Creating PDF files in Lazarus and Free Pascal

Michaël Van Canneyt

December 27, 2021

Abstract

In this article we show how to create PDF files in Lazarus and Free Pascal, in particular the `TPDFDocument` component that is distributed with Free Pascal.

1 Introduction

Lazarus has had a PDF creating component since a very long time: PowerPDF. It was ported from Delphi, and is used in LazReport to create PDFs from reports. However, it has several drawbacks: it requires the LCL and a GUI, has no support for embedding fonts, and does not support unicode fonts.

To remedy this, recently, a new implementation was added to Free Pascal: `fpPDF`. This new component has the following features:

- Pure Object Pascal code.
- No dependencies on a GUI, external libraries or OS calls.
- Many drawing primitives :
  - Images (any FPC supported format)
  - Lines (pen styles)
  - Circles
  - Rectangles (optionally rounded)
  - Polylines, i.e. draw multiple line segments in a single command.
  - Polygons, with 2 fill modes: non-zero winding rule and the even-odd fill rule.
  - Bezier curve support.
- Full TTF support.
- Unicode support.
- Optional Font Embedding (Partial embedding is being worked on).
- Coordinate transformation support (rotation, scaling and translation).
- Compression support for text and images.
- Embed HTML links

Because it requires no GUI this component is very suitable for use in e.g. a web application, where no GUI is available. Since it has no external dependencies, and does its own font handling, it can be used on any platform that Free Pascal and lazarus support, including the various Raspberry Pis. A new Lazarus lazReport PDF export has already been made with this component.

The use of this component is quite simple and pretty straightforward, as we'll demonstrate. The code is simple enough to port the component to Delphi with little or no effort, if one would like to.

2 Installing the component

The PDF component can be installed in the Lazarus IDE, the package is available from the Blaise Pascal website. It suffices to install lazpdf.lpk in the IDE and the `TPDFDocument` component will appear on the PDF tab of the component palette. The component has a few published properties:

LineStyles This is a collection of line styles which can be used when drawing lines. A line style defines a color, linewidth and pen style.

PageLayout This property determines how a PDF viewer should open the file: showing a single page, 2 pages or in continuous mode.

Infos This property has some properties that can be set (producer, author etc.) which will show up in the document properties in A PDF viewer.

DefaultPaperType This is the default paper type to use when creating a new page. By default this is set to A4.

DefaultOrientation This property determines whether a newly added page will be using landscape or portrait orientation.

DefaultUnitOfMeasure This property determines the default unit of measure used by a page in the PDF document. It is one of Inches, Millimeters (the default), Centimeters, Pixels (see below for more information).

FontDirectory The location of any fonts that need to be embedded.

Options Various options can be set which will influence the kind of PDF code that is generated.

The various options currently available include:

poOutLine Create a document outline which will be shown in a PDF viewer.

poCompressText Compress any strings that are written to the document.

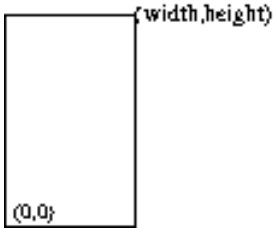
poCompressFonts Compress embedded fonts.

poCompressImages Compress images that are included in the document.

poUseRawJPEG When adding a JPEG image, the image is included as-is. The default is to read the image and convert it to PDF native format.

poNoEmbeddedFonts Do not embed fonts in the document. The default is to embed all fonts in the document.

Figure 1: PDF coordinate system



poPageOriginAtTop Flip the coordinate system so the origin is at the top,left corner, instead of the bottom, left corner. See below for more information.

Most of the work, however, is done in code: there is no point-and-click interface for this component.

3 Drawing: the coordinate system

The PDF standard is derived from the PostScript standard. This means that the origin of the drawing canvas is the bottom-left corner of the page, as shown in figure 1 on page 3. A bigger Y coordinate value means higher up the page.

This in differs from the coordinate system used in a Canvas of a LCL or VCL application, where the origin is at the top-left, and where higher Y coordinate values mean lower on the screen.

The coordinate system can be made compatible to the LCL or VCL value by specifying the `poPageOriginAtTop` option in the document options. Then a new page will be started with the coordinate system starting at the (top,left) corner. The coordinate system of the current page will not be changed.

The coordinate system uses a floating point value for coordinates, and uses 2 digits for precision. It can use various units, which can be set for each page separately in the `UnitOfMeasure` property of a page object. It has one of the following values:

uomInches 1 inch is 25.4 millimieters.

uomMillimeters which is the default unit.

uomCentimeters Centimeters.

uomPixels Pixels. The number of pixels per inch is calculated using a DPI of 72.

A default unit of measure can be specified at the document level.

All drawing commands will first transform the coordinates using a transformation matrix: When a drawing command has coordinates `(X,Y)`, then the `X` and `Y` are scaled and translated using

```
X := XScale * X + XTranslation;
Y := YScale * Y + YTranslation;
```

Note that this does not allow rotation, which requires a slightly more complex matrix. Instead, rotation must be specified separately when drawing objects.

4 Drawing: Getting started

Before anything can be drawn, the `StartDocument` method must be called. This will do some housekeeping tasks, and will prepare everything for drawing.

Once this is done, a section must be added to the document. A section is needed for creating document structure: every section will contain one or more pages. It is acceptable to have only 1 section in the document, that contains all pages.

The PDF component is page oriented. That means that before anything can be drawn, a page must be started. A page is started by calling the `Pages.AddPage` method, which returns a `TPDFPage` object. This page must then be added to a section.

In summary, the following code will start a document, and add a section and page to it:

```
Var
  D: TPDFDocument;
  S: TPDFSection;
  P :TPDFPage;

begin
  // Create document
  D:=TPDFDocument.Create(Nil);
  // Start document
  D.StartDocument;
  // Add a section, at least one section is needed.
  S:=D.Sections.AddSection;
  P:=P.Pages.AddPage;
  // Add the Page to the Section
  S.AddPage(P);
  // Set some properties:
  P.PaperType := ptA4;
  P.UnitOfMeasure := uomMillimeters;
end;
```

There are 2 ways to specify the dimension of a page in `TPDFpage`:

1. Using `PaperType` and `Orientation`. This will set the dimensions in `Paper`.
2. Setting `PaperType` to `ptCustom` and explicitly setting dimensions in `Paper`.

The `Paper` property contains the size of the paper, in PDF points (pixels).

In order to convert the PDF sizes to millimeters, centimerers or inches, one can use the conversion routines `mmToPDF`, `PDFToMM` and so on. The names speak for themselves.

5 Drawing: Colors, linewidths and styles

Drawing commands use a certain color, linewidth and pen style. These can be set on the `TPDFPage` object with the obvious methods:

```
Procedure SetColor(AColor : TARGBColor; AStroke : Boolean = True);
Procedure SetPenStyle(AStyle : TPDFPenStyle; ALineWidth: TPDFFloat = 1.0);
```

As soon as one of these properties is set, it remains valid for all subsequent drawing commands.

The colors used in a PDF document are specified using RGB notation. The fpPDF unit contains some constants for common colors. When setting the color, one must specify whether this is the stroking (or line drawing) color (`AStroke=True`), or the fill color (`AStroke=False`).

The pen style determines how lines are drawn and can be one of

```
TPDFPenStyle = (ppsSolid,ppsDash,ppsDot,ppsDashDot,ppsDashDotDot);
```

The meaning of these are obvious.

To make it easier to manage colors and line styles, the `TPDFDocument` component can maintain a set of line styles in the `LineStyles` collection property. Each item in the collection specifies a color, width and pen style. The method

```
Procedure SetLineStyle(AIndex: Integer; AStroke: Boolean = True);
Procedure SetLineStyle(S: TPDFLineStyleDef; AStroke: Boolean = True);
```

can be used to set the color, width and pen style in one command. The style can be set using the `AIndex` of an item in the document `LineStyles` property, or the item can be specified directly.

6 Drawing: Line drawing commands

The `TPDFPage` class is in many ways similar to the `TCanvas` class in the LCL or VCL: it offers many commands to actually draw something on the page.

There are 4 line drawing commands:

```
procedure DrawLine(X1, Y1, X2, Y2, ALineWidth : TPDFFloat;
                   const AStroke: Boolean = True);
procedure DrawLine(APos1, APos2: TPDFCoord; ALineWidth: TPDFFloat;
                   const AStroke: Boolean = True);
Procedure DrawLineStyle(X1, Y1, X2, Y2: TPDFFloat; AStyle: Integer);
Procedure DrawLineStyle(APos1, APos2: TPDFCoord; AStyle: Integer);
```

The `TPDFCoord` is a record describing an X,Y position using `TPDFFloat` type. All commands come in 2 overloaded forms: one with explicit X,Y coordinates, one where coordinates are specified using a `TPDFCoord` structure. We will only present the former in this document.

The `DrawLine` command will draw a line in the current color, style and line width. The `DrawLineStyle` command will draw a line in the color, style and linewidth defined in the specified item in the `TPDFDocument` property `LineStyles`. Note that the style will remain set for the subsequent commands.

The `Stroke` option, when set to `True` will tell the PDF engine to actually generate the line. If a lot of line segments must be drawn, then the lines can be drawn without stroke (called drawing the path), and when the last line is drawn, it can be stroked (the path is actually drawn).

The following code which draws a border around a page, demonstrates the difference:

```
Procedure TMainForm.GenerateLines(P : TPDFPage);

Var
  W,M,T,R : TPDFFloat;
```

```
begin
  W:=1.0; // Line width
  M:=10; // Margin
  T:=PDFToMM(P.Paper.H)-M; // Top
  R:=PDFToMM(P.Paper.W)-M; // Right
  // Page margins
  P.DrawLine(M,M,M,T,W,True);
  P.DrawLine(M,M,R,M,W,True);
  P.DrawLine(M,T,R,T,W,True);
  P.DrawLine(R,M,R,T,W,True);
end;
```

4 lines are drawn in sequence and generated at once. Note that the lines are not drawn in a logical order: left, bottom, top, right. When drawing this by hand on a paper, one needs to lift the hand to go to the starting position for the next line.

The following achieves the same, but draws the lines in a logical order: left, top, right, bottom. When drawing this by hand on paper, one can draw the 4 segments in 1 fluid motion (in 1 stroke).

```
Procedure TMainForm.GenerateLinesNoStroke(P : TPDFPage);

Var
  W,M,T,R : TPDFFloat;

begin
  W:=1.0; // Line width
  M:=10; // Margin
  T:=PDFToMM(P.Paper.H)-M; // Top
  R:=PDFToMM(P.Paper.W)-M; // Right
  // Page margins
  P.MoveTo(M,M);
  P.DrawLine(M,M,M,T,W,False);
  P.DrawLine(M,T,R,T,W,False);
  P.DrawLine(R,T,R,M,W,False);
  P.DrawLine(R,M,M,M,W,False);
  P.StrokePath;
end;
```

The first coordinate of `DrawLine` is ignored when the `Stroke` parameter is `False`, the drawing starts from the current position. Because of this, a `MoveTo` command is needed, which sets the initial position. The last command `StrokePath`, tells the engine that the drawing is done, and that the line may actually be drawn.

To generate multiple lines at once, a Polyline command can be used:

```
procedure DrawPolyLine(const APoints: array of TPDFCoord;
                       const ALineWidth: TPDFFloat);
```

It accepts an array of points, and will draw lines between the N and N+1-th points.

Using this, the page rectangle can now be drawn using the following:

```
Procedure TMainForm.GeneratePolyLines(P : TPDFPage);

Var
```

```
  W,M,T,R : TPDFFloat;
  A : Array[1..5] of TPDFCoord;

begin
  W:=1.0; // Line width
  M:=10; // Margin
  T:=PDFToMM(P.Paper.H)-M; // Top
  R:=PDFToMM(P.Paper.W)-M; // Right
  // Page margins
  A[1].X:=M; A[1].Y:=M;
  A[2].X:=M; A[2].Y:=T;
  A[3].X:=R; A[3].Y:=T;
  A[4].X:=R; A[4].Y:=M;
  A[5].X:=M; A[5].Y:=M;
  P.DrawPolyLine(A,1);
  P.StrokePath;
end;
```

Note the `StrokePath`: the `DrawPolyline` command will not yet draw the path.

Although there are only 4 corners, the array needs 5 positions: the last position is the same as the first. It is possible to avoid this by using the `DrawPolygon` function: this will close the line by drawing a segment from the last to the first line.

```
procedure DrawPolygon(const APoints: array of TPDFCoord;
                      const ALineWidth: TPDFFloat);
```

Note that again, `StrokePath` must be called after a call to `DrawPolygon`.

When using `DrawPolygon` (or indeed when drawing any closed path), the inside of the drawn path can be filled. This can be done using 2 commands, each uses a different algorithm:

FillStrokePath This uses the nonzero-winding number rule.

FillEvenOddStrokePath This uses the even-odd number rule.

Detailed descriptions of these rules can be found e.g. on wikipedia. They may produce different results, depending on the path.

The color for filling must be set using `SetColor`, with the `AStroke` parameter set to false. The following will draw a page rectangle and fill it using a yellow color:

```
procedure TMainForm.GeneratePolygonsFill(P: TPDFPage);
Var
  W,M,T,R : TPDFFloat;
  L : Integer;
  A : Array[1..4] of TPDFCoord;

begin
  W:=1.0; // Line width
  M:=10; // Margin
  T:=PDFToMM(P.Paper.H)-M; // Top
  R:=PDFToMM(P.Paper.W)-M; // Right
  // Page margins
  A[1].X:=M; A[1].Y:=M;
```

```
  A[2].X:=M; A[2].Y:=T;
  A[3].X:=R; A[3].Y:=T;
  A[4].X:=R; A[4].Y:=M;
  P.DrawPolygon(A,1);
  P.SetColor(clYellow,False);
  P.FillStrokePath;
end;
```

7 Drawing: shapes - rectangles, circles and ellipses

A rectangle can be drawn using polyline, but this is tedious. PDF defines a rectangle command, making it easier to draw (and optionally fill) a rect. The command needs the coordinates of the bottom-left corner, and a with and height.

Additionally, fill and stroke can be specified, as well as a rotation. The rotation (in degrees) is not performed around the center of the rectangle, but around the (bottom,left) corner of the rectangle.

Drawing an ellipse (or circle, which is just an ellipse with equal radii) is done in much the same way as a rectangle: the bounding rectangle of the ellipse is specified, and the ellipse will be drawn inside the rectangle.

The following are the definitions of the various drawing commands:

```
Procedure DrawRect(const X, Y, W, H, ALineWidth: TPDFFloat;
                   const AFill, AStroke : Boolean;
                   const ADegrees: single = 0.0);
procedure DrawRoundedRect(const X, Y, W, H,
                          ARadius, ALineWidth: TPDFFloat;
                          const AFill, AStroke : Boolean;
                          const ADegrees: single = 0.0);
Procedure DrawEllipse(const APosX, APosY, AWidth, AHeight,
                      ALineWidth: TPDFFloat;
                      const AFill: Boolean = True;
                       AStroke: Boolean = True;
                      const ADegrees: single = 0.0);
```

The following code will draw 2 ellipses with the same (bottom,left) corner, but the second is rotated 45 degrees. For each ellipsis, it draws the bounding rectangle around it in a different color:

```
procedure TMainForm.GenerateEllipses(P: TPDFPage);
Var
  W,M,T,R : TPDFFloat;
  L : Integer;
  CX,CY : TPDFFloat;

begin
  W:=1.0; // Line width
  M:=10; // Margin
  T:=PDFToMM(P.Paper.H)-M; // Top
  R:=PDFToMM(P.Paper.W)-M; // Right
  CX:=R/4;
  CY:=T/2;
  P.DrawEllipse(CX-10,CY-10,40,20,W,False,True);
```

```
  // Rotation is around bottom,left corner
  P.SetColor(clRed);
  P.DrawEllipse(CX-10,CY-10,40,20,W,False,True,45);
  P.SetColor(clDkGray);
  P.SetPenStyle(ppsDash);
  P.DrawRect(CX-10,CY-10,40,20,W,False,True);
  P.DrawRect(CX-10,CY-10,40,20,W,False,True,45);
```

In figure 2 on page 10 one can see the result of this code, together with some variations.

The PDF specification does not actually have commands to draw a circle or an ellipse. Instead, it draws Bezier curves. Circles and ellipses are in fact drawn with an approximation using Bezier curves. There are 3 bezier-curve drawing commands in the PDF specification (C, V and Y), they differ only in how the control points for the bezier curve are specified. The C form explicitly specifies the 2 control points.

```
procedure CubicCurveTo(ACtrl1, ACtrl2, ATo: TPDFCoord;
                       const ALineWidth: TPDFFloat;
                       AStroke: Boolean = True); overload;
```

For the V form, the first control point coincides with the start of the curve, for the Y form, the second control points coincices with the final point of the curve.

```
procedure CubicCurveToV(ACtrl2, ATo: TPDFCoord;
                        const ALineWidth: TPDFFloat;
                        AStroke: Boolean = True); overload;
procedure CubicCurveToY(ACtrl1, ATo: TPDFCoord;
                        const ALineWidth: TPDFFloat;
                        AStroke: Boolean = True); overload;
```

In all cases, the first point of the Bezier curve is the current position.

The demo application can be consulted to see how this works.

8 Images

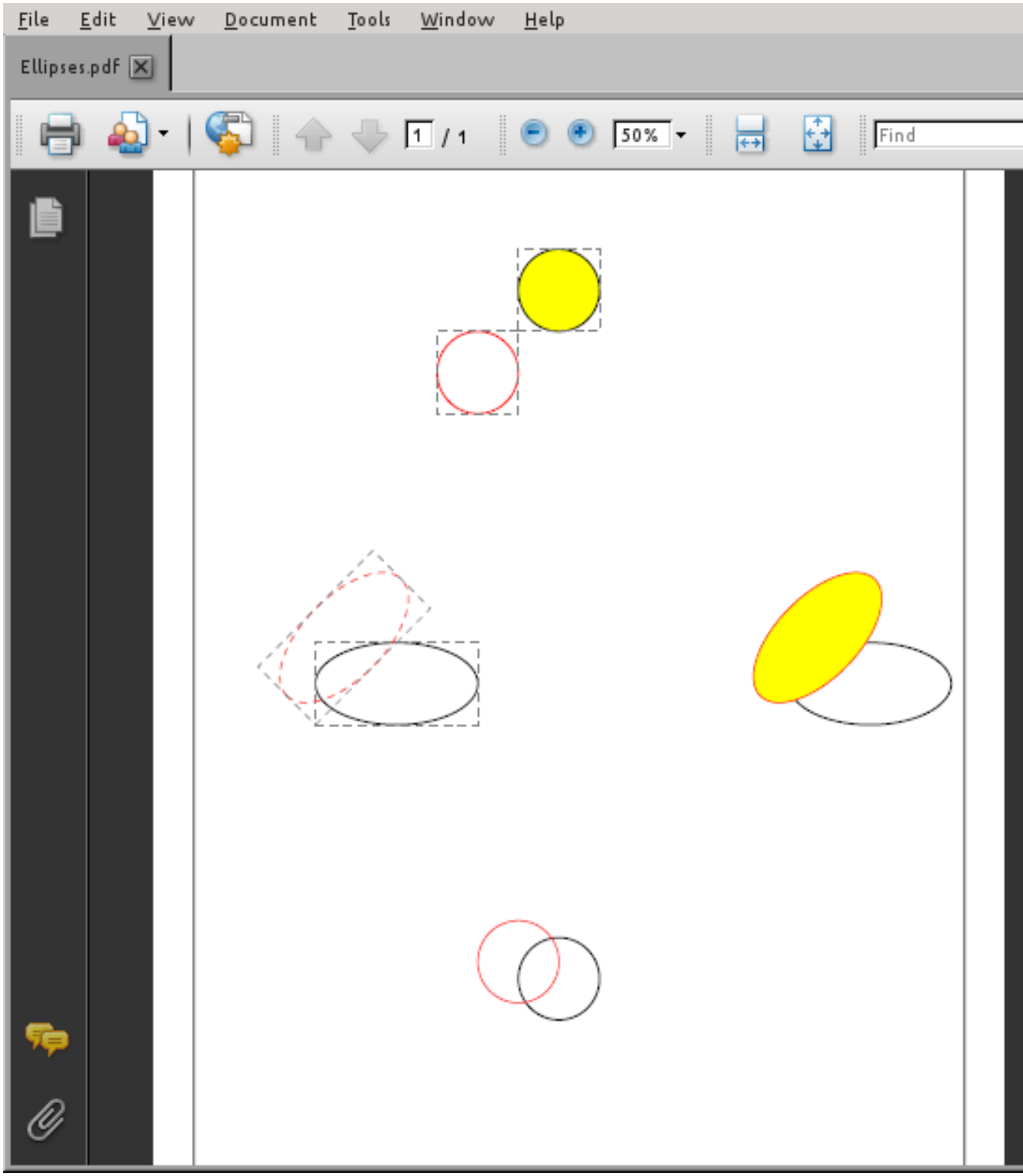
A PDF engine would not be clear without image support. Drawing an image on a PDF document is a 2 step process:

1. Add the image to the document. This results in an numerical ID for the image.
2. Draw the image on a page, using the ID obtained in step 1.

The PDF document maintains a list of images in the `Images` property. This collection keeps a list of images. An image can be added using one of the following methods:

```
Function AddJPEGStream(Const AStream : TStream;
                       Width,Height : Integer): Integer;
Function AddFromStream(Const AStream : TStream;
                       Handler : TFPCustomImageReaderClass;
                       KeepImage : Boolean = False): Integer;
Function AddFromFile(Const AFileName : String;
                     KeepImage : Boolean = False): Integer;
```

Figure 2: Various ellipses and their bounding rectangles



The methods speak for themselves. Any image format supported by Free Pascal can be added to the document, the component will do what is necessary to convert it. PDF supports compressed JPEG streams natively, so raw JPEG files can be added to the document as-is.

Once added to the list of images, the image can be used. It can be drawn using one of 2 methods:

```
Procedure DrawImageRawSize(const X, Y: TPDFFloat;
                           const APixelWidth, APixelHeight,
                           ANumber: integer;
                           const ADegrees: single = 0.0);
Procedure DrawImage(const X, Y: TPDFFloat;
                    const AWidth, AHeight: TPDFFloat;
                    const ANumber: integer;
                    const ADegrees: single = 0.0);
```

The first one (`DrawImageRawSize`) will draw the images using their dimensions in pixels. `DrawImage` uses dimensions in the current units of the page.

As with all elements, the image can be rotated. The following code will draw the same image 4 times on the page: in each corner of the page, rotating the image for each corner. The bottom-right and top-left images are resized, where the bottom-right image retains the aspect ratio of the image.

```
procedure TMainForm.GenerateImages(P: TPDFPage);

Var
  M,T,R : TPDFFloat;
  I : Integer;
  W,H : TPDFFloat;
  FN : String;

begin
  M:=10; // Margin
  T:=PDFToMM(P.Paper.H)-M; // Top
  R:=PDFToMM(P.Paper.W)-M; // Right
  DrawPageMargin(P);
  FN:=ExtractFilePath(ParamStr(0))+'poppy.jpg';
  I:=D.Images.AddFromFile(FN,False);
  W:=PDFtoMM(D.Images[i].Width);
  H:=PDFtoMM(D.Images[i].Height);
  // Bottom left
  P.DrawImage(2*M,2*M,W,H,I);
  // Top right, rotated 180 degrees
  P.DrawImage(R-M,T-M,W,H,I,180);
  // Bottom right, keep aspect ratio.
  P.DrawImage(R-M,2*M,4*M,4*M * H/W,I,90);
  // Top left, rotated 270 degrees. Distorted.
  P.DrawImage(M,T-M,4*M,4*M,I,270);
end;
```

The result of this can be seen in figure 3 on page 12.

Figure 3: Images drawn on the page

9 Text support

Probably one of the more difficult topics in PDF is writing text on the page. Text must be written using a font. The PDF standard defines a few built-in fonts, which will be emulated when not available on the machine where the PDF is viewed.

Additionally, PDF can also use TTF fonts. If the font is available on the system where the PDF is viewed, then the font need not be included in the file. If the font is not available, then the PDF cannot be viewed propertly. For these cases, the font can be included in the PDF file (embedded).

As with images, to write a text on a PDF page is again a 2 step process:

1. Add the font definition to the document. This results in an numerical ID for the font.
2. Write the text on a page, using the ID obtained in step 1 to specify the font.

Adding a font can be done using the `AddFont` method:

```
Function AddFont(AName : String) : Integer;
Function AddFont(AFontFile: String; AName : String) : Integer;
```

The first form just adds a font definition to the document. It is meant for standard fonts. The second form adds a font file to the list of available fonts. The font metrics will be read from the font file. To avoid specifying the full path to the font, the font file name can be relative to a font directory; the font directory can be set in the `FontDirectory` property.

When it is time to actually draw a text on the screen, the font must be set using index of the font in the font list additionally the size of the font (in points) must be specified:

```
Procedure SetFont(AFontIndex : Integer; AFontSize : Integer);
```

And then the text can be drawn using `WriteText`:

```
Procedure WriteText(X,Y: TPDFFloat; AText: UTF8String;
                    const ADegrees: single = 0.0);
```

The text will be placed at `X,Y`, with the baseline of the text at the `Y` coordinate, which means the text can extend above and below `Y`. The text can be unicode text, and the encoding is in UTF8. As with all other drawing commands, the text can be rotated. Text is drawn with the fill color.

The following will draw a text at the 4 corners of the page:

```
procedure TMainForm.GenerateText(P: TPDFPage);

Var
  M,T,R : TPDFFloat;
  I : Integer;

begin
  M:=10; // Margin
  T:=PDFToMM(P.Paper.H)-M; // Top
  R:=PDFToMM(P.Paper.W)-M; // Right
  I:=D.AddFont('Helvetica');
  // Set font, Helvetica, 12 pt
  P.SetFont(I,12);
```

```
  // Bottom left
  P.WriteText(2*M,2*M,'(Bottom,left)');
  // Top left, rotated 270 degrees.
  P.WriteText(2*M,T-M,'(Top,Left)',270);
  // Top right, rotated 180 degrees
  P.WriteText(R-M,T-M,'(Top,Right)',180);
  // Bottom right, rotated 90 degrees
  P.WriteText(R-M,2*M,'(Bottom,Right)',90);
end;
```

10 More Text & HTML

The text drawing shown above circumvented width and height calculations, the text was just drawn at a particular spot. This is not always possible. For instance, when drawing a large text, it is necessary to know what the width of a text is (e.g. to perform word wrap) and the height, to know where to draw the following line of text.

Often, text is placed in a rectangle (border) to make it stand out. To be able to draw a border around a text, one needs to know the width and height of a text: it must be calculated.

Additionally, PDF supports embedding of hyperlinks. This works by designating a rectangle on the page as a hotspot. This is done using the `AddExternalLink` method of the `TPDFPage` method:

```
Procedure AddExternalLink(const APosX, APosY,
                          AWidth, AHeight: TPDFFloat;
                          const AURI: string;
                          ABorder: boolean = false);
```

So, in order to mark a piece of text as a hyperlink, one first needs to know the extent of the text.

To be able to calculate text sizes, and to be able to embed fonts, a font manager is implemented in the `fpttf` unit. The font manager will read a font file and look up the metrics of a font, i.e. the sizes of the various characters in the font. Using these metrics, the font manager can then calculate the height and width of a text. These metrics are also needed when embedding a font definition in a PDF.

The font manager must be initialized. The easiest way is to use the `BuildFontCache` method, which will build a cache of fonts found in one or more directories. For the sample program we'll assume that all fonts are located in a directory called 'fonts' below the example executable:

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

Var
  FontDir : String;

begin
  FontDir:=ExtractFilePath(ParamStr(0))+'fonts';
  D.FontDirectory:=FontDir;
  gTTFontCache.SearchPath.Add(FontDir);
  gTTFontCache.BuildFontCache;
end;
```

After this is done, the `TextWidth` and `TextHeight` methods of a font item can used to calculate the width and height (in pixels) of a given text:

```
function TextWidth(const AStr: utf8string;
                   const APointSize: single): single;
function TextHeight(const AText: utf8string;
                    const APointSize: single;
                    out ADescender: single): single;
```

The text height is the height of the text above the baseline, the descender is the amount of pixels the text extends below the baseline. The amount of pixels is determined by the 'DPI' setting of the font cache (72 by default).

Putting all this together, we can now write some text on a Page, draw a rectangle around it, and mark it as a hyperlink:

```
Procedure TMainForm.GenerateHTML(P : TPDFPage);

Const
  aText1 = 'Hello, Free Pascal!';
  aURL = 'http://www.freepascal.org/';
  FontName = 'FreeSans';
  FontSize = 12;

var
  DH,H,W,M,T,AY,AX : TPDFFloat;
  I : integer;
  lFC: TFPFontCacheItem;

begin
  M:=10; // Margin
  T:=PDFToMM(P.Paper.H)-M; // Top
  // Add the font to the document.
  I:=D.AddFont('FreeSans.ttf', FontName);
  P.SetFont(I, FontSize);
  P.SetColor(clBlack, false);
  AX:=2*M;
  AY:=T-T/4-M;
  P.WriteText(AX,AY, aText1);
  // Find the font metrics:
  lFC := gTTFontCache.Find(FontName, False, False);
  // Calculate height, in pixels according to DPI in font cache
  H:=lFC.TextHeight(Atext1,FontSize,DH);
  W := lFC.TextWidth(aText1,FontSize);
  // convert to mm
  H:=(H * cInchToMM)/gTTFontCache.DPI ;
  DH:=(DH * cInchToMM)/gTTFontCache.DPI ;
  W:=(W * cInchToMM)/gTTFontCache.DPI;
  // Draw a border around it
  P.SetColor(cldkGray, true);
  P.DrawRect(AX,AY-DH,W,H+DH,1,false,true);
  // Now create a hyperlink at the same place:
  P.AddExternalLink(AX,AY-DH,W,H+DH,aURL,False);
end;
```

11 Conclusion

Anyone who can draw something in the LCL or VCL can create a basic PDF: Creating a PDF is much like a drawing on a canvas. The methods differ somewhat, but the ideas are pretty much the same. Currently, the PDF API of Free Pascal does not support extensions to PDF such as forms, video and sound, but it offers all methods needed to create basic printable PDFs.