The fpGUI widget set for Free Pascal

Michaël Van Canneyt

March 31, 2008

Abstract

When designing a GUI application, there is a wealth of GUI toolkits (widget sets) that one can use, with various levels of abstraction. Some are cross-platform, some are not. There are few, however, that are implemented completely in Object Pascal. fpGUI is a relatively new widget set that is implemented 100% in object pascal, and in this article we'll have a closer look at it.

1 Introduction

Writing a GUI application would be a very difficult task if one had to use the native windowing system of the operating system. Therefor, GUI toolkits or widget sets have been created, that hide the gory details of GUI programming. The widget sets provide commonly a set of pre-defined widgets (buttons, edit fields, bitmaps, comboboxes, treeviews) that one sees in most programs: these widgets are easy to manipulate, and are usually highly customizable by means of some specialized calls.

When creating GUI (graphical) applications in Object Pascal, one has a lot of options. Depending on the required functionality, one has the choice of many GUI toolkits:

Native Windows

Carbon

GTK (version 1 or 2)

Qt

Lazarus

MSEGui

PasGF (formerly LPTK)

fpGUI

The first one will only work on Windows, unless one wishes to use WineLib on unixes. Carbon works only on Mac. Both Qt and GTK are geared towards C or C++, and require large support libraries. Lazarus uses whatever of these 4 is available, to create a native look on all platforms using a single codebase, as much as possible compatible with the VCL from Delphi.

The last three toolkits: MESGui, PasGF or fpGUI aim to write a toolkit that works on all platforms, and which presents a consistent interface on these platforms: an application running on one platform will look and behave identical when run on another platform. Additionally, the application should need as little as possible extra libraries.

This is a distinctly different goal than e.g. Lazarus uses. This is also the reason that a new toolkit was written: 2 of the design targets were:

1. Uniform look on all platforms.
2. As little dependencies on extra libraries as possible.

fpGUI aims to fullfill these requirements. It currently works on Windows and Linux (any BSD flavour should work too, but this is untested currently), 32-bit and 64-bit.

While in its current incarnation fpGUI is rather new (9 months old), the history of fpGUI goes back to almost pre-Lazarus times; it was then developed as an alternative to Lazarus' LCL, but later abandoned. Little over a year ago, Graeme Geldenhuys revived the codebase, and has been maintaining it ever since. It is in use in commercial products, which should guarantee its stability, and the continued development of the toolkit.

2 Installation

fpGUI can be downloaded from its homepage:

```
http://opensoft.homeip.net/fpgui/
```

The latest sources can always be downloaded from the Subversion repository at:

```
https://fpgui.svn.sourceforge.net/svnroot/fpgui/trunk
```

The distribution has a simple structure, with a few main directories:

src the sources of the whole library, with 2 subdirectories: corelib (the graphics layer) and gui (the actual widgets).

examples many sample applications.

docs documentation in fpDoc format

extras this contains extra components, notably support for tiOPF mediators.

images with the default used images.

lib the compiled units end up here.

Compilation is quite simple: in the src directory, a build.bat file is provided for windows, and a build.sh shell script in for Linux/Unix. The compiled units end up in the lib. Most examples (and indeed, the library itself) gace a extrafpc.cfg file, which contains all options needed to compile the sources: it contains, for instance, a reference to the lib directory.

3 Architecture

To accomplish its design goals, fpGUI is built in 2 layers:

1. A low-level graphics layer: this layer interacts with the graphical subsystem of the operating system (GDI on Windows, X-Windows on unix-like systems), and handles drawing and communication of events (keystrokes, mouse clicks and so on). This part is implemented in the fpGfx unit. This system also contains a small set of OS-abstraction functions.

2. A widget set with actual widgets. It relies only on the functions in the graphics layer, and does not access the underlying graphical system.

This design ensures that if fpGUI must be ported to a new platform (DOS, Carbon, the linux framebuffer device), only the first layer needs to be implemented: the second layer will work as it was.

The graphics layer introduces the following important classes:

tfpgWindow This encapsulates a graphical window, which means it receives messages from the operating system windowing system, and can be used to draw upon.

tfpgCanvas This encapsulates the drawing routines which can be used to draw on a window.

tfpgImage This represents bitmaps that can be drawn on a canvas.

TfpgApplication This object encapsulates the event loop of the application and the connection with the windowing system.

It is possible to create applications using simply these low-level graphics layer classes; This would be much like creating an application directly using the OS provided calls, with the difference that `fpGfx` is cross-platform.

To show this, we'll implement a 'Hello World!' application using the drawing layer only. to do this, a simple descendent of `TfpgWindow` is implemented, called `TMainWindow`.

```
TMainWindow = class(TfpgWindow)
Protected
  procedure MsgPaint(var msg: TfpgMessageRec); message FPGM_PAINT;
  procedure MsgClose(var msg: TfpgMessageRec); message FPGM_CLOSE;
  procedure MsgResize(var msg: TfpgMessageRec); message FPGM_RESIZE;
public
  constructor Create(AOwner: TComponent); override;
end;
```

As can be seen from this code, the Tfpg prefix is used in all types declared in fpGUI, and FPG is used as a prefix for constants.

Note the use of the message directive: it is used to dispatch messages to the correct method: the event loop implementation in `TfpgApplication` relies on this. There are more messages a window can respond to, but the above ones suffice for our implementation. The complete list of messages as supported by fpGFX is defined in the `GFXbase` unit.

The constructor is rather simple. It sets up the window, and shows it.

```
constructor TMainWindow.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FWidth    := 350;
  FHeight   := 200;
  WindowAttributes := [waSizeable, waScreenCenterPos];
  AllocateWindowHandle;
  DoSetWindowVisible(True);
  SetWindowTitle(HelloWorldDemo);
end;
```

It sets the initial height and width of the window, and specifies some window attributes, namely: the window should be sizeable, and should initially appear at the center of the screen. There are other attributes (stay on top, full screen etc), the `TWindowAttribute` enumerated type is defined in gfxBase.

After this, the operating system window handle is allocated with `AllocateWindowHandle`. This is necessary to make the window visible, and to set the title: these things cannot be done if no window handle is present.

In the constructor, the window was created. However, nothing is drawn on this window. Whenever the operating system wants the window to be drawn (for instance, when it is made visible), it sends a paint message to the window. The paint message is caught by fpGUI, which will then call the paint message handler, to do the actual painting. The painting routine could be coded as follows:

```
procedure TMainWindow.MsgPaint(var msg: TfpgMessageRec);

var
  r: TfpgRect;

begin
  Canvas.BeginDraw;
  try
    r.SetRect(0, 0, FWidth, FHeight);
    Canvas.Color:=clGray;
    Canvas.FillRectangle(r);
    Canvas.Font := fpgGetFont('Arial-30');
    Canvas.SetTextColor(clBlack);
    Canvas.DrawString((Width - Canvas.Font.TextWidth(HelloWorld)) div 2 + 1,
      (Height - Canvas.Font.Height) div 2 + 1, HelloWorld);
    Canvas.SetTextColor(clWhite);
    Canvas.DrawString((Width - Canvas.Font.TextWidth(HelloWorld)) div 2 - 1,
      (Height - Canvas.Font.Height) div 2 - 1, HelloWorld);
  finally
    Canvas.EndDraw;
  end;
end;
```

Each window has a `Canvas` property, of type `TfpgCanvas`, which offers a lot of methods to draw on the window surface. All painting should be done in a pair of calls to the `BeginDraw`/`EndDraw` methods of the canvas: this allows to optimize the output and perform double buffering if need be.

For the 'Hello, World!' application we first draw the background in the gray color; This means simply filling a rectangle having the size of the form with the gray color. Then, to write the text, a font object is requested. The `fpgGetFont` function returns a font object with the requested font: the parameter is a string representation of the font, as X11 handles it:

```
FaceName-Height:Prop=Value[:Prop=Value]
```

Where `Prop` is one of `Bold`,`Italic`,`Antialias`, and `Value` can be `True` or `False`. The code above requests an Arial font at height 30, and then draws a shaded 'Hello world' text: it simply draws the text twice, once in black, once in White, with the second text shifted 2 pixels both horizontally and vertically.

fpGUI has a nice mechanism where a set of stock fonts can be defined. These font definitions can always be retrieved to by `fpgGetNamedFont`. By default, the following fonts are defined:

Label1 standard font for labels.

Label2 the same as Label1, but bold.

Edit1 A font used in edit controls.

Edit2 A fixed-width font for use in edit controls.

List A font for listboxes.

Grid A font for data cells in grids.

GridHeader A font for header cells in grids.

Menu A font for menu items.

MenuAccel the font for indicating the shortcut keys in menus.

MenuDisabled the font for a disabled menu item.

It is possible to define your own fonts with `fpgSetNamedFont`, which takes a name and a description string as it's arguments. A similar system exists for colors.

The window was declared sizeable with the `waSizeable` windo attribute. When the user resizes the window, a `FPGM_RESIZE` message is sent to the window, and the parameter will contain the new size of the window. All that must be done with this message is to store the new size of the window:

```
procedure TMainWindow.MsgResize(var msg: TfpgMessageRec);
begin
  FWidth:=msg.Params.rect.Width;
  FHeight:=msg.Params.rect.Height;
end;
```

if need be (when the window is made bigger), the windowing system will send a paint message to request a redraw the window. The Params property of the message is a record that looks as follows:

```
TfpgMessageParams = record
  case integer of
    0: (mouse: TfpgMsgParmMouse);
    1: (keyboard: TfpgMsgParmKeyboard);
    2: (rect: TfpgRect);
end;
```

Depending on what message is being handled, one of the 3 structures is populated with the message data. In the case of a size message, the `rect` field is filled with the new size, and the fields of the `rect` record are used to update the size properties of the window.

Lastly, when the user clicks the close button on the window border, a `FPGM_CLOSE` message is sent to the window: the proper response is that the window handle should be freed. In the case of the hello world program, this is also a signal that the application should stop, as the window is the only window of the application. This is accomplished in the following method:

```
procedure TMainWindow.MsgClose(var msg: TfpgMessageRec);
begin
  ReleaseWindowHandle;
  fpgApplication.Terminated:=True;
end;
```

Setting the `Terminated` property of `TfpgApplication` signals that it should stop the message loop, and exit the `Run` method.

That's it. Our window class is ready to be used. All that remains to be done is write the main routine of our program:

```
begin
  fpgApplication.Initialize;
  TMainWindow.Create(fpgApplication);
  fpgApplication.run;
end.
```

The global `fpgApplication` variable contains an initialized version of the `TfpgApplication` class. The `Initialize` routine sets up the connection with the windowing system of the operating system. Note that this call can go wrong, for instance if the application cannot connect to an X server. After this, the main window is created - and it will be immediatly shown, as can be seen from the constructor code. After that the `Run` method of the global application instance will run the message loop, till the user clicks the close button. Since the `fpgApplication` instance owns the window, the TMainWindow instance will automatically be freed. The application should look something like figure 1 on page 7.

4 Widgets

While it is possible to create entire visual applications using the low level routines, it is tedious work, and apart from the fact that it is closs-platform, it is not much easier than using the OS calls directly. It is only when using the second layer of fpGUI - the widgets or controls, that it becomes interesting.

fpGUI currently does not offer the wealth of controls that e.g. lazarus offers, but it offers enough controls to easily make good-looking applications, as can be seen in table 1 on page 7. Almost each control is in it's own unit, which makes it easy to remember which units must be added to the uses clause of your file. On the other hand, it makes the uses clause quite long.

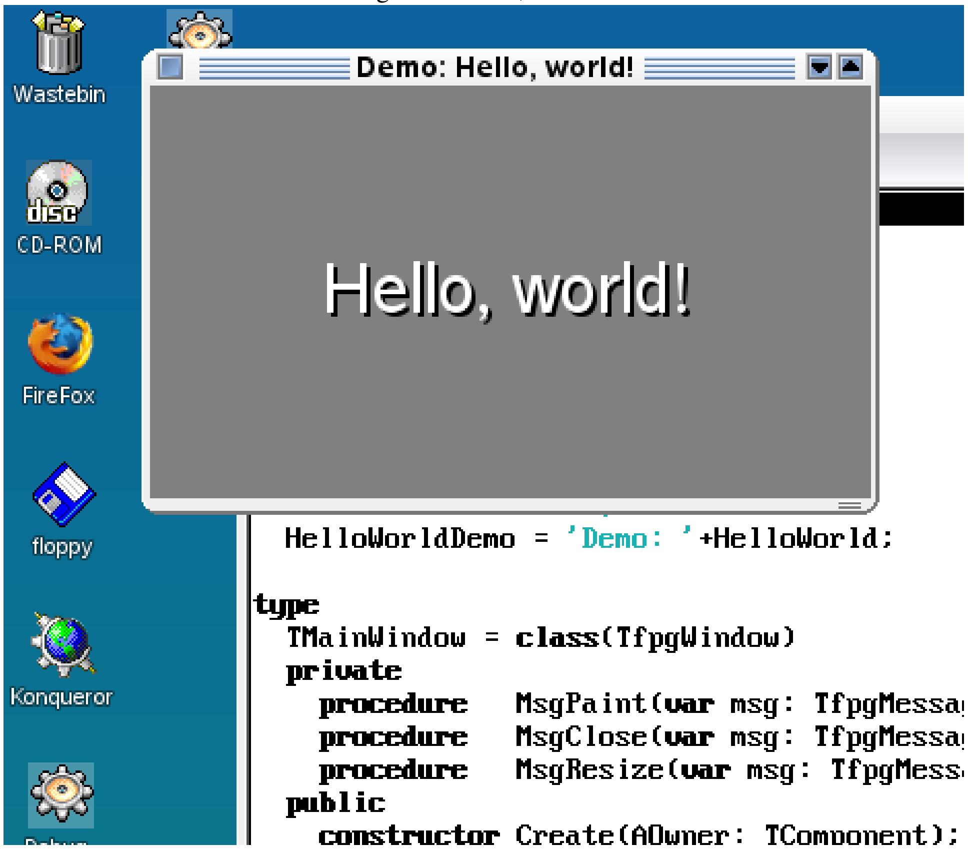
fpGUI is not geared towards streaming controls but instead assumes that all controls are created in code. This has an advantage: multiple forms can be declared in a single unit (although this may not be a good idea in practice).

The `TfpgForm` class is the ancestor for all forms - a descendent class must be created for each window in the application. In the constructor, or in the `AfterCreate` procedure, the form must be built in code: all controls that will appear on the form must be created.

For a 'hello, world!' application, this would lead to the following code:

```
type
  TMainForm = Class(TfpgForm)
  Private
    FLabel : TfpgLabel;
  Public
    Procedure AfterCreate; override;
```

Figure 1: Hello, World !



```
    HelloWorldDemo = 'Demo: '+HelloWorld;

type
  TMainWindow = class(TfpgWindow)
  private
    procedure   MsgPaint(var msg: TfpgMessa
    procedure   MsgClose(var msg: TfpgMessa
    procedure   MsgResize(var msg: TfpgMess
  public
    constructor Create(AOwner: TComponent);
```

Table 1: Available widgets

Control	unit	what
tfpgForm	gui_form	A form
tfpgButton	gui_button	buttons (and speedbutton)
tfpgCheckbox	gui_checkbox	checkboxes
TfpgEdit	gui_edit	Edit control, numerical edit.
tfpgListBox	gui_listbox	listboxes
tfpgProgressBar	gui_progressbar	Progress bar
tfpgTreeView	gui_tree	Treeview
tfpgCombobox	gui_combobox	combobox
tfpgGauge	gui_gauge	A gauge-like control
tfpgListView	gui_listview	a listview (report view only)
tfpgRadioButton	gui_radiobutton	A radio button
tfpgStringGrid	gui_grid	A stringgrid
tfpgMemo	gui_memo	A Memo
tfpgScrollbar	gui_scrollbar	A scrollbar
tfpgmenuBar	gui_menu	A main menu
tfpgPopupMenu	gui_menu	A popup menu
tfpgBevel	gui_bevel	Bevel
tfpgPageControl	gui_tab	Pagecontrol and tabsheets
tfpgLabel	gui_label	Label control, hyperlink label
tfpgTrackbar	gui_trackbar	Trackbar control
TfpgPopupCalendar	gui_popupcalendar	Popup calendar

```
  end;

Procedure TMainForm.AfterCreate;

begin
  WindowTitle:=HelloWorldDemo;
  SetPosition(100, 100, 350, 200);
  FLabel:=TfpgLabel.create(Self);
  FLabel.SetPosition(0,0,Width,Height);
  FLabel.Text:=HelloWorld;
  FLabel.Anchors:=[anTop,anLeft,anBottom,anRight];
  FLabel.FontDesc:='Arial-30';
  FLabel.TextColor:=clWhite;
  FLabel.BackgroundColor:=clGray;
  FLabel.Alignment:=taCenter;
  FLabel.Layout:=tlCenter;
  Show;
end;
```

Which is considerably smaller and more comprehensible than the code in the first version. Note that the `FLabel` field is a private field of the class: This means that it is possible to hide the internals of a form. In Delphi or lazarus, controls are always published fields of the form, exposing them to all other forms (units) in the application.

The code to run the application is exactly the same as the one for the low-level version.

Handling events in fpGUI is much as it is in Delphi's VCL or as in the LCL; The names of the events and their prototype are the same. To show this, the application is modified so that instead of a label, a button is displayed:

```
Procedure TMainForm.AfterCreate;

begin
  WindowTitle:=HelloWorldDemo;
  SetPosition(100, 100, 350, 200);
  FButton:=CreateButton(Self,5,5,Width-10,HelloWorld,@DoClick);
  FButton.Height:=Height-10;
  FButton.Anchors:=[anTop,anLeft,anBottom,anRight];
  FButton.FontDesc:='Arial-30';
  FButton.TextColor:=clWhite;
  FButton.BackgroundColor:=clGray;
  Show;
end;
```

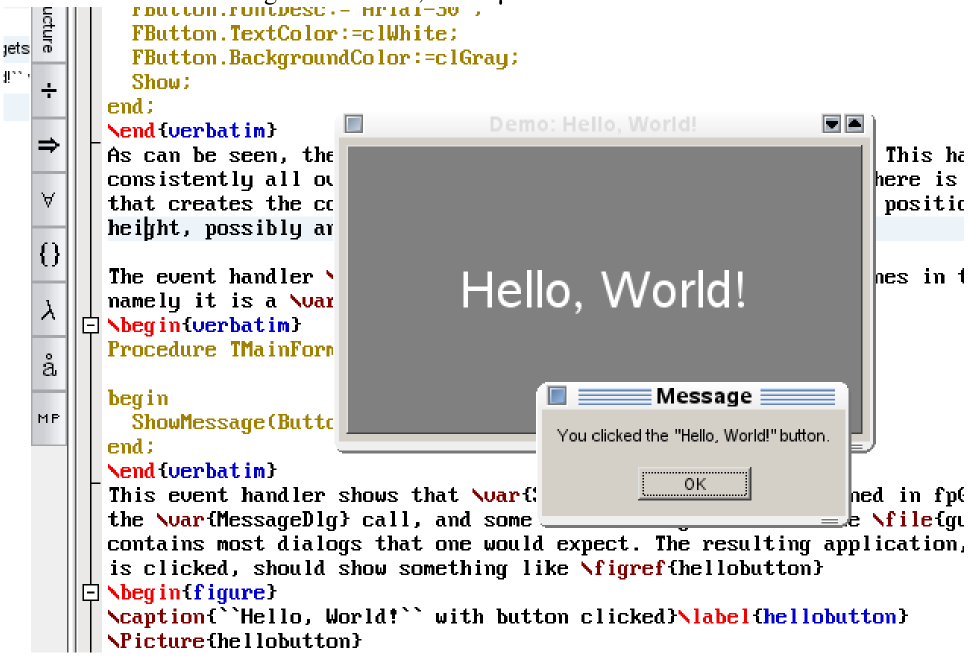
As can be seen, the button is created with a CreateButton call. This has been done consistently all over the widgetset: for almost all controls, there is a function that creates the control, and sets the main properties: parent, position, width, height, possibly an onclick event. Note the font selection. It is possible here to select a stock font by specifying it's name with a # sign in front of it:

```
FButton.FontDesc:='#Label1';
```

Using this feature together with the named colors is the start of theming an application.

The event handler `DoClick` has the same interface as the ones in the LCL of VCL, namely it is a `TNotifyEvent`:

Figure 2: “Hello, World¡‘ with button clicked



```
    FButton.FontDesc:='Arial-30';
    FButton.TextColor:=clWhite;
    FButton.BackgroundColor:=clGray;
    Show;
  end;
\end{verbatim}
As can be seen, the                                        This ha
consistently all ou                                       here is
that creates the cc                                        positic
height, possibly ar

The event handler `                                       nes in t
namely it is a \var
\begin{verbatim}
Procedure TMainForm

begin
  ShowMessage(Buttc
end;
\end{verbatim}
This event handler shows that \var{S                      ned in fpG
the \var{MessageDlg} call, and some                     e \file{gu
contains most dialogs that one would expect. The resulting application,
is clicked, should show something like \figref{hellobutton}
\begin{figure}
\caption{``Hello, World!`` with button clicked}\label{hellobutton}
\Picture{hellobutton}
```

```
Procedure TMainForm.DoClick(Sender : TObject);

begin
  ShowMessage(ButtonClicked);
end;
```

This event handler shows that `ShowMessage` has been defined in fpGUI, as well as the `MessageDlg` call, and some other dialogs as well: the gui_dialogs unit contains most dialogs that one would expect. The resulting application, when the button is clicked, should show something like figure 2 on page 9

Obviously, there are a lot more controls than just labels and buttons, but it would take too far to describe them all. Luckily, fpGUI comes with a lot of small demo applications, that show the possibilities of each of the controls that fpGUI offers: worth a look, and a rich source of information on how to use the controls.

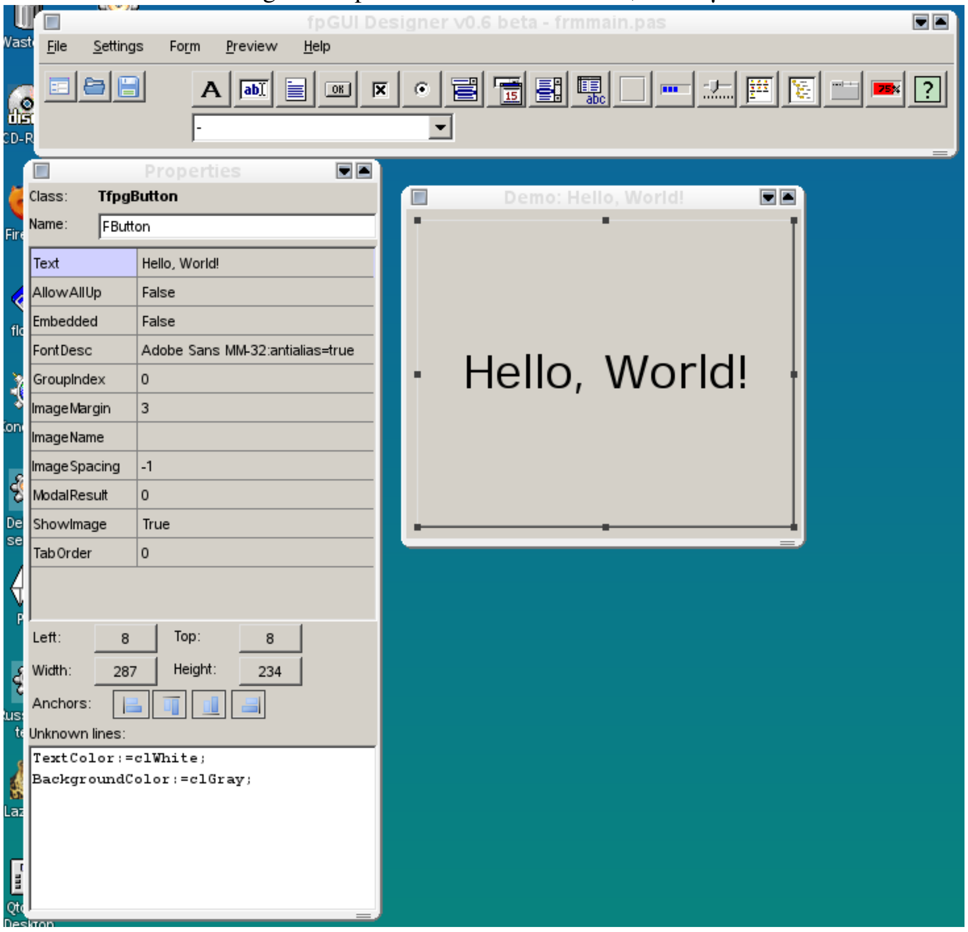
5 Visual Designer

Obviously, creating all these controls in code is a tedious job, requiring a lot of typing. Positioning controls and setting properties is also easier done visually than in code. So, to ease working with the widget set, fpGui comes with a small application called 'uiDesigner': it allows to create (and modify) a form just as one would do in Delphi, Lazarus or one of the many GUI designer tools for the other toolkits such as glade for GTK.

Unlike lazarus or Delphi, it does not create a resource-like .dfm form file, but instead creates a source file with a single method (`AfterCreate`, by default) that completely builds the form in code, much as has been displayed above.

While not a 2-way tool such as the Delphi IDE, it is powerful enough to be able to parse the pascal code that was generated (within some limitations, obviously), and so it is possible to modify the file in an editor, but still be able to edit the form visually.

Figure 3: fpGUI desiner with “Hello, World¡‘



Class: **TfpgButton**
Name: FButton

Text	Hello, World!
AllowAllUp	False
Embedded	False
FontDesc	Adobe Sans MM-32:antialias=true
GroupIndex	0
ImageMargin	3
ImageName	
ImageSpacing	-1
ModalResult	0
ShowImage	True
TabOrder	0

Left: 8 Top: 8
Width: 287 Height: 234
Anchors:
Unknown lines:

```
TextColor:=clWhite;
BackgroundColor:=clGray;
```

To use the tool, it should first be compiled: it is located in the examples/apps/uidesigner directory of the sources, and has a lazarus project file, or a @extrafpc.cfg configuration file for when compiling on the command-line.

Not all properties can be set using the Visual editor. The creator of the tool provided for this eventuality: code lines that it does not recognize, are displayed in a small memo below the ’object inspector’: it is possible to edit the lines there. They will be saved with the definition of the object, and will be reloaded when the file is opened, as can be seen in the screenshot figure 3 on page 10. The tool can handle multiple forms in a file, unlike Delphi or lazarus. When opening a file that contains 2 form declarations, both forms will be shown in the designer. When saving a form to a file that already contains a form definition, the definition is merged into the file.

The code generated by the UI designer tool looks like this:

```
begin
  {@VFD_BODY_BEGIN: MainForm}
  Name := 'MainForm';
  SetPosition(363, 286, 300, 250);
  WindowTitle := 'Demo: Hello, World!';
  FButton := TfpgButton.Create(self);
  with FButton do
  begin
```

```
    Name := 'FButton';
    SetPosition(8, 8, 287, 234);
    Anchors := [anLeft,anRight,anTop,anBottom];
    Text := 'Hello, World!';
    FontDesc := 'Adobe Sans MM-32:antialias=true';
    ImageName := '';
    TextColor:=clWhite;
    BackgroundColor:=clGray;
  end;
  {@VFD_BODY_END: MainForm}
end;
```

Note the merged lines. The need for markers and the fact that it does not handle all properties is a sign that this tool is far from perfect: it isn't intended to be. It is also not intended as a replacement for the Lazarus (or Delphi) IDEs - it has no code window for instance. But as a compagnon tool for the Lazarus IDE it is well suited, making it easy to develop fpGUI applications with lazarus: Indeed, fpGUI ships with a package for the Lazarus IDE, and also with some code templates that make coding fpGUI applications a lot faster. The Lazarus IDE will detect changes made to files by the ui Designer when switching between Lazarus and the IDE. While not as integrated as editing form files in Lazarus itself, it softens the working experience considerably.

6 The future

fpGUI is far from finished. There are lots of things on the TODO list, such as (in random order):

- support for MDI.
- a lazarus widgetset that bases itself on fpGUI.
- Much more controls: toolbars and splitters.
- More utilities: hints or tooltips, new graphics formats.
- New platforms: Mac and Windows CE.
- Theming support is one of the next things scheduled.

7 Conclusion

Why another GUI Toolkit ? fpGUI fills a void: an all-pascal GUI toolkit that is portable across platforms and that behaves the same on all platforms. It frees the GUI programmer from the need to resort to toolkits that are implemented in C. While this does not reduce the number of needed libraries significantly – a Lazarus 'Hello World' program uses 16 libraries, the same program in fpGUI still uses 13 – it does allow the programmer to check, possibly modify, the internals of the GUI library without (eventually) having to resort to C. It also means that creating new controls is very easy: the toolkit is very light-weight. Since its reincarnation the list of people developing has been increasing: A sure sign that it fills a need. The fact that it does not have to be Delphi compatible is an advantage - but also a weakness: it shuts out the possiblity of incorporating the huge number of existing third-party components out there. Nevertheless, fpGUI is one of these projects that show that Pascal is still very much alive and kicking, and if its design goals coincide with the goals for your projects, it's worth a look.