Stretching a Canvas: Image support for Free Pascal

Michaël Van Canneyt

February 4, 2005

Abstract

In this article, the Free Pascal support for images and simple drawing support is presented. The support is split out in 2 parts: support for image loading and support for drawing, which is modelled after the Delphi TCanvas implentation. They are tightly bound together, resulting in easy image manipulation.

1 Introduction

Free Pascal comes with the Free Classes Library, a set of non-visual worker classes, spread over several domains: Compression and encryption, XML treatment (DOM), Sockets - combined into support for XML-RPC. One of the domains is also image treatment: Abstract classes for image support, with some simple descendents. Likewise there are some abstract classes for drawing support: Mainly an abstract definition of the Delphi `TCanvas` class, which includes support for resource managing: fonts, pens and brushes.

All these image and drawing classes are defined independent of any visual elements such as a screen or printer: The idea is that there is a set of classes which can be used in all environments: command-line programs, web-page generating programs, or visual programs such as created with Lazarus.

Support for visual elements or specific implementations is left up to descendent classes: indeed, Lazarus defines decendents of most classes in the FCL image support units.

The advantage of this abstract approach is that if one wishes to use the same code for applications that must e.g. display bitmaps on screen, put some text on the bitmap, print them or write them to file, the routines that do the work must be written only once, as long as one sticks to methods offered by the FCL image and drawing interface. This will be demonstrated later in the article.

All source code for the components described here is located in the directory image of the FCL sources, which can be downloaded from the Free Pascal Website.

Since the drawing support is built on top of the image support, the image support is treated first in this article.

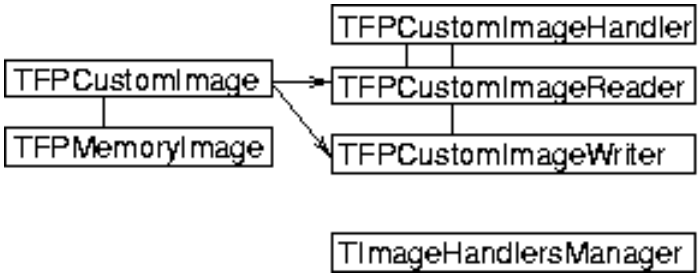
2 Image support

The image support is built around 4 basic structures, all defined in the fpImage unit:

TFPColor this record defines a color.

TFPCustomImage This class defines an image: it is basically a 2-dimensional array of colored dots; The colors may be specified using a palette, or not.

Figure 1: Classes in fpImage unit



TFPCustomImageReader A class to read an image from a stream (or file) in a specific format.

TFPCustomImageWriter A class to write an image from a stream (or file) in a specific format.

With these classes, images can be created, loaded, saved and optionally worked over on a pixel-by-pixel basis. A schematical overview of the classes involved is presented in figure 1 on page 2.

The `TFPColor` record is fundamental to all operations: it contains the color definitions. It is defined as follows:

```
TFPColor = record
  red,green,blue,alpha : word;
end;
```

The meaning of this definition should be obvious: The color components red,green and blue are specified as 16 bit colors, and the ALPHA (transparency) value is also specified as a 16 bit value. Put together, each pixel in an image is represented by a 64-bit value. This should be enough for most current and future applications. As more and more 64-bit processors will enter the market, speed should also no longer be an issue when using 64-bit color values.

The support for 64-bit colors does not mean that an actual image needs to store the pixel data as 64-bit data: The colors may be specified in a palette, or it can be stored using as much data as needed for the current screen display. The `TFPColor` definition is used to represent a pixel of arbitrary color in an image. When reading from or writing to an image, the image component may reduce the size of each pixel to whatever is needed. The same is true when drawing an image on screen: Memory sizes can be reduced as much as needed.

Various functions exist to convert a `TFPColor` value, or do operations on `TFPColor` values; e.g. the operators `=`, `and`, `or` and `not` have been overloaded so they can be used directly on `TFPColor` values. The following functions can be used to quickly construct a `TFPColor` value from red, green and blue values:

```
function FPColor (r,g,b,a:word) : TFPColor;
function FPColor (r,g,b:word) : TFPColor;
```

By default, `a` has the value `alphaOpaque`. Color constants have been defined for many colors: `colTransparant`, `colBlack`, `colWhite`, `colBlue` - in general, the color name with the prefix `col`. This is done to avoid confusion with the Delphi definitons which start with the prefix `cl`, and which are only 32-bit.

A 16-bit grayscale value can be computed with the function

```
function CalculateGray (const From : TFPColor) : word;
```

By default, the JPEG grayscale conversion method is used to calculate the gray value. Other methods are also available; The `GrayConvMatrix` constant defines the way in which grayscales are calculated. The interested reader can consult the fpImage unit for some constants which define the grayscaling method used.

An image is defined in the `TFPCustomImage` class. The following is part of the public declaration of this class:

```
constructor create (AWidth,AHeight:integer); virtual;
property Height : integer;
property Width : integer;
property Colors [x,y:integer]: TFPColor; default;
property UsePalette : boolean;
property Palette : TFPPalette;
property Pixels [x,y:integer] : integer;
property Extra [const key:string] : string;
property ExtraValue [index:integer] : string;
property ExtraKey [index:integer] : string;
property OnProgress: TFPImgProgressEvent;
```

The constructor create takes as the sole argument the width and height for the image. In case these are not known yet (for instance when loading an image from file), they can be set to zero.

The main (and hence default) property of the image is the `Colors` property: it represents the 64-bit color value for each pixel in the image. If the image is palette based, (indicated by the `UsePalette` property) then the `Pixels` property contains the index in the palette for the corresponding pixel. The palette itself is available as the `Palette` property. The location of the pixel in the image is specified by the index specifiers `x`,`y`, which are the - zero based - horizontal and vertical position of the pixel.

Some images contain extra information, such as the name of the software that created the image, or a date. To accomodate for this, a series of `Name=Value` pairs can be stored in the `TFPCustomImage` class and they are accessible through the `Extra`, `ExtraValue` and `ExtraKey` properties. The `ExtraCount` function returns the number of available pairs.

All the properties except `Palette` can be read and written. The methods that handle the writing of the properties are either abstract or virtual, so they can (or must) be overridden in descendent classes. Indeed, the `TFPCustomImage` class is not usable in itself; a descendent must be used to hold the actual image data.

The `TFPPalette` class has the following declaration:

```
constructor Create(ACount : integer);
procedure Build(Img : TFPCustomImage);
procedure Merge(pal : TFPPalette);
function IndexOf(const AColor: TFPColor) : integer;
function Add(const Value: TFPColor) : integer;
procedure Clear;
property Color[Index : integer] : TFPColor; default;
property Count : integer;
```

The constructor takes an initial size for the palette; The size can be changed at any time by setting the `Count` property, and the actual colors are available through the `Color` property.

The palette can be constructed quite easily using the `Build` method: it will create a palette, based on the various colors found in the image (`Img` parameter). The `Merge` method will merge two palettes together, and will make sure two identical colors do not appear twice in the palette. The meaning of the `IndexOf`, `Add` and `Clear` methods should be obvious.

Contrary to the `TFPCustomImage` class, the palette class is fully usable, but could be improved by a descendent class, in order to make it faster if e.g. the colors are known to be 16-bit only.

The `TFPCustomImage` class has also some methods to load the image from file or stream, and to save it to file or stream:

```
procedure LoadFromStream(Str:TStream;
                         Handler:TFPCustomImageReader);
procedure LoadFromStream(Str:TStream);
procedure LoadFromFile(const filename:String;
                       Handler:TFPCustomImageReader);
procedure LoadFromFile(const filename:String);
procedure SaveToStream(Str:TStream;
                       Handler:TFPCustomImageWriter);
procedure SaveToFile(const filename:String);
```

The `Handler` parameter is an instance of a class that does the actual reading or writing: If none is specified, then the method will try to deduce it from the filename extension or the header of the stream when loading an image from stream. If there is no class available to read the data, or to write the data in the requested form, then an exception will be raised.

The `fpImage` unit does not contain any reading or writing class for image data; only abstract classes are defined. Implementations for reading or writing image data should be implemented in separate units, which should be included in the `uses` clause of the program that needs them.

The definition of the reader and writer classes is as follows:

```
TFPCustomImageReader = class (TFPCustomImageHandler)
  function ImageRead (Str:TStream;
                      Img:TFPCustomImage) : TFPCustomImage;
  function CheckContents (Str:TStream) : boolean;
end;

TFPCustomImageWriter = class (TFPCustomImageHandler)
  procedure ImageWrite (Str:TStream; Img:TFPCustomImage);
end;
```

The meaning of the `ImageRead` and `ImageWrite` methods should be clear: they are called to do the actual reading and writing. The `CheckContents` method of the `TFPCustomImageReader` class should read the initial bytes of a stream, in order to determine whether it is a valid header for the kind of data it can read; this way, the `LoadFromStream` method of `TFPCustomImage` can determine which reader class it should use to read the image from a stream.

When creating a new reader or writer class, it should be registered in the image system. The `TImageHandlersManager` class is responsible for keeping a list of image readers and writers. It has the following interface:

```
procedure RegisterImageHandlers(const ATypeName,
                                TheExtentions:string;
```

```
                              AReader:TFPCustomImageReaderClass;
                              AWriter:TFPCustomImageWriterClass);
procedure RegisterImageReader(const ATypeName,
                              TheExtentions:string;
                              AReader:TFPCustomImageReaderClass);
procedure RegisterImageWriter(const ATypeName,
                              TheExtentions:string;
                              AWriter:TFPCustomImageWriterClass);
property Count : integer;
property ImageReader [const TypeName:string] :
               TFPCustomImageReaderClass;
property ImageWriter [const TypeName:string] :
               TFPCustomImageWriterClass;
property Extentions [const TypeName:string] : string;
property DefaultExtention [const TypeName:string] : string;
property TypeNames [index:integer] : string;
```

The various `Register` calls serve to register an implementation of a reader or writer class. The various properties can be used to query the available readers and writers in the executable; they should be self-explanatory. There is one instance of the `TImageHandlersManager` class created in the fpImage unit. The instance is called `ImageHandlers`: this instance should be used to register new image readers in the initialization code of the unit implementing a reader.

One reader can handle multiple extensions of filenames; they should be specified in a semicolon separated list, as follows:

```
ImageHandlers.RegisterImageReader('JPEG Graphics',
                                  'jpg;jpeg',
                                  TFPReaderJPEG);
```

As can be seen, readers and writers can be registered together or separately. This is done to save code: many applications will only need to be able to read images, not write them.

The following image types are implemented and distributed together with FPImage. The code is written without support of external libraries:

BMP reads and write support for Windows BMP files. Several bit-sizes are supported, RLE bitmaps are not (yet) supported. The relevant units are fpreadbmp and fpwritebmp.

JPEG Read and write support for JPEG format, 100The relevant units are (fpreadjpeg) and written (fpwritejpeg).

XPM Read and write support for XPM (X-Windows Pixmap) in the units fpreadxpm and fpwritexpm.

TARGA Read and write support for targa files is implemented in fpreadtga and fpwritetga.

PNG Read and write support for PNG files is implemented in fpreadpng and fpwritepng.

PNM The PNM (Portable aNyMaps) is a generic name for the following three formats: PBM (Portable BitMaps), PGM (Portable GrayMaps) and PPM (Portable PixMaps). They can be read and written.

To add support for reading or writing of images in one of the above formats, it is sufficient to add the unit to the uses clause of the program. Note that there may be several handlers for the same image type. All handlers will be tried till a handler is encountered that does not give an error. The last registered handler is tried first.

How can all this be used now ? The following program loads a bitmap from file, and saves it again in another file. It determines the format from the filename extensions:

```
{$mode objfpc}{$h+}
program ImgConv;

uses
  FPWriteXPM, FPWritePNG, FPWriteBMP, fpwritejpeg, fpwritetga,
  FPReadXPM, FPReadPNG, FPReadBMP, fpreadjpeg, fpreadtga,
  fpreadpnm, FPImage, sysutils;

var
  img : TFPMemoryImage;
  ReadFile, WriteFile : string;

begin
  if paramcount = 2 then
    begin
    ReadFile := paramstr(1);
    WriteFile := paramstr(2);
    end
  else
    begin
    Writeln('Usage: imconv infile outfile');
    Halt(1);
    end;
  If CompareText(ReadFile,WriteFile)=0 then
    begin
    Writeln('Input file cannot be the same as output file');
    Halt(1);
    end;
  Img:=TFPMemoryImage.Create(0,0);
  try
    Img.LoadFromFile(ReadFile);
    Img.SaveToFile(WriteFile);
  Finally
    Img.Free;
  end;
end.
```

So the following command would convert a PNG image to a BMP image:

```
imconv DrawTest.png DrawTest.bmp
```

Obviously, the above program does not do anything with the image, but the following code will for instance replace all cyan pixels with green pixels:

```
Img.LoadFromFile(ReadFile);
For x:=0 to Img.Width-1 do
  for y:=0 to img.height-1 do
```

```
    if Img[x,y]=colCyan then
      Img[x,y]:=colGreen;
Img.SaveToFile(WriteFile);
```

The image support in FPC does not add any algorithms to transform images; However, it should not be hard to add a small library with image routines such as sharpening, softening, mirroring or rotating.

The `TFPMemoryImage` component used in the above program is a simple descendent of the `TFPCustomImage` class, defined in the fpImage unit. The implementation is very simple, but illustrates how an image component should be made. To support the default Colors property, the `GetInternalColor` and `SetInternalColor` must be implemented:

```
function TFPMemoryImage.GetInternalColor(x,y:integer):TFPColor;

begin
  if Assigned(FPalette) then
    Result:=inherited GetInternalColor(x,y)
  else
    Result:=PFPColorArray(FData)^[y*FWidth+x];
end;

procedure TFPMemoryImage.SetInternalColor(x,y:integer;
                                          const Value:TFPColor);
begin
  if Assigned(FPalette) then
    inherited SetInternalColor(x,y,Value)
  else
    PFPColorArray(FData)^[y*FWidth+x]:=Value;
end;
```

If the image is palette based, then the inherited method should be called: the implementation in `TFPCustomImage` knows how to retrieve a pixel from the palette. If it is not pixel based, then the colors are stored in a long array in memory: the methods simply retrieve or store the color at the appropriate location in the array. There is no need to check the X,Y indexes: they have been checked before `GetInternalColor` or `SetInternalColor` are called.

To support a palette-based image, the `GetInternalPixel` and `SetInternalPixel` methods must be implemented:

```
function TFPMemoryImage.GetInternalPixel(x,y:integer): integer;
begin
  Result:=FData^[y*FWidth+x];
end;

procedure TFPMemoryImage.SetInternalPixel(x,y:integer;
                                          Value:integer);
begin
  FData^[y*FWidth+x]:=Value;
end;
```

They simply retrieve or store the palette index in an array in memory.

The array (`FData`) which is used to store the image is allocated in the `SetSize` method:

```
procedure TFPMemoryImage.SetSize (AWidth, AHeight : integer);
var w, h, r, old : integer;
    NewData : PFPIntegerArray;
begin
  if (AWidth <> Width) or (AHeight <> Height) then
    begin
    old := Height * Width;
    r:=AWidth*AHeight;
    if Assigned(FPalette)
    then
      r:=SizeOf(integer)*r
    else
      r:=SizeOf(TFPColor)*r;
    if r = 0 then
      NewData := nil
    else
      begin
      GetMem (NewData, r);
      FillWord (Newdata^[0], R div sizeof(Word), 0);
      end;
    if (old <> 0) and assigned(FData) and (NewData<>nil) then
      begin
      if r <> 0 then
        begin
        w := Lowest(Width, AWidth);
        h := Lowest(Height, AHeight);
        for r := 0 to h-1 do
          move (FData^[r*Width], NewData^[r*AWidth], w);
        end;
      FreeMem (FData);
      end;
    FData := NewData;
    inherited;
    end;
end;
```

The procedure is quite straightforward: after determining the number of slots in the array needed to keep the image data, the size of each slot is calculated. A new array is allocated, and as much as possible of the old data is copied to the new data. Note that at the end, the inherited method is called; it stores the new size settings.

The above 5 methods are the only methods that must be implemented in order to create a working image class. For efficiency, other methods can be overridden too, but all methods have a default implementation.

The interested reader can look at the Lazarus LCL code to see another (more complicated) implementation of a `TFPCustomImage`, as well as several reader and writer classes.

3 Drawing support: Canvas

The basis for drawing support is implemented in the fpcanvas unit. It contains an abstract definition of the following classes:

TFPCustomCanvas corresponds to the `TCanvas` class as defined in the Delphi VCL.

TFPCustomPen a pen to draw things on the canvas, corresponding to `TPen` in Delphi.

TFPCustomBrush To fill areas of the canvas with solid colors or patterns. Corresponds to `TBrush` in Delphi.

TFPCustomFont for putting text on the canvas. Corresponds to `TFont` in Delphi.

Lazarus defines descendent classes with the same name and behaviour as the Delphi VCL counterparts, making Delphi code easy to port.

For compatibility, these classes have the same properties (such as width, height, penstyle etc) and methods (MoveTo, LineTo, Rectangle) as their Delphi counterparts; they will not be elaborated here. The sole exception is the `Color` property: it is called `FPColor`. This is done to distinguish it from the Delphi `Color` property, which is an integer value. However, Lazarus defines descendents `TFont`, `TBrush` and `TPen` which do have a `Color` property, compatible to Delphi. Setting this property will also set the `FPColor` property and vice versa.

An important difference with the Delphi implementation of `TCanvas` is that the resources used to draw on the canvas (the pen, brush, font) do not need to be managed by the canvas itself, although this can be the case.

What does this mean ? In Lazarus, `TCanvas` manages the resources: it creates a pen, a brush and a font as needed: it obtains these resources from the windowing system, and manages them automatically: when the font name is changed, it will detect this, and request another font from (X) windows.

However, in a CGI program which simply draws on a bitmap in memory, the resources may (indeed must) come from elsewhere: The programmer may create a font, and use this font to put a text on the canvas. Therefore `TFPCustomCanvas` introduces a property `ManageResources` which determines whether the canvas itself will manage the resources, or whether the user (programmer) must do this himself. This property is determined by the descendent of `TFPCustumCanvas` and should not be set by the programmer.

The following code is part of a CGI program which displays the results of the FPC testsuite. It illustrates the case where the user manually allocates a font resource (using the FreeType engine), assigns it to the canvas, and then puts a text on the canvas:

```
Cnv:=TFPImageCanvas.Create(Img);
Cnv.Brush.Style:=bsSolid;
Cnv.Brush.Color:=colTransparent;
Cnv.Pen.Color:=colWhite;
F:=TFreeTypeFont.Create;
With F do
  begin
  Name:='arial';
  FontIndex:=0;
  Size:=12;
  Color:=colred;
  AntiAliased:=False;
  Resolution:=96;
  end;
Cnv.Font:=F;
Cnv.Textout(1,28,Format('%d Failed (%3.1f%%)',[Failed,Fr*100]));
```

The canvas will not free the font resource; It is the responsability of the user to free the font resource.

To do this correctly, TFPCustomCanvas contains a lot of logic to determine how the resources are allocated before they are drawn on the canvas.

Once resources have been allocated, the actual drawing must still be done. Obviously, if a canvas receives a pen resource from the programmer, how can it 'draw' using this resource ? On Windows (or X-Windows) all this is done by (X-)Windows itself: The pen is simply a series of properties which are passed to (X-)Windows. Windows then knows how to draw the line or circle on the screen, based on these properties.

In a CGI program, the situation is different. There the canvas gets a font or a pen, and must then decide how to 'draw' a circle, line or rectangle. There are 2 options:

1. The resource itself contains all logic to draw on the canvas.
2. The canvas contains the logic to draw on itself, using the given resource.

How does the TFPCustomCanvas decide which of these 2 possibilities must be used? For this, some descendents from the standard pen, font and brush classes, are introduced. They are called the TFPCustomDrawPen, TFPCustomDrawBrush and TFPCustomDrawFont classes: These classes have a method which will be called by TFPCustomCanvas. For TFPCustomDrawFont, these methods are:

```
procedure DoDrawText (x,y:integer; text:string);
procedure DoGetTextSize (text:string; var w,h:integer);
function DoGetTextHeight (text:string) : integer;
function DoGetTextWidth (text:string) : integer;
```

The purpose of the calls should be obvious.

When a canvas needs to draw something (e.g. a text), it checks whether the current font descends of TFPCustomDrawFont. If it does, then the methods defined in TFPCustomDrawFont are called. If the current font is not a descendent of TFPCustomDrawFont, then the internal TFPCustomCanvas method for drawing a text is called: in this case, DoTextOut. Descendents of TFPCustomCanvas should implement these calls: for instance the Lazarus implementation of TCanvas calls the Windows or X-Windows routines to draw the text on the canvas.

A descendents of TFPCustomDrawFont which uses this mechanism is the TFreeTypeFont font: it is a wrapper around the FreeType library: This font 'knows' how to draw a text on a canvas. To make this more clear, consider the code presented above. Now, when the following line is executed:

```
Cnv.Textout(1,28,Format('%d Failed (%3.1f%%)',[Failed,Fr*100]));
```

then the following chain of events will occur:

1. The canvas detects that the font is a descendent of TFPCustomDrawFont. It tells the font instance to allocate the necessary resources for drawing a text.
2. As a consequence, the TFreeTypeFont instance asks the freetype engine to load the requested font with the asked characteristics ('Times New Roman', bold etc.)
3. The canvas calls the DoDrawText method from TFreeTypeFont
4. As a consequence, the font instance asks the FreeType engine to draw the text as a grayscale bitmap (a glyph, in FreeType terms).
5. The glyph is then copied, pixel per pixel, on the canvas, using the font color, on the requested position.

Note that the font resource only knows how to put pixels on the canvas. It uses the `Colors` property of the canvas for this. How these pixels are actually drawn is determined (again) by the current pen of the canvas, and the font resource should not make any assumptions on this: Thus it is for instance possible that a FreeType font could be used to draw on a MS-Windows canvas (a Display Context), completely bypassing Windows text drawing routines.

The FreeType font as described here is implemented in the ftfont.pp unit, and is a standard part of the image library.

The whole canvas story would not be of much use if there were not some descendents implemented which actually do something. Currently, there are several descendents:

TCanvas defined in the LCL of lazarus.

TPixelCanvas defined in the image library, is an abstract canvas which consist of a 2-dimensional array of pixels. All drawing operations are done by setting the pixels to the required colors according to the current pen and brush.

TFPImageCanvas is a `TPixelCanvas` descendent: The pixels are stored in the image. It is implemented in the fpimgcanv unit.

TPostScriptCanvas is a canvas implementation which outputs postscript code to a stream. It is implemented in the pscanvas unit.

With these 4 (actually 3) classes, pretty much everything needed is possible:

- Draw on screen in a Lazarus program, using `TCanvas`.
- Export the drawing to an image, using `TImageCanvas`.
- Print (on Unix) using the `TPostScriptCanvas`.

And all this using one routine.

All this is illustrated in a small lazarus program. The program in itself is not very useful, it simply draws some geometrical figures. It draws this on the form canvas (in the `OnPaint` event), on an image (which is subsequently written to disk), and on a postscript canvas, which is also written to file. The drawing on an image and postscript file is triggered by 2 menu items.

The drawing code is quite simple:

```
procedure TMainForm.DoDraw(ACanvas : TFPCustomCanvas);

Var
  I : Integer;
  R : TRect;
  APolyGon : Array [0..8] of TPoint;

begin
  R.Top:=10;
  R.Left:=10;
  R.Right:=100;
  R.Bottom:=100;
  ACanvas.Pen.FPColor:=colRed;
  ACanvas.Rectangle(R);
  ACanvas.Ellipse(R);
```

```
  ACanvas.Pen.FPColor:=colGreen;
  ACanvas.Line(10,10,100,100);
  ACanvas.Line(10,100,100,10);
  ACanvas.Pen.FPColor:=colBlue;
  ACanvas.MoveTo(10,55);
  ACanvas.LineTo(100,55);
  ACanvas.MoveTo(55,10);
  ACanvas.LineTo(55,100);
  ACanvas.Pen.FPColor:=colYellow;
  For I:=0 to 8 do
    with APolyGon[i] do
      begin
      X:=55+Trunc(Cos(I*pi/4)*45);
      Y:=55+Trunc(Sin(I*pi/4)*45);
      end;
  ACanvas.PolyLine(APolyGon);
end;
```

The code draws a rectangle which encompasses a circle, and draws some mirroring lines. After this, an octagone with sides of equal length is drawn. Note that the `ACanvas` parameter is of type `TFPCustomCanvas`, not of type `TCanvas`.

To draw the image on an image and save the image to disk, the following code is put in the `OnClick` handler of the `MIBitmap` menu item:

```
procedure TMainForm.MIBitmapClick(Sender: TObject);

Var
  Image : TFPMemoryImage;
  ICanvas : TFPImageCanvas;
  FN  : String;

begin
  With SDDraw do
    begin
    Filter:='JPEG files|*.jpg|All Files|*.*';
    If Execute then
      FN:=FileName
    else
      exit;
    end;
  image := TFPMemoryImage.Create (110,110);
  Try
    ICanvas:=TFPImageCanvas.Create (image);
    Try
      DoDraw(ICanvas);
    Finally
      ICanvas.Free;
    end;
    Image.SaveToFile(FN);
  Finally
    Image.Free;
  end;
end;
```

The code is not so different from the code shown earlier to load and save an image. The central line is the call to `DoDraw`.

To print the drawing, a postscript file is created:

```
procedure TMainForm.MISavePostScriptClick(Sender: TObject);

Var
  F : TFileStream;
  PSCanvas : TPostScriptCanvas;

begin
  With SDDraw do
    begin
    Filter:='Postscript files|*.ps|All Files|*.*';
    If Execute then
      begin
      F:=TFileStream.Create(FileName,fmCreate);
      Try
        PSCanvas:= TPostScriptCanvas.Create(F);
        Try
          DoDraw(PSCanvas);
        Finally
          PSCanvas.Free;
        end;
      Finally
        F.Free;
      end;
      end;
    end;
end;
```

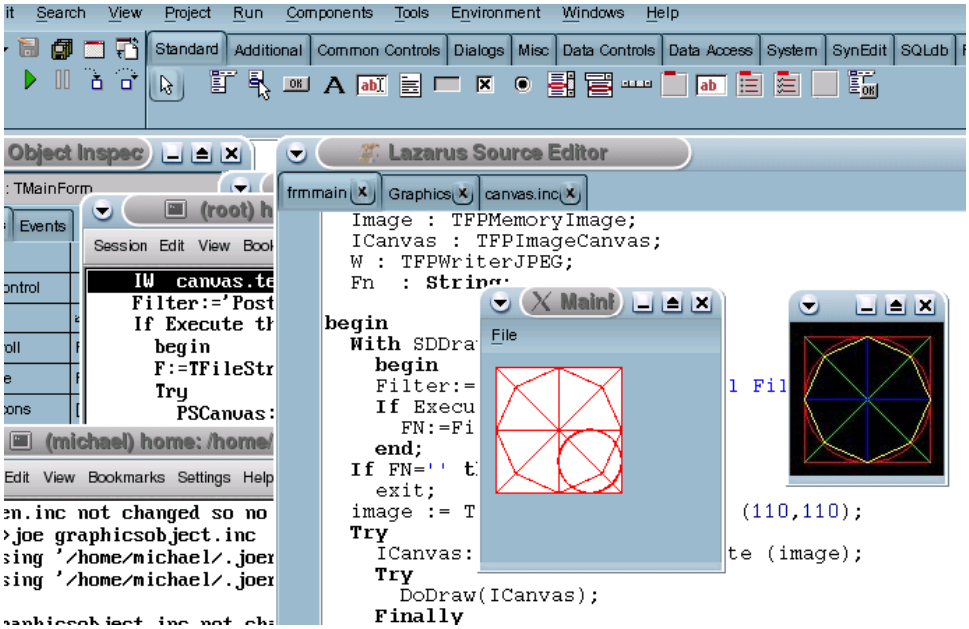
The code is straightforward: a filename is requested, a file stream is created with the returned filename, and then a `TPostScriptCanvas` is created, and the `DoDraw` method is called.

Finally, the image is drawn on screen in the `OnPaint` method of the form:

```
procedure TMainForm.MainFormPaint(Sender: TObject);
begin
  DoDraw(Canvas);
end;
```

The result can be seen in figure 2 on page 14. As can be seen, there are some differences in the two images. The background is different; To remedy this, the canvas should be 'cleared' in a uniform color. Secondly, the on-screen drawing is in the first color. The reason is that the conversion of the Lazarus LCL to use `TFPCustomCanvas` and friends is still in progress; there are some synchronization issues left. But the basics have been laid out, and by the time FPC 2.0 is released, the `TFPCustomCanvas` code is expected to be fully used by Lazarus. Currently there are also 3 postscript canvas implementations; They will be reduced to 1 implementation in the FCL (pscanvas).

Figure 2: The geometrical figure on screen and as image



4 Conclusion

Since its initial implementation by Luk Vandelaer, the image library has come a long way. But it is still a work in progress. Nevertheless it is already fully usable; the FPC team uses it in some CGI scripts, the Image code is fully integrated in Lazarus since a long time, and recently the conversion of the Lazarus `TCanvas` to a descendent of `TFPCustomCanvas` was started: Lazarus itself compiles and works using this new version; there are some minor issues left to be resolved. There are some number of things which are planned, but which still need to be tackled:

- Integrate the FreeType font in the PixelCanvas. For this, the freetype library should be loaded dynamically.
- Reduce the current number of PostScript canvases to 1.
- Create an interface to gdk_pixbuf to load additional images.
- Create an interface to imlib to load additional images.
- Create an interface to ImageMagick to load additional images.
- Integrate the delphi 32-bit color in the basic color handling, to improve Delphi compatibility.

As usual in FPC and Lazarus development, available time is the biggest restriction in the realization of these projects.

Even without these projects realized, the image library is already a powerful instrument: it can completely replace the `gd` library used in e.g. PHP. People that plan on developing some graphical components for Lazarus can easily add printing and image export support to their component by basing it on `FPCustomCanvas`. The design of the image library is more powerful than the VCL counterpart: The latter will only function on a system where a

GUI is present (X or Windows), while the Image library of FPC can run on a server without graphical system such as a web server. This goal was initially set, and has been reached. When the other goals outlined above are realized, Free Pascal will once more prove to be an all-purpose development tool.