRLabel;

      // ¿Es distinta a la etiqueta original y está
seleccionada?
      if (Etiqueta.Tag <> ID) and (Etiqueta.Hint <> '' )
then
      begin
        dx2 := x - xAnterior;
        dy2 := y - yAnterior;
        Etiqueta.Left := Etiqueta.Left + dx2;
        Etiqueta.Top := Etiqueta.Top + dy2;

        // Movemos la selección
        Seleccion := BuscarSeleccion(Etiqueta.Tag);
        if Seleccion <> nil then
        begin
          Seleccion.Left := Etiqueta.Left-1;
          Seleccion.Top := Etiqueta.Top-1;
        end;
      end;
    end;
  end;
end;
```

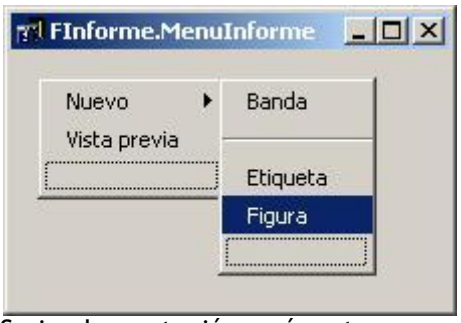
Lo que hace es recorrer todas las etiquetas de la banda actual (menos la seleccionada) y las arrastra según lo que se ha movido la etiqueta original:

Si movemos la etiqueta 3 entonces tienen que venirse detrás las etiquetas 1 y 2.

CREANDO FIGURAS GRÁFICAS

Otro elemento que se hace imprescindible en todo editor de informes son las figuras gráficas (líneas y rectángulos). Vamos a seguir la misma filosofía que para crear etiquetas.

Primero añadimos la opción **Figura** a nuestro menú contextual:



Su implementación sería esta:

```
procedure TFInforme.FiguraClick(Sender: TObject);
begin
  if BandaActual = nil then
  begin
    Application.MessageBox( 'Debe crear una banda.',
      'Acceso denegado', MB_ICONSTOP );
    Exit;
  end;

  QuitarSeleccionados;
  with TQRShape.Create(BandaActual) do
  begin
    Parent := BandaActual;
    Left := 10;
    Top := 10;
    Inc(UltimoID);
    Tag := UltimoID;
```

```
    Name := 'Figura' + IntToStr(UltimoID);
  end;
end;
```

Al igual que con las etiquetas, creamos un objeto **TQRShape** y le damos el siguiente **ID** que le corresponda.

Si ejecutamos el programa y creamos una nueva figura, aparecerá en la parte superior izquierda de la banda con borde negro y fondo blanco:

SELECCIONANDO LAS FIGURAS

Ahora tenemos que modificar las rutinas de selección para que se adapten a nuestro nuevo objeto **TQRShape**. Empezamos por el procedimiento **SeleccionarComponente**:

```
procedure TFInforme.SeleccionarComponente;
var
  Punto: TPoint;
begin
  bPulsado := True;
  x := Cursor.X;
  y := Cursor.Y;

  // ¿Ha pinchado una etiqueta o una figura?
  if (Seleccionado is TQRLabel) or (Seleccionado is
TQRShape) then
  begin
    xInicial := Seleccionado.Left;
    yInicial := Seleccionado.Top;

    if Seleccionado.Hint = '' then
    begin
      // ¿No esta pulsado la tacla control?
      if HiWord(GetAsyncKeyState(VK_LCONTROL)) = 0 then
        QuitarSeleccionados;

      Seleccionado.Hint := 'X';
    end;
  end;
...
```

Realmente sólo he cambiado esta línea:

```
  if (Seleccionado is TQRLabel) or (Seleccionado is
TQRShape) then
```

ya que no afecta a lo demás. Lo que si hay que ampliar bien es el procedimiento encargado de dibujar los componentes seleccionados:

```
procedure TFInforme.DibujarSeleccionados;
var
  i: Integer;
  Seleccion: TShape;
  Etiqueta: TQRLabel;
  Figura: TQRShape;
begin
  if BandaActual = nil then
  Exit;

  for i := 0 to BandaActual.ComponentCount-1 do
  begin
    // ¿Es una etiqueta?
    if BandaActual.Components[i] is TQRLabel then
    begin
      Etiqueta := BandaActual.Components[i] as TQRLabel;

      if Etiqueta.Hint <> '' then
      begin
        // Antes de crearla comprobamos si ya tiene
selección
        Seleccion := BuscarSeleccion(Etiqueta.Tag);

        if Seleccion = nil then
        begin
          Seleccion := TShape.Create(BandaActual);
          Seleccion.Parent := BandaActual;
          Seleccion.Pen.Color := clRed;
          Seleccion.Width := Etiqueta.Width+2;
          Seleccion.Height := Etiqueta.Height+2;
          Seleccion.Tag := Etiqueta.Tag;
          Seleccion.Name := 'Seleccion' +
IntToStr(Etiqueta.Tag);
        end;

        Seleccion.Left := Etiqueta.Left-1;
        Seleccion.Top := Etiqueta.Top-1;
      end;
    end;

    // ¿Es una figura?
    if BandaActual.Components[i] is TQRShape then
    begin
      Figura := BandaActual.Components[i] as TQRShape;
```

```
      if Figura.Hint <> '' then
      begin
        // Antes de crearla comprobamos si ya tiene
selección
        Seleccion := BuscarSeleccion(Figura.Tag);

        if Seleccion = nil then
        begin
          Seleccion := TShape.Create(BandaActual);
          Seleccion.Parent := BandaActual;
          Seleccion.Pen.Color := clRed;
          Seleccion.Width := Figura.Width+2;
          Seleccion.Height := Figura.Height+2;
          Seleccion.Tag := Figura.Tag;
          Seleccion.Name := 'Seleccion' +
IntToStr(Figura.Tag);
        end;

        Seleccion.Left := Figura.Left-1;
        Seleccion.Top := Figura.Top-1;
      end;
    end;
  end;
end;
```

Comprobamos que si es una figura creamos una selección para la misma también de color rojo. Aunque estos dos componentes podíamos haberlos metido en una sólo bloque de código, conviene dejarlos separados para futuras ampliaciones en cada uno de ellos.

Igualmente tenemos que contemplar también las figuras a la hora de quitar los seleccionados:

```
procedure TFInforme.QuitarSeleccionados;
var
  i: Integer;
begin
  if BandaActual = nil then
    Exit;

  // Le quitamos la selección a los componentes
  for i := 0 to BandaActual.ComponentCount-1 do
  begin
    if BandaActual.Components[i] is TQRLabel then
      (BandaActual.Components[i] as TQRLabel).Hint := '';

    if BandaActual.Components[i] is TQRShape then
      (BandaActual.Components[i] as TQRShape).Hint := '';
  end;

...
```

MOVIENDO LAS FIGURAS

Otra cosa que también tenemos hacer es cambiar el procedimiento encargado de arrastrar los componentes para que pueda hacer lo mismo con las figuras:

```
procedure TFInforme.MoverRatonPulsado;
var
  dx, dy, xAnterior, yAnterior: Integer;
  Seleccion: TShape;
begin
  // ¿Ha movido el ratón estándo el botón izquierdo
pulsado?
  if ( ( x <> Cursor.X ) or ( y <> Cursor.Y ) ) and (
Seleccionado <> nil ) then
  begin
    // ¿Estamos moviendo una etiqueta o una figura?
    if (Seleccionado is TQRLabel) or (Seleccionado is
TQRShape) then
    begin
...
```

Igualmente tenemos que ampliar el procedimiento encargado de seleccionar varios componentes:

```
procedure TFInforme.MoverOtrosSeleccionados(ID, x, y,
xAnterior, yAnterior: Integer);
var
  i, dx2, dy2: Integer;
  Etiqueta: TQRLabel;
  Figura: TQRShape;
  Seleccion: TShape;
begin
  for i := 0 to BandaActual.ComponentCount - 1 do
  begin
    // ¿Es una etiqueta?
    if (BandaActual.Components[i] is TQRLabel) then
    begin
      Etiqueta := BandaActual.Components[i] as TQRLabel;

      // ¿Es distinta a la etiqueta original y está
seleccionada?
      if (Etiqueta.Tag <> ID) and (Etiqueta.Hint <> '')
then
      begin
        dx2 := x - xAnterior;
        dy2 := y - yAnterior;
        Etiqueta.Left := Etiqueta.Left + dx2;
        Etiqueta.Top := Etiqueta.Top + dy2;
```

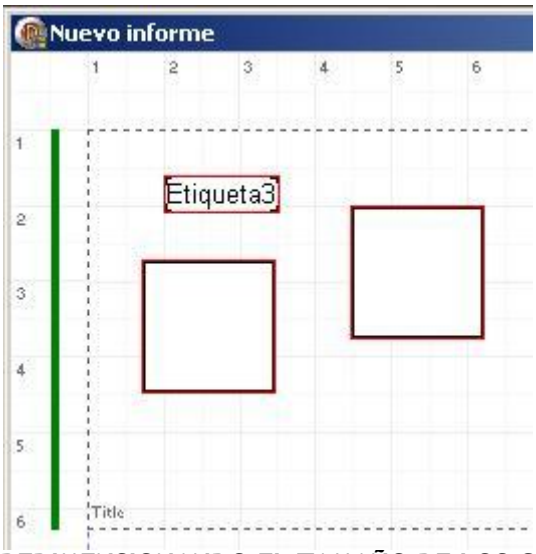
```
        // Movemos la selección
        Seleccion := BuscarSeleccion(Etiqueta.Tag);
        if Seleccion <> nil then
        begin
          Seleccion.Left := Etiqueta.Left-1;
          Seleccion.Top := Etiqueta.Top-1;
        end;
      end;
    end;

    // ¿Es una figura?
    if (BandaActual.Components[i] is TQRShape) then
    begin
      Figura := BandaActual.Components[i] as TQRShape;

      // ¿Es distinta a la Figura original y está
seleccionada?
      if (Figura.Tag <> ID) and (Figura.Hint <> '' ) then
      begin
        dx2 := x - xAnterior;
        dy2 := y - yAnterior;
        Figura.Left := Figura.Left + dx2;
        Figura.Top := Figura.Top + dy2;

        // Movemos la selección
        Seleccion := BuscarSeleccion(Figura.Tag);
        if Seleccion <> nil then
        begin
          Seleccion.Left := Figura.Left-1;
          Seleccion.Top := Figura.Top-1;
        end;
      end;
    end;
  end;
end;
```

Y como no, hay que modificar la selección de varios componentes con el rectángulo azul. Eso estaba en el procedimiento **ComprobarSeleccionComponetes**:

```
procedure TFInforme.ComprobarSeleccionComponentes;
var
  Sel: TShape; // Rectángulo azul de selección
  i: Integer;
  Eti: TQRLabel;
  Fig: TQRShape;
begin
  if BandaActual = nil then
    Exit;

  // ¿Hemos abierto una selección azul?
  Sel := BandaActual.FindComponent('Seleccionar') as
```

```
TShape;

  if Sel <> nil then
  begin
    // Recorremos todas las etiquetas de esta banda en
busca
    // las cuales están dentro de nuestro rectángulo de
selección
    for i := 0 to BandaActual.ComponentCount-1 do
    begin
      // ¿Es una etiqueta?
      if BandaActual.Components[i] is TQRLabel then
      begin
        Eti := BandaActual.Components[i] as TQRLabel;

        // ¿Está dentro de nuestro rectángulo azul de
selección?
        if ( Eti.Left >= Sel.Left ) and
          ( Eti.Top >= Sel.Top ) and
          ( Eti.Left+Eti.Width <= Sel.Left+Sel.Width ) and
          ( Eti.Top+Eti.Height <= Sel.Top+Sel.Height ) then
          Eti.Hint := 'X';
      end;

      // ¿Es una figura?
      if BandaActual.Components[i] is TQRShape then
      begin
        Fig := BandaActual.Components[i] as TQRShape;

        // ¿Está dentro de nuestro rectángulo azul de
selección?
        if ( Fig.Left >= Sel.Left ) and
          ( Fig.Top >= Sel.Top ) and
          ( Fig.Left+Fig.Width <= Sel.Left+Sel.Width ) and
          ( Fig.Top+Fig.Height <= Sel.Top+Sel.Height ) then
          Fig.Hint := 'X';
      end;
    end;

    DibujarSeleccionados;
    FreeAndNil(Sel); // Eliminamos la selección
  end;
end;
```

El mismo rollo he tenido que hacer en el procedimiento **MoverComponentes** (el que movía los componentes con el teclado). De este modo podemos seleccionar y arrastrar a la vez tanto etiquetas como figuras:



REDIMENSIONANDO EL TAMAÑO DE LOS COMPONENTES

Ya tenemos hecho que si el usuario tiene seleccionado uno o más componentes y pulsa los cursores entonces mueve los componentes. Ahora vamos a hacer que si pulsa también la tecla SHIFT entonces lo que hace es redimensionarlos. Esto los controlamos en el evento **FormKeyDown**:

```
procedure TFInforme.FormKeyDown(Sender: TObject; var Key:
Word;
 Shift: TShiftState);
begin
  // ¿Está pulsada la tecla SHIFT?
  if ssShift in Shift then
  begin
    // Redimensionamos el tamaño de los componentes
    case key of
      VK_RIGHT: AmpliarComponentes( 1, 0 );
      VK_LEFT: AmpliarComponentes( -1, 0 );
      VK_UP: AmpliarComponentes( 0, -1 );
      VK_DOWN: AmpliarComponentes( 0, 1 );
    end;
  end
  else
  begin
    // Los movemos
    case key of
      VK_RIGHT: MoverComponentes( 1, 0 );
      VK_LEFT: MoverComponentes( -1, 0 );
      VK_UP: MoverComponentes( 0, -1 );
      VK_DOWN: MoverComponentes( 0, 1 );
    end;
  end;
end;
```

El procedimiento **AmpliarComponentes** es similar al de moverlos:

```
procedure TFInforme.AmpliarComponentes(x, y: Integer);
```

```
var
  i: Integer;
  Etiqueta: TQRLabel;
  Figura: TQRShape;
  Seleccion: TShape;
begin
  if BandaActual = nil then
    Exit;

  // Recorremos todos los componentes
  for i := 0 to BandaActual.ComponentCount-1 do
  begin
    // ¿Es una etiqueta?
    if BandaActual.Components[i] is TQRLabel then
    begin
      Etiqueta := BandaActual.Components[i] as TQRLabel;

      // ¿Está seleccionada?
      if Etiqueta.Hint <> '' then
      begin
        // Movemos la etiqueta
        Etiqueta.Width := Etiqueta.Width + x;
        Etiqueta.Height := Etiqueta.Height + y;

        // Movemos su selección
        Seleccion :=
BandaActual.FindComponent('Seleccion'+IntToStr(Etiqueta.Tag
)) as TShape;
        Seleccion.Width := Etiqueta.Width+2;
        Seleccion.Height := Etiqueta.Height+2;
      end;
    end;

    // ¿Es una figura?
    if BandaActual.Components[i] is TQRShape then
    begin
      Figura := BandaActual.Components[i] as TQRShape;

      // ¿Está seleccionada?
      if Figura.Hint <> '' then
      begin
        // Movemos la Figura
        Figura.Width := Figura.Width + x;
        Figura.Height := Figura.Height + y;

        // Movemos su selección
        Seleccion :=
BandaActual.FindComponent('Seleccion'+IntToStr(Figura.Tag))
as TShape;
        Seleccion.Width := Figura.Width+2;
        Seleccion.Height := Figura.Height+2;
```

```
      end;
    end;
  end;
end;
```

También se podría haber hecho con el ratón cogiendo las esquinas del componente y estirándolas, pero me llevaría un artículo sólo para eso (y la vaca no da para más).

Ya veis la movida que hay que hacer para hacer un simple editor de informes. En el próximo artículo vamos a añadir el componente **TQRDBText** para conectarnos con bases de datos Internase/Firebird.

DESCARGA DEL PROYECTO

Aquí van los enlaces para bajarse todo el proyecto en RapidShare y Megaupload:

http://rapidshare.com/files/208654755/EditorInformes3_DelphiAlLimite.rar.html

http://www.megaupload.com/?d=3BPPEI7H

Pruebas realizadas en RAD Studio 2007.

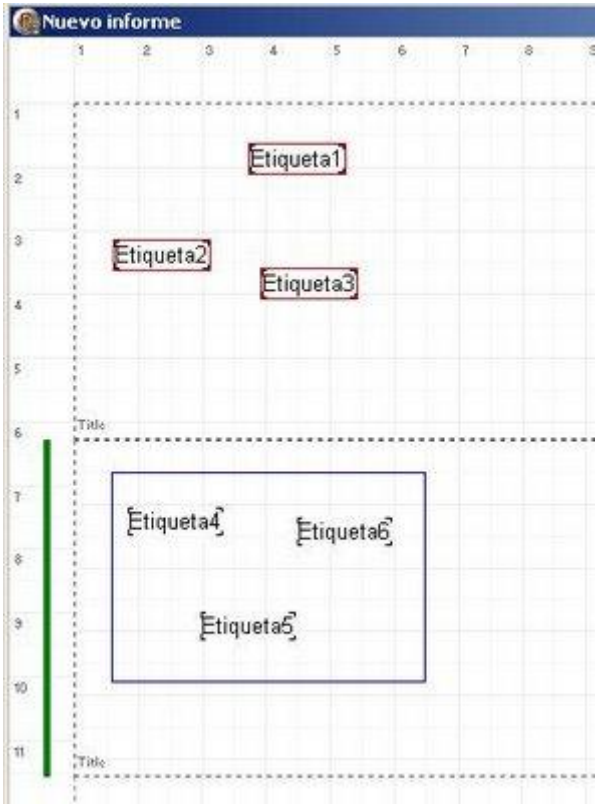
Publicado por Administrador en 09:32 2 comentarios
Etiquetas: impresión

27 febrero 2009

Crea tu propio editor de informes (II)

En el artículo de hoy vamos a conseguir seleccionar varios componentes a la vez pulsando la tecla control así como poder seleccionarlos con un rectángulo. También vamos a crear una selección lateral para saber con que banda estamos trabajando.

Una vez seleccionados los componentes vamos a dar la posibilidad de arrastrarlos todos a la vez utilizando los cursores del teclado:



Pero antes de nada, vamos a comenzar a pegar hachazos en el código fuente para dejarlo todo bien organizado. Al igual que en el artículo anterior daré al final del mismo los enlaces a RapidShare y Megaupload para que no tengáis que picar código, ya que veréis como se va complicando el asunto.

MODIFICANDO EL NUCLEO DE LA APLICACIÓN

Vamos a comenzar modificando el evento **OnTimer** de nuestro **Temporizador** simplificando el código fuente de este modo:

```
procedure TFInforme.TemporizadorTimer(Sender: TObject);
begin
  GetCursorPos( Cursor );

  // ¿Está pulsado el botón izquierdo del ratón?
  if HiWord( GetAsyncKeyState( VK_LBUTTON ) ) <> 0 then
  begin
    if Seleccionado = nil then
      Seleccionado := FindControl( WindowFromPoint( Cursor
) );

    // ¿No estaba pulsado el botón izquierdo del ratón
anteriormente?
    if not bPulsado then
      SeleccionarComponente
    else
      MoverRatonPulsado;
  end
  else
  begin
```

```
    Seleccionado := nil;
    bPulsado := False;
    ComprobarSeleccionComponentes;
  end;
end;
```

Primero comprobamos si el usuario ha pulsado el botón izquierdo del ratón. En ese caso guardamos en la variable**Seleccionado** el componente donde ha pinchado y llamamos al procedimiento **SeleccionarComponente** que se encarga de comprobar que componente hemos pulsado (banda o etiqueta).

Si el ratón ya estaba pulsado de antes significa que vamos a arrastrar un componente o que vamos a abrir una selección. Esto se hace con el procedimiento **MoverRatonPulsado**.

En el caso de que el usuario suelte el botón izquierdo del ratón entonces anulamos tanto la selección individual de componentes como la colectiva mediante el procedimiento **ComprobarSeleccionComponentes**.

Veamos la implementación de cada procedimiento.

SELECCIONANDO COMPONENTES

El procedimiento **SeleccionarComponente** comprobará si hemos pulsado una etiqueta o la banda para efectuar una acción u otra:

```
procedure TFInforme.SeleccionarComponente;
var
  Punto: TPoint;
begin
  bPulsado := True;
  x := Cursor.X;
  y := Cursor.Y;

  // ¿Ha pinchado sobre una etiqueta?
  if Seleccionado is TQRLabel then
  begin
    xInicial := Seleccionado.Left;
    yInicial := Seleccionado.Top;

    if Seleccionado.Hint = '' then
    begin
      // ¿No esta pulsado la tacla control?
      if HiWord(GetAsyncKeyState(VK_LCONTROL)) = 0 then
        QuitarSeleccionados;

      Seleccionado.Hint := 'X';
    end;
  end;

  // ¿Ha pinchado sobre una banda?
  if Seleccionado is TQRBand then
  begin
    BandaActual := Seleccionado as TQRBand;
```

```
    SeleccionarBandaActual;

    // Transformamos las coordenadas del cursor del
formulario a la banda
    Punto.X := x;
    Punto.Y := y;
    Punto := ScreenToClient( Punto );
    xInicial := Punto.X-BandaActual.Left;
    yInicial := Punto.Y-BandaActual.Top;
    QuitarSeleccionados;
  end;

  DibujarSeleccionados;
end;
```

Primero lee las coordenadas del ratón. Si hemos pinchado sobre una etiqueta entonces la seleccionamos y deseleccionamos las demás, pero aquí hemos añadido una novedad. Si el usuario mantiene pulsado la tecla CONTROL entonces seguimos seleccionando sin quitar la selección a las otras etiquetas.

En el caso de que pulsemos sobre una banda significa que vamos a deseleccionar todas las etiquetas y abrir un rectángulo de selección. Por eso guardo en las variables **xInicial** e **yInicial** donde comienza la selección. Aquí he tenido que convertir las coordenadas del cursor del ratón a coordenadas del formulario para que no se desplacen las coordenadas mediante la función **ScreenToClient**.

El procedimiento **DibujarSeleccionados** no ha cambiado.

SELECCIONANDO LA BANDA SOBRE LA QUE ESTAMOS

El procedimiento **SeleccionarBandaActual** crea un rectángulo a la izquierda de la banda actual de color verde:

```
procedure TFInforme.SeleccionarBandaActual;
var
  SeleccionBanda: TShape;
begin
  // Si ya había seleccionada una banda la eliminamos (la
selección)
  SeleccionBanda := Informe.FindComponent('SeleccionBanda')
as TShape;
  if SeleccionBanda <> nil then
    FreeAndNil(SeleccionBanda);

  // Seleccionamos de nuevo la banda
  SeleccionBanda := TShape.Create(Informe);
  with SeleccionBanda do
  begin
    Name := 'SeleccionBanda';
    Parent := Informe;
    Left := 20;
    Top := BandaActual.Top;
    Width := 4;
```

```
    Pen.Width := 2;
    Pen.Color := clGreen;
    Height := BandaActual.Height;
  end;
end;
```

A este procedimiento lo vamos a llamar también al crear una banda:

```
procedure TFInforme.BandaClick(Sender: TObject);
begin
  BandaActual := TQRBand.Create(Informe);

  with BandaActual do
  begin
    Parent := Informe;
    Height := 200;
  end;

  SeleccionarBandaActual;
end;
```

Cuando creamos una nueva banda la damos por la banda de trabajo actual.

ARRASTRANDO UN COMPONENTE O LA SELECCIÓN

El procedimiento **MoverRatonPulsado** se ejecutará cuando dejamos pulsado el botón izquierdo del ratón sobre un componente o si estamos abriendo un rectángulo de selección:

```
procedure TFInforme.MoverRatonPulsado;
var
  dx, dy: Integer;
  Seleccion: TShape;
begin
  // ¿Ha movido el ratón estándo el botón izquierdo
pulsado?
  if ( ( x <> Cursor.X ) or ( y <> Cursor.Y ) ) and (
Seleccionado <> nil ) then
  begin
    // ¿Estamos moviendo una etiqueta?
    if Seleccionado is TQRLabel then
    begin
      // Movemos el componente seleccionado
      dx := Cursor.X - x;
      dy := Cursor.Y - y;
      Seleccionado.Left := xInicial + dx;
      Seleccionado.Top := yInicial + dy;

      // Movemos la selección
      Seleccion := BuscarSeleccion(Seleccionado.Tag);
      if Seleccion <> nil then
      begin
```

```
        Seleccion.Left := xInicial+dx-1;
        Seleccion.Top := yInicial+dy-1;
      end;
    end;

    // ¿Estamos sobre una banda?
    if Seleccionado is TQRBand then
    begin
      // si no existe un rectángulo de selección lo creamos
      Seleccion := BandaActual.FindComponent('Seleccionar')
as TShape;

      if Seleccion = nil then
      begin
        Seleccion := TShape.Create(BandaActual);
        Seleccion.Parent := BandaActual;
        Seleccion.Pen.Color := clBlue;
        Seleccion.Name := 'Seleccionar';
      end;

      dx := Cursor.X - x;
      dy := Cursor.Y - y;
      Seleccion.Left := xInicial;
      Seleccion.Top := yInicial;
      Seleccion.Width := dx;
      Seleccion.Height := dy;
      Seleccion.Brush.Style := bsClear;
    end;
  end;
end;
```

Si lo que arrastramos es una etiqueta, el código fuente no cambia mucho respecto al artículo anterior. Pero si lo hacemos sobre el fondo de una banda entonces comprobamos si existe un rectángulo de selección. Si no lo hay abrimos uno mediante un componente **TShape** de color azul.

La selección tendrá lugar cuando el usuario suelte el botón izquierdo del ratón.

SELECCIONANDO UNO O MAS COMPONENTES

El procedimiento **ComprobarSeleccionComponentes** se ejecuta inmediatamente después de soltar el botón izquierdo del ratón:

```
procedure TFInforme.ComprobarSeleccionComponentes;
var
  Sel: TShape; // Rectángulo azul de selección
  i: Integer;
  Eti: TQRLabel;
begin
  if BandaActual = nil then
    Exit;
```

```
  // ¿Hemos abierto una selección azul?
  Sel := BandaActual.FindComponent('Seleccionar') as
TShape;

  if Sel <> nil then
  begin
    // Recorremos todas las etiquetas de esta banda en
busca
    // las cuales están dentro de nuestro rectángulo de
selección
    for i := 0 to BandaActual.ComponentCount-1 do
      // ¿Es una etiqueta?
      if BandaActual.Components[i] is TQRLabel then
      begin
        Eti := BandaActual.Components[i] as TQRLabel;

        // ¿Está dentro de nuestro rectángulo azul de
selección?
        if ( Eti.Left >= Sel.Left ) and
          ( Eti.Top >= Sel.Top ) and
          ( Eti.Left+Eti.Width <= Sel.Left+Sel.Width ) and
          ( Eti.Top+Eti.Height <= Sel.Top+Sel.Height ) then
          Eti.Hint := 'X';
      end;

    DibujarSeleccionados;
    FreeAndNil(Sel); // Eliminamos la selección
  end;
end;
```

Comprueba si hay un rectángulo azul llamado **Seleccionar** y en ese caso recorre todas las etiquetas de la banda y comprueba si alguna está dentro del rectángulo de selección. Si es así, le metemos una **X** en el **Hint**. Luego llamamos al procedimiento **DibujarSeleccionados** para que se encargue del resto y eliminamos la selección.

MOVIENDO LOS COMPONENTES CON LOS CURSORES

Para poder mover los componentes seleccionados con el teclado podemos aprovechar el evento **OnKeyDown** del formulario:

```
procedure TFInforme.FormKeyDown(Sender: TObject; var Key:
Word;
  Shift: TShiftState);
begin
  case key of
    VK_RIGHT: MoverComponentes( 1, 0 );
    VK_LEFT: MoverComponentes( -1, 0 );
    VK_UP: MoverComponentes( 0, -1 );
    VK_DOWN: MoverComponentes( 0, 1 );
  end;
end;
```

El procedimiento **MoverComponentes** desplaza todos las etiquetas seleccionadas horizontal o verticalmente según los parámetros:

```
procedure TFInforme.MoverComponentes(x, y: Integer);
var
  i: Integer;
  Etiqueta: TQRLabel;
  Seleccion: TShape;
begin
  if BandaActual = nil then
    Exit;

  // Recorremos todos los componentes
  for i := 0 to BandaActual.ComponentCount-1 do
    // ¿Es una etiqueta?
    if BandaActual.Components[i] is TQRLabel then
    begin
      Etiqueta := BandaActual.Components[i] as TQRLabel;

      // ¿Está seleccionada?
      if Etiqueta.Hint <> '' then
      begin
        // Movemos la etiqueta
        Etiqueta.Left := Etiqueta.Left + x;
        Etiqueta.Top := Etiqueta.Top + y;

        // Movemos su selección
        Seleccion :=
BandaActual.FindComponent('Seleccion'+IntToStr(Etiqueta.Tag
)) as TShape;
        Seleccion.Left := Etiqueta.Left-1;
        Seleccion.Top := Etiqueta.Top-1;
      end;
    end;
end;
```

Aparte de mover la etiqueta también tenemos que arrastrar detrás su selección (**TShape**).

MODIFICANDO LA DESELECCION DE COMPONENTES

Tenemos que modificar también la deselección de componentes ya que si no también eliminaría la selección verde de la banda:

```
procedure TFInforme.QuitarSeleccionados;
var
  i: Integer;
begin
  if BandaActual = nil then
    Exit;
```

```
  // Quitamos las etiquetas seleccionadas
  for i := 0 to BandaActual.ComponentCount-1 do
    if BandaActual.Components[i] is TQRLabel then
      (BandaActual.Components[i] as TQRLabel).Hint := '';

  // Eliminamos las selecciones
  for i := BandaActual.ComponentCount-1 downto 0 do
    if BandaActual.Components[i] is TShape then
      if BandaActual.Components[i].Name <> 'SeleccionBanda'
then
        BandaActual.Components[i].Free;
end;
```

Lo hacemos sólo para ese caso en especial.

EL PROYECTO EN INTERNET

Aquí va el proyecto comprimido con rar y subido a RapidShare y Megaupload:

http://rapidshare.com/files/203119248/EditorInformes2_DelphiAlLimite.rar.html

http://www.megaupload.com/?d=2FCA8P9C

CONTINUARA

Como podéis ver con estos ejemplos, crear un editor de informes no es cosa fácil (hacer algo como Microsoft Word tiene que doler). Aunque no me meteré a fondo con todo lo que engloba un editor de informes, todavía nos quedan muchas características básicas que implementar. Eso será en el próximo artículo.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 10:05 0 comentarios
Etiquetas: impresión

13 febrero 2009

Crea tu propio editor de informes (I)

Después de haber visto todas las características de QuickReport a uno se le queda un sabor agridulce por dos razones: es uno de los editores de informes más completos que hay pero no permite exportar plantillas de informes, es decir, por cada informe hay que hacer un formulario.

Eso sin contar que los clientes de nuestros programas no pueden personalizar los informes a su gusto. Todo tiene que pasar por nosotros mediante Delphi.

Para solucionar esto podemos crear un pequeño editor de informes que permita a cualquier usuario diseñar o modificar cualquier informe aunque internamente funcione con QuickReport.

Veamos todos los pasos que se necesitan para crear un editor.

CREANDO UN NUEVO PROYECTO

Para iniciar el proyecto vamos a hacer lo siguiente:

1º Creamos un nuevo proyecto y guardamos el formulario principal con el nombre **FPrincipal**. Luego guardamos este formulario con el nombre **UPrincipal.pas** y el proyecto con el nombre **EditorInformes.dproj**.

2º Como nos interesa hacer una aplicación que permita manejar varios informes a la vez, vamos a modificar la propiedad **FormStyle** del formulario con el valor **fsMDIForm**.

Para los que no lo sepan, una aplicación MDI se compone de una ventana padre (**fsMDIForm**) y varias ventanas hijas (**fsMDIChild**). Por ejemplo, los programas de Microsoft Office (Word, Excel y Access) son aplicaciones MDI.

3º También vamos a hacer que cuando se ejecute el programa aparezca la ventana maximizada. Eso se hace poniendo el valor **wsMaximized** en la propiedad **WindowState** del formulario.

Una vez tenemos el formulario principal, vamos a ir añadiendo las opciones generales del programa.

AÑADIENDO EL MENU DE OPCIONES

Insertamos en el formulario el componente de la clase **TMainMenu** (que se encuentra en la sección **Standard**) y lo vamos a llamar **MenuPrincipal**:

Hacemos doble clic sobre dicho componente y añadimos el siguiente menú **Archivo**:

La opción de salir ya la podemos implementar cerrando el formulario:

```
procedure TFPrincipal.SalirClick(Sender: TObject);
begin
 Close;
end;
```

CREANDO UN NUEVO INFORME

Vamos a crear un nuevo formulario llamado **FInforme** que va a ser el alma del proyecto. Es el que vamos a utilizar para diseñar el informe. Guardamos el informe con el

nombre **UInforme.pas**.

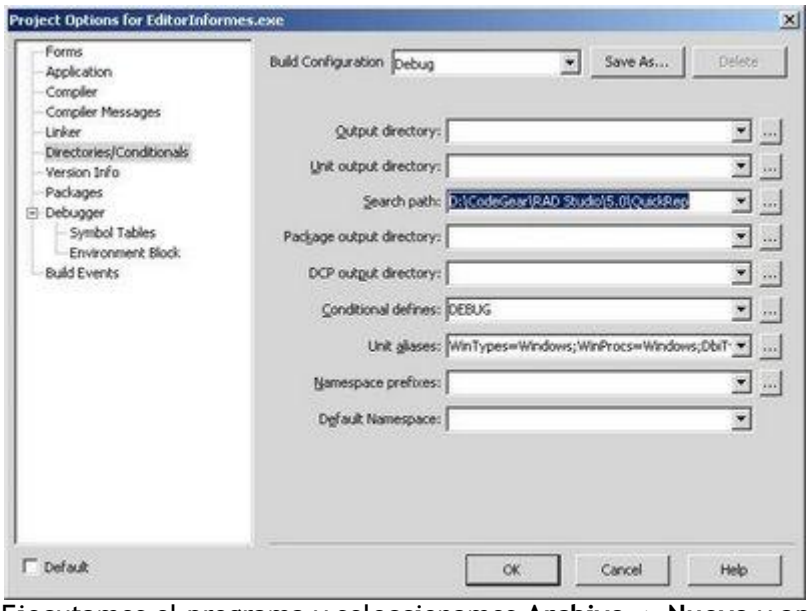
En la propiedad **Caption** del formulario le ponemos **Nuevo informe**. También tenemos que poner su propiedad**FormStyle** a **fsMDIChild**, para indicarle que es un formulario hijo de **FPrincipal**.

Ahora añadimos al formulario el componente **TQuickRep** que se encuentra en la sección de componentes **QReport**y lo colocamos en la esquina superior izquierda del formulario (Left=0, Top=0):

Al componente **TQuickRep** lo vamos a llamar **Informe**. Ahora volvemos al formulario principal y vamos a implementar la opción **Archivo** -> **Nuevo** para crear nuestro informe:

```
procedure TFPrincipal.NuevoClick(Sender: TObject);
begin
 Application.CreateForm(TFInforme,FInforme);
 FInforme.Show;
end;
```

Al compilar el proyecto os dará un error quejándose de que no encuentra la unidad **QuickRpt**. Tenéis que añadir la ruta de búsqueda del proyecto con el directorio donde se encuentra instalado QuickReport:



Ejecutamos el programa y seleccionamos **Archivo** -> **Nuevo** y aparecerá esta ventana

hija:

Si intentáis cerrar el formulario hijo veréis que no se cierra, se minimiza. Esto es normal en las aplicaciones MDI. Para solucionar esto tenemos que forzar a que se libere el formulario utilizando el evento **OnClose** del formulario**FInforme**:

```
procedure TFInforme.FormClose(Sender: TObject; var Action:
TCloseAction);
begin
 Action := caFree;
end;
```

Ahora pasemos a insertar componentes visuales en nuestro informe.

AÑADIENDO COMPONENTES AL INFORME

Vamos a hacer que el usuario pueda añadir componentes utilizando un menú contextual. Para ello añadimos un componente de la clase **TPopupMenu** que se encuentra en la pestaña **Standard**:

A este componente lo vamos a llamar **MenuInforme** y va tener estas opciones:

Como el componente **TQuickRep** no es un componente visual para manejar en tiempo de ejecución, no podemos vincular nuestro menú popup directamente a este componente

sino que hay que hacerlo con el formulario**FInforme**.

Para arrastrar los componentes tampoco podemos utilizar las propiedades de Drag and Drop que suelen llevar los componentes visuales de Delphi, ya que ningún componente de QuickReport tiene estas propiedades.

Vamos a necesitar estas variables en la sección **private** de nuestro formulario:

```
private
 { Private declarations }
 bPulsado: Boolean;
 Seleccionado: TControl;
 x, y, xInicial, yInicial, UltimoID: Integer;
 Cursor: TPoint;
```

Veamos para que sirve cada variable:

bPulsado: Nos va a indicar cuando el usuario ha dejado pulsado el botón izquierdo del ratón.

Seleccionado: Esta variable va a puntar al control que tenemos seleccionado (por ejemplo TQRLabel).

x, y: Utilizamos estas variables para guardar la posición actual del puntero del ratón.

xInicial, yInicial: Cuando arrastramos un componente utilizamos estas variables para conservar la posición inicial del mismo.

UltimoID: Con cada componente que creamos le asignamos un identificador (un número único) que nos va a servir para asignar un rectángulo de selección.

Cursor: Lo utilizamos para leer las coordenadas del ratón.

También tenemos que añadir la unidad **QRCtrls** en el apartado uses de nuestra unidad.

CREANDO UNA NUEVA BANDA

Ya sabemos que para que se muestre algo en QuickReport lo primero que tenemos que hacer es crear una banda. Este es el código fuente que vamos a introducir en la opción **Nuevo** -> **Banda**:

```
procedure TFInforme.BandaClick(Sender: TObject);
begin
 BandaActual := TQRBand.Create(Informe);

 with BandaActual do
 begin
   Parent := Informe;
   Height := 200;
 end;
end;
```

Aparte de crear el objeto le hemos asignado como padre el componente **TQuickRep** y le hemos dado un tamaño vertical de 200. Aprovechamos para guardar en la variable **BandaActual** el objeto **TQRBand** con el que estamos trabajando.

Ahora pasemos a la creación de una etiqueta.

CREANDO UNA ETIQUETA

Aquí va el código para la opción **Nuevo** -> **Etiqueta**:

```
procedure TFInforme.EtiquetaClick(Sender: TObject);
begin
 if BandaActual = nil then
 begin
   Application.MessageBox( 'Debe crear una banda.',
     'Acceso denegado', MB_ICONSTOP );
   Exit;
 end;

 QuitarSeleccionados;
 with TQRLabel.Create(BandaActual) do
 begin
   Parent := BandaActual;
   Left := 10;
   Top := 10;
   Inc(UltimoID);
   Tag := UltimoID;
   Caption := 'Etiqueta' + IntToStr(UltimoID);
   Name := Caption;
 end;
end;
```

Lo primero que hacemos es comprobar si primero hemos creado una banda. Si es así, entonces llamamos a otro procedimiento llamado **QuitarSeleccionados**. Eso lo veremos más adelante.

Después creo un componente de la clase **TQRLabel**, le asigno el padre (**TQuickRep**) y le doy un tamaño predeterminado. También lo coloco en la esquina superior izquierda de la banda actual. Le asigno un nuevo entero identificador que vamos a utilizar posteriormente para poder seleccionarlo y por último le doy el nombre **Etiqueta** +**UltimoID**.

LA SELECCION DE COMPONENTES

Al igual que el editor de formularios de Delphi vamos a poder seleccionar componentes si hacemos clic sobre los mismos. Esta selección la vamos a hacer utilizando componentes de la clase **TShape** con la ventaja de que los podemos ver en el editor pero no se imprimen cuando se lanza el informe a la impresora o se hace la vista previa.

Como el usuario va a poder seleccionar uno o más componentes, lo que hago es asociar cada etiqueta **TQRLabel** a cada componente **TShape** mediante un identificador (**UltimoID**) y lo guardamos en la propiedad **Tag** de cada componente.

Por ejemplo, si creo tres etiquetas, este sería su **Tag**:

```
TQRLabel            TShape          Tag
---------           ----------      ---
```

```
Etiqueta1          Seleccion1     1
Etiqueta2          Seleccion2     2
Etiqueta3          Seleccion3     3
```

Este sería el procedimiento para eliminar todos los seleccionados:

```
procedure TFInforme.QuitarSeleccionados;
var
 i: Integer;
begin
 if BandaActual = nil then
   Exit;

 // Quitamos las etiquetas seleccionadas
 for i := 0 to BandaActual.ComponentCount-1 do
   if BandaActual.Components[i] is TQRLabel then
     (BandaActual.Components[i] as TQRLabel).Hint := '';

 // Eliminamos las selecciones
 for i := BandaActual.ComponentCount-1 downto 0 do
   if BandaActual.Components[i] is TShape then
     BandaActual.Components[i].Free;
end;
```

Quitar los seleccionados implica eliminar todos los componentes **TShape** y decirle a cada etiqueta **TQRLabel** que no está seleccionada. ¿Cómo sabemos si una etiqueta está seleccionada o no? Pues como el **Tag** ya lo tenemos ocupado con su identificador, lo que hacemos es utilizar su propiedad **Hint**. Si el **Hint** tiene una **X** lo damos por seleccionado, en caso contrario estará vacío.

MOVIENDO LOS COMPONENTES POR LA PANTALLA

Esto es lo que más me ha costado. ¿Cómo mover componentes que no soportan la propiedad Drag and Drop?. ¿Cómo averiguar a que componente hemos pinchado en pantalla?

Gracias a Internet y un poco de paciencia pude encontrar piezas de código que me solucionaron todos estos problemas. Vayamos por partes. Como no podemos aprovechar los eventos MouseDown, MouseMove, etc. del formulario (por que tenemos el objeto **TQuickRep** encima que no tiene estos eventos) entonces opté por utilizar un temporizador (**TTimer**) para controlar el todo momento las acciones del usuario:

A este componente lo he llamado **Temporizador**. Le he puesto su propiedad **Interval** a 1 milisegundo y este sería su evento **OnTimer**:

```
procedure TFInforme.TemporizadorTimer(Sender: TObject);
var
 dx, dy: Integer;
```

```
 Seleccion: TShape;
begin
 GetCursorPos( Cursor );

 // ¿Está pulsado el botón izquierdo del ratón?
 if HiWord( GetAsyncKeyState( VK_LBUTTON ) ) <> 0 then
 begin
   if Seleccionado = nil then
     Seleccionado := FindControl( WindowFromPoint( Cursor )
);

   // ¿No estaba pulsado el botón izquierdo del ratón
anteriormente?
   if not bPulsado then
   begin
     bPulsado := True;
     x := Cursor.X;
     y := Cursor.Y;

     // ¿Ha seleccionado una etiqueta?
     if Seleccionado is TQRLabel then
     begin
       xInicial := Seleccionado.Left;
       yInicial := Seleccionado.Top;

       if Seleccionado.Hint = '' then
       begin
         QuitarSeleccionados;
         Seleccionado.Hint := 'X';
       end;
     end;

     if Seleccionado is TQRBand then
       QuitarSeleccionados;

     DibujarSeleccionados;
   end
   else
   begin
     // ¿Ha movido el ratón estándo el botón izquierdo
pulsado?
     if ( ( x <> Cursor.X ) or ( y <> Cursor.Y ) ) and (
Seleccionado <> nil ) then
       if Seleccionado is TQRLabel then
       begin
         // Movemos el componente seleccionado
         dx := Cursor.X - x;
         dy := Cursor.Y - y;
         Seleccionado.Left := xInicial + dx;
         Seleccionado.Top := yInicial + dy;
```

```
          // Movemos la selección
          Seleccion := BuscarSeleccion(Seleccionado.Tag);
          if Seleccion <> nil then
          begin
            Seleccion.Left := xInicial+dx-1;
            Seleccion.Top := yInicial+dy-1;
          end;
        end;
    end;
 end
 else
 begin
    bPulsado := False;
    Seleccionado := nil;
 end;
end;
```

¿Vaya ladrillo, no? No es tan difícil como parece. Vamos a ver este procedimiento por partes:

1º Leemos la posición del cursor en pantalla:

```
GetCursorPos( Cursor );
```

2º Para averiguar si está pulsado el botón izquierdo del ratón utilizo esta función:

```
if HiWord( GetAsyncKeyState( VK_LBUTTON ) ) <> 0 then
```

GetAsyncKeyState es una función asíncrona de la API de Windows que nos devuelve el estado de una tecla o el botón del ratón.

3º Averiguamos sobre que componente esta el puntero del ratón con esta función:

```
if Seleccionado = nil then
 Seleccionado := FindControl( WindowFromPoint( Cursor ) );
```

4º Si no estaba pulsado el botón izquierdo del ratón entonces compruebo si lo que ha pulsado es una etiqueta o la banda. Si es la etiqueta guardamos su posición original (por si no da por moverla) y la selecciono guardando una **X**en su propieda **Hint**:

```
if Seleccionado is TQRLabel then
begin
 xInicial := Seleccionado.Left;
 yInicial := Seleccionado.Top;

 if Seleccionado.Hint = '' then
 begin
    QuitarSeleccionados;
    Seleccionado.Hint := 'X';
 end;
end;
```

5º Si lo que hemos pulsado es una banda deseleccionamos todos las etiquetas:

```
if Seleccionado is TQRBand then
 QuitarSeleccionados;
```

6º Llamamos al procedimiento encargado de dibujar las etiquetas seleccionadas:

```
DibujarSeleccionados;
```

La implementación de este procedimiento la veremos más adelante.

7º La otra parte del procedimiento se ejecuta si mantenemos pulsando el botón izquierdo del ratón y arrastramos el componente. Pero por ahora sólo dejamos arrastrar una etiqueta:

```
// ¿Ha movido el ratón estándo el botón izquierdo pulsado?
if ( ( x <> Cursor.X ) or ( y <> Cursor.Y ) ) and (
Seleccionado <> nil ) then
 if Seleccionado is TQRLabel then
 begin
   // Movemos el componente seleccionado
   dx := Cursor.X - x;
   dy := Cursor.Y - y;
   Seleccionado.Left := xInicial + dx;
   Seleccionado.Top := yInicial + dy;

   // Movemos la selección
   Seleccion := BuscarSeleccion(Seleccionado.Tag);
   if Seleccion <> nil then
   begin
     Seleccion.Left := xInicial+dx-1;
     Seleccion.Top := yInicial+dy-1;
   end;
 end;
```

Para moverla calculo la diferencia entre la posición original y le sumo el desplazamiento del ratón (variales **dx** y **dy**). Para evitar que una etiqueta tenga varias selecciones he creado la fución **BuscarSeleccion** a la cual le pasamos el ID de la etiqueta y nos devuelve el objeto **TShape** para poder moverlo. Si no fuera así, cada vez que movemos una etiqueta se queda su selección abandonada.

Este sería el procedimiento de buscar una selección:

```
function TFInforme.BuscarSeleccion( ID: Integer ): TShape;
var
 i: Integer;
begin
 Result := nil;

 if BandaActual = nil then
   Exit;

 for i := 0 to BandaActual.ComponentCount-1 do
   if BandaActual.Components[i] is TShape then
```

```
      if BandaActual.Components[i].Tag = ID then
      begin
        Result := BandaActual.Components[i] as TShape;
        Exit;
      end;
end;
```

CREANDO LA SELECCIÓN PARA CADA ETIQUETA

El procedimiento **DibujarSeleccionados** comprueba primero si una etiqueta ya tiene selección (por su ID) y si no es así la creamos:

```
procedure TFInforme.DibujarSeleccionados;
var
 i: Integer;
 Seleccion: TShape;
 Etiqueta: TQRLabel;
begin
 if BandaActual = nil then
   Exit;

 for i := 0 to BandaActual.ComponentCount-1 do
   if BandaActual.Components[i] is TQRLabel then
   begin
     Etiqueta := BandaActual.Components[i] as TQRLabel;

     if Etiqueta.Hint <> '' then
     begin
       // Antes de crearla comprobamos si ya tiene
selección
       Seleccion := BuscarSeleccion(Etiqueta.Tag);

       if Seleccion = nil then
       begin
         Seleccion := TShape.Create(BandaActual);
         Seleccion.Parent := BandaActual;
         Seleccion.Pen.Color := clRed;
         Seleccion.Width := Etiqueta.Width+2;
         Seleccion.Height := Etiqueta.Height+2;
         Seleccion.Tag := Etiqueta.Tag;
         Seleccion.Name := 'Seleccion' +
IntToStr(Etiqueta.Tag);
       end;

       Seleccion.Left := Etiqueta.Left-1;
       Seleccion.Top := Etiqueta.Top-1;
     end;
   end;
end;
```

Para que un componente tenga selección tienen que cumplirse dos condiciones:

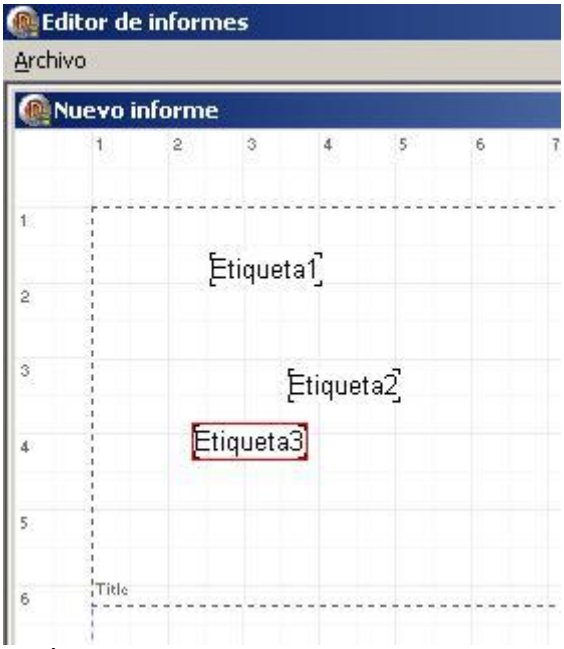
1º Que sea una etiqueta.
2º Que su propiedad Hint = 'X'

Si la selección (**TShape**) ya estaba creada de antes sólo tenemos que actualizar su posición.

Por último implementamos la opción de vista previa:

```
procedure TFInforme.VistaPreviaClick(Sender: TObject);
begin
 Informe.Preview;
end;
```

Y aquí tenemos el resultado:



Aquí os dejo el proyecto comprimido con RAR y subido a RapidShape y Megaupload:

http://rapidshare.com/files/197571065/EditorInformes_DelphiAlLimite.rar.html

http://www.megaupload.com/?d=UO2IU2MV

En la siguiente parte de este artículo seguiremos mejorando nuestro editor.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 12:40 4 comentarios
Etiquetas: impresión

30 enero 2009

Creación de informes con QuickReport (y V)

En artículos anteriores vimos que mediante etiquetas podemos llevar al informe cualquier tipo de dato (fecha, hora, totales, etc.) generado por nuestro código fuente, pero hay funciones que lleva QuickReport que evitan tener que hacer muchas de estas cosas a mano.

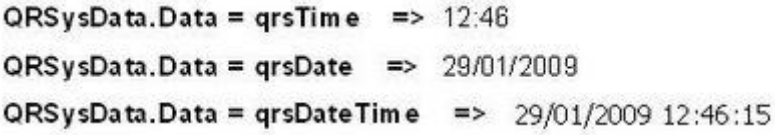
Esto se hace utilizando campos del tipo **TQRExpr** y **TQRSysData**. Veamos los tipos de funciones.

FUNCIONES DE FECHA Y HORA

Las funciones de fecha y hora que tenemos con **TQRExpr** son **TIME** y **DATE**:

Estas funciones hay que introducirlas en la propiedad **Expression**.

Si utilizamos el campo **TQRSysData** podemos obtener la fecha y la hora también:



En este tipo de campos seleccionamos los valores predeterminados mediante su propiedad **Data**.

FUNCIONES MATEMATICAS

Tenemos estas cuatro funciones matemáticas:

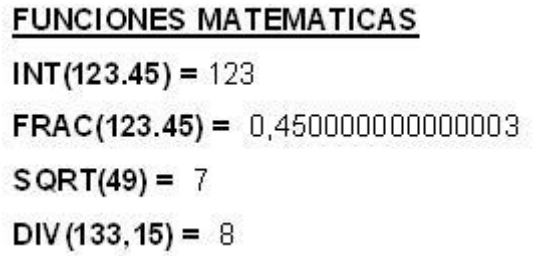
INT(x) -> Devuelve la parte entera de un número real x

FRAC(x) -> Devuelve la parte fraccionaria de un número real x

SQRT(x) -> Calcula la raíz cuadrada de x

DIV(x, y) -> Realiza la división entera de x entre y.

Este sería un ejemplo al ejecutarlas:



FUNCIONES ESTADISTICAS

Las funciones estadísticas van aplicadas a campos del **DataSet** que estén dentro de una banda tipo detalle o subdetalle (la banda padre):

SUM(campo) -> Calcula la suma de toda la columna del campo. Hace suma y sigue.

COUNT -> Cuenta el número de registros aparecidos.

MAX(campo) -> Muestra el valor máximo del campo de entre todos los registros aparecidos.

MIN(campo) -> Muestra el valor mínimo.

AVERAGE(campo) -> Calcula el valor medio del campo respecto a todos los registros aparecidos.

Este ejemplo lo he realizado en una banda de tipo **Summary** y debajo de una banda de tipo **Detail**:

ID	Fecha	Cliente	Base Imponible	Importe I.V.A.	TOTAL
1	30/01/2009	PABLO AGUILAR RUIZ	300,00	48,00	**348,00**
2	30/01/2009	MIGUEL HERNANDEZ ROJO	55,00	8,80	**63,80**
3	30/01/2009	MARIA MARTINEZ BERNAL	100,00	16,00	**116,00**
4	30/01/2009	CONSTRUCCIONES DOLYDEN	200,00	32,00	**232,00**
5	30/01/2009	PABLO AGUILAR RUIZ	50,00	8,00	**58,00**
6	30/01/2009	MIGUEL HERNANDEZ ROJO	100,00	16,00	**116,00**

Y este sería el resultado:

FUNCIONES ESTADISCICAS

SUM(TOTAL) = 933,8

COUNT = 6

MAX(IVA) = 48

MIN(IVA) = 8

AVERAGE(BASEIMPONIBLE) = 134,166666666667

FUNCION PARA AVERIGUAR EL TIPO DE CAMPO

La función **TYPEOF** nos dice el tipo de campo que le pasamos como parámetro, aunque por las pruebas que he realizado no trabaja muy fino que digamos:

TIPOS DE DATOS

TYPEOF(ID) = INTEGER

TYPEOF(FECHA) = STRING

TYPEOF(TOTAL) = FLOAT

El tipo fecha me lo ha dado como **STRING**. De todas formas no le encuentro mucho sentido a esta función, ya que rara vez se trabaja con tipos **Variant** en los informes.

FUNCIONES LOGICAS

Podemos condicionar resultados mediante la función:

IF (condición, valor si es verdadero, valor si es falso)

Por ejemplo:

IF(SUM(IVA)> 100,'MAS DE 100 €','MENOS DE 100€') = MAS DE 100 €

Esta función evalúa el primer parámetro y si es verdadero entonces devuelve el segundo parámetro y en caso contrario el tercero.

OTRAS FUNCIONES

En el apartado **Other** del asistente de la propiedad **Expression** tenemos estas funciones:

STR(x) -> Convierte un número entero o real a tipo cadena de texto (STRING).

UPPER(s) -> Pasa el contenido de una cadena de texto a mayúsculas.

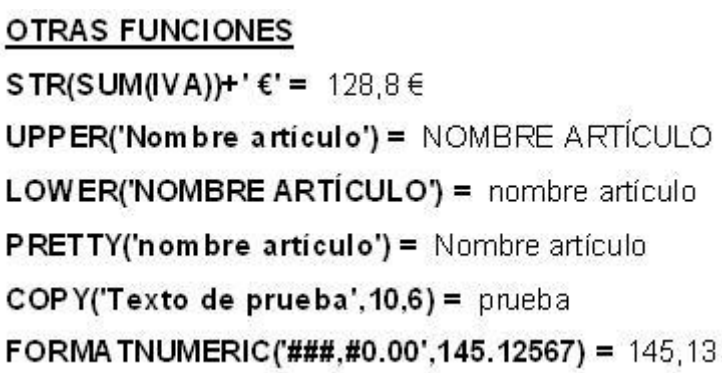
LOWER(s) -> Convierte todo el texto a minúsculas.

PRETTY(s) -> Pasa a mayúsculas la primera letra del texto (capitalización).

COPY(s, inicio, longitud) -> Copia un trozo de cadena de texto que comience en la posición inicio y con una longitud predeterminada.

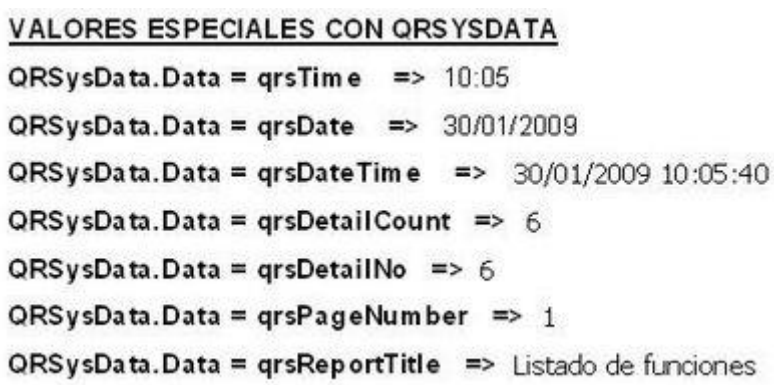
FORMATNUMERIC(formato, numeroreal) -> Convierte un número entero o real a STRING con el formato que le pasemos como parámetro.

Veamos un ejemplo de todas estas funciones:



OTROS USOS DEL CAMPO TQRSYSDATA

Aparte de mostrar los valores de fecha y hora mediante la propiedad **Data**, el campo de la clase **TQRSysData** permite mostrar esta otra información:



Las propiedades **qrsDetailCount** y **qrsDetailNo** devuelven el número de registros encontrados (la verdad es que no se donde está la diferencia entre ambas). Luego tenemos **qrsPageNumber** que devuelve el número de página actual y **qrsReportTitle** que imprime el título del informe (**TQuickRep.ReportTitle**).

En el caso de que estas funciones sean insuficientes siempre podemos echar mano a una etiqueta **TQRLabel**enviando a su propiedad **Caption** el formato o información que nos interese en tiempo real con los eventos**BefotePrint** de las bandas.

INFORMES COMPUESTOS

Otra de las ventajas que incorpora **QuickReport** es la posibilidad de imprimir varios informes juntos (uno detrás de otro) como si fueran el mismo (divide y vencerás).

Para ello añado al formulario principal de la aplicación el componente **TQRCompositeReport**:

A este componente lo voy a llamar **InformeCompuesto** y voy a juntar en un solo informe los listados de facturas y de ventas:

```
Application.CreateForm( TFListadoFacturas, FListadoFacturas
);
Application.CreateForm( TFListadoVentas, FListadoVentas );
InformeCompuesto.Preview;
```

En el evento **OnAddReports** de este componente añadimos todos los informes que queremos imprimir:

```
procedure TForm1.InformeCompuestoAddReports(Sender:
TObject);
begin
  InformeCompuesto.Reports.Add( FListadoFacturas.Informe );
  InformeCompuesto.Reports.Add( FListadoVentas.Informe );
end;
```

De modo que al ejecutarlo mostraría este aspecto:

LISTADO DE FACTURAS DESGLOSADAS

ID	Fecha	Cliente	Base Imponible	Importe I.V.A.	TOTAL
1	28/01/2009	PABLO AGUILAR RUIZ	300,00	48,00	348,00

Unidades	Artículo	Precio	Total Línea
1,00	MOVIL SAMSUNG	100,00	100,00
1,00	MOVIL NOKIA	200,00	200,00

ID	Fecha	Cliente	Base Imponible	Importe I.V.A.	TOTAL
2	28/01/2009	MIGUEL HERNANDEZ ROJO	55,00	8,80	63,80

Unidades	Artículo	Precio	Total Línea
1,00	CARGADOR SAMSUNG	20,00	20,00
1,00	CARGADOR NOKIA	35,00	35,00

LISTADO DE VENTAS POR CLIENTE

Nombre	Nº factura	Base Imponible	Importe I.V.A.	Total
MIGUEL HERNANDEZ ROJO	2	55,00	8,80	63,80
		55,00	8,80	63,80
	Suma y sigue...	55,00	8,80	63,80
	Totales.....	55,00	8,80	63,80

Esto es algo ideal cuando hay que volcar todos los informes en un archivo PDF.

INFORMES ABSTRACTOS

Un informe abstracto es un componentes de la clase **TQuickAbstractRep** que viene a ser el equivalente a**TQuickRep**. Este componente no va vinculado con bases de datos ni con campos. Los informes abstractos sólo son útiles para imprimir rápidamente información fija que podemos cargar de un archivo de texto.

Para poder utilizar este tipo de informes tenemos que añadir una banda de tipo **TQRStringBand** y dentro de esta última metemos expresiones **TQRExpr**. También permite añadir etiquetas **TQRLabel** o campos **TQRSysData**.

En resumen, los informes abstractos son una versión “light” de **TQuickRep** cuando necesitamos imprimir textos fijos rápidamente y sin complicaciones de bases de datos.

CONCLUSIONES

Con estos puntos finalizo la iniciación a la creación de Informes con QuickReport. Creo que con esto cubro más o menos las partes más importantes. Podía haber realizado cientos de ejemplos pero eso lleva un tiempo no tengo.

De todas formas, si alguien tiene alguna duda sobre como realizar un cierto tipo informe que lo escriba en el blog o me mande un correo y veré que puedo hacer. O si hay algún usuario avanzado que controle a tope QuickReport y se le ocurra algún ejemplo que me lo mande y lo publicaré encantado.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 09:37 4 comentarios
Etiquetas: impresión

23 enero 2009

Creación de informes con QuickReport (IV)

Si creéis que con los informes que hemos hecho hemos cubierto casi todas las posibilidades estáis equivocados. La mente retorcida de nuestros clientes siempre hace dar otra vuelta de tuerca a nuestros informes hasta límites insospechados. Simplemente quieren verlo todo en un folio.

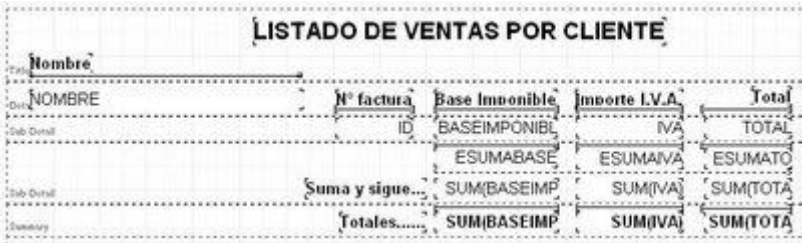
MAS DIFICIL TODAVIA

A partir de las tablas de **CLIENTES** y **FACTURAS** vamos a hacer un informe de ventas por cliente que nos va a mostrar lo que ha facturado cada uno (desglosado), con el total y un suma y sigue. Además vamos a totalizar toda la facturación:

LISTADO DE VENTAS POR CLIENTE

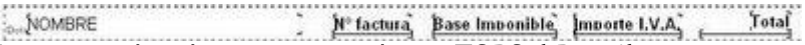
Nombre	Nº factura	Base Imponible	Importe I.V.A.	Total
PABLO AGUILAR RUIZ	1	300,00	48,00	348,00
	5	50,00	8,00	58,00
		350,00	56,00	406,00
	Suma y sigue...	350,00	56,00	406,00
MIGUEL HERNANDEZ ROJO	Nº factura	Base Imponible	Importe I.V.A.	Total
	2	55,00	8,80	63,80
	6	100,00	16,00	116,00
		155,00	24,80	179,80
	Suma y sigue...	505,00	80,80	585,80
MARIA MARTINEZ BERNAL	Nº factura	Base Imponible	Importe I.V.A.	Total
	3	100,00	16,00	116,00
		100,00	16,00	116,00
	Suma y sigue...	605,00	96,80	701,80
CONSTRUCCIONES DOLYDEN S.L.	Nº factura	Base Imponible	Importe I.V.A.	Total
	4	200,00	32,00	232,00
		200,00	32,00	232,00
	Suma y sigue...	805,00	128,80	933,80
	Totales.....	805,00	128,80	933,80

Podéis verlo mejor si hacéis clic sobre la imagen. El diseño del informe sería este:



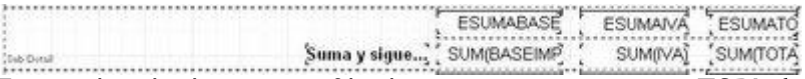
A la primera banda la vamos a llamar **Cabecera** y va a ser un objeto **TQRBand** cuya propiedad **BandType** será **rbTitle**. Sólo va a tener el título del listado y el encabezado:

La segunda banda se va a llamar **Detalle** y también será de tipo **TQRBand**. La banda va a ser de tipo **rbDetail** y va a mostrar los nombres de los clientes y las cabeceras de la factura mediante campos **TQRDBText**:



La tercera banda va a ser un objeto **TQRSubDetail** que vamos a llamar **Subdetalle**. En esta banda vamos a mostrar los importes de cada factura (también con **TQRBText**):

La cuarta banda también va a ser un objeto **TQRSubDetalle** que llamaremos **Subdetalle2**. Esta banda va a encargarse de totalizar las facturas de cada cliente y hacer un suma y sigue:



En esta banda, la primera fila de campos son etiquetas **TQRLabel** que vamos a reprogramar para que nos sumen el total de facturación de cada cliente. Para cumplir este cometido creamos en la sección privada del formulario de este informe estas variables:

```
private
  { Private declarations }
  dSumaBase, dSumaIva, dSumaTotal: Double;
```

Ahora programamos el evento **OnBeforePrint** de la banda **Detalle** (**rbDetail**) para que inicialice estas variables y además nos filtre las facturas del cliente actual:

```
procedure TFListadoVentas.DetalleBeforePrint(Sender:
TQRCustomBand;
var PrintBand: Boolean);
begin
  with Form1 do
  begin
    Facturas.Filter := 'IDCLIENTE=' + #39 +
      Clientes.FieldByName('ID').AsString + #39;
    Facturas.Filtered := True;
  end;

  dSumaBase := 0;
  dSumaIva := 0;
```

```
  dSumaTotal := 0;
end;
```

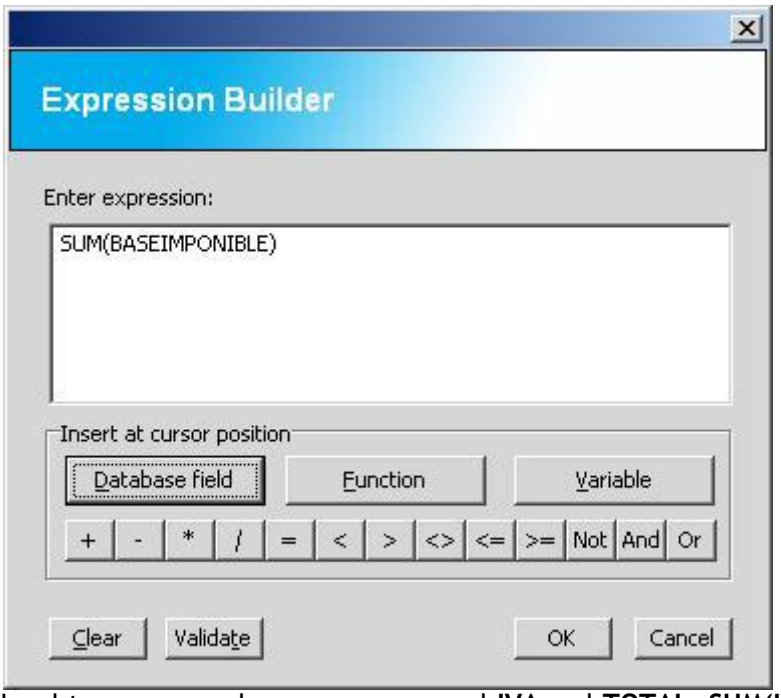
Ahora le decimos antes de imprimir la banda **Subdetalle2** que cambie las etiquetas por lo que llevamos aculumado:

```
procedure TFListadoVentas.Subdetalle2BeforePrint(Sender:
TQRCustomBand;
var PrintBand: Boolean);
begin
  ESUMABASE.Caption := FormatFloat( '###,###,#0.00',
dSumaBase );
  ESUMAIVA.Caption := FormatFloat( '###,###,#0.00',
dSumaIva );
  ESUMATOTAL.Caption := FormatFloat( '###,###,#0.00',
dSumaTotal );
end;
```

Y para que haga la suma le decimos en el evento **AfterPrint** (después de imprimir) de la banda **Detalle** que vaya sumando los totales de cada registro:

```
procedure TFListadoVentas.SubdetalleAfterPrint(Sender:
TQRCustomBand;
BandPrinted: Boolean);
begin
  dSumaBase := dSumaBase +
Form1.Facturas.FieldByName('BASEIMPONIBLE').AsFloat;
  dSumaIva := dSumaIva +
Form1.Facturas.FieldByName('IVA').AsFloat;
  dSumaTotal := dSumaTotal +
Form1.Facturas.FieldByName('TOTAL').AsFloat;
end;
```

Esta banda de totales también contiene campos de tipo **TQRExpr** (expresiones) que no van vinculados a bases de datos pero que permiten realizar operaciones con los campos del formulario. Estos campos los utilizamos para hacer el suma y sigue de todos los clientes. Para acumular la suma del campo **BASEIMPONIBLE** le decimos a la propiedad**Expression** del objeto **TQRExpr** que haga esto:



Igual tenemos que hacer para sumar el **IVA** y el **TOTAL**: **SUM(IVA)** y **SUM(TOTAL),** cada uno en su campo correspondiente. Como puede verse hay otras funciones para calcular la media, el máximo, pasar a mayúsculas, etc.

Y la quinta y última banda va a ser un objeto **TQRBand** cuyo tipo va a ser **rbSummary**. Aquí vamos a meter campos de tipo **TQRExpr** para sumar Base Imponible, Importe IVA y Total.

Aunque todo esto parezca una paranoia mental os puedo asegurar que se hace rápidamente respecto a otros editores como Rave (el que tenga valor que intente hacer este ejemplo).

Hay que procurar utilizar las funciones que trae el objeto **TQRExpr** para que nos resuelva los problemas, pero en casos como el que hemos visto de totales los podemos solucionar metiendo etiquetas y haciendo el trabajo a mano por programación.

EXPORTANDO EL INFORME A PDF

Aunque hoy en día se suelen utilizar programas como Primo PDF para convertir los documentos impresos a PDF, a los usuarios de los programas de gestión no les suele hacer mucha gracia porque cada vez que van a imprimir tiene que seleccionar la impresora Primo PDF y elegir la ubicación.

Por fortuna, las últimas versiones de QuickReport permiten exportar el formato a PDF. Esto podemos hacerlo directamente mediante código cuando vamos a imprimir el documento:

```
Application.CreateForm( TFListadoVentas, FListadoVentas );
FListadoVentas.Informe.ExportToFilter(
  TQRPDFDocumentFilter.Create( 'C:\ListadoVentas.pdf' ) );
```

Para poder compilar esto tenemos que añadir la unidad **QRPDFFilt** en nuestro formulario. Con una sola línea de código hemos solucionado el problema.

EXPORTANDO A OTROS FORMATOS

Igualmente podemos pasarlo a **HTML** de este modo:

```
Application.CreateForm( TFListadoVentas, FListadoVentas );
FListadoVentas.Informe.ExportToFilter(
  TQRGHTMLDocumentFilter.Create( 'C:\ListadoVentas.html' )
);
```

Este filtro necesita la unidad **QRWebFilt**.

Para exportar el informe a un documento **RTF** compatible con **Microsoft Word** sería así:

```
Application.CreateForm( TFListadoVentas, FListadoVentas );
FListadoVentas.Informe.ExportToFilter(
TQRRTFExportFilter.Create( 'C:\ListadoVentas.rtf' ) );
```

Ese filtro necesita la unidad **QRExport**.

También podemos exportar a **Microsoft Excel** de este modo:

```
Application.CreateForm( TFListadoVentas, FListadoVentas );
FListadoVentas.Informe.ExportToFilter( TQRXLSFilter.Create(
'C:\ListadoVentas.xls' ) );
```

Utiliza también la unidad **QRExport**.

Si nos interesa extraer tanto los datos como los metadatos se puede exportar todo a **XML**:

```
Application.CreateForm( TFListadoVentas, FListadoVentas );
FListadoVentas.Informe.ExportToFilter(
TQRXDocumentFilter.Create( 'C:\ListadoVentas.xml' ) );
```

Este filtro está en la unidad **QRXMLSFilt**.

Se puede exportar a un formato gráfico de 16 bits de gráficos vectoriales (aunque también soporta bitmaps) con extensión **WMF** que era antiguamente utilizado por Windows 3.0. Este formato puede verse con programas de diseño gráfico como GIMP. Se exporta de este modo:

```
var
  FiltroWMF: TQRWMFExportFilter;
begin
  FiltroWMF := TQRWMFExportFilter.Create(
'C:\ListadoVentas.wmf' );
  Application.CreateForm( TFListadoVentas, FListadoVentas
);
  FListadoVentas.Informe.Prepare;
  FListadoVentas.Informe.ExportToFilter( FiltroWMF );
end;
```

Un formato que considero muy interesante es pasarlo a texto plano para impresoras matriciales de tickets:

```
Application.CreateForm( TFListadoVentas, FListadoVentas );
FListadoVentas.Informe.ExportToFilter(
TQRASCIIExportFilter.Create( 'C:\ListadoVentas.txt' ) );
```

Este formato de texto también viene muy bien para impresoras matriciales de papel continuo que tango nos castigan cuando se desajustan verticalmente.

Por último y no menos importante, tenemos el formato CSV que permite exportar los datos en texto plano entre comillas y separados por comas (ideal para exportar datos a Excel, Access, MySQL, etc.):

```
Application.CreateForm( TFListadoVentas, FListadoVentas );
FListadoVentas.Informe.ExportToFilter(
TQRCommaSeparatedFilter.Create( 'C:\ListadoVentas.csv' ) );
```

Con esto cubrimos todas las necesidades de exportación de nuestros informes a otros programas ofimáticos y de terceros. Estos últimos filtros que hemos visto también tiran de la unidad **QRExport**.

No os podéis ni imaginar la cantidad de horas que he tenido que echar para averiguar como se exporta a todos estos formatos (mirando el código fuente). Todavía no entiendo por que narices los que fabrican componentes tan buenos como estos no les sale de los ... de escribir una documentación y ejemplos decentes (y eso también va por los Indy). No me extraña de que algunos programadores abandonen Delphi y elijan otros lenguajes por esta dificultad. No entiendo que a estas alturas (Delphi 2009) los programadores de Delphi tengamos que sufrir tanto para averiguar cuatro tonterías. A ver si "patera" technlogies se pone las pilas de una vez.

La semana que viene seguiré investigando para sacarle todo el partido posible a QuickRerport.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 11:11 12 comentarios
Etiquetas: impresión

16 enero 2009

Creación de informes con QuickReport (III)

Hoy vamos a ver como modificar la impresión de un informe en tiempo real para poder cambiar su aspecto. También veremos otro ejemplo de un informe con detalle y subdetalle.

MODIFICANDO LA IMPRESIÓN EN TIEMPO REAL

Supongamos que en el listado de clientes que vimos en ejemplos anteriores hemos añadido la columna **SALDO INICIAL** y nos interesa que salgan todos los importes en negro menos aquellos con saldo negativo que aparecerán en rojo:

LISTADO DE CLIENTES

	Población	Provincia	Alta	Saldo
SPAÑA, 8	MADRID	MADRID	12/05/2008	4.200,28
R, 8	MADRID	MADRID	21/06/2008	1.000,00
NTEROS, 16	VALENCIA	VALENCIA	03/05/2008	-280,45
LO, 23	MURCIA	MURCIA	08/03/2009	580,99

Cuando se insertan

campos en un informe, Delphi suele darles el nombre QRDBText1, QRDBText2, etc. Debemos acostumbrarnos a poner en la propiedad **Name** de estos componentes el mismo nombre que tiene el campo en la tabla, en nuestro caso **SALDOINICIAL**. Sólo para los campos de la base de datos, no es necesario para etiquetas y figuras gráficas.

También sería recomendable ponerle nombre a la banda donde se imprimen los campos (**Detalle**).

Luego programamos el evento **BeforePrint** de la banda **Detalle** del siguiente modo:

```
procedure TFListadoClientes.DetalleBeforePrint(Sender:
TQRCustomBand;
var PrintBand: Boolean);
begin
  if Form1.Clientes['SALDOINICIAL'] >= 0 then
    SALDOINICIAL.Font.Color := clBlack
  else
    SALDOINICIAL.Font.Color := clRed;
end;
```

El evento **BeforePrint** (antes de imprimir) se va a ejecutar antes de dibujar cada registro en la banda. De este modo podemos cambiar en tiempo real la visualización de cada campo antes de que se plasme en el folio.

Otro caso especial que nos puede solucionar este método es el de modificar los campos booleanos para que no imprima True o False. Por ejemplo, si quiero imprimir el campo activo pasaría esto:

DE CLIENTES

Población	Alta	Saldo	Activo
MADRID	12/05/2008	4.200,28	False
MADRID	21/06/2008	1.000,00	True
VALENCIA	03/05/2008	-280,45	True
MURCIA	08/03/2009	580,99	True

Para solucionar esto en vez de imprimir directamente el campo **ACTIVO** utilizando un objeto **TQRBDText** lo que vamos a hacer es meter una etiqueta llamada **EActivo** (**TQRLabel**):

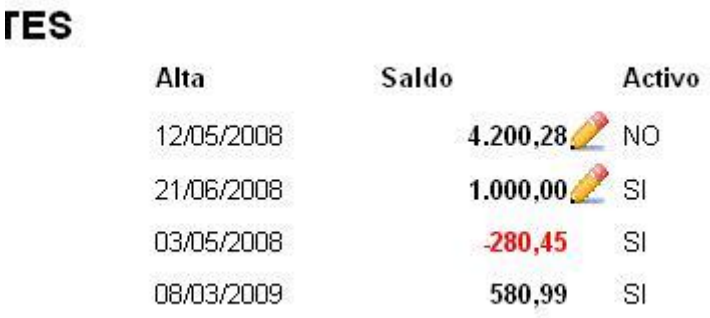
Ahora modificamos el método **BeforePrint** de la banda **Detalle**:

```
procedure TFListadoClientes.DetalleBeforePrint(Sender:
TQRCustomBand;
 var PrintBand: Boolean);
begin
  if Form1.Clientes['SALDOINICIAL'] >= 0 then
    SALDOINICIAL.Font.Color := clBlack
```

```
  else
    SALDOINICIAL.Font.Color := clRed;

  if Form1.Clientes['ACTIVO'] then
    EActivo.Caption := 'SI'
  else
    EActivo.Caption := 'NO';
end;
```

Si el campo **ACTIVO** está a **True** imprimimos **SI**, **NO** en canso contrario. Estos truquillos nos ahorran mucho tiempo y evitan tener que modificar los datos de la tabla original.

Hasta modemos meter en la banda figuras gráficas o iconos según los datos del registro actual. Pongamos por ejemplo que a los clientes cuyo saldo inicial sea superor a 800 euros le ponemos el icono de un lápiz a lado como advertencia para su edición:



Lo que he hecho es insertar el objeto imagen **TQRImage** con el icono del lápiz (a través de su propiedad **Picture**) y luego vuelvo a ampliar el evento **BeforePrint** de este modo:

```
procedure TFListadoClientes.DetalleBeforePrint(Sender:
TQRCustomBand;
  var PrintBand: Boolean);
begin
  if Form1.Clientes['SALDOINICIAL'] >= 0 then
    SALDOINICIAL.Font.Color := clBlack
  else
    SALDOINICIAL.Font.Color := clRed;

  if Form1.Clientes['ACTIVO'] then
    EActivo.Caption := 'SI'
  else
    EActivo.Caption := 'NO';

  if Form1.Clientes['SALDOINICIAL'] >= 800 then
    Icono.Enabled := True
  else
    Icono.Enabled := False;
end;
```

Como puede verse en el código he utilizado la propiedad **Enabled** en vez de la propiedad **Visible** para ocultar el icono. La propiedad **Visible** no tiene ningún efecto en QuickReport (por lo menos a mi no me ha funcionado en ningún caso).

Con estos pequeños retoques nuestros informes parecerán mucho más profesionales (y a nuestros clientes les sacamos más pasta diciendo que nos ha costado mucho hacerlo :)

CREANDO UN INFORME CON DETALLE Y SUBDETALLE

Ahora vamos a complicar un poco más el asunto y vamos hacer un listado desglosado de facturas con dos bandas de tipo detalle. Más bien con un detalle y un subdetalle:

Este listado muestra las cabeceras de las facturas (los totales) y sus líneas de detalle. Este sería su diseño en QuickReport:

El documento se compone de tres bandas:

Cabecera: esta banda es un objeto **TRQBand** y va a ser de tipo **rbTitle**. En esta banda sólo vamos a meter el título del listado:

Detalle: también va a ser un objeto **TRQBand** y su tipo será **rbDetail**. En esta banda necesitamos los títulos de la factura así como los campos de la misma:

El único campo que no hay que vincular con la tabla **Factura** es el **NOMBRE** que va vinculado a la tabla **CLIENTES**.

SubDetalle: aquí introducimos una banda de tipo **TQRSubDetail**. No es lo mismo una banda **TRQBand** de tipo**rbSubDetail** que una banda **TQRSubDetail**. La primera sólo suele utilizarse para totalizar datos (sin vincularlo a tablas) pero esta última tiene la propiedad **DataSet** que permite vincular sus campos a otra tabla:

Ahora viene la vinculación con las tablas. Primero hay que seleccionar el objeto **TQuickRep** y vincular a su propiedad **DataSet** la tabla **Facturas** (la cabecera). En la banda de **Detalle** vinculamos todos los campos a la tabla**Facturas**.

En la banda **SubDetail** tenemos que vincular los campos a la tabla **Detalle** (el contenido de la factura).

Si ejecutáramos el listado como está, nos pasaría esto:

LISTADO DE FACTURAS DESGLOSADAS

ID	Fecha	Cliente	Base Imponible	Importe I.V.A.	TOTAL
1	15/01/2009	CONSTRUCCIONES DOLYDEN	300,00	48,00	348,00

Unidades	Artículo	Precio	Total Linea
1,00	MOVIL SAMSUNG	100,00	100,00
1,00	MOVIL NOKIA	200,00	200,00
1,00	CARGARDOR SAMSUNG	20,00	20,00
1,00	CARGADOR NOKIA	35,00	35,00

ID	Fecha	Cliente	Base Imponible	Importe I.V.A.	TOTAL
2	15/01/2009	CONSTRUCCIONES DOLYDEN	55,00	8,80	63,80

Unidades	Artículo	Precio	Total Linea
1,00	MOVIL SAMSUNG	100,00	100,00
1,00	MOVIL NOKIA	200,00	200,00
1,00	CARGARDOR SAMSUNG	20,00	20,00
1,00	CARGADOR NOKIA	35,00	35,00

Han ocurrido dos cosas:

1º Al no filtrar la tabla de **Clientes** por cada factura resulta que aparece el mismo cliente en ambas facturas. Debería ser: **CLIENTES.ID** = **FACTURAS.IDCLIENTE**.

2º Al no filtrar la tabla **Detalle** por cada factura nos aparece el detalle de ambas facturas debajo de cada cabecera. Debería ser así: **DETALLE.IDFACTURA** = **FACTURAS.ID**.

Para solucionar esto tenemos que escribir este código en el evento **BeforePrint** (antes de imprimir) de la banda**Detalle**:

```
procedure TFListadoFacturas.DetalleBeforePrint(Sender:
TQRCustomBand;
var PrintBand: Boolean);
begin
  with Form1 do
  begin
    Clientes.Filter := 'ID=' +
IntToStr(Facturas['IDCLIENTE']);
    Clientes.Filtered := True;

    Detalle.Filter := 'IDFACTURA=' +
IntToStr(Facturas['ID']);
    Detalle.Filtered := True;
  end;
end;
```

Lo que hacemos es filtrar el cliente y el detalle de la factura respecto a la factura que estamos imprimiendo en cada momento. De este modo no hay que recurrir a sentencias SQL para filtrar la tablas (aunque también se podría hacer así).

En la siguiente parte de este artículo veremos más ejemplos de informes así como la forma de exportar a PDF y otros formatos.

Pruebas realizadas en RAD Studio 2007.

09 enero 2009

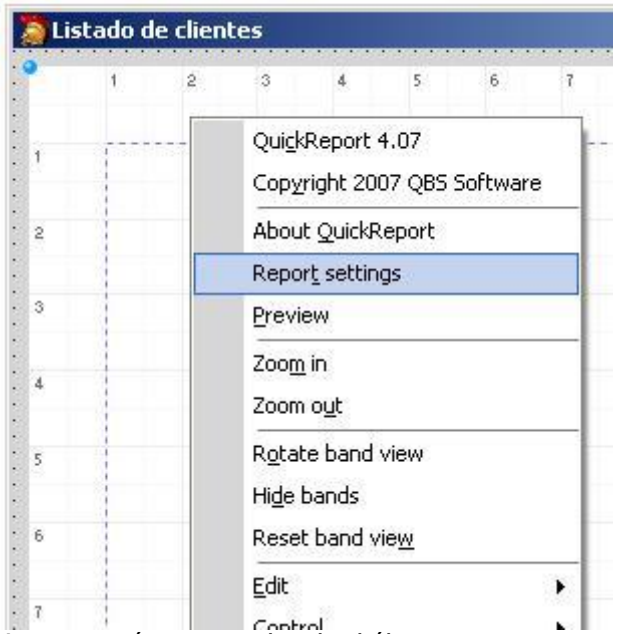
Creación de informes con QuickReport (II)

Después de crear nuestro primer informe con QuickReport vamos a hacer otro a dos bandas el cual va a ser un listado de clientes. En un listado cualquiera necesitamos que la banda superior se mantenga fija en todas las hojas que se impriman y que la banda que va a imprimir lo datos se adapte al folio según los mismos.

CREANDO UN LISTADO SIMPLE A DOS BANDAS

Para cumplir nuestro cometido vamos a crear un nuevo formulario llamado **FListadoClientes** y dentro del mismo volvemos a añadir el objeto de la clase **TQuickRep** que llamaremos **Informe**.

Como necesito imprimir columnas bien anchas entonces debemos imprimir el folio de manera apaisado. Esto se hace pinchando el objeto **QuickRep** con el botón derecho del ratón y seleccionando **Report Settings**:

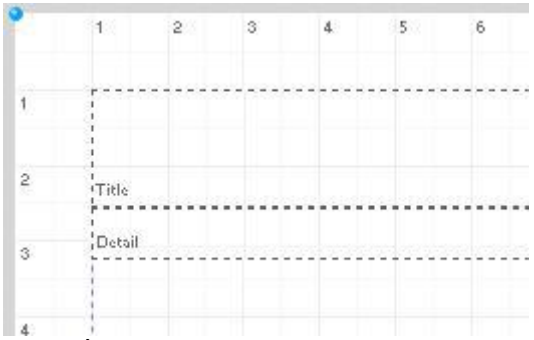


Y aparecerá este cuadro de diálogo:

Como puede verse en la imagen no sólo podemos seleccionar el tipo de folio (A4, A5, B4, etc.) sino que además se puede configurar el folio a un ancho y alto determinado en milímetros que viene muy bien para formatos de documentos que se imprimen en impresoras matriciales con papel continuo (lo más asqueroso para un programador).

En la parte derecha de esta ventana tenemos un ComboBox que por defecto tiene seleccionada la opción **Portrait**. Esto significa que va a imprimir el folio verticalmente. Como lo vamos a imprimir apaisado seleccionamos**LandScape**.

Ahora añadimos al objeto **TQuickRep** un par de bandas (**TQRBand**) y a la segunda banda le podemos mediante el inspector de objetos a su propiedad **BandType** el valor **rbDetail**:



También es buena costumbre ponerle nombre a las bandas para luego poder modificar sus acciones mediante código. A la primera banda le ponemos en su propiedad **Name** el nombre **Cabecera** y a la segunda banda la llamamos**Detalle**.

Insertamos etiquetas en la cabecera mediante objetos **TQRLabel** y los campos en el detalle mediante **TQRDBText**:

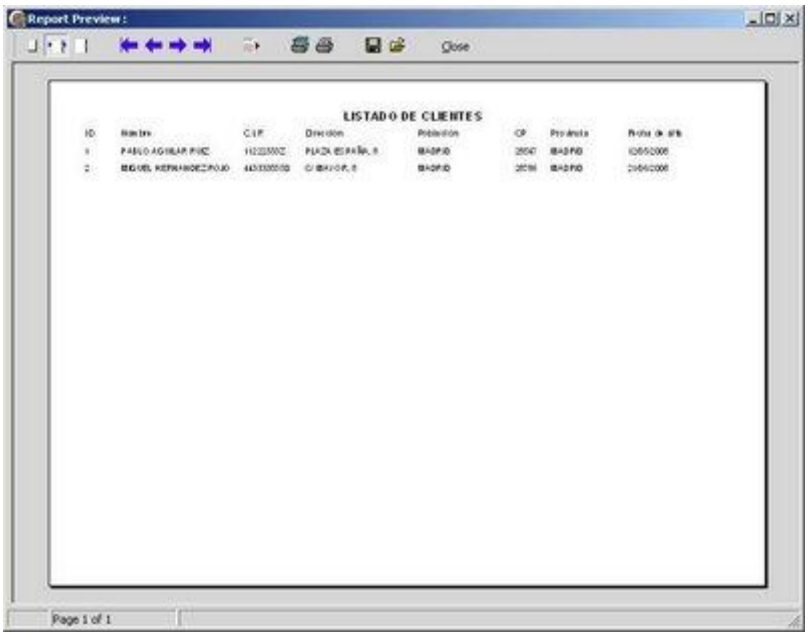
Al igual que hicimos en el artículo anterior debemos vincular este formulario a la unidad donde están los dados (**uses Unit1**) y añadir en la propiedad **DataSet** del objeto **TQuickReport** la tabla **Clientes**.

Hay que procurar que la banda de **Detalle** tenga la altura ajustada a los campos a imprimir, para que la separación de las líneas no sea muy grande y no se desperdicien folios (algunos jefes de empresas son muy tacaños).

Creamos el formulario del informe y lo ejecutamos desde la ventana principal:

```
 Application.CreateForm( TFListadoClientes,
FListadoClientes );
 FListadoClientes.Informe.Preview;
```

Al ejecutar nuestro informe ya lo tendremos apaisado:

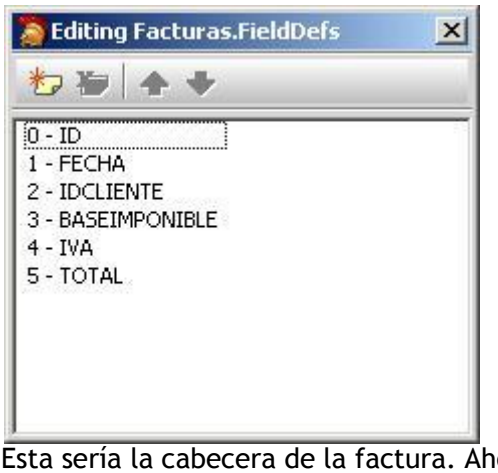


INFORMES MAESTRO/DETALLE A TRES BANDAS

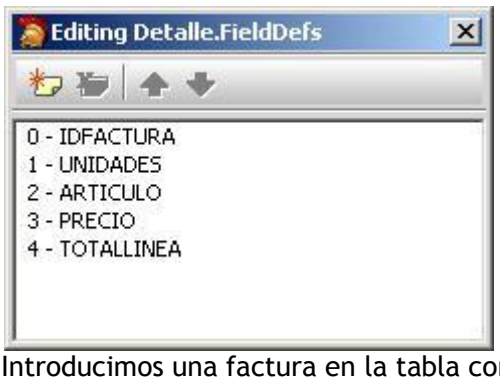
El siguiente ejemplo que vamos a ver es la impresión de una factura que tiene cabecera, detalle y pie así como los datos de un cliente que viene de otra tabla, es decir, vamos a imprimir datos de tres tablas a la vez.

Como en los casos anteriores creamos un nuevo formulario llamado **FImprimirFactura** e insertamos el objeto**TQuickRep** y lo llamamos **Informe**.

Para este ejemplo voy a añadir al formulario principal un nuevo objeto **TClientDataSet** llamado **Facturas** con los siguientes campos:



Esta sería la cabecera de la factura. Ahora creamos otro **ClientDataSet** para el **Detalle** de la factura con estos campos:



Introducimos una factura en la tabla con un par de líneas de detalle:

```
Facturas.Append;
Facturas['ID'] := 1;
Facturas['FECHA'] := Date;
Facturas['IDCLIENTE'] := 1;
Facturas['BASEIMPONIBLE'] := 300;
Facturas['IVA'] := 48;
Facturas['TOTAL'] := 348;
Facturas.Post;

Detalle.Append;
Detalle['IDFACTURA'] := 1;
Detalle['UNIDADES'] := 1;
Detalle['ARTICULO'] := 'MOVIL SAMSUNG';
Detalle['PRECIO'] := 100;
Detalle['TOTALLINEA'] := 100;
Detalle.Post;

Detalle.Append;
Detalle['IDFACTURA'] := 1;
Detalle['UNIDADES'] := 1;
Detalle['ARTICULO'] := 'MOVIL NOKIA';
Detalle['PRECIO'] := 200;
Detalle['TOTALLINEA'] := 200;
Detalle.Post;
```

A continuación creamos un nuevo formulario llamado **FImprimirFactura** e insertamos un objeto **TQuickRep** con tres bandas:

A las bandas las vamos a llamar **Cabecera**, **Detalle**, **Pie** y serán del tipo (BandType) **rbTitle**, **rbDetail** y **rbPageFooter**respectivamente.

Como hicimos en ejemplos anteriores, vinculamos a formulario con el formulario principal para poder traernos los datos de las tablas **(uses Unit1**). Seleccionamos el objeto **TQuickRep** y vinculamos mediante la propiedad **DataSet** la tabla **Detalle**.

Aunque pueda parecer extraño, cuando vamos a imprimir un documento maestro/detalle debemos vincular a la propiedad **DataSet** del objeto **TQuickRep** la tabla **Detalle** y no la del maestro. Esto se debe a que es el detalle la banda que se va a repetir en la factura, es decir, el centro de la misma. La cabecera y el pie son sólo datos adicionales. Eso no quita que luego cada campo se vincule con una tabla distinta. No hay limitaciones a la hora de vincular campos con tablas.

Pero vayamos por partes. Primero implementamos la cabecera de la factura en la primera banda:

Los campos **ID** y **FECHA** están vinculados con la tabla **FACTURAS**. En cambio todos los campos encerrados en el rectángulo están vinculados con la tabla **CLIENTES** (sólo hay que procurar tener filtrado que **FACTURA.IDCLIENTE** =**CLIENTES.ID**).

Lo demás son todo etiquetas **TQRLabel**. El rectángulo y las líneas que he puesto debajo de los títulos de los campos del detalle lo he hecho con un objeto **TQRShape** que en su propiedad **Shape** soporta estos formatos:

qrsRectangle: Un rectángulo.
qrsCircle: Un círculo.
qrsHorLine: Línea horizontal.
qrsVerLine: Línea vertical.
qrsRoundRect: Rectángulo con las esquinas redondeadas.
qrsRightAndLeft: dos líneas verticales (un rectángulo sin techo ni suelo).
qrsTopAndBottom: dos líneas horizontales (un rectángulo sin paredes).

En mi formato de factura sólo he utilizado líneas horizontales y un rectángulo.

Vamos ahora con el detalle de la factura en la segunda banda:

Esta banda sólo va a contener campos **TQRDBText** vinculados a la tabla **DETALLE**. Es importante quitar en este tipo de campos el valor **Autosize** y ajustar el ancho manualmente para que por ejemplo el nombre del artículo no chafe el precio en el caso de que el nombre sea demasiado largo.

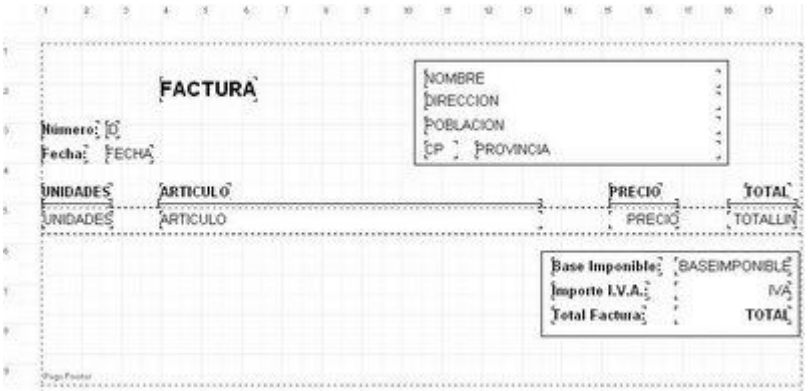
También es importante que todos los campos de tipo float estén alineados a la derecha mediante su propiedad**Alingment** = **taRightJustify**.

Hay que procurar que la altura de la banda de detalle sea un poco más grande que la altura de los campos para que el espacio entre líneas no sea ni muy ancho ni muy estrecho.

Vamos con la última banda, el pie:

Esta banda sólo contiene los tres campos que totalizan la factura y que están vinculados con la tabla **FACTURAS**. Un detalle que hay que tener presente es que la posición vertical a la que van a salir estos totales es inversamente proporcional a la altura de la banda de tipo **rbPageFooter**, es decir, si queremos que los totales salgan más arriba hay que ampliar verticalmente la altura de esta banda. Y si queremos que los totales salgan pegados en la parte inferior del folio entonces hay que procurar que esta banda sea lo más estrecha posible (verticalmente).

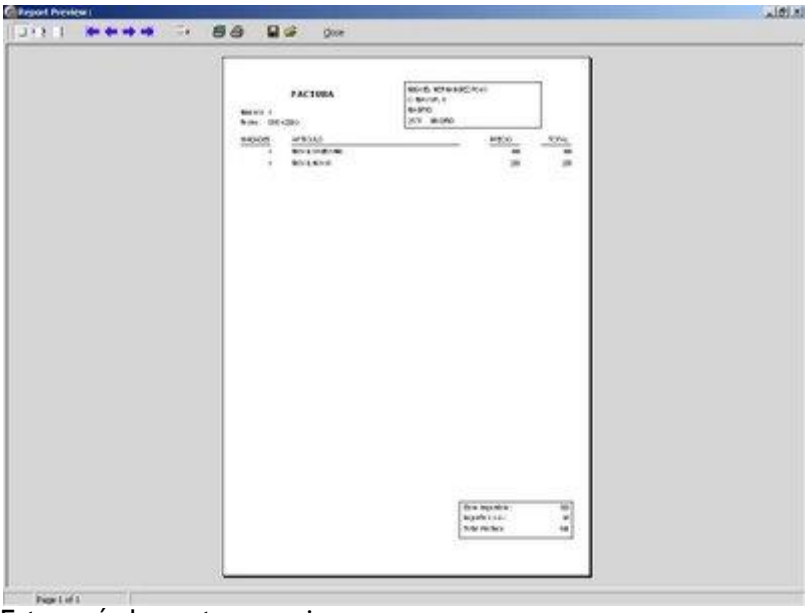
Este seria nuestro diseño de la factura:



Llamamos a la vista previa como siempre:

```
 Application.CreateForm( TFImprimirFactura,
FImprimirFactura );
 FImprimirFactura.Informe.Preview;
```

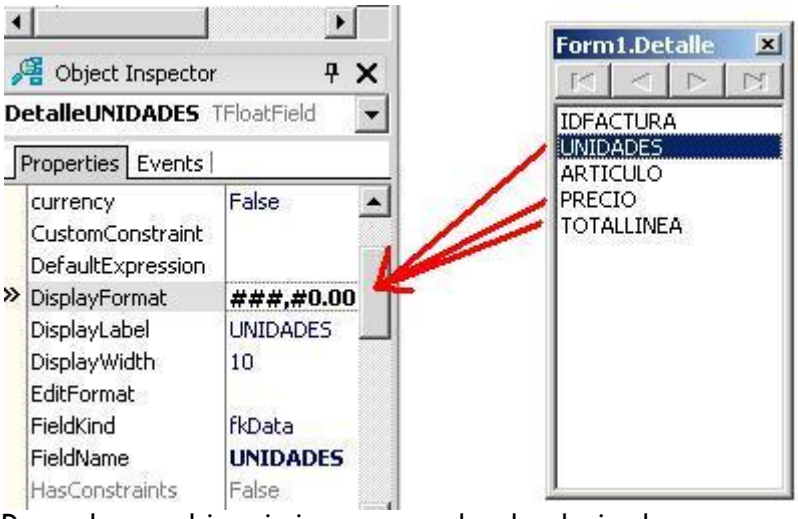
Y esta sería la vista previa al imprimirla:



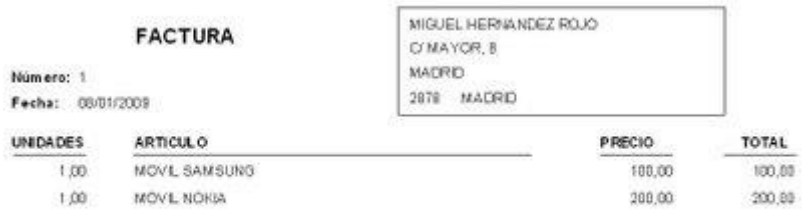
Esta sería la parte superior:

Y la parte inferior:

También sería recomendable que los campos que vamos a imprimir de tipo moneda tengan definida su máscara a dos decimales para evitar problemas de redondeo. Esto se hace en el diseño de la tabla original a través de la propiedad **DisplayFormat**:



De modo que al imprimir aparezcan los dos decimales:



Estos tres ejemplos que hemos visto son los tres típicos informes que nos suelen pedir: un formulario, un listado o un documento maestro/detalle (y pie). Aunque aquí no se acaban nuestras pesadillas ya que esto solo son juguetes comparado con lo que los clientes suelen pedir de verdad.

En el próximo artículo veremos casos más complejos y algunos truquillos para modificar el aspecto del informe en tiempo real.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 09:37 5 comentarios
Etiquetas: impresión

02 enero 2009

Creación de informes con QuickReport (I)

Aunque en los tiempos que estamos disponemos de gran cantidad de generadores de informes para Delphi (Rave Reports, Report Maganer, FastReport, etc.) hay que reconocer que QuickReport es el que tiene mayor predilección por parte de los usuarios debido a que venía incorporado de serie el las primeras versiones de Delphi.

Luego Borland-Inprise-CodeGear se deshizo de él a partir de Delphi 7 (no se porque) e incorporó Rave Reports, donde este último destacaba por tener su propio editor, por tener una documentación cutre e inexistente así como por dar Access Violations si lo mirabas mal.

Pero la empresa QBS Software siguió ampliando estos componentes hasta incluso las últimas versiones de Delphi 2009.

Los componentes QuickReport son comerciales estando actualmente en la versión 5.0 (podemos descargar una versión trial para probarlos). Para la serie de artículos que voy a escribir es suficiente con tener la versión 4.xx que está disponible desde para las

versiones de Delphi 5, 6 y 7.

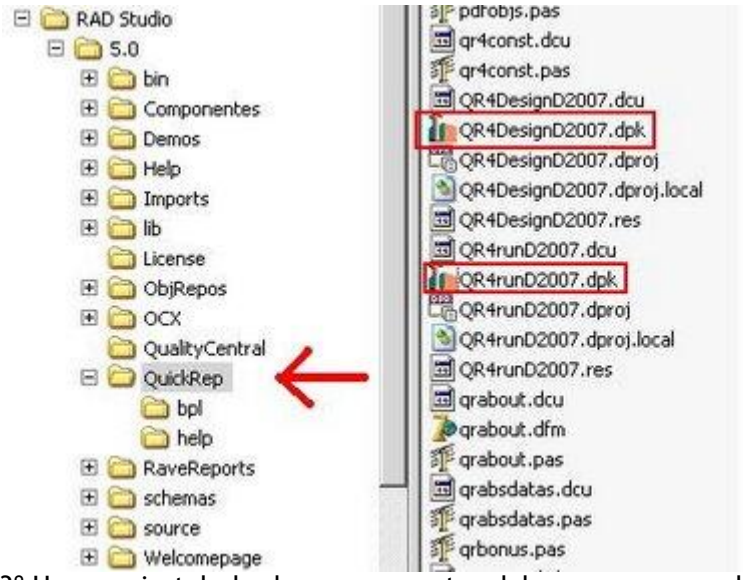
Para escribir este artículo me he basado en la versión **4.07** de QuickReport para Delphi 2007 instalándolo en **RAD Studio 2007**.

INSTALANDO LOS COMPONENTES QUICKREPORT

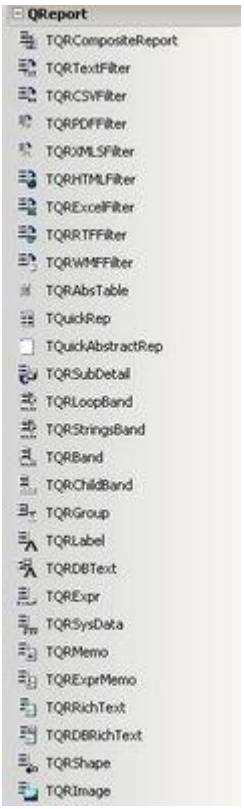
Aunque las instalaciones suelen variar en aspecto de una versión a otra, la forma de instalarlas suele ser la misma:

1º Descargamos el archivo de la instalación en un exe o archivo zip y al ejecutarla se instalará en un subdirectorio según la ruta donde tengamos instalado Delphi o RAD Studio:

2º Cuando finalice la instalación abrimos nuestra versión de Delphi habitual y abrimos los archivos con extensión**DPK** para instalarlos:

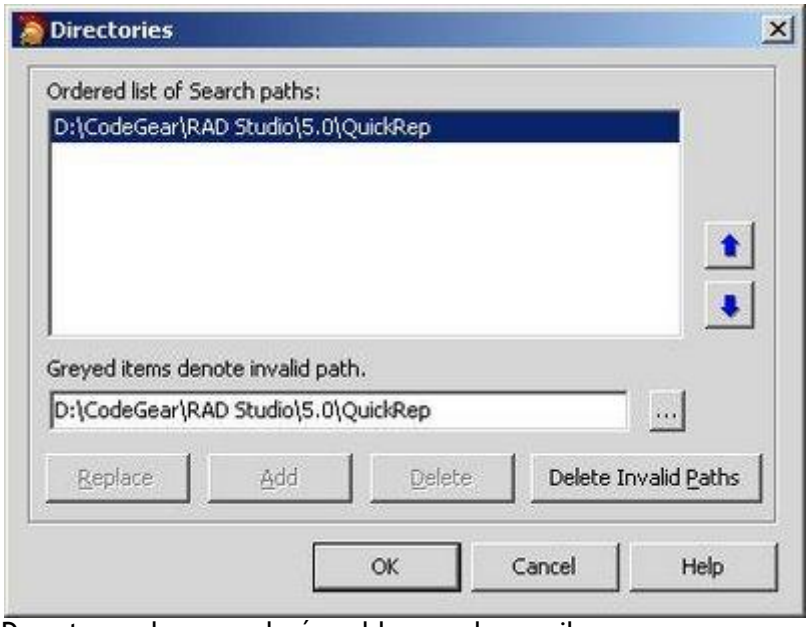


3º Una vez instalados los componentes deben aparecer en la paleta de herramientas:



CREANDO UN NUEVO PROYECTO

Al crear un nuevo proyecto debemos configurar en las opciones de nuestro proyecto la ruta donde se encuentran los componentes QuickReport. En Delphi 2007 se haría seleccionando **Project** -> **Options** y en el apartado**Directories/Conditionals** introducimos la ruta donde se han instalado los componentes, por ejemplo:



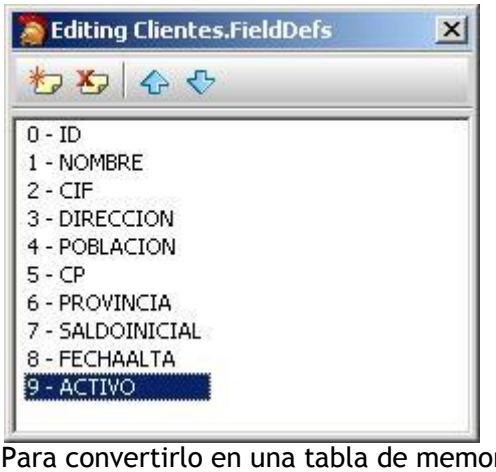
De este modo ya no dará problemas al compilar.

CREANDO LA BASE DE DATOS PARA IMPRIMIR EL INFORME

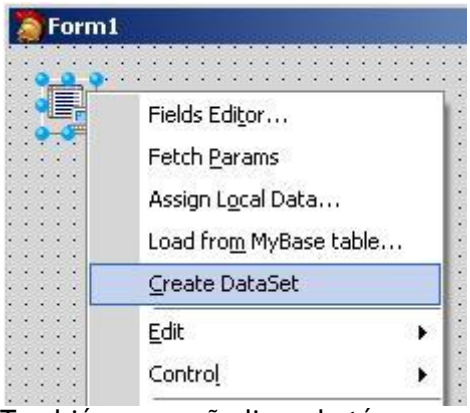
A modo de ejemplo vamos a crear una tabla de memoria utilizando los componentes de la clase **TClientDataSet** para crear una base de datos de clientes para luego imprimir su formulario en QuickReport.

De la pestaña **Data Access** extraemos el componente **ClientDataSet** y los insertamos en el formulario con el nombre**Clientes**:

En la propiedad **FieldDefs** de este componente vamos a definir los siguientes campos:



Para convertirlo en una tabla de memoria pinchamos el componente con el botón derecho del ratón y seleccionamos **Create Dataset**:



También voy a añadir un botón cuyo evento sea añadir un registro a la tabla:

```
procedure TForm1.BIntroducirDatosClick(Sender: TObject);
begin
  Clientes.Append;
  Clientes['ID'] := 1;
  Clientes['NOMBRE'] := 'PABLO AGUILAR RUIZ';
  Clientes['CIF'] := '11222333Z';
  Clientes['DIRECCION'] := 'PLAZA ESPAÑA, 8';
  Clientes['POBLACION'] := 'MADRID';
  Clientes['CP'] := '28547';
  Clientes['PROVINCIA'] := 'MADRID';
  Clientes['SALDOINICIAL'] := 4200.28;
  Clientes['FECHAALTA'] := '12/05/2008';
  Clientes['ACTIVO'] := False;
```

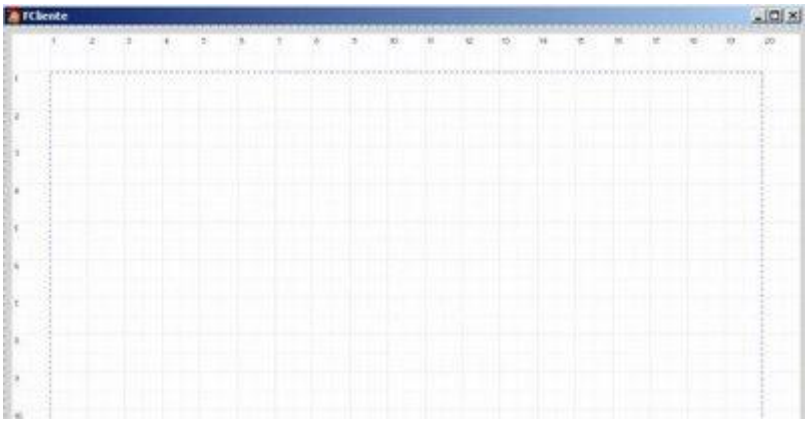
```
  Clientes.Post;
end;
```

Ahora pasemos a imprimir un informe con estos datos.

CREANDO EL INFORME

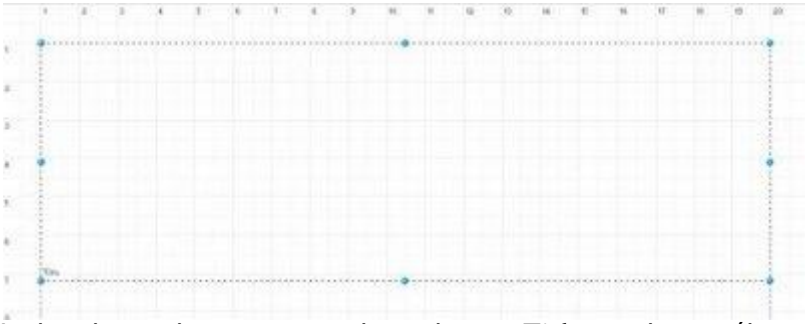
Un informe creado en QuickReport va vinculado directamente a un formulario de Delphi por lo que obligatoriamente debemos utilizar como mínimo un formulario por cada informe. También tiene sus ventajas y es que aprovecha las nuevas características del editor de formularios de los nuevos Delphi.

Creamos un nuevo formulario que vamos a llamar **FCliente** y dentro del mismo añadimos el componente**TQuickRept**:



Hay que ampliar bien el formulario ya que nos inserta el equivalente a un folio A4 donde vamos a diseñar nuestro informe. Encima de este componente es donde debemos añadir sólo componentes de la pestaña QuickReport (no vale meter por ejemplo un componente **TLabel**, si lo deja pero no imprimirá nada).

Lo siguiente que tenemos que hacer es insertar una banda la cual va a contener los datos que queremos imprimir. Esto se hace añadiendo el componente de la clase **TQRBand** dentro del informe:



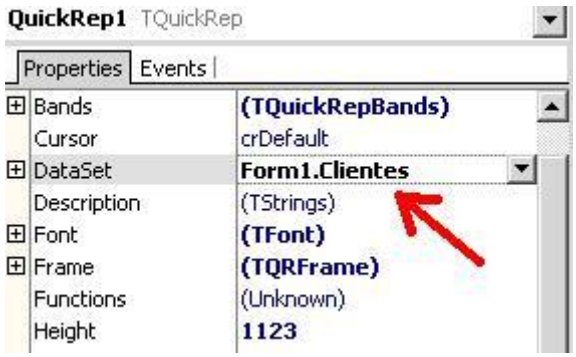
La banda que hemos insertado es de tipo **Title**, es decir, sólo se va a imprimir una sola vez independientemente del número de registros tenga nuestra tabla. Esto puede verse en el inspector de objetos (cuando tenemos la banda seleccionada) con su propiedad **BandType**:

Como vamos a imprimir la ficha de un cliente podemos dejarla como esta y empezar a meter componentes. Comenzamos añadiendo etiquetas para los nombres de los campos utilizando el componente **TQRLabel**:

Para poder añadir los campos de la base de datos de clientes debemos añadir a este formulario la unidad donde se halla la tabla que queremos imprimir (en mi ejemplo **Form1**):

```
uses Unit1;
```

Ahora pinchamos sobre cualquier borde del objeto **TQuickRep** y en el inspector de objetos le ponemos el su propiedad **DataSet** la tabla que queremos imprimir:



Una vez hecho esto ya podemos introducir campos de base de datos para traernos la información. Una de las ventajas de las que goza QuickReport es que podemos vincular directamente el informe a una tabla de datos sin tener que utilizar el objeto **DataSource** de la pestaña **DataAccess**.

Para introducir los campos vamos a utilizar el componente **TQRDBText**:

Sólo hay que procurar que cada campo tenga las propiedades **DataSet** (**Form1.Clientes**) y **DataSource** (el nombre del campo) rellenas.

EJECUTANDO LA VISTA PREVIA DEL INFORME

Desde nuestro formulario principal (**Form1**) voy a crear un botón para abrir la vista previa del informe que hemos hecho. Antes de esto tengo que vincular la unidad **UClientes.pas** a nuestro formulario para poder llamarlo.

Para programar de una manera más clara voy a llamar al objeto **TQuickRep** con el nombre **Informe**. Así se llama a la vista previa:

```
procedure TForm1.BInformeClientesClick(Sender: TObject);
begin
  FCliente.Informe.Preview;
end;
```

Con algo tan simple como esto ya tenemos el informe en pantalla:

En principio nos aparece la vista previa del informe en una ventana pequeña en el centro de la pantalla. En la parte izquierda de la ventana nos aparecen dos pestañas:

Thumbails: contiene la vista previa cada folio que se imprime (al estilo PowerPoint).

Search Result: permite hacer búsquedas de texto en las hojas impresas.

Si nos interesa que se vea sólo el informe a pantalla completa entonces seleccionamos el objeto **TQuickRep** llamado**Informe** y en su propiedad **PreviewInitialState** seleccionamos **wsMaximized**.

Para quitar las pestañas **Thumbails** y **Search result** hay que deshabilitar las opciones **PrevShowThumbs** y**PrevShowSearch**. Y por último vamos a ampliar la visualización del folio (el Zoom) seleccionando en su propiedad**PrevInitialZoom** el valor **qrZoomToWith**. Este sería el resultado:

También sería conveniente darle un título al informe sobre todo porque si lo mandamos a imprimir aparecerá sin título en la cola de impresión. Para ello escribimos en su propiedad **ReportTitle** por ejemplo **Datos del cliente**. Puede verse al abrir la vista previa:

Esta barra superior contiene los típicos botones para modificar el zoom, avanzar o retroceder por las páginas, modificar las propiedades de la impresión o imprimir directamente. QuickReport también permite guardar el informe generado en un archivo con extensión **QRP** para poder abrirlo posteriormente y reimprimirlo.

No hay que confundir esto con una plantilla de informe ya que lo que guardamos el todo el informe (incluyendo los datos). Esto puede ser útil para guardar copias de factura, recibos, etc. cuando los datos ya no existen en nuestra base de datos y necesitamos reimprimirlos por pérdida del documento original. También nos evita volver a procesar listados lentos (informe de estadísticas de enero, febrero, etc.). Con procesarlo una vez sobra.

Esta sería la parte básica de generación de informes aunque veremos en artículos posteriores que QuickReport es mucho más potente que esto.

Pruebas realizadas en RAD Studio 2007.

Publicado por Administrador en 10:14 9 comentarios
Etiquetas: impresión