COPIA LITERAL

Extracto del libro “GAMBAS, programación visual con Software Libre”, de la editorial EDIT LIN EDITORIAL S.L., cuyos autores son Daniel Campos Fernández y José Luis Redrejo.

LICENCIA DE ESTE DOCUMENTO

Se permite la copia y distribución de la totalidad o parte de esta obra sin ánimo de lucro. Toda copia total o parcial deberá citar expresamente el nombre del autor, nombre de la editorial e incluir esta misma licencia, añadiendo, si es copia literal, la mención “copia literal”. Se autoriza la modificación y traducción de la obra sin ánimo de lucro siempre que se haga constar en la obra resultante de la modificación el nombre de la obra originaria, el autor de la obra originaria y el nombre de la editorial. La obra resultante también será libremente reproducida, distribuida, comunicada al público y transformada en términos similares a los expuestos en esta licencia.

3. La interfaz gráfica

3. La interfaz gráfica
3. La interfaz gráfica
3. La interfaz gráfica

3.1 Concepto

Si bien Gambas, como cualquier otro lenguaje de programación bien diseñado, puede trabajar perfectamente desligado de toda librería gráfica, creando programas de consola, uno de sus puntos fuertes es la sencillez para crear interfaces gráficas de usuario.

En el mundo GNU/Linux ha existido, a lo largo de la historia, diversas librerías que facilitan la creación de dichas interfaces. Al nivel más bajo, el tradicional sistema X-Window tan sólo proporciona una API para mostrar ventanas, dibujar líneas, copiar mapas de bits y poco más. Sobre dicho sistema, una de las primeras librerías de apoyo que proporcionaba controles completos, tales como botones o etiquetas, fue la Motif™, tradicional de sistemas UNIX™. Su clon libre, Lesstif, probablemente llegó demasiado tarde al escenario para tener un papel relevante.

Las interfaces creadas con estas dos librerías han tenido siempre fama, además, de ser bastante feas o incómodas, especialmente a ojos de los usuarios de Windows™, y no hay que olvidar que de este sistema propietario proviene buena parte de los actuales usuarios de escritorio GNU/Linux.

La compañía TrollTech™, por su parte, creo las librerías QT, para el desarrollo de aplicaciones gráficas con C++. Estas librerías, en principio, se distribuían bajo una licencia, la QPL, no totalmente compatible con el proyecto de la Free Software Foundation. Su inclusión como base del proyecto de escritorio KDE generó un gran revuelo, y el rechazo de un sector de la comunidad hacia ambos proyectos (QT y KDE). En la actualidad, las librerías QT en su edición no comercial, se distribuyen bajo licencia GPL, lo cual implica que los programas desarrollados y compilados con QT como base, han de ser también Software Libre compatible con la GPL. Un programa que no cumpla esta norma, habría de ser compilado sobre la versión comercial de QT, que la compañía antes citada vende y soporta.

En parte como rechazo al tándem QT/KDE, y en parte para crear una alternativa al popular escritorio KDE, surgió el proyecto GNOME, que está basado en las librerías GTK+.

GTK+ se desarrolló al principio como una librería gráfica escrita únicamente para el popular programa de dibujo The Gimp (de hecho, GTK significa *Gimp Tool Kit*). Pero más tarde se escindió de este proyecto para convertirse en una librería de propósito general, especialmente diseñada para desarrollos en lenguaje C. Hoy en día, todo el proyecto GTK+ está dividido en varios bloques y niveles: Glib, utilidades de carácter general sin relación con la interfaz gráfica: Gobject para dotar, de cierta orientación a objetos al lenguaje C; Atk, que es un kit de accesibilidad; Pango, para la gestión de fuentes; Gdk, para el dibujo de bajo nivel; y, finalmente, GTK, que proporciona los elementos de la interfaz gráfica habitual. La licencia de GTK+ es LGPL, por lo que ha sido utilizada, además de en muchos proyectos de Software Libre, en programas gráficos privativos que no desean contar con el soporte de la versión comercial de QT porque la han visto como una alternativa más cómoda en sus desarrollos, o han decidido reducir los costes al no tener que pagar por la librería gráfica.

Al margen de estos pesos pesados, hay nombres como FOX, FLTK o WxWidgets, que también resuenan como alternativas para el desarrollo de programas gráficos.

Por tanto, hay muchas alternativas (toolkits) para desarrollar interfaces, y al menos dos de ellas (QT y GTK+) sirven como base para los dos escritorios más comunes (KDE y GNOME), que a su vez aportan un aspecto y funcionalidades diferentes. No obstante, una aplicación KDE puede funcionar en un entorno GNOME y viceversa, a costa, quizá, de perder homogeneidad en el entorno.

Gambas ha decidido ser neutral al respecto. Aporta una interfaz de alto nivel, sencilla para el diseño y programación habituales, que no está ligada a los conceptos de QT, GTK+, ni a ninguna otra librería gráfica subyacente.

No obstante, a la hora de implementar dicha interfaz sí que es necesario emplear alguna librería de sustento por debajo, escrita en C o C++. Por tanto, existen dos componentes gráficos que proporcionan al programador libertad de elección: *gb.qt* y *gb.gtk*. Como se puede suponer por sus nombres, el primero utiliza código compilado con QT y el segundo código compilado con GTK+.

La particularidad de Gambas es que el código escrito para *gb.qt* funcionará exactamente igual si reemplazamos este componente por *gb.gtk* y viceversa. Por lo tanto, el programador en cada momento puede elegir el que más se adapte a sus necesidades por diversos motivos, por ejemplo:

- Integración con KDE, GNOME o XFCE (este último, un escritorio ligero basado en GTK+).
- Aspecto final. Algunos programadores y usuarios se sienten más a gusto con una aplicación QT, otros con GTK+.
- Cuestiones de rendimiento, uso de recursos.
- Licencia, costes en software comercial.

- Necesidades especiales. GTK+ se puede compilar sobre DirectFB, un sistema gráfico alternativo a X-Window, ligero y apto para sistemas empotrados. Tal vez *gb.qt* se pueda compilar en el futuro sobre Qtopia, la librería de TrollTech™ generalmente empleada en PDAs.

Partiendo de la consola

Vamos a empezar por el camino difícil para comprender cómo la interfaz gráfica no es más que otro accesorio de Gambas, que al igual que el resto de componentes, aporta clases a partir de las cuales crearemos objetos, en este caso *Controles* o *Widgets*.

Crearemos un proyecto de consola, (sí, de consola), llamado **Ventana**. Añadiremos un módulo **modMain** y una referencia al componente **gb.qt**.

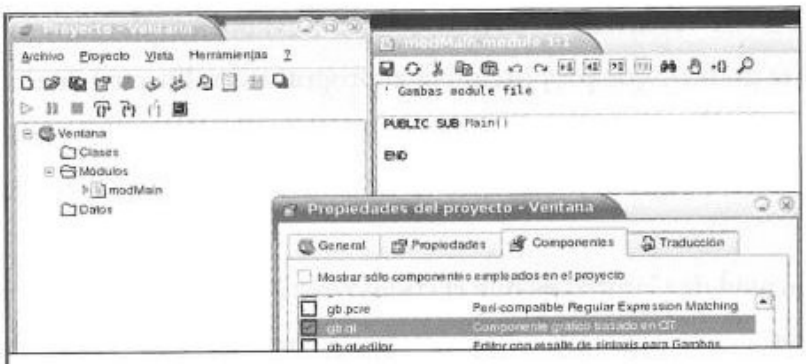


Figura 1. Proyecto **Ventana**.

Escribiremos ahora el siguiente código:

```
PUBLIC SUB Main()

  DIM hWin AS Window

  hWin = NEW Window
  hWin.Show()

END
```

Al ejecutarlo, aparece una ventana solitaria en la pantalla, que desaparecerá cuando pulsemos el botón **Cerrar** del gestor de ventanas.

Las ventanas son contenedores de primer nivel. Un *contenedor* es un control que permite situar otros en su interior (botones, cuadros de texto, etc.). Y el adjetivo de *primer nivel* se refiere a que no tiene un *padre*, es decir, que no cuelga de otro control de nuestra aplicación, sino directamente del escritorio.

En este programa podemos ver varios efectos, a primera vista, extraños. El primero es que hemos declarado **hWin** como una variable local, así pues, parece que al finalizar la función **Main** el objeto debería destruirse. Esto no así ya que la ventana, al ser un control, mantiene una referencia interna (los objetos Gambas sólo se destruyen si no existe una referencia en todo el programa). Dicha referencia podemos decir que se corresponde con la ventana que está dibujada en el servidor gráfico, con la intermediación de la librería gráfica (en este caso QT). Por otra parte, el programa debería haber finalizado al terminar la función **Main**() y no ha sido así.

El segundo efecto se debe a que tanto el componente *gb.qt* como el componente *gb.gtk* son llamados automáticamente tras el método **Main**() del programa, y queda en el lazo principal de la librería gráfica, esperando a que se produzcan eventos gráficos (por ejemplo, una pulsación del ratón sobre un control).

En la práctica, para crear un programa gráfico indicaremos directamente en el asistente que deseamos crear un proyecto gráfico, de modo que el entorno de desarrollo incluye ya, por defecto, el componente *qb.qt*, y nos permite crear un *formulario de inicio*. Un formulario de inicio es una clase de inicio que hereda las características de la clase *Form*, que no necesita de un método *Main* en el programa porque el intérprete llama directamente al formulario para mostrarlo, y entra en el lazo de gestión de eventos.

El entorno de desarrollo

Para crear un nuevo formulario nos situaremos en la ventana principal del IDE y, tras pulsar el botón derecho, elegiremos **Nuevo | Formulario**.

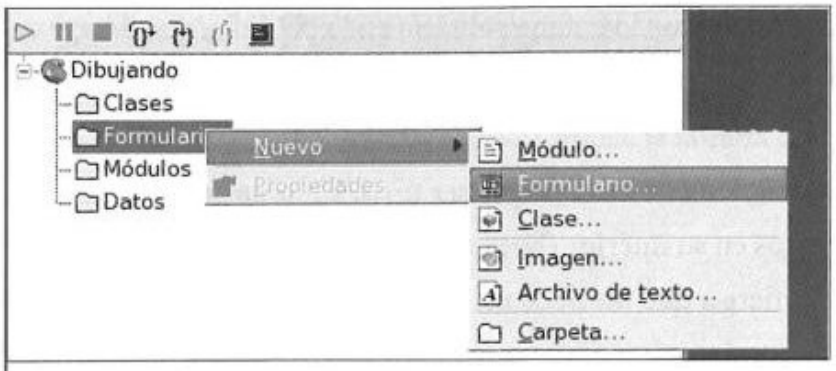


Figura 2. Pasos para crear un nuevo formulario.

Ahora pasaremos a situar los diferentes controles, para lo cual los seleccionaremos de la Caja de herramientas y los posicionaremos sobre los formularios manteniendo el botón izquierdo pulsado, mientras le damos el tamaño deseado. El código se irá escribiendo en función de los eventos que generen los controles, por ejemplo, el evento *Click* de los botones o los eventos de ratón y teclado de cada control.

Figura 3. Controles disponibles en la **Caja de herramientas**.

Al pulsar con doble clic sobre un control, se genera, automáticamente, el encabezado del código correspondiente al evento por defecto (el más característico) del control seleccionado.

Además, pulsando con el botón derecho se puede seleccionar la lista de eventos disponibles para un control.

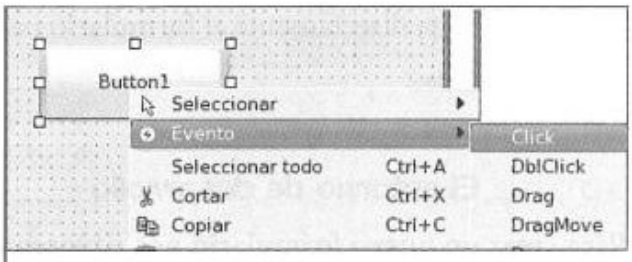


Figura 4. Eventos del control **Button1**.

3. 2 Manejo básico de los controles

A pesar de que cada control ha sido diseñado para cumplir una función específica, comparten buena parte de su interfaz de programación, de modo que aprender a manejar un nuevo control sea tarea sencilla para un programador que ya ha trabajado con otros controles.

Los controles heredan métodos, propiedades y eventos de la clase *Control*, por lo que todas las características heredadas son aplicables a todos ellos.

En cuanto a los contenedores, proceden de la clase *Container* (que a su vez procede de *Control*) e igualmente tienen muchas características comunes.

Posición y tamaño

Todos los controles disponen de una serie de propiedades que permiten conocer y modificar su posición y tamaño dentro de su contenedor, o del escritorio en el caso de las ventanas:

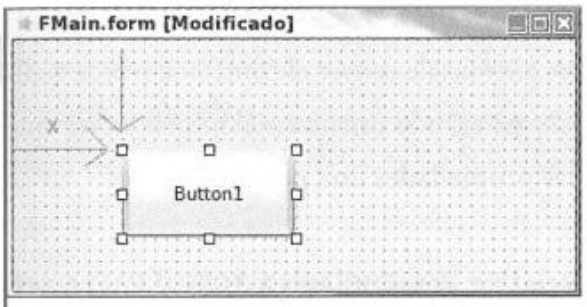


Figura 5. Propiedades *X* e *Y* del control.

- **Propiedades X e Y**: son de lectura y escritura, y determinan la posición del control, es decir, su punto superior izquierdo. En los controles comunes, la posición indicada es relativa a su contenedor, y en el caso de las ventanas es relativa a la esquina superior izquierda del escritorio. Los controles disponen de otras dos propiedades, *Left* y *Top*, que son sinónimas de *X* e *Y*, respectivamente. Usar una u otra se deja a la elección del programador.

- **Propiedades W y H**: son de lectura y escritura, y determinan el ancho y alto del control, respectivamente. Dispone de dos propiedades sinónimas. *Width* y *Height* con el mismo significado.

- **Propiedades ScreenX y ScreenY**: son de sólo lectura, y permiten conocer la posición de cualquier control relativa al escritorio, en lugar de a su contenedor *padre*.

Los contenedores disponen, además, de las propiedades *ClientX*, *ClientY*, *ClientWidth* y *ClientHeight*, que determinan, respectivamente, el inicio y la dimensión del área útil para contener controles *hijo*. Por ejemplo, el control *TabStrip*, que dispone de unas pestañas en la parte superior, sitúa a sus *hijos* por debajo de ellas; y *ScrollView*, que puede mostrar barras de Scroll, ve su área útil reducida por dichas barras.

Existen también una serie de métodos para modificar los controles:

- **Método Resize(W,H)**: con él podemos cambiar el tamaño de un control modificando su alto y ancho de una sola vez, en lugar de hacerlo en dos pasos modificando las propiedades *W* y *H*, lo que mejora el efecto gráfico de la redimensión ante el usuario.

- **Método Move(X,Y)**: mueve de una sola vez el control a la posición indicada, en lugar de hacerlo en dos pasos. También dispone de dos parámetros adicionales, *Move(X,Y,W,H)*, con los cuales además de mover el control se puede redimensionar, todo ello en un solo paso, generando una transición más suave ante el usuario, que si modificáramos las propiedades una por una.

- **Métodos MoveRelative y ResizeRelative**: son similares a *Move* y *Resize*, respectivamente, pero en este caso las unidades no son píxeles, sino unidades relativas al tamaño de la fuente por defecto del escritorio. Con esta capacidad, el aspecto del formulario será similar para usuarios que utilicen distintas configuraciones de fuentes (por ejemplo, grandes en un escritorio a 1024x768 o más pequeñas en un escritorio a 800x600).

Las ventanas (controles *Window* y *Form*) disponen, por su parte, de varios eventos relacionados con la posición y tamaño. El evento Resize se genera cada vez que el usuario redimensiona una ventana, y *Move* cuando se mueve.

Visibilidad

Todo control puede encontrarse en uno de estos dos estados en un momento dado: *Visible*, que significa que aparece representado ante el usuario; e *Invisible*, que mantiene todas sus propiedades, pero no aparece de cara al usuario.

La propiedad *Visible* sirve para conocer el estado de visibilidad, en todo momento.

Por ejemplo, los métodos *Show()* y *Hide()* muestran y ocultan, respectivamente, el control.

Con lo que respecta a las ventanas, disponen de dos eventos, *Show* y *Hide*, que se disparan cuando se muestran o dejan de estar visibles.

Además, la apertura por primera vez de un formulario o ventana genera el evento *Open*.

> Crear efectos de *blinking*, controles tales como etiquetas que aparecen y desaparecen para llamar la atención del usuario, puede ser útil en alguna ocasión especial, si se reclama atención por algún motivo crítico. Pero, en general, abusar de esta técnica molesta a los usuarios y se considera poco apropiada en una interfaz correctamente diseñada.

Textos relacionados

Todos los controles pueden tener diversas cadenas de texto, que muestran de una u otra forma.

Para todos ellos, la propiedad *ToolTip* se encarga de mantener un texto que aparecerá como una pequeña ventana flotante, cuando el usuario sitúe el ratón por encima del área ocupada con el control.

De esta forma, se pueden dar pequeñas informaciones de guía para que el usuario conozca la función de dicho control dentro del programa.

Muchos controles disponen de un texto que muestran en su espacio principal, como es el caso de las etiquetas, las cajas de texto o los botones. Para todos estos controles, la propiedad *Text* es la que determina la cadena a mostrar.

Puesto que Gambas trabaja con codificación UTF-8 para la interfaz gráfica, puede que sea necesario utilizar la función *Conv$* para convertir desde otras codificaciones, a la hora de representar texto procedente, por ejemplo, de una base de datos o de un proceso en ejecución en segundo plano.

Estos controles, por comodidad para programadores de otros entornos, tienen un sinónimo para esta propiedad, denominado *Caption*, salvo en el caso de las ventanas cuyo sinónimo es *Title*.

Colores

Para cada control se definen dos colores: *ForeGround*, que es el color de primer plano en el que normalmente se mostrará el texto del control o parte de sus líneas y dibujos; y *Background*, que es el color de fondo.

Por herencia de otros lenguajes de programación, disponen de dos sinónimos: *BackColor* y *ForeColor*.

Los colores en Gambas son valores numéricos enteros, desde el 0 al hexadecimal FFFFFF, de forma que cada componente de color viene determinado por sus componentes de rojo, verde y azul.

La intensidad de cada uno de estos colores básicos varía entre el 0 (mínimo) y el 255 (máximo).

La gestión del color se realiza en Gambas a través de la clase *Color*, que dispone de varios métodos estáticos, así como de una serie de constantes.

Las constantes de la clase *Color* determinan una serie de colores básicos, en su codificación numérica:

COLOR	TONALIDAD NORMAL	TONALIDAD OSCURA
Negro	Color.Black	—
Blanco	Color.White	—
Azul	Color.Blue	Color.DarkBlue
Cian	Color.Cyan	Color.DarkCyan
Magenta	color.Magenta	Color.DarkMagenta
Naranja	color.Orange	—
Rosa	color.Pink	—
Rojo	color.Red	Color.DarkRed
Violeta	color.Violet	—
Amarillo	color.Yellow	Color.DarkYellow
Gris	Color.Gray	—
Verde	Color.Green	Color.DarkGreen
Gris Claro	Color.LightGray	—

El método RGB recibe tres parámetros, las componentes de rojo, verde y azul, en este orden, con valores entre 0 y 255 para cada uno, y devuelve un número representando al color.

El método HSV recibe tres parámetros, las componentes de tonalidad (0-360), saturación (0-255) y brillo (0-255), en este orden, y lo devuelve traducido a un color con su codificación numérica habitual.

Existen también unas propiedades que determinan los colores del sistema.

Por ejemplo, y dependiendo del tema que haya seleccionado el usuario, los cuadros de texto pueden aparecer de color blanco con letra negra, o los formularios de color gris.

Estas propiedades permiten conocer los colores actuales para los elementos de la interfaz gráfica:

Color de fondo general	Color.Background
Color de fondo de los botones	Color.ButtonBackground
Color de primer plano de los botones	Color.ButtonForeground
Color de primer plano general	Color.Foreground
Color de fondo de un elemento seleccionado	Color.SelectedBackground
Color de primer plano de un elemento seleccionado	Color.SelectedForeground
Color del texto en cajas de texto	Color.TextBackground
Color de fondo en cajas de texto	Color.TextForeground

Es poco recomendable cambiar los colores de la interfaz por capricho o cuestiones de estética particulares del programador. Cada usuario elige el tema que más se adapta a sus gustos o necesidades visuales, y puede sentirse incómodo si le forzamos a usar otros colores. Por otra parte, algunos temas pueden interferir sus colores con los que hemos definido en nuestra aplicación, resultando la combinación difícil de ver, molesta o desagradable. Sólo se cambiarán colores de los controles, cuando sea expresamente necesario por algún motivo de diseño (por ejemplo, resaltar de forma clara un texto en una etiqueta).

Ratón

El ratón es la interfaz por excelencia de cualquier escritorio actual. Hemos de distinguir aquí dos apartados.

En primer lugar se encuentra la representación de éste en el escritorio, el puntero, que habitualmente aparece como una flecha, blanca o negra. Al respecto, cada control dispone de una propiedad *Mouse*, la cual puede tomar como valores las constantes de la clase *Mouse* para cambiar su aspecto.

Se consideran constantes para que el ratón adopte diversos aspectos, como puede ser un reloj (espera), un cursor de texto, flechas en diversas orientaciones, etc.

Las constantes de la clase *Mouse* son las mismas para *gb.qt y gb.gtk*, no obstante, sus valores numéricos son distintos. Por tanto, un código bien escrito y escalable, no debe usar valores numéricos para indicar un tipo de puntero, sino las constantes de esta clase.

Cada control dispone, además, de una propiedad *Cursor*, que acepta una imagen y permite dibujar un puntero totalmente personalizado (a partir de un archivo *.png* o *.xpm*, por ejemplo) para cada control. Dependiendo del servidor gráfico utilizado en el sistema, es posible que el cursor pueda tener también varios colores, no sólo en blanco y negro como los tradicionales.

La clase estática *Application*, aporta una propiedad *Busy*, que es un número entero. Si su valor es mayor que cero, todos los controles y ventanas de la aplicación mostrarán el puntero con un reloj (indicando al usuario que ha de esperar), independientemente del cursor empleado para cada control en concreto. Si el valor pasa a cero, se retorna a los cursores habituales.

Por otra parte, cada control recibe eventos del ratón, que podemos manejar desde el programa. Los eventos *MouseDown*, *MouseUp*, *MouseWheel* y *MouseMove*, determinan, respectivamente, si el usuario pulsó un botón, lo levantó, movió la rueda del ratón o movió el ratón de posición.

Dentro de estos eventos, y sólo dentro de ellos, se puede emplear la clase *Mouse*, para determinar qué botón se pulsó (izquierdo, derecho o central) mediante las propiedades *Mouse.Left*, *Mouse.Right* o *Mouse.Middle*, que toman el valor *true* si el botón correspondiente ha sido pulsado o levantado; la posición del ratón dentro del control (*Mouse.X* y *Mouse.Y*); la posición del ratón respecto al escritorio (*Mouse.ScreenX* y *Mouse.ScreenY*); así como el movimiento sobre la rueda del ratón (*Mouse.Delta* y *Mouse.Orientation*).

En la mayor parte de controles, el evento *MouseMove*, se produce sólo si hay, al menos, un botón pulsado del ratón por parte del usuario.

Este pequeño ejemplo permite mover un botón de posición dentro de un formulario, cuando el usuario lo arrastra mientras mantiene pulsado el botón izquierdo.

Creamos un formulario *Form1*, con un botón **Button1**, que incluye este código:

```
PRIVATE pX AS Integer
PRIVATE pY AS Integer

PUBLIC SUB Button1_MouseDown()

  IF Mouse.Left THEN
    '************************************
    ' Almacenamos posiciones iniciales
    '************************************
    pX = Mouse.X
    pY = Mouse.Y
  END IF

END


PUBLIC SUB Button1_MouseMove()

  IF Mouse.Left THEN
    '************************************
    ' Desplazamos el botón de acuerdo
    ' con la variación de X e Y
    '************************************
    Button1.Move(Button1.X - px + Mouse.X, Button1.Y -
    py + Mouse.Y)
  END IF

END
```

A diferencia de otros entornos de desarrollo, donde el evento recibe varios parámetros indicando el botón, posición, etc., en Gambas todo ello se conoce a partir de la información almacenada en la clase *Mouse*, que sólo está disponible en estos eventos.

El evento *MouseDown* es cancelable, lo que significa que empleando la instrucción *STOP EVENT* dentro de su código, se evita que se propague y, por tanto, que el control actúe en consecuencia, según su funcionamiento interno habitual (por ejemplo, un botón no lanzaría el evento *Click* si se cancela *MouseDown*).

Otros eventos comunes en los controles son *DblClick* (doble pulsación del ratón), *Menu* (pulsación del botón derecho) o *Click* (pulsación del botón izquierdo). Algunos controles pueden no disponer del evento *Click*.

Teclado

De forma similar al ratón, el teclado se controla con los eventos *KeyPress* y *KeyRelease*. Estos no tienen parámetros.

La clase estática *Key* proporciona la información necesaria para controlar el teclado dentro de estos eventos, de igual modo que la clase *Mouse* dentro de los eventos de ratón.

El evento *KeyPress* es cancelable con la instrucción *STOP EVENT*, de modo que se puede impedir, por ejemplo, que en una caja de texto se impriman determinados caracteres.

En el siguiente ejemplo, se bloquea un **TextBox**, de forma que sólo permita la entrada de números, la pulsación de las teclas **Supr** (borrado) y **BackSpace** (borrado hacia atrás) y el **tabulador** para pasar el foco a otro control.

Para ello llama a **STOP EVENT** cuando el código de la tecla pulsada no es ninguno de los deseados:

```
PUBLIC SUB TextBox1_KeyPress()

  SELECT CASE Key.Code

  CASE 48 TO 57 ' Códigos ASCII de los números
    CASE Key.BackSpace ' Retroceso
    CASE Key.Delete ' Borrado
    CASE Key.Tab ' Tabulador
    CASE ELSE
      STOP EVENT

  END SELECT

END
```


3.3 Galería de controles

Controles básicos

Tanto *gb.qt* como *gb.gtk* aportan una serie de controles básicos para desarrollar una interfaz gráfica. A continuación se detallan estos controles y sus características principales.

- **Label:** es una etiqueta simple que contiene una línea de texto de poca longitud. Su única función es mostrar un texto en una posición dentro de un formulario. La propiedad *Text* es la que determina el texto a mostrar en cada momento. Al margen de este uso básico, se puede modificar tanto su color de fondo (*Background*), como el color de primer plano (*ForeGround*). Como el resto de controles,

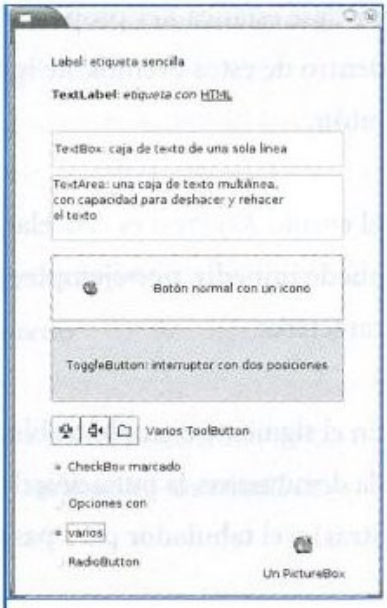


Figura 6. Controles de *gb.qt* y *gb.gtk*.

responde a los eventos de ratón, por lo que en un momento dado, ayudado de *MousePress* y *MouseRelease*, pueden servir para implementar un botón personalizado.

- **TextLabel:** es muy similar al control *Label*, pero tiene la particularidad de que es capaz de mostrar texto formateado en HTML. De esta forma, indicando una cadena con etiquetas HTML en la propiedad *Text*, podremos tener texto que combine negrita, itálica, subrayado y otras características de texto enriquecido.

```
TextLabel1.Text="<b>Texto con HTML</b><br>Dentro de una
<i>etiqueta."
```

El componente *gb.gtk* es capaz de representar menos etiquetas HTML, en las versiones actuales. Por su parte, *gb.qt* permite la inclusión de tablas e imágenes desde un archivo. Un programa desarrollado para ser independiente del toolkit, no debería abusar de esto. Pero un programa expresamente diseñado para trabajar con *gb.qt* puede, por el contrario, sacar provecho de sus características extra.

- **TextBox:** es una caja de texto, de una sola línea, en la cual el usuario puede modificar, copiar, cortar o borrar texto. El texto introducido se recibe o modifica por código mediante la propiedad *Text*. Además, el método *Select* permite seleccionar o resaltar por código una parte del texto, y *Selection* dota de algunas propiedades para conocer el texto que el usuario ha seleccionado.

Gambas emplea codificación UTF-8 en la interfaz gráfica, por lo que un carácter puede suponer 1, 2 o 3 bytes de longitud. Así, escribir *Len(TextBox1.Text)* para conocer la longitud en caracteres de un texto, no siempre dará el resultado esperado. En su lugar, se debe emplear el método *TextBox.Length*, que da siempre la longitud en caracteres del texto.

- **TextArea**: se trata de una caja de texto que es capaz contener múltiples líneas. Se permite también los retornos de carro. Como en el caso del *TextBox*, el método *Selection* y la propiedad *Select* determinan el texto seleccionado. Además, este control dispone de los métodos *Undo* y *Redo* que equivalen a las órdenes *Deshacer* y *Rehacer* de cualquier editor de textos. Es decir, eliminan los últimos cambios del usuario o los vuelven a situar en el texto.

- **Botones**: Gambas tiene tres tipos de botones. El primero, *Button*, es un botón normal, dispone de una propiedad *Text* que indica el texto a mostrar, así como una propiedad *Picture* para mostrar un icono identificativo. Este control dispone del evento *Click* que se dispara cuando se pulsa con el botón izquierdo (o derecho si está configurado para zurdos). Otro tipo de botón es el *ToggleButton*, que mantiene su estado tras una pulsación, es decir, cuando se pulsa una vez, queda presionado, y al pulsarlo otra vez, sale de ese estado. La propiedad *Value* sirve para conocer o variar su estado: *FALSE* significa 'no presionado' y *TRUE* 'presionado'. El siguiente botón, *ToolButton*, es similar, pero sólo muestra un pequeño icono, sin texto. Está diseñado para insertarse en barras de herramientas, habituales en la parte superior de las interfaces, como acceso rápido a ciertas funciones comunes. Puede actuar como un botón normal, si su propiedad *Toggle* vale *FALSE*, o como un interruptor (como un *ToggleButton*), si *Toggle* vale *TRUE*. Así mismo dispone de una propiedad *Border* que de valer *FALSE* dará apariencia plana al botón, y de valer *TRUE* lo mostrará con relieve, como un botón normal.

Los botones *Toggle* pueden ser confusos con algunos temas de KDE o GNOME, siendo difícil para el usuario determinar si se encuentra pulsado o no. Cuando sea posible se empleará un *CheckBox* en lugar de un *ToggleButton*, o bien se resaltará su estado de algún modo, por ejemplo, cambiando el color de fondo o el icono que se muestra. Esto evitará problemas con los futuros usuarios de la aplicación.

- **CheckBox**: muestra un texto determinado por la propiedad *Text*, junto con una caja donde el usuario puede pulsar para marcar o desmarcar la opción.

La propiedad *Value* indica si el usuario ha marcado o no el *CheckBox*, y el evento *Change* informa de cada cambio. Se suele emplear para opciones de configuración que sólo disponen de dos posibles valores: 'Activado o Desactivado', 'Sí o No', 'Verdadero o Falso'.

- **RadioButton:** es similar a *CheckBox*, pero tiene la particularidad de que todos los *RadioButton* existentes, *hijos* de un mismo contenedor, están internamente agrupados, y en cada momento sólo puede haber uno activado. Cuando el usuario activa uno de ellos, el resto se desactiva, por lo que se emplea para seleccionar una opción que excluye a otras dentro de un menú de opciones.

- **PictureBox:** este control tiene la función de mostrar una imagen. Responde a eventos de ratón, por lo que se puede emplear como botón personalizado. Su propiedad *Stretch* permite adaptar la imagen al tamaño del *PictureBox* en cada momento, la propiedad *Border* determina su apariencia plana o con relieve y la propiedad *Picture* representa la imagen a mostrar.

Otros controles básicos misceláneos

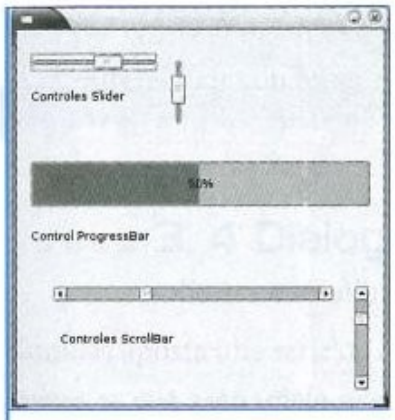


Figura 7. Otro tipo de controles.

Otros controles que pueden ayudar a diseñar una interfaz y tienen un propósito mucho más concreto son los que se presentan en la imagen de la izquierda.

- **ProgressBar:** barra de progreso que muestra un porcentaje de forma gráfica. Sirve para dar idea del avance de un proceso que dura mucho tiempo, de forma que el usuario no sienta que la aplicación está colgada mientras trabaja en segundo plano.

- **Slider:** es similar a *ProgressBar* en el sentido de que muestra un porcentaje, pero en este caso el usuario es quien varía su valor. Un buen ejemplo es su uso para subir o bajar el volumen en una aplicación que reproduzca audio. Los valores de volumen se definen entre un rango máximo y mínimo, y el usuario

lo cambia a su gusto. El evento *Change* señala un cambio por parte del usuario en el valor de la escala.

- **MovieBox**: a pesar de su sugerente nombre, no se trata de un reproductor multimedia, sino algo más humilde. Muestra una animación en formato GIF o MNG, sin que el programador deba preocuparse de la sucesión de frames del archivo que se muestra. La propiedad *Path* determina el archivo a reproducir, y *Playing* permite el control de la reproducción, con los valores *TRUE* (reproducir) o *FALSE* (detenido).

- **ScrollBar**: se trata de una barra de scroll para desplazar otro control, habitualmente, de forma que sea el usuario quien determine la posición de éste. La propiedad *Value* indica el valor elegido por el usuario, y el evento *Change* señala cada cambio.

A diferencia de otras interfaces gráficas, donde existen variantes horizontales y verticales para determinados controles, en Gambas los controles *Slider* y *ScrollBar* determinan su orientación automáticamente: si son más anchos que altos serán horizontales, y verticales en caso contrario.

Listas de datos

Existen tres controles diseñados para mostrar listas de diferentes modos:

1. **ListBox**: es una lista simple. Se añaden o eliminan elementos que se representan como una línea de texto cada uno. El usuario tiene capacidad para seleccionarlos o deseleccionarlos. La propiedad *Mode* determina si el usuario no puede seleccionar ninguno, sólo uno o varios.

2. **ListView**: similar a *ListBox*, dispone de capacidades adicionales. Puede representar un icono junto con cada elemento de la lista, y cada uno de ellos está identificado por una clave única de texto, que nos permite hacer búsquedas

de los elementos por su clave. Así mismo dispone de un cursor interno que puede moverse hacia adelante y hacia atrás, lo que lo hace apropiado para interactuar con una aplicación de bases de datos, donde puede representar un campo de una tabla.

3. **ComboBox**: es una lista desplegable. El usuario sólo ve el elemento seleccionado en cada momento y puede desplegar la lista para seleccionar uno u otro.

Otros controles avanzados

A continuación mostramos otros controles más avanzados:

- **TreeView**: sirve para representar elementos en un árbol, de forma que cada nodo puede tener otros nodos *hijos*.

- **ColumnView**: es similar al anterior, pero cada nodo puede disponer de varias columnas.

- **GridView**: sirve como representación de parrilla, de forma que disponemos de registros agrupados en filas y columnas. Es empleado, habitualmente, para interactuar con bases de datos.

3. 4 Diálogos

Gambas aporta una serie de diálogos auxiliares para mostrar o recabar información, interactuando con el usuario.

La clase Message

La clase *Message* se encarga de mostrar una ventana modal al usuario, en la cual podemos definir un texto, que será una información o una pregunta, y una serie de botones para elegir una opción. La clase *Message* es estática, y dispone de una serie de métodos para mostrar distintos tipos de mensajes, que serán reconocibles para el usuario gracias al icono que acompaña a la ventana y que da una idea del carácter del

mensaje. En estos métodos se situará siempre, como primer parámetro, el texto a mostrar y, a continuación, el texto de los botones, hasta un máximo de tres. Si no se indica el texto de los botones, aparecerá sólo un botón indicando OK para que el usuario acepte la lectura del mensaje.

- **Message.Info** (Texto, Boton1): se utiliza para mostrar un mensaje meramente informativo. Sólo permite definir un botón, que normalmente tendrá un texto tal como OK o Aceptar.

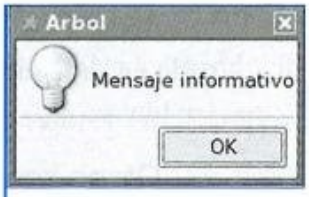


Figura 8. Mensaje informativo.

La clase *Message* también puede ser llamada como una función, de modo que el código:

```
Message.Info ("Mensaje")
```

Es equivalente a:

```
Message("Mensaje")
```


- **Message.Delete** (Texto, Boton1, Boton2, Boton3): se utiliza para indicar que se va a proceder a eliminar algo (archivo, registro de una tabla...), y se solicita al usuario su confirmación.

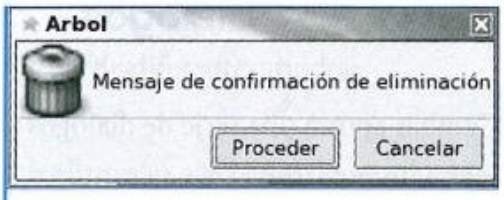


Figura 9. Mensaje de eliminación.

- **Message.Error** (Texto, Boton1, Boton2, Boton3): se emplea para indicar un mensaje de error.

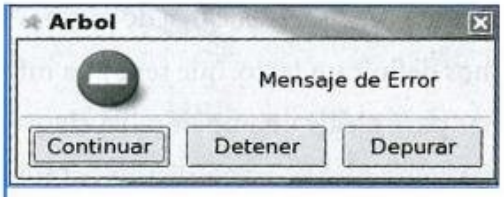


Figura 10. Mensaje de error.

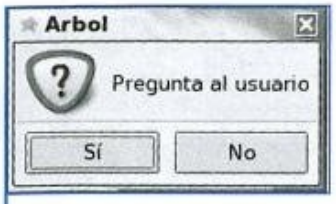


Figura 11. Mensaje para preguntar.

- **Message.Question** (Texto, Boton1, Boton2, Boton3): es una pregunta al usuario, generalmente para confirmar una acción o una opción de configuración.

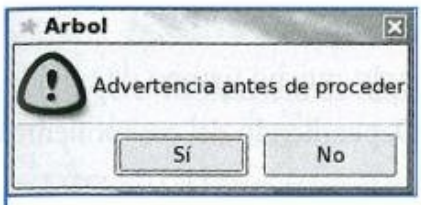


Figura 12. Mensaje de advertencia.

- **Message.Warning** (Texto, Boton1, Boton2, Boton3): advierte al usuario de que la acción que va a realizar supone un cierto peligro, por ejemplo, pérdidas de datos de una tabla que podrían ser útiles aún.

Los métodos de la clase *Message* devuelven un número entero que denota el botón que el usuario pulsó. El primer botón comienza en el número 1. Los mensajes son modales, lo que quiere decir que la interacción de la interfaz de usuario con el programa, así como el flujo de éste, quedan bloqueados hasta que se pulse uno de los botones.


```
...
DIM hRes AS INTEGER
hRes=Message.Warning("¿Formatear el disco duro?",
"Sí","No")
IF hRes=1 THEN Formatea_Disco()
...
```

Figura 13. Cuadro de diálogo con botón de cerrar.

Dependiendo del gestor de ventanas del sistema, es posible que los cuadros de diálogo tengan un botón de cerrar. Si el usuario cierra el mensaje de este modo, se devolverá el número del botón existente más alto (en el ejemplo anterior el 2), por tanto la opción menos peligrosa, o que se estima a ejecutar por defecto, se habrá de indicar en el botón más alto.

- **Dialog.Filter**: permite indicar filtros para los archivos a mostrar. Se trata de una matriz de cadenas en la que podemos situar, por ejemplo, las extensiones de los archivos a elegir, usando comodines si lo deseamos.

- **Dialog.Title**: permite establecer un título para la ventana, que por defecto corresponderá a la acción a realizar (*Select Font*, *Select Color*, etc.).

```
...
Dialog.Title = "Seleccione imágenes a procesar"
Dialog.Filter = ["*.png", ".jpg"]
IF Dialog.OpenFile(TRUE) THEN
  Message.Info("Acción cancelada")
ELSE
  Procesa_Imagenes(Dialog.Paths)
END IF
...
```


Diálogos personalizados

Al margen de los diálogos ya mencionados, el programador puede crear otros personalizados. Cuando un formulario se muestra de forma *Modal*, es decir, con los métodos *ShowModal()* o *ShowDialog()*, puede devolver un valor entero, que sirve como indicación de la opción elegida por el usuario. Veamos un pequeño ejemplo.

Creamos un programa gráfico, con un formulario principal de inicio llamado **FMain** y otro formulario llamado **FDialogo**. Creamos también tres pequeños iconos en formato *png* o los copiamos de la carpeta */usr/share/pixmaps* del sistema, y los renombramos como *a.png*, *b.png* y *c.png*, de forma que los tengamos disponibles en la carpeta del programa. El formulario principal **FMain** tendrá un *PictureBox* llamado **pImage**, y un botón denominado *btnSelect* con el texto **Icono**. El formulario **FDialogo** dispondrá de tres controles

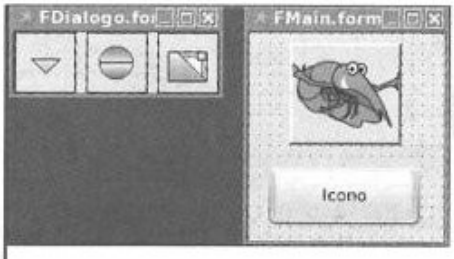
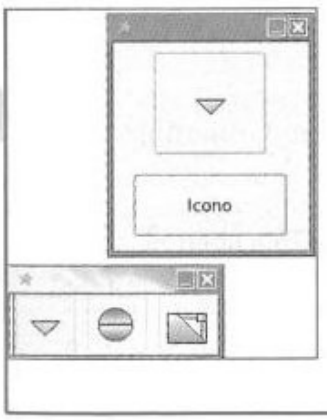


Figura 19. Formularios **FMain** y **FDialogo**.

PictureBox, cada uno de ellos conteniendo uno de los iconos *png* que habíamos situado en el proyecto, y los llamados **pic1**, **pic2** y **pic3**.

Con el código del formulario **FDialogo** lo que conseguiremos es que cada vez que el usuario pulse uno de los controles *PictureBox*, el formulario, que se mostrará de forma modal, devuelva un valor entero que identifquea el icono pulsado.

```
PUBLIC SUB pic1_MouseDown()

  ME.Close(1)

END

PUBLIC SUB pic2_MouseDown()

  ME.Close(2)

END

PUBLIC SUB pic3_MouseDown()

  ME.Close(3)

END
```



En cuanto al formulario **FMain**, la pulsación del botón conducirá a mostrar el formulario **FDialogo** de forma modal y, en función del valor devuelto, situará una imagen u otra en el *PictureBox* llamado **pImage**:

Figura 20. Imágenes que hemos situado en el *PictureBox*.

```
PUBLIC SUB BtnSelect_Click()

  SELECT CASE FDialogo.ShowDialog()

    CASE 1
      pImage.Picture = Picture["a.png"]

    CASE 2
      pImage.Picture = Picture["b.png"]

    CASE 3
      pImage.Picture = Picture["c.png"]

  END SELECT

END
```


Tras ejecutarlo, podremos comprobar el resultado. Si el usuario cierra el formulario modal pulsando el aspa del gestor de ventanas, se devolverá el valor por defecto, es decir, cero, lo que equivale a cancelar la selección. Puede plantearse otro problema más complejo: la necesidad de devolver otro tipo de valores, tales como cadenas o referencias a objetos. En este caso, la solución del entero no es válida. Si tratamos de mantener una cadena en una variable del formulario, ésta se liberará tras cerrar el formulario, lo cual no nos sirve. Por ejemplo, modificando el código anterior, de la siguiente manera, para que el formulario **FDialogo** mantenga una cadena con el valor elegido:

```
PUBLIC Valor AS String

PUBLIC SUB pic1_MouseDown()

  Valor = "a.png"
  ME.Close()
```

```
END

PUBLIC SUB pic2_MouseDown()

  Valor = "b.png"
  ME.Close()

END

PUBLIC SUB pic3_MouseDown()

  Valor = "c.png"
  ME.Close()

END
```

Y modificando el formulario principal para que tome el valor de la cadena:

```
PUBLIC SUB BtnSelect_Click()

  FDialogo.ShowDialog()
  IF FDialogo.Valor <> "" THEN
    pImage.Picture = Picture[FDialogo.Valor]
  END IF

END
```

No conseguiremos que funcione, ya que la secuencia es la siguiente:

1. La llamada a **FDialogo.ShowDialog()** crea una instancia de la clase *FDialogo*.

2. Tras la pulsación del usuario, se destruye esa instancia y con ella el valor almacenado en la variable pública *Valor*.

3. Al llamar a **FDialogo.Valor**, se crea otra instancia de *FDialogo*, que tiene la variable *Valor* con su valor por defecto (cadena vacía).

Podemos solventar este problema por dos caminos:

1. El primero consiste en declarar la variable *Valor* como estática. Así la variable no depende de cada instancia, si no de la clase, de forma que no se crea ni se destruye en cada llamada a *FDialogo.ShowModal()*.

2. El segundo consiste en aprovechar la propiedad *Persistent* de los formularios. Poniendo su valor a *TRUE*, un formulario no se destruye cuando el usuario lo cierra pulsando el aspa del gestor de ventanas, ni se llama al método *Close()*, si no que simplemente se oculta. Y si estaba en pantalla de forma modal, el programa abandona este modo y continua su ejecución normal.

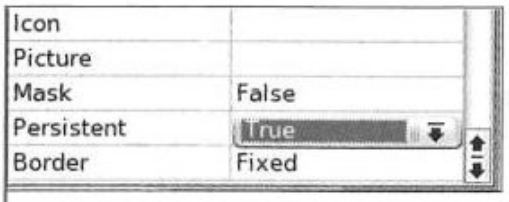


Figura 21. Propiedad *Persistent* con valor *TRUE*.

Así pues, en tiempo de diseño situaremos la propiedad *Persistent* del formulario **FDialogo** a *TRUE*, y modificaremos el código del formulario **FMain** para que destruya el formulario de forma explícita tras haber leído el valor que interesaba.

```
PUBLIC SUB BtnSelect_Click()

  FDialogo.ShowDialog()
  IF FDialogo.Valor <> "" THEN
    pImage.Picture = Picture[FDialogo.Valor]
  END IF

  FDialogo.Delete()

END
```

3. 5 Menús

Figura 22. Localización del **Editor de menú**.

La creación de menús es realmente sencilla ya que un asistente del IDE permite diseñarlos. Tan sólo hay que situarse sobre un formulario, pulsar el botón derecho, y seleccionar la opción **Editor de menu...**

Los menús se crean como árboles, es decir, cada menú de primer nivel, por ejemplo los típicos menús de la barra superior de muchos programas como **Archivo**, **Editar**, **Ver**, **Ayuda**, etc., tendrán menús *hijos* que se sitúan un nivel más profundo que éste y, a su vez, si estos tienen *hijos*, se situarán un nivel más profundo.

Todo ello se controla con los botones con forma de flecha, las verticales permiten cambiar el orden de aparición de los menús, y con las horizontales modificamos la profundidad de estos.

Las propiedades más importantes son el *Nombre*, que es el nombre del objeto menú y que corresponderá con su gestor de eventos, el *Título*, que es el texto que aparecerá en la pantalla, un icono a elegir si lo deseamos, y un posible atajo de teclado para acceder sin necesidad del ratón.

Figura 23. Ejemplo de menús con submenús.

Si el nombre de un menú se deja en blanco, éste se muestra como una barra separadora en lugar de una entrada de menú normal.

En la **Figura 23** podemos ver un ejemplo con un menú principal que tiene tres opciones, y sus correspondientes submenús.

El formulario tendrá el aspecto de la **Figura 24**. Si en tiempo de diseño pulsamos sobre uno de los menús que no tienen *hijos*, el IDE nos llevará directamente al evento *Click* del menú, que es donde podemos crear el código que se ejecutará cuando el usuario pulse el menú.

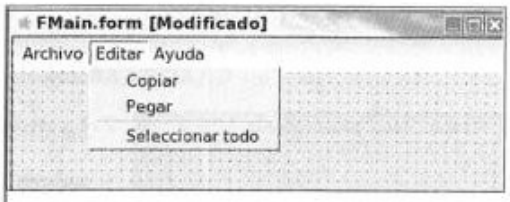


Figura 24. Formulario *FMain*.

```
PUBLIC SUB mnuSeleccionar_Click()

    ...

END
```

Si lo que deseamos es crear un menú de los que se muestran cuando el usuario pulsa, por ejemplo, el botón derecho sobre un formulario u otro control, tendremos que crear un menú de primer nivel con su propiedad visible a *FALSE*, y sus correspondientes *hijos*. Después, al lanzarse el evento del formulario o control que nos interesa, por ejemplo, la pulsación del botón derecho sobre un formulario, que se detecta mediante la gestión del evento *Menu*, indicaremos a nuestro menú invisible que debe mostrarse como un menú contextual:

Figura 25. Menú que vemos al pulsar el botón derecho de un formulario.

```
PUBLIC SUB Form_Menu()

  mnuEditar.Popup()

END
```

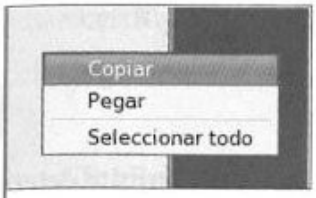


Figura 26. Resultado del menú contextual.

Al ejecutar el código, veremos el resultado al pulsar el botón derecho sobre el formulario.

En cuanto a la naturaleza de los menús, no son más que objetos, aunque en este caso no provienen de la clase *Control*, si bien disponen de algunas propiedades comunes como *Text* o *Picture*.

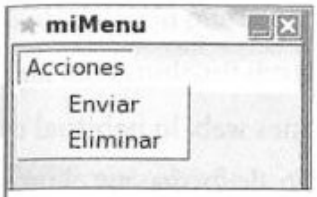


Figura 27. Acciones del menú creado.

A la hora de crear un menú, habrá de indicarse su objeto *padre*, que podrá ser una ventana o formulario para los menús de primer nivel, u otro menú para los *hijos* del primero. Los menús se pueden crear o destruir también directamente por código:

```
PUBLIC SUB Form_Open()


    DIM h1 AS Menu
    DIM h2 AS Menu
    DIM h3 AS Menu


    h1 = NEW Menu(ME)
    h1.Text = "Acciones"

    h2 = NEW Menu(h1) AS "h2"
    h2.Text = "Enviar"

    h3 = NEW Menu(h1) AS "h3"
    h3.Text = "Eliminar"


END
```

3.6 Alineación de los controles

Propiedades de alineación

Algunos contenedores disponen de una propiedad *Arrangement* que permite determinar cómo se alinean los controles dentro de un contenedor. Por defecto, el valor de la propiedad es *None*, lo que significa que las posiciones de los controles son libres, como es típico en las interfaces para Windows™ o en algunas librerías como QT.

Sin embargo, en otras librerías gráficas como GTK+, o en las definiciones del lenguaje XUL diseñado por el proyecto Mozilla para aplicaciones web, lo habitual es encontrar contenedores que definen dónde se situará cada *hijo*, de forma que el programador sólo ha de indicar el modo general de alineamiento y los controles se adaptarán en todo momento a dicha alineación.

En principio, diseñar interfaces de esta manera puede ser algo complicado para gente sin experiencia, no obstante, una vez que se toma algo de pericia, aporta grandes ventajas. La principal es que el diseño de ventanas redimensionables con controles variados en su interior es trivial.

Cada usuario puede agrandar o hacer más pequeña la ventana, o variar su relación alto/ancho, y la aplicación seguirá manteniendo un aspecto coherente en cada momento, dentro de unos límites razonables de tamaño.

Gambas define varias posibilidades de alineamiento para los controles:

- **None**: alineamiento libre, el contenedor no decide nada acerca de la posición de sus *hijos*.

- **Horizontal**: todos los controles se alinean de izquierda a derecha, ocupando todo el espacio en vertical dentro del contenedor.

- **Vertical**: todos los controles se alinean de arriba a abajo, ocupando todo el espacio en horizontal dentro del contenedor.

- **LeftRigth**: los controles tratan de alinearse de izquierda a derecha, y si falta espacio, de arriba a abajo.

- **TopBottom**: los controles tratan de alinearse de arriba a abajo, y si falta espacio, de izquierda a derecha.

Además de la propiedad general *Arrangement*, se aporta la propiedad *Padding*, que es un espacio que queda libre en el borde del contenedor, y una propiedad *Spacing*, que determina un espacio de separación entre control y control.

Cada control, por su parte, dispone de la propiedad *Expand*. Si el control se sitúa sobre un contenedor cuya propiedad *Arrangement* es distinta de *None*, el valor *Expand* determina si éste, junto con el resto de controles del contenedor que tengan la propiedad *Expand* a *TRUE*, tratarán de ocupar el espacio libre que quede dentro del contenedor.

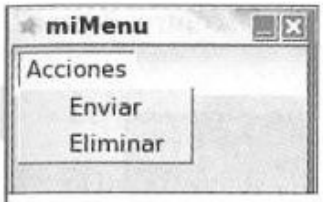


Figura 28. Proyecto **Alineación**.

Para comprobar el efecto de todas estas propiedades, creamos un nuevo proyecto gráfico llamado **Alineacion**, con un sólo formulario de inicio *FMain*, en cuyo interior situaremos dos botones y un *RadioButton*. Tras ejecutarlo veremos el resultado habitual: un formulario con tres controles algo desordenados.

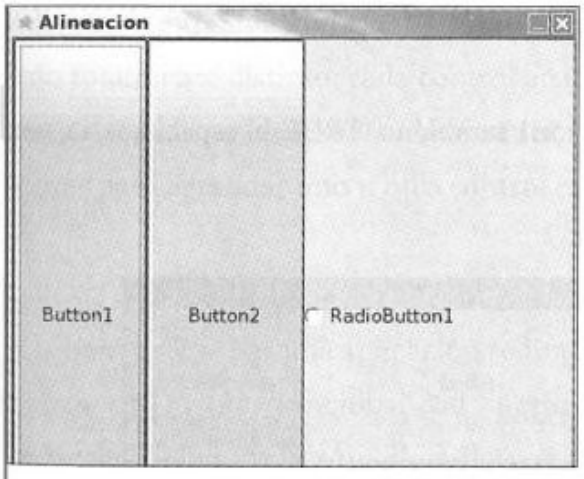


Figura 29. Controles alineados en horizontal.

Cambiamos el valor de la propiedad *Arrangement* a *Horizontal* y ejecutamos el programa.

Ahora los controles se han situado alineados en horizontal, ocupando todo el espacio vertical del contenedor (**Figura 29**).

Ponemos la propiedad *Border* del formulario a *Resizable*, de modo que podamos variar su tamaño. Hacemos varias pruebas de ejecución cambiando su relación altura/anchura.

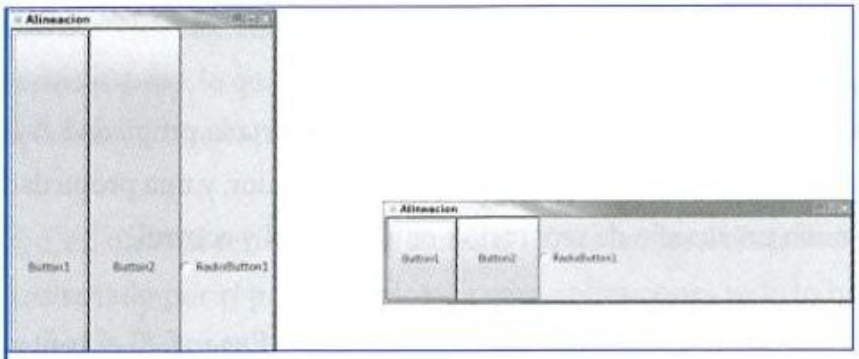


Figura 30. Variación del formulario según altura/anchura.

Como podemos observar, los controles siguen ocupando todo el ancho del contenedor, mientras que el ancho extra queda libre. Situamos ahora la propiedad *Expand* del control **Button2** a *TRUE*, y volvemos a ejecutarlo. Con esta nueva configuración, todo el espacio del contenedor es aprovechado, modificando el ancho del control **Button2**.

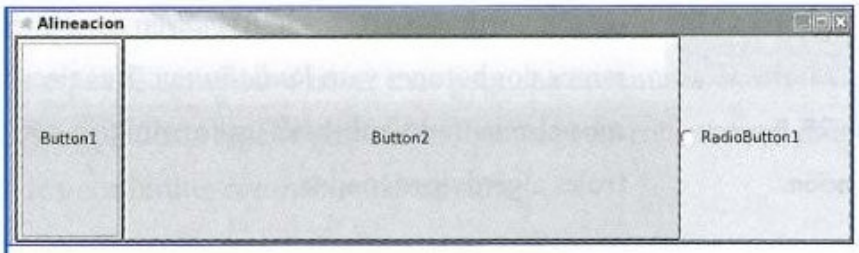


Figura 31. Aprovechamiento del espacio del contenedor.

Si situamos la propiedad *Expand* de **Button1** también a *TRUE*, el espacio extra será compartido por ambos controles.

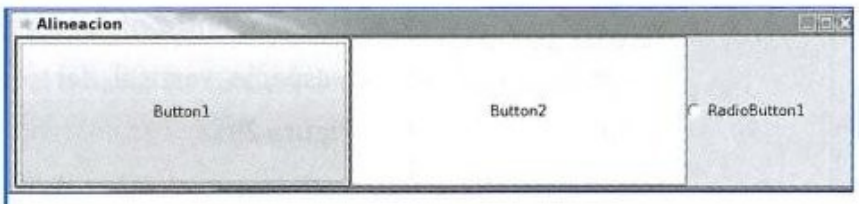


Figura 32. Espacio compartido entre los controladores **Button1** y **Button2**.

Para dejar espacio visible por el borde del contenedor, podemos dar un valor a la propiedad *Padding*, y para separar un poco cada control utilizaremos la propiedad *Spacing*. Los valores indicados son píxeles de separación.

El efecto de la propiedad *Arrangement* no se aprecia en tiempo de diseño, de este modo si por error se cambia a un valor no deseado, podremos revertirlo sin perder las posiciones de cada control dentro del contenedor.

Controles con alineación predefinida

Los controles *Form* (Window), *Panel*, *TabStrip* y *ScrollView* permiten definir las propiedades de alineación, sin embargo, otros controles tienen esta propiedad implícita y no es modificable. Estos son los siguientes:

- **Hbox**: es un panel que siempre tiene alineación horizontal.
- **Vbox**: es un panel con alineación vertical.
- **Hpanel**: sigue el modelo *RightToLeft*.
- **Vpanel**: sigue el modelo *Vpanel*.

Además de estos, los controles *Hsplit* y *Vsplit* son contenedores con un modo de trabajo totalmente distinto: cada control añadido se muestra separado por una barra vertical, en el caso de *Hsplit*, u horizontal, en el caso de *Vsplit*, que el usuario puede mover para agrandar uno u otro control en detrimento del tamaño de su vecino.

Diseño de una aplicación que aprovecha este recurso

A la hora de diseñar una aplicación redimensionable, lo mejor es plantear áreas de trabajo con distinta funcionalidad y agruparlas en distintos paneles horizontales y verticales. Supongamos un clon de los exploradores de archivos habituales. Una zona de trabajo estará formada los típicos botones de menú, que permiten realizar

las tareas más comunes y que presentan un icono y un tooltip explicativo. Otra zona de trabajo puede ser la parte inferior, en la que se muestran datos de estado. Por último, la zona central muestra los archivos y, a su vez, es una zona de trabajo que comprende otras dos: un árbol con las carpetas a la izquierda, y una zona más grande a la derecha con la vista en detalle de los archivos. No crearemos en este ejemplo el código correspondiente, pero sí seguiremos los pasos necesarios para crear la interfaz de un modo adecuado, para conseguir que cada usuario pueda disponer de sus ventanas como mejor lo desee.

En primer lugar, hay tres grandes grupos de trabajo, que van de arriba abajo. Por tanto, lo mejor es definir un formulario con la propiedad *Arrangement* situada a *Vertical*.

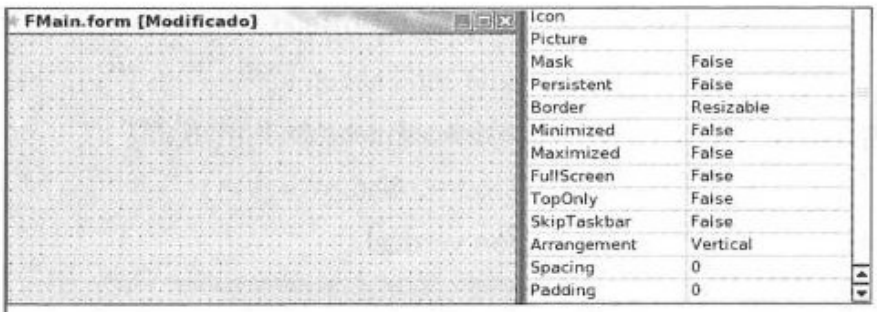


Figura 33. Formulario con la propiedad *Arrangement* a *Vertical*.

Dentro de ésta, situaremos tres paneles con alineación horizontal, es decir, tres contenedores *Hbox*. Deseamos que la parte superior con los botones y la inferior con la barra de estado, tengan un ancho fijo, pero la parte central que contiene el cuerpo de la información útil del programa, será redimensionable. Por tanto, la propiedad *Expand* del panel central habrá de tener el valor *TRUE*.

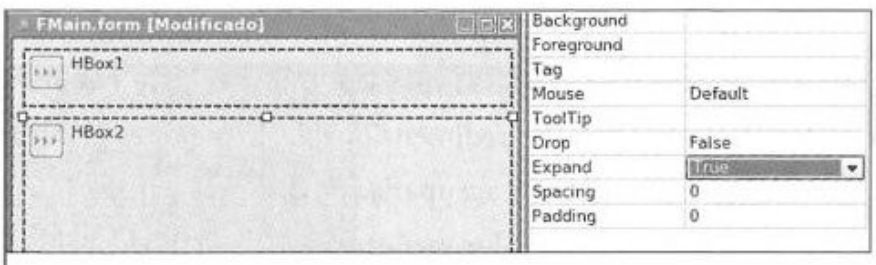


Figura 34. Paneles *Hbox* con alineación horizontal.

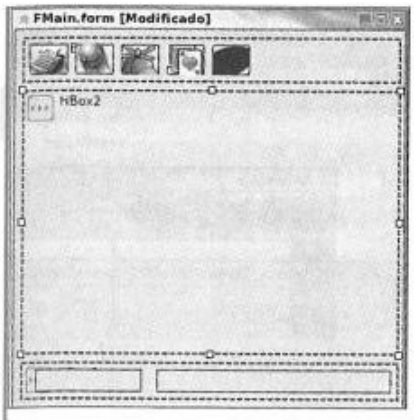


Figura 35. Diseño del formulario **FMain**.

La parte superior contendrá diversos botones tipo *ToolButton*, que podemos resaltar con distintos iconos. La parte inferior dispondrá de dos etiquetas *Label* con borde *Sunken*, una de las cuales, según qué parte se quiere hacer extensible, tendrá su propiedad *Expand* a *TRUE*.

Figura 36. Botones auxiliares del formulario.

Dentro del cuerpo se situará una caja vertical a la izquierda (*Vbox*) con otros botones auxiliares. Es el mismo diseño que la zona de botones principal, pero alineada en vertical.

Figura 37. Introducción del control *Hsplit*.

Tras esto, situaremos un contenedor *Hsplit*, de modo que el usuario pueda disponer de una barra para modificar el tamaño relativo del árbol y la zona principal de trabajo. Dentro de él se colocará, a la izquierda, el control *TreeView* para el árbol, así como el control *IconView* para la zona principal.

El control *Hsplit* debe tener la propiedad *Expand* a *TRUE*, para que aproveche todo el espacio libre disponible dentro de su contenedor.

Ya podemos ejecutar el programa para ver la interfaz, añadiendo, si lo deseamos, algo de código para rellenar el árbol y la vista de iconos.

Podemos modificar el alto y ancho de la aplicación, la cual se encargará de mantener, en todo momento, la relación de los distintos controles, sin que sea necesario añadir ningún código de cálculo de posiciones por nuestra parte.

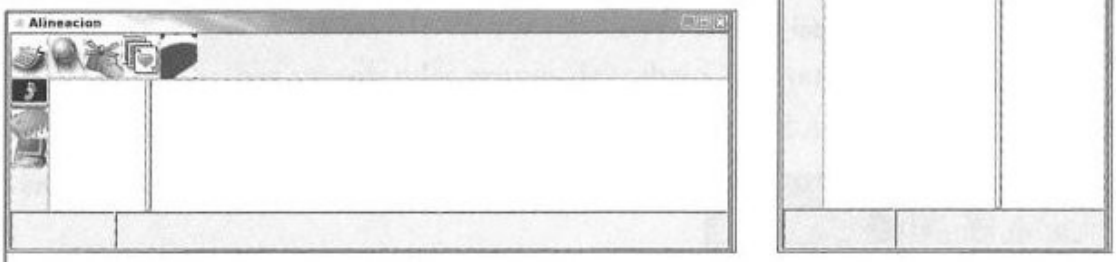


Figura 38. Modificación del alto y ancho de la aplicación **Alineción**.

3.7 Introducción al dibujo de primitivas

Además de los controles ya diseñados, el programador puede necesitar dibujar gráficos personalizados. La clase estática *Draw* se emplea para dibujar sobre un control, que puede ser un formulario o el control *DrawingArea*.

Dibujar directamente sobre un formulario no tiene demasiado sentido, ya que una vez se refresque la interfaz, por ejemplo tras pasar otra ventana por encima o minimizar y maximizar, se pierde el dibujo realizado sin que tengamos control sobre la situación.

Sin embargo, *DrawingArea* dispone de dos modos de trabajo: en el primero, como en el caso del formulario, se pierde el dibujo una vez se refresca el control, pero el evento *Refresh* nos informa para que redibujemos la parte eliminada; en el segundo, que se activa poniendo la propiedad *Cached* a *TRUE*, el control guarda una caché del dibujo realizado y no genera eventos *Refresh*, sino que redibuja automáticamente la zona clareada.

Figura 39. Proyecto **Dibujando**.

Como ejemplo, crearemos un proyecto gráfico **Dibujando**, que contenga un formulario *FMain*, y en su interior un control *DrawingArea* llamado **Lienzo** y un botón llamado **Dibujar**.

La propiedad *Cached* del control *DrawingArea* se pondrá a *TRUE*.

El código del botón **Dibujar** será el siguiente:

```
PUBLIC SUB Dibujar_Click()

  Draw.Begin(Lienzo)
  Draw.Line(0, 0, Lienzo.W, Lienzo.H)
  Draw.Line(0, Lienzo.H, Lienzo.W, 0)
  Draw.End()

END
```

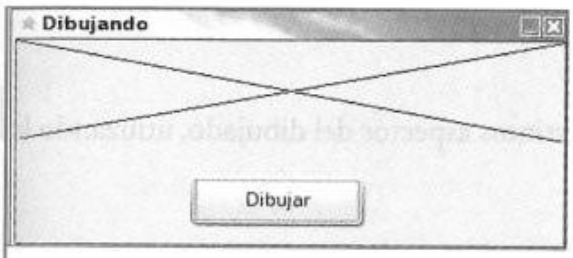



Figura 40. Resultado de nuestro código.

Al ejecutar el programa y pulsar el botón, aparecerá un aspa creada por nuestro código.

Este código contiene, para cualquier dibujo, el método de trabajo siguiente:

- En primer lugar, se ha de especificar a la clase *Draw* qué control será el empleado para dibujar los elementos que indiquemos, para lo cual se emplea el método **Draw.Begin**() pasando como parámetro el control deseado, en nuestro caso el control *Lienzo*.

- Tras esto, pasamos a dibujar las primitivas que deseamos. Aquí hemos empleado el método **Draw.Line()**, el cual dibuja líneas cuando especificamos los puntos de origen y destino.

- Finalmente, siempre hay que llamar al método **Draw.End()** para que la caché del control *DrawingArea* se dibuje en la pantalla, y se liberen los recursos asociados al proceso de dibujado del control.

La clase *Draw* permite dibujar distintos tipos de primitivas:

- **Draw.Ellipse**: elipses.

- **Draw.Line**: líneas.

- **Draw.Point**: puntos.

- **Draw.Polyline**: varias líneas enlazadas.

- **Draw.Polygon**: polígonos.

- **Draw.Rect**: rectángulos.

En todo momento podemos controlar distintos aspectos del dibujado, utilizando lo siguiente:

- **Draw.BackGround**: color de fondo del pincel.

- **Draw.ForeGround**: color de primer plano del pincel.

- **Draw.FillColor**: color para relleno de elipses o rectángulos.

- **Draw.FillStyle**: utiliza las constantes de la clase *Fill* para determinar el patrón de dibujado (relleno, rayado en horizontal, líneas y puntos, etc.).

Además de las primitivas, podemos dibujar elementos complejos con:

- **Draw.Text**: dibuja un texto en una posición indicada y con la fuente seleccionada por *Draw.Font*.

- **Draw.Image**: dibuja un gráfico almacenado en un objeto *Image*.

- **Draw.Picture**: dibuja un gráfico almacenado en un objeto *Picture*.

Por último, **Draw.Clip** permite seleccionar una área de *Clipping*, es decir, reducir el área útil de dibujo a un rectángulo que especifiquemos, de modo que todo trazo que quede fuera de él se excluirá de ser realmente dibujado.