sgcPDF

Getting Started

Create, show and save PDF documents from a Delphi form, one small demo at a time

About this guide

This guide is for Delphi developers who know how to build a VCL form and want their application to write PDF files, show them on screen, or both, with sgcPDF. You do not need to know anything about the PDF format in advance. Chapter 3 explains the handful of PDF ideas you meet in the first hour, such as points and the bottom left origin, in plain words.

The guide follows the five QuickStart demos. Each demo adds one idea to the previous one, from a single page with a title and two lines of text to a small invoice application with a form for the data, a live preview and a Save button. Chapter 4 walks through them one at a time: what you will build, the components on the form, the code, how it works and a few changes to try.

Read it with Delphi open next to it. Open a demo, run it, look at the PDF it produces, then read its section and try the suggested changes. Chapters 1 to 3 explain what sgcPDF is, how to install it and the ideas behind it, chapter 5 lists the main components, chapter 6 collects common problems and their fixes, and chapter 7 shows where to go next.

The demos are in the `Demos\PDF\Delphi\00.QuickStart` folder of your sgcUI installation. Each demo has its own subfolder, from `01.HelloPDF` to `05.PDFInvoiceApp`, with a Delphi project (`.dpr`) that you can open and run directly. All of them are VCL applications for Windows, and all of them build their documents in code, so they need no network, no sample files and no input from you.

The screenshots were taken on a Windows machine with Spanish regional settings. Numbers and dates that the demos format themselves, such as the amounts of the invoice, follow the Windows regional settings of the machine that runs the demo, so on your machine you may see a decimal point where the figures show a comma.

Contents

1. What sgcPDF is

The idea in one minute
Your first PDF in one event handler
What you need
Part of sgcUI
Coming from other ways of making PDFs

2. Install and the palette

Installing
The packages
The palette page
Licences and the trial
Where the demos are
Running your first demo

3. Core concepts

The document and its pages
Points and the bottom left origin
Drawing: text, shapes, colours and images
Fonts: standard and TrueType
Tables and page breaks
Saving: file or stream
Showing a PDF: the viewer control
Opening and changing existing files
Summary: who does what

4. The QuickStart demos

4.1 Demo 01: Hello PDF
4.2 Demo 02: PDF Drawing
4.3 Demo 03: PDF Table Pages
4.4 Demo 04: PDF Open View
4.5 Demo 05: PDF Invoice App

5. Main components

Reference table
Creating documents
Showing documents
Working with existing files
Security and signatures
Beyond the basics

6. Common problems

Text and fonts
Layout and tables
Viewing and saving

7. Where next

The other Delphi demos
Help topics
On the website
Getting help

1. What sgcPDF is

sgcPDF is a Delphi library to create, open, edit, sign, render and view PDF documents. It is written in Object Pascal and compiles into your executable, so there is no PDF engine DLL to deploy next to your program and no third party library to license.

For a first application you need two components. You drop a `TsgcPDFDocument` on a form, add pages to it and draw on them from code: text, lines, rectangles, images and tables. Then you save the document to a file or a stream. If the user should see the result, you add a `TsgcPDFViewerControl`, a visual control that shows a PDF document the way the viewer of a web browser does, with zoom, scrolling and page navigation built in.

This guide is for Delphi developers who are new to sgcPDF. By the end you will know how the pieces fit together, which methods to reach for first, and how to avoid the mistakes that most people make in their first hour.

The idea in one minute

A PDF file is a list of pages, and every page is a drawing. When you write a PDF with sgcPDF you do not fill in a template or describe a layout that something else arranges for you. You say where things go, in the same way you would paint on a `TCanvas`:

- **Add a page** of the size you want. A4 is 595 by 842 points.
- **Draw on it.** Write a text at a position with a font and a size, draw a line from one point to another, fill a rectangle with a colour, place an image.
- **Save it** to a file or a stream. Any PDF reader can open the result.

Two things feel different from a `TCanvas`. The unit is the point, 72 points per inch, not the pixel. And the point (0, 0) is the *bottom* left corner of the page, so the Y coordinate grows upwards. Chapter 3 shows both ideas with a picture. Once you have them, the rest is a list of drawing methods with readable names.

For the parts of a document that do not have a fixed size, such as a list of products that may need one page or ten, sgcPDF has a table class that measures the rows, starts a new page when the current one is full and repeats the header row there.

Your first PDF in one event handler

The first QuickStart demo is the smallest sgcPDF application there is. The form holds a `TsgcPDFDocument` named `PDF` and a `TsgcPDFViewerControl` named `Viewer`. Everything happens in `FormCreate`:

```
procedure TfrmHelloPDF.FormCreate(Sender: TObject);
var
  vPage: TsgcPDFPage;
  vFileName: string;
begin
  PDF.Info.Title := 'Hello PDF';
  PDF.Info.Author := 'sgcPDF QuickStart';

  vPage := PDF.AddPage(595, 842);

  vPage.DrawText('Hello PDF', 72, 760, sfHelveticaBold, 36);
  vPage.DrawText('This page was created with sgcPDF.', 72, 720, sfHelvetica,
    14);
  vPage.DrawText('It was saved to a file and loaded into the viewer.', 72,
    700, sfHelvetica, 14);

  vPage.DrawLine(72, 745, 523, 745, 1);

  vFileName := ExtractFilePath(ParamStr(0)) + 'HelloPDF.pdf';
  PDF.SaveToFile(vFileName);

  Viewer.LoadFromFile(vFileName);
end;
```

That is all. One A4 page, three lines of text in Helvetica, a rule under the title, a file named `HelloPDF.pdf` next to the executable, and the same file on screen. You did not write any PDF syntax, you did not embed a font and you did not need a printer driver.



Figure 1.1. The result of demo 01: the page is created in memory, saved to a file and shown in the viewer.

What you need

- **Delphi.** The packages in the `Packages` folder cover Delphi 7, 2007, 2009, 2010, XE to XE8, 10 Seattle to 10.4 Sydney, 11, 12 and 13.
- **VCL.** `TsgcPDFViewerControl`, the thumbnail and bookmark views and the barcode control run on Windows, Win32 in every version and Win64 where your Delphi version can build it.
- **FireMonkey.** `TsgcPDFViewerFMX` and the FireMonkey barcode control have their own packages from Delphi XE5 on.
- **No window at all.** The core package uses neither the VCL nor FireMonkey. Creating, editing, signing and rendering documents works in console applications, services and web servers, and the help describes the core units for Linux 64-bit, macOS, iOS and Android as well as Windows.

The QuickStart demos are VCL applications for Windows and use only units that ship with sgcPDF and with Delphi.

Part of sgcUI

sgcPDF is one product of sgcUI, the eSeGeCe suite of visual components (grids, charts, reports, dashboards, planners and more). It is installed by the same sgcUI installer as the other products, and its demos and help live in the same folders.

Unlike most sgcUI products, the sgcPDF runtime packages do not need sgcUICore, the shared drawing and theme layer of the visual products: the core package requires only the Delphi RTL. The connection goes the other way. Other sgcUI products use sgcPDF to export to PDF (sgcReports, for example, installs sgcPDF), and a small bridge, sgcUIPDF, lets the sgcUICore based controls write PDF files. With that bridge linked, the viewer control can also follow the theme of the application: set its `UseUITheme` property to `True` and the background around the pages, the selection and the annotation popup take the colours of the sgcUI theme. You do not need the bridge for anything in this guide.

Coming from other ways of making PDFs

Delphi developers usually reach PDF in one of three ways: a report generator with a PDF export, a "print to PDF" printer, or a PDF library. sgcPDF is the third kind. It is the right tool when your program decides what goes on each page, for example invoices, labels, certificates or generated documents, and when it needs to read, change, sign or show PDF files that already exist. If you need a banded report designer, look at sgcReports, which uses sgcPDF for its PDF output.

If you are moving an application from Quick PDF Library, the `MigrateFromQuickPDF` demo in `Demos\PDF\Delphi` shows the everyday tasks of such an application done with sgcPDF, one button each. Chapter 7 lists it with the other demos.

2. Install and the palette

This chapter gets sgcPDF onto your Tool Palette and the first demo running. If the components are already installed, skim the palette section and jump to Running your first demo.

Installing

sgcPDF is installed by the sgcUI installer, the same installer for every sgcUI product:

1. Run the sgcUI installer and sign in with the user name and password of your eSeGeCe account.
2. Choose the Delphi versions to install into.
3. On the components page, select **sgcPDF**. The installer shows one checkbox per product and enables the products your licence includes. Selecting sgcReports, sgcRichEdit or sgcChat selects sgcPDF as well, because those products use it.
4. The installer copies the files, compiles the packages on your machine for each selected Delphi version and installs the design time packages.

Everything goes into one sgcUI folder. The installer adds its `Source` folder and the folder of the compiled units to the library path of each selected Delphi version, so a new project finds the sgcPDF units without any setting on your side.

The packages

You only need this section if you build the packages yourself. sgcPDF is split into a framework neutral core package and one package per framework, so a FireMonkey application never links the VCL, and a service that only writes PDF files links neither. Every package name ends with the suffix of the Delphi version, for example `D13` for Delphi 13 or `D7` for Delphi 7.

Package	Kind	Contains
`sgcPDF<V>`	Runtime	Every platform independent unit: the document, the editor, the parser and writer, fonts, images, tables, forms, annotations, encryption, signatures, PDF/A, text and image extraction, the optimizer and the raster renderer. Requires only `rtl`.
`sgcPDFVCL<V>`	Runtime	The Windows and VCL units: `TsgcPDFViewerControl`, `TsgcPDFThumbnailView`, `TsgcPDFOutlineView`, `TsgcPDFViewer`, `TsgcPDFExporter`, `TsgcPDFBarcodeControl` and the GDI renderer.
`sgcPDFFMX<V>`	Runtime, XE5 and later	`TsgcPDFViewerFMX`, the FireMonkey barcode control and the FireMonkey renderer.
`dclsgcPDF<V>`	Design time	Registers the core and VCL components on the palette.
`dclsgcPDFFMX<V>`	Design time, XE5 and later	Registers the FireMonkey components.
`sgcUIPDF<V>`	Runtime	The bridge between sgcUICore and sgcPDF, used by the other sgcUI products for their PDF export. It has no components of its own.

Build the runtime packages in dependency order: `sgcPDF`, then `sgcPDFVCL` and, for FireMonkey, `sgcPDFFMX`. The design time packages also require the sgcUI design package (`dclsgcUI<V>`), so build sgcUICore first if you build everything by hand. Then install `dclsgcPDF` and add `sgcUI\Source` to the library path of your Delphi version (**Tools** > **Options** > **Language** > **Delphi** > **Library**) for every platform you use.

> **Tip.** Every sgcPDF component can also be created in code, with no form and no installed package. `TsgcPDFDocument.Create(nil)` is enough, as demos 04 and 05 show, so console applications and services only need the `Source` folder on the search path.

The palette page

After the installation the Tool Palette has a page named **sgcPDF**. In a VCL project it shows these components:

Component	Visual	What it does
`TsgcPDFDocument`	no	The PDF document: create pages, draw content, load, save, encrypt.
`TsgcPDFEditor`	no	Editing of existing PDF files: watermarks, page numbers, merge, forms.
`TsgcPDFViewer`	no	A native window that displays, searches and prints a PDF.
`TsgcPDFTextExtractor`	no	The text of a page, with fonts and character positions.
`TsgcPDFExporter`	no	Renders pages to BMP, PNG or JPEG files and thumbnails.
`TsgcPDFOptimizer`	no	Reduces the size of a PDF file.
`TsgcPDFSignatureHandler`	no	Signs PDF documents.
`TsgcPDFSignatureVerifier`	no	Verifies the signatures of a PDF document.
`TsgcPDFRedactor`	no	Removes text, images and graphics from areas of the pages.
`TsgcPDFTextOutliner`	no	Converts all the text of a PDF to vector outlines.
`TsgcPDFImageExtractor`	no	Lists and saves the images of a PDF.
`TsgcPDFHTMLRenderer`	no	Converts HTML and CSS to PDF pages.
`TsgcPDFViewerControl`	yes	The viewer control: zoom, view modes, selection, search, form filling, printing.
`TsgcPDFThumbnailView`	yes	Page thumbnails for a `TsgcPDFViewerControl`.
`TsgcPDFOutlineView`	yes	The bookmarks tree for a `TsgcPDFViewerControl`.
`TsgcPDFCommentsView`	yes	The list of comments (annotations) for a `TsgcPDFViewerControl`.
`TsgcPDFBarcodeControl`	yes	A barcode and QR Code control, with PNG, SVG and PDF output.

In a FireMonkey project the same page shows `TsgcPDFViewerFMX`, `TsgcPDFBarcodeFMX` and `TsgcPDFCommentsViewFMX`. The QuickStart demos use three components of the list: `TsgcPDFDocument`, `TsgcPDFViewerControl` and `TsgcPDFThumbnailView`. They also use `TsgcPDFTable`, which is a plain class created in code and therefore not on the palette.

Licences and the trial

sgcPDF is sold on its own, with Single, Team and Site licences, and it is also part of sgcUI Enterprise and of the All-Access bundle. Every licence includes the full source code. A free trial is available from the product page for evaluation; it has the same components, so every demo in this guide runs with it. The current prices and conditions are on the product page listed in chapter 7.

Where the demos are

The demos live in `Demos\PDF\Delphi` under your sgcUI folder. The QuickStart is the first subfolder:

Folder	Project	What it shows
01.HelloPDF	`HelloPDF.dpr`	One page with a title, two lines and a rule, saved to a file and shown in the viewer
02.PDFDrawing	`PDFDrawing.dpr`	The drawing API: fonts, a TrueType font, shapes, colours, a gradient, transparency and an image
03.PDFTablePages	`PDFTablePages.dpr`	A 70 row table that flows over several pages, with a title and page numbers
04.PDFOpenView	`PDFOpenView.dpr`	A small PDF viewer with a toolbar and page thumbnails that opens any PDF file
05.PDFInvoiceApp	`PDFInvoiceApp.dpr`	An invoice generator: data typed on the form, a live preview and Save to PDF

Every demo folder holds the project, its unit and form, and a `readme.html` with a short description. The `readme.html` in `00.QuickStart` itself is the index of the five demos. The other folders in `Demos\PDF\Delphi` are larger demos, listed in chapter 7.

Running your first demo

1. In Delphi, choose **File** > **Open Project** and open `Demos\PDF\Delphi\00.QuickStart\01.HelloPDF\HelloPDF.dpr`.
2. Press **F9**. The window opens with the page already on screen.
3. Look in the output folder of the project: `HelloPDF.pdf` is there. Open it with any PDF reader; its document properties show the title “Hello PDF”, the value of `PDF.Info.Title`.
4. Back in the IDE, open `uHelloPDF.pas`. The comment at the top explains the two components; the event handler below it is the whole program.

> **Tip.** Every QuickStart unit starts with a comment that describes the form and the idea of the demo. Read it first. To see every property of a form at once, right click the form in the designer and choose **View as Text**; the DFM excerpts in this guide come from that view.

3. Core concepts

Eight ideas explain almost everything the QuickStart demos do. Read them once now, and come back to them when a demo surprises you.

The document and its pages

Everything starts with a `TsgcPDFDocument` (unit `sgcPDF_Document`). It holds one PDF document in memory: its pages, the fonts and images the pages use, and the document information such as the title and the author (`Info.Title`, `Info.Author`). You can drop it on a form, as demos 01 to 03 do, or create it in code with `TsgcPDFDocument.Create(nil)`, as demos 04 and 05 do.

A new document has no pages. `AddPage(Width, Height)` appends one and returns it as a `TsgcPDFPage`, the object you draw on. Keep that reference in a local variable while you draw. To get back to a page later, use `Pages[Index]`, which counts from 0, together with `PageCount`. Demo 03 uses both to put a footer on every page after the table has created them.

Member	What it does
`AddPage(W, H)`	Appends a page of W by H points and returns it.
`Pages[i]`, `PageCount`	The pages of the document, from 0 to `PageCount - 1`.
`InsertPage`, `DeletePage`, `MovePage`	Change the order of the pages.
`Info`	Title, author and the other document information saved with the file.
`AddTrueTypeFont`, `AddImage`, `AddImageFromStream`	Add a font or an image to the document and return its index.
`SaveToFile`, `SaveToStream`	Write the document.
`LoadFromFile`, `LoadFromStream`	Read an existing PDF file into the document.

Points and the bottom left origin

A PDF page is measured in points, 72 points per inch, whatever the resolution of the screen or the printer. An A4 page is 595 points wide and 842 points high; a US Letter page is 612 by 792. For a landscape page, swap the two numbers: `AddPage(842, 595)`.

The point (0, 0) is the **bottom left** corner of the page, and Y grows upwards. The top of an A4 page is Y = 842. This is the opposite of a VCL canvas, where Y grows downwards, and it is the one idea that most often puts a text in an unexpected place.

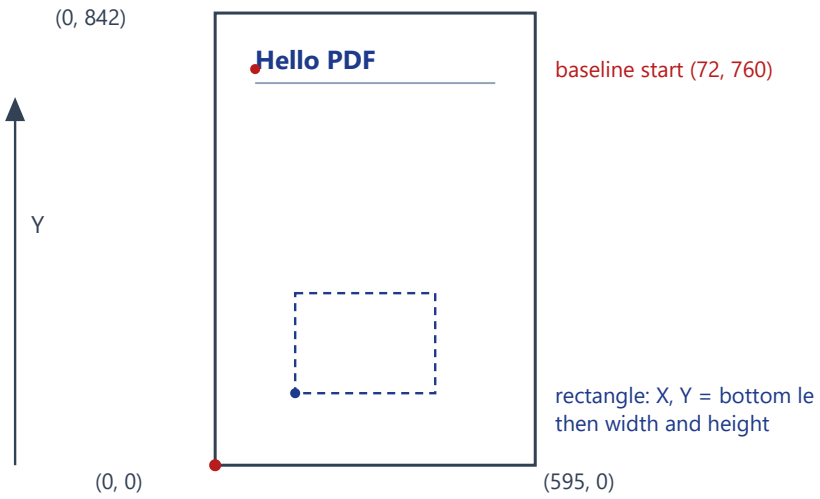


Figure 3.1. Page coordinates. The origin is the bottom left corner; a text is positioned by the start of its baseline, a rectangle by its bottom left corner.

Three rules follow from this:

- **Text.** X and Y of `DrawText` are the start of the *baseline* of the text, the line the letters sit on. Descenders such as g and p go below it.
- **Rectangles and images.** X and Y are the *bottom left* corner, followed by the width and the height. In demo 02 the title band is `DrawRectangleFill(0, 762, 595, 80, ...)`: it starts 762 points above the bottom and is 80 points high, so it reaches the top edge at 842.
- **Tables.** The exception: a table is placed by its *top* left corner, because it grows downwards. Section 3.5 shows it.

> **Tip.** When you lay out a page, write down the Y of the top margin (for A4 with a 40 point margin, 802) and count downwards from there. All the demos use this habit.

Drawing: text, shapes, colours and images

All drawing is done with methods of `TsgcPDFPage`. Their names say what they draw, and the demos use these:

Method	Draws
`DrawText`, `DrawTextColor`	Text with a standard font, in black or in a colour.
`DrawTextTT`, `DrawTextTTColor`	Text with an embedded TrueType font, any Unicode character.
`DrawLine`, `DrawLineColor`	A line between two points, with a width.
`DrawRectangle`, `DrawRectangleFill`	The outline of a rectangle, or a filled rectangle.
`DrawCircle`, `DrawCircleFill`	The outline of a circle, or a filled circle, from its centre and radius.
`DrawLinearGradient`	A rectangle filled with a gradient between two colours.
`DrawRectangleTransparent`	A filled rectangle with an opacity from 0 (invisible) to 1 (solid).
`DrawImageFromIndex`	An image added to the document, scaled to a width and a height.
`SetStrokeColor`	The colour of the outlines drawn after it.

Colours are three numbers from 0 to 1 for red, green and blue, not a `TColor`. So 0, 0, 0 is black, 1, 1, 1 is white and 0.13, 0.29, 0.53 is the dark blue the demos use for their headers. To convert a colour you know in the 0 to 255 range, divide each component by 255.

Images are added to the document once and drawn as often as you like. `AddImage(FileName)` or `AddImageFromStream(Stream)` accepts JPEG, PNG, BMP and TIFF and returns an index; `DrawImageFromIndex(Index, X, Y, W, H)` draws that image with its bottom left corner at X, Y, scaled to W by H points. Demo 02 paints a logo on a `TBitmap`, saves it to a stream and adds it this way.

Fonts: standard and TrueType

There are two kinds of fonts, and the choice matters more than it seems.

The 14 standard fonts are Helvetica, Times and Courier with their bold and italic variants, plus Symbol and ZapfDingbats. Every PDF reader has them, so nothing is embedded and the files stay small. You select them with the enumeration `TsgcPDFStandardFont` (unit `sgcPDF_Fonts`): `sfHelvetica`, `sfHelveticaBold`, `sfHelveticaOblique`, `sfTimesRoman`, `sfTimesBold`, `sfTimesItalic`, `sfCourier` and so on. `DrawText` and `DrawTextColor` use them. Two limits come with them:

- The text parameter is an `AnsiString`. A string literal is fine, but a `string` variable in Delphi 2009 and later needs an explicit `AnsiString(...)` cast, otherwise the compiler warns about an implicit string conversion. That is why the demos write `DrawText(AnsiString(edtCustomer.Text), ...)`.
- They only cover the Western (WinAnsi) character set. Accented letters outside it, Greek, Cyrillic, Asian scripts and other non ASCII text need a TrueType font.

TrueType fonts are loaded from the fonts installed in Windows by their family name. `AddTrueTypeFont('Segoe UI', ABold, AItalic)` returns a font index, and `DrawTextTT` or `DrawTextTTColor` draw with it. Their text is a `WideString`, so any character the font contains works. When the document is saved, only the characters you used are embedded.

The standard fonts also let you measure a text: `sgcPDFMeasureTextWidth(Font, Text, Size)` returns its width in points, which is how you right align or centre it. This short procedure, compiled for this guide, shows the cast, the measuring and a TrueType font with Unicode text:

```
uses
  SysUtils,
  sgcPDF_Document, sgcPDF_Fonts;

procedure CreateReport(const aFileName: string);
var
  oPDF: TsgcPDFDocument;
  vPage: TsgcPDFPage;
  vFont: Integer;
  vText: string;
  vWidth: Double;
begin
  oPDF := TsgcPDFDocument.Create(nil);
  try
    vPage := oPDF.AddPage(595, 842);
    { a string variable needs an AnsiString cast for the standard fonts }
    vText := 'Total: 1,234.50';
    vWidth := sgcPDFMeasureTextWidth(sfHelvetica, AnsiString(vText), 12);
    { right aligned: the text ends at X = 555 }
    vPage.DrawText(AnsiString(vText), 555 - vWidth, 780, sfHelvetica, 12);
    { accented and non Latin text needs a TrueType font }
    vFont := oPDF.AddTrueTypeFont('Arial', False, False);
    vPage.DrawTextTT('Espa'#$00F1'a, M'#$00FC'nchen, '#$041C#$043E#$0441#$043A#$0432#$0430,
      40, 740, vFont, 12);
    oPDF.SaveToFile(aFileName);
  finally
    oPDF.Free;
  end;
end;
```

Tables and page breaks

Drawing every cell of a table with `DrawText` and `DrawLine` is possible, but you would have to measure the rows and decide where a page ends yourself. `TsgcPDFTable` (unit `sgcPDF_Tables`) does that for you. It is a plain class, not a component, so you create and free it in code. The steps are always the same:

1. **Create** it with the number of columns, `TsgcPDFTable.Create(5)`, and set the width of each column in points with `SetColumnWidths`.
2. **Fill** it: `SetHeaders` adds the header row, `AddRowData` adds one data row from an array of strings. The cells are `AnsiString`, so cast string variables as in section 3.4.
3. **Style** it with `SetDefaultFont`, `SetHeaderFont`, `SetHeaderBackground`, `SetAlternateRowColors`, `SetBorder` and `SetPadding`. These apply to the whole table.
4. **Draw** it. `DrawOnPage(Page, X, Y)` draws the whole table on one page, with X, Y as its top left corner. `DrawMultiPage(Document, X, Y, MaxHeight, PageWidth, PageHeight)` adds a page, draws as many rows as fit in `MaxHeight`, adds another page when needed and repeats the header row on each one.

Demo 03 uses `DrawMultiPage` for a 70 row list, demo 05 uses `DrawOnPage` for an invoice that always fits on one page.

Cell styles. The style calls above apply to the whole table. To change one cell, read its `Style` record, change the fields you want and write it back. `Align`, `Font`, `FontSize`, the text colour (`TextR`, `TextG`, `TextB`), the background (`HasBackground` with `BackR`, `BackG`, `BackB`), the padding, the `Border` and `WordWrap` of the cell

win over the table style and over the alternate row colour. A field you leave at its default value counts as not set, so the table style still applies to it. The cells of a row are reached with `GetRow(Index).GetCell(Index)`. This is how demo 05 right aligns the numbers and makes the TOTAL row bold:

```
{ A cell Style overrides the table style for that cell only. This helper right
  aligns the number columns (1 to 3) of a row and, for the TOTAL row, also makes
  the text bold on a light background. }
procedure StyleInvoiceRow(aRow: TsgcPDFTableRow; aTotal: Boolean);
var
  i: Integer;
  vStyle: TsgcPDFTableCellStyle;
begin
  for i := 0 to aRow.CellCount - 1 do
  begin
    vStyle := aRow.GetCell(i).Style;
    if i > 0 then
      vStyle.Align := caRight;
    if aTotal then
    begin
      vStyle.Font := sfHelveticaBold;
      vStyle.HasBackground := True;
      vStyle.BackR := 0.85;
      vStyle.BackG := 0.9;
      vStyle.BackB := 1.0;
    end;
    aRow.GetCell(i).Style := vStyle;
  end;
end;
```

> **Note.** The header text of a table is white by default, and it takes its colour from `HeaderStyle` (`TextR`, `TextG`, `TextB`). The demos give the header a dark background with `SetHeaderBackground`. For a light header background, set a dark text colour in `HeaderStyle`.

Saving: file or stream

A document is written with `SaveToFile(FileName)` or `SaveToStream(Stream)`. The stream version is the one to use when the PDF does not belong on disk: to show it in the viewer, to send it as an HTTP response or to store it in a database blob. After `SaveToStream`, set `Stream.Position := 0` before you hand the stream to anything that reads it.

Nothing is written until you save, so the order of the drawing is up to you. Demo 03 relies on this: the table creates the pages first, then the title and the footers are drawn on those pages, and only then is the document saved, once.

Showing a PDF: the viewer control

`TsgcPDFViewerControl` (unit `sgcPDF_ViewerVCL`) shows a PDF document on a form. It loads a document in one of two ways:

- `Viewer.LoadFromFile(FileName)` opens any PDF file, as demos 01 and 04 do.
- `Viewer.LoadFromStream(Stream)` reads a PDF from a stream. Demos 02, 03 and 05 save their document to a `TMemoryStream` and load it from there, so nothing is written to disk.

In both cases the viewer parses the PDF into a document of its own, which it owns and frees. The `TsgcPDFDocument` you built is not connected to the viewer afterwards: drawing on it again changes nothing on screen until you save and load it again. That is also why demo 05 can free its document right after the preview.

Three properties decide how the pages appear:

Property	Values	Default
`ZoomMode`	`zmFitWidth`, `zmFitPage`, `zmFitHeight`, `zmActualSize`, `zmCustom`	`zmFitWidth`
`ViewMode`	`pvmContinuous` (all pages in a scrolling column), `pvmSinglePage`, `pvmTwoPage` (side by side, like a book)	`pvmContinuous`
`PageGap`, `BackgroundColor`	The space between pages in pixels and the colour around them	`10`, `clAppWorkSpace`

Navigation and zoom are methods: `FirstPage`, `PrevPage`, `NextPage`, `LastPage`, `GotoPage`, `ZoomIn`, `ZoomOut`, `RotateLeft` and `RotateRight`. `PageIndex` is the current page, counted from 0, `PageCount` the number of pages and `Scale` the current zoom, where 1 means 100%. Two events tell you when these change: `OnPageChange` and `OnZoomChange`. Scrolling with the mouse wheel, selecting text and following links are built in. `Print` prints the document, all pages by default.

A `TsgcPDFThumbnailView` next to the viewer shows small images of the pages. Set its `Viewer` property to the viewer control and it follows the viewer on its own: it shows the pages of the loaded document, marks the current page and moves the viewer when the user clicks a thumbnail. `TsgcPDFOutlineView` works the same way for the bookmarks of a document.

Opening and changing existing files

`TsgcPDFDocument` reads PDF files too. After `LoadFromFile`, the existing pages are in `Pages[]` and you can draw on them with the same methods, then save the result under a new name. This procedure, compiled and run for this guide, stamps “Page N of M” at the bottom of every page of an existing file:

```
procedure NumberPages(const aFileName, aOutFileName: string);
var
  oPDF: TsgcPDFDocument;
  i: Integer;
begin
  oPDF := TsgcPDFDocument.Create(nil);
  try
    oPDF.LoadFromFile(aFileName);
    for i := 0 to oPDF.PageCount - 1 do
      oPDF.Pages[i].DrawText(AnsiString(Format('Page %d of %d',
        [i + 1, oPDF.PageCount])), oPDF.Pages[i].Width - 100, 20,
        sfHelvetica, 9);
    oPDF.SaveToFile(aOutFileName);
  finally
    oPDF.Free;
  end;
end;
```

The page size is read from the file (`Width` and `Height` of the page), so the same code works for A4, Letter and landscape pages. For the common editing jobs on existing files, such as watermarks, headers and footers, merging and splitting, there is a dedicated component, `TsgcPDFEditor`, described in chapter 5.

Summary: who does what

Piece	Its job	Where you meet it
`TsgcPDFDocument`	Holds the document: pages, fonts, images, information. Loads and saves.	Every demo
`TsgcPDFPage`	One page. All the `Draw...` methods.	Every demo
`TsgcPDFStandardFont`	Selects one of the 14 standard fonts.	Every demo
`AddTrueTypeFont`, `DrawTextTT`	Embedded fonts and Unicode text.	Demo 02
`TsgcPDFTable`	Rows and columns, page breaks, repeated headers.	Demos 03 and 05
`TsgcPDFViewerControl`	Shows a PDF file or stream, zoom and navigation.	Every demo
`TsgcPDFThumbnailView`	Page thumbnails linked to a viewer.	Demo 04

4. The QuickStart demos

The five QuickStart demos build on each other. Run them in order: each one adds one or two ideas to the previous one, and the sections below assume you have seen the earlier ones.

Demo	Project	New ideas
01.HelloPDF	`HelloPDF.dpr`	The document, a page, points and coordinates, standard fonts, save to a file, the viewer
02.PDFDrawing	`PDFDrawing.dpr`	Colours, shapes, a TrueType font, a gradient, transparency, an image, showing a document from memory
03.PDFTablePages	`PDFTablePages.dpr`	A table, automatic page breaks, walking over the pages, the two page view
04.PDFOpenView	`PDFOpenView.dpr`	Opening any PDF file, viewer methods and events, thumbnails
05.PDFInvoiceApp	`PDFInvoiceApp.dpr`	A complete generate, preview and save workflow driven by data typed on the form

> **Note.** The demos need sgcPDF installed, so the components appear on the sgcPDF palette page and the `sgcUI\Source` folder is on the library path. They build for Delphi 7 as well as for the current versions, Win32 and Win64. Demos 01 and 04 write a PDF file next to the executable, so run them from a folder where you can write.

4.1 Demo 01: Hello PDF

Folder `01.HelloPDF`, project `HelloPDF.dpr`.

What you will build

The smallest sgcPDF application. When the form is created, the program builds a one page A4 document with a title, two lines of text and a rule under the title, saves it as `HelloPDF.pdf` next to the executable and opens that file in the viewer.



Figure 4.1. Demo 01. With `ZoomMode = zmFitPage` the whole page is visible and scales with the window.

Components on the form

Component	Class	Properties you set
`PDF`	`TsgcPDFDocument`	Nothing. It is a non visual component, everything is done in code.
`Viewer`	`TsgcPDFViewerControl`	`Align = alClient`, `ZoomMode = zmFitPage`.

In the form file the two components look like this:

```
  object Viewer: TsgcPDFViewerControl
    Left = 0
    Top = 0
    Width = 480
    Height = 560
    Align = alClient
    TabOrder = 0
    ZoomMode = zmFitPage
  end
  object PDF: TsgcPDFDocument
    Left = 40
    Top = 40
  end
```

The code

The `uses` clause names three sgcPDF units: the document, the standard fonts (for `sfHelvetica`) and the VCL viewer:

```
uses
  SysUtils, Classes, Controls, Forms,
  // sgc
  sgcPDF_Document, sgcPDF_Fonts, sgcPDF_ViewerVCL;
```

And this is the only event handler of the demo:

```
procedure TfrmHelloPDF.FormCreate(Sender: TObject);
var
  vPage: TsgcPDFPage;
  vFileName: string;
begin
  { The document information is saved with the file: a PDF reader shows the
    title in its caption and in the document properties. }
  PDF.Info.Title := 'Hello PDF';
  PDF.Info.Author := 'sgcPDF QuickStart';

  { AddPage returns the new page, 595 x 842 points is A4. }
  vPage := PDF.AddPage(595, 842);

  { DrawText writes a text with one of the 14 standard PDF fonts
    (Helvetica, Times, Courier...), which every PDF reader knows, so nothing
    has to be embedded. X and Y are the start of the baseline of the text. }
  vPage.DrawText('Hello PDF', 72, 760, sfHelveticaBold, 36);
  vPage.DrawText('This page was created with sgcPDF.', 72, 720, sfHelvetica,
    14);
  vPage.DrawText('It was saved to a file and loaded into the viewer.', 72,
    700, sfHelvetica, 14);

  { A line under the title: from (72, 745) to (523, 745), 1 point wide. }
```

```
  vPage.DrawLine(72, 745, 523, 745, 1);

  { Save the document. The same document can also be written to any TStream
    with SaveToStream. }
  vFileName := ExtractFilePath(ParamStr(0)) + 'HelloPDF.pdf';
  PDF.SaveToFile(vFileName);

  { And show the file. LoadFromFile works with any PDF file. }
  Viewer.LoadFromFile(vFileName);
end;
```

How it works

1. The form loads from its DFM. The document is empty and the viewer shows nothing.
2. `FormCreate` fills in the document information. It is saved with the file, and PDF readers show it in the document properties.
3. `AddPage(595, 842)` appends an A4 page and returns it. All the drawing goes through `vPage`.
4. The three `DrawText` calls write the text. The title starts at X = 72, one inch from the left edge, and its baseline is at Y = 760, which is 82 points below the top of the page because Y counts from the bottom.
5. `DrawLine` draws the rule from (72, 745) to (523, 745), one point wide: one inch of margin on each side.
6. `SaveToFile` writes `HelloPDF.pdf`, and `Viewer.LoadFromFile` opens that file, as it would open any other PDF.

Try this

- Write the title in `sfTimesBold` with a size of 48.
- Replace the first `DrawText` with `vPage.DrawTextColor('Hello PDF', 72, 760, sfHelveticaBold, 36, 0.13, 0.29, 0.53)` for a blue title.
- Make the page landscape with `PDF.AddPage(842, 595)`. The text stays near the top only if you also lower its Y values, because the page is now 595 points high.

4.2 Demo 02: PDF Drawing

Folder `02.PDFDrawing`, project `PDFDrawing.dpr`.

What you will build

One page that shows the drawing API: a coloured title band with white text, a sample of the standard fonts, a line in an embedded TrueType font, a rectangle, a filled rectangle, a circle, a filled circle and a coloured line, a gradient with a half transparent rectangle over it, and a small logo painted in code and placed as an image. This time nothing is written to disk: the document goes to the viewer through a memory stream.

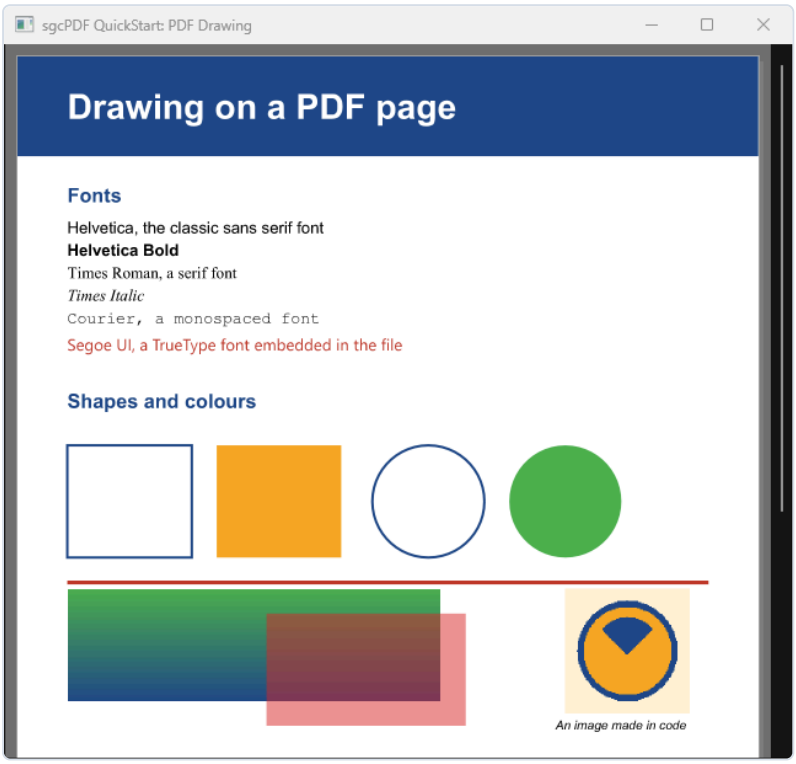


Figure 4.2. Demo 02. The viewer uses the default `ZoomMode = zmFitWidth`, so the page fills the width and you scroll down for the rest.

Components on the form

Component	Class	Properties you set
`PDF`	`TsgcPDFDocument`	Nothing.
`Viewer`	`TsgcPDFViewerControl`	`Align = alClient`, `BackgroundColor` set to a dark grey (the colour around the page). `ZoomMode` keeps its default, `zmFitWidth`.

The code

The `uses` clause adds `Graphics` for the bitmap of the logo:

```
uses
  SysUtils, Classes, Graphics, Controls, Forms,
  // sgc
  sgcPDF_Document, sgcPDF_Fonts, sgcPDF_ViewerVCL;
```

The logo is painted on a `TBitmap` with the usual VCL canvas calls, saved to a stream and added to the document. The function returns the index of the image:

```
{ Paints a small logo on a TBitmap and adds it to the document. A document
  accepts JPEG, PNG, BMP and TIFF images, from a file (AddImage) or from a
  stream (AddImageFromStream). The result is the index of the image, used
  later to draw it on a page. }
function TfrmPDFDrawing.AddLogoImage: Integer;
var
  vBitmap: TBitmap;
  vStream: TMemoryStream;
begin
  vBitmap := TBitmap.Create;
  try
    vBitmap.PixelFormat := pf24bit;
    vBitmap.Width := 120;
    vBitmap.Height := 120;
    { TColor values are $00BBGGRR: blue, green, red }
    vBitmap.Canvas.Pen.Color := $00D6F4FF;
    vBitmap.Canvas.Brush.Color := $00D6F4FF;
    vBitmap.Canvas.Rectangle(0, 0, 120, 120);
    vBitmap.Canvas.Pen.Color := $00874A20;
    vBitmap.Canvas.Pen.Width := 6;
    vBitmap.Canvas.Brush.Color := $0023A6F5;
    vBitmap.Canvas.Ellipse(14, 14, 106, 106);
    vBitmap.Canvas.Brush.Color := $00874A20;
    vBitmap.Canvas.Pie(30, 30, 90, 90, 90, 30, 30, 30);

    vStream := TMemoryStream.Create;
    try
      vBitmap.SaveToStream(vStream);
      vStream.Position := 0;
      Result := PDF.AddImageFromStream(vStream);
    finally
      vStream.Free;
    end;
  finally
    vBitmap.Free;
  end;
end;
```

Everything else happens in `FormCreate`:

```
procedure TfrmPDFDrawing.FormCreate(Sender: TObject);
var
  vPage: TsgcPDFPage;
  vFont, vImage: Integer;
  vStream: TMemoryStream;
begin
  vPage := PDF.AddPage(595, 842);

  { A coloured band at the top: a filled rectangle as wide as the page, and
    a white title on it. }
  vPage.DrawRectangleFill(0, 762, 595, 80, 0.13, 0.29, 0.53);
  vPage.DrawTextColor('Drawing on a PDF page', 40, 792, sfHelveticaBold, 28,
    1, 1, 1);

  { Fonts. The 14 standard fonts need nothing embedded. }
  vPage.DrawTextColor('Fonts', 40, 725, sfHelveticaBold, 16, 0.13, 0.29, 0.53);
  vPage.DrawText('Helvetica, the classic sans serif font', 40, 700,
```

```
    sfHelvetica, 13);
  vPage.DrawText('Helvetica Bold', 40, 682, sfHelveticaBold, 13);
  vPage.DrawText('Times Roman, a serif font', 40, 664, sfTimesRoman, 13);
  vPage.DrawText('Times Italic', 40, 646, sfTimesItalic, 13);
  vPage.DrawText('Courier, a monospaced font', 40, 628, sfCourier, 13);

  { A TrueType font installed in Windows. AddTrueTypeFont embeds the font in
    the file (only the characters used) and returns its index; DrawTextTT
    and DrawTextTTColor draw with it and accept any Unicode text. }
  vFont := PDF.AddTrueTypeFont('Segoe UI', False, False);
  vPage.DrawTextTTColor('Segoe UI, a TrueType font embedded in the file', 40,
    606, vFont, 13, 0.75, 0.22, 0.17);

  { Shapes. SetStrokeColor sets the colour of the lines drawn after it. }
  vPage.DrawTextColor('Shapes and colours', 40, 560, sfHelveticaBold, 16,
    0.13, 0.29, 0.53);
  vPage.SetStrokeColor(0.13, 0.29, 0.53);
  vPage.DrawRectangle(40, 440, 100, 90, 2);
  vPage.DrawRectangleFill(160, 440, 100, 90, 0.96, 0.65, 0.14);
  vPage.DrawCircle(330, 485, 45, 2);
  vPage.DrawCircleFill(440, 485, 45, 0.30, 0.69, 0.31);
  vPage.DrawLineColor(40, 420, 555, 420, 3, 0.75, 0.22, 0.17);

  { A gradient from blue (bottom edge) to green (top edge), and a transparent
    red rectangle on top of it: the opacity goes from 0 (invisible) to 1
    (solid). }
  vPage.DrawLinearGradient(40, 325, 300, 90, 0.13, 0.29, 0.53, 0.30, 0.69,
    0.31);
  vPage.DrawRectangleTransparent(200, 305, 160, 90, 0.86, 0.15, 0.15, 0.5);

  { The image, drawn 100 x 100 points with its bottom left corner at
    (440, 315). The width and height scale the image. }
  vImage := AddLogoImage;
  vPage.DrawImageFromIndex(vImage, 440, 315, 100, 100);
  vPage.DrawText('An image made in code', 432, 302, sfHelveticaOblique, 10);

  { Save to memory and show it. }
  vStream := TMemoryStream.Create;
  try
    PDF.SaveToStream(vStream);
    vStream.Position := 0;
    Viewer.LoadFromStream(vStream);
  finally
    vStream.Free;
  end;
end;
```

How it works

1. The title band is a filled rectangle as wide as the page: bottom left corner at (0, 762), 595 points wide and 80 high, so it touches the top edge. The title is drawn on top of it in white (1, 1, 1).
2. The font samples use the standard fonts with `DrawText`. Each line is 18 points below the previous one.
3. `AddTrueTypeFont('Segoe UI', False, False)` loads the Windows font and returns an index. `DrawTextTTColor` draws with it; only the characters used are embedded when the document is saved.
4. `SetStrokeColor` sets the colour of the outlines drawn after it, so the rectangle and the circle outlines are dark blue. The filled shapes take their colour from their own parameters.
5. `DrawLinearGradient` fills its rectangle from the first colour at the bottom edge to the second colour at the top edge. `DrawRectangleTransparent` then paints a red rectangle with an opacity of 0.5 over part of it.

6. `AddLogoImage` adds the bitmap to the document once; `DrawImageFromIndex` draws it at (440, 315), scaled to 100 by 100 points.
7. The document is saved to a `TMemoryStream`. After `Position := 0`, `Viewer.LoadFromStream` reads it. The viewer keeps its own copy, so the stream can be freed at once.

> **Tip.** A `TColor` is `$00BBGGRR`, blue first, while the PDF drawing methods take red, green and blue from 0 to 1. The logo uses the first kind because it is painted on a VCL canvas; the page uses the second kind.

Try this

- Use a photo instead of the painted logo: `vImage := PDF.AddImage('photo.jpg');` with a JPEG file of your own.
- Change the opacity of `DrawRectangleTransparent` from 0.5 to 0.2.
- Load the bold variant of the TrueType font with `PDF.AddTrueTypeFont('Segoe UI', True, False)`.

4.3 Demo 03: PDF Table Pages

Folder `03.PDFTablePages`, project `PDFTablePages.dpr`.

What you will build

A product list of 70 rows that does not fit on one page. A `TsgcPDFTable` lays the rows out, starts a new page whenever the current one is full and repeats the header row at the top of each page. After the table, the program adds a title to the first page and a footer with "Page N of M" to every page. The viewer shows two pages side by side, and the window caption shows how many pages the table needed.

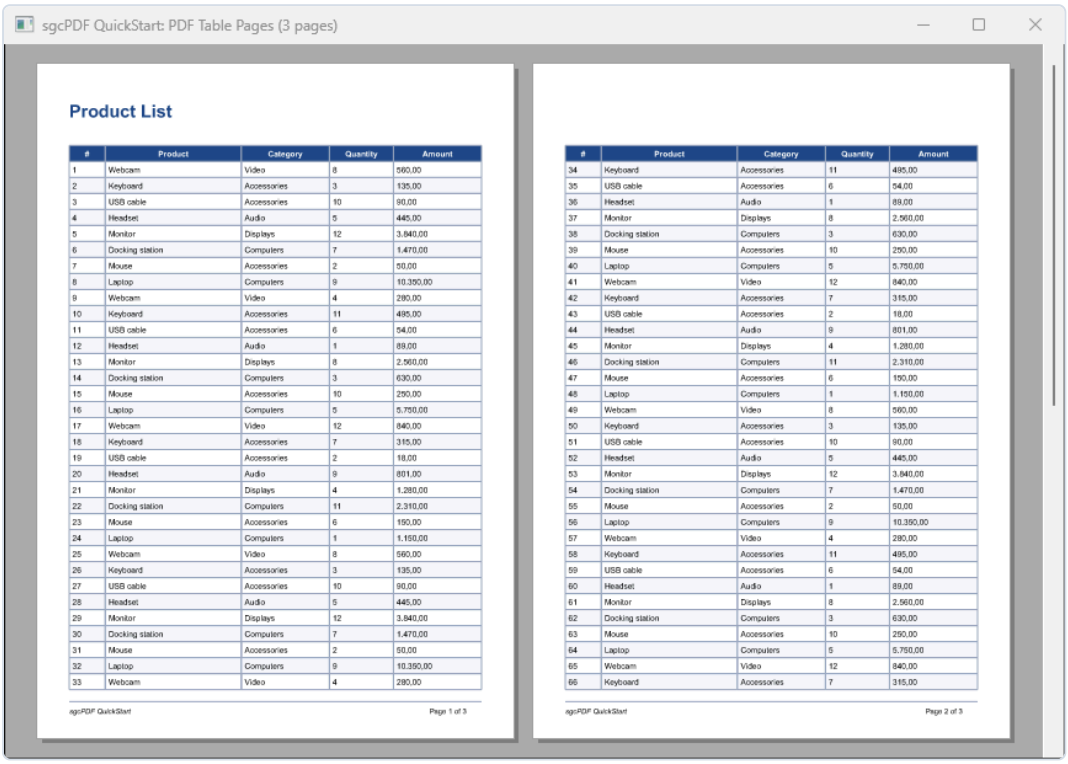


Figure 4.3. Demo 03 with `ViewMode = pvmTwoPage`. The header row is repeated on the second page.

Components on the form

Component	Class	Properties you set
`PDF`	`TsgcPDFDocument`	Nothing.
`Viewer`	`TsgcPDFViewerControl`	`Align = alClient`, `ViewMode = pvmTwoPage`, `ZoomMode = zmFitPage`, `PageGap = 16`.

```
object Viewer: TsgcPDFViewerControl
  Left = 0
  Top = 0
  Width = 860
  Height = 580
  Align = alClient
  TabOrder = 0
  ZoomMode = zmFitPage
  ViewMode = pvmTwoPage
  PageGap = 16
end
```

The code

The `uses` clause adds `sgcPDF_Tables`. The sample data is three constant arrays:

```
uses
  SysUtils, Classes, Controls, Forms,
  // sgc
  sgcPDF_Document, sgcPDF_Fonts, sgcPDF_Tables, sgcPDF_ViewerVCL;

  // ...

const
  CS_PRODUCTS: array [0 .. 7] of string = ('Laptop', 'Monitor', 'Keyboard',
    'Mouse', 'Headset', 'Webcam', 'Docking station', 'USB cable');
  CS_CATEGORIES: array [0 .. 7] of string = ('Computers', 'Displays',
    'Accessories', 'Accessories', 'Audio', 'Video', 'Computers',
    'Accessories');
  CS_PRICES: array [0 .. 7] of Double = (1150, 320, 45, 25, 89, 70, 210, 9);
```

`DrawTable` follows the four steps of section 3.5: create, fill, style, draw:

```
procedure TfrmPDFTablePages.DrawTable;
var
  oTable: TsgcPDFTable;
  i, vProduct, vQuantity: Integer;
begin
  oTable := TsgcPDFTable.Create(5);
  try
    { 515 points: the width of an A4 page (595) less two margins of 40. }
    oTable.SetColumnWidths([45, 170, 110, 80, 110]);

    oTable.SetHeaders(['#', 'Product', 'Category', 'Quantity', 'Amount']);
    for i := 1 to 70 do
    begin
      vProduct := (i * 5) mod 8;
      vQuantity := 1 + (i * 7) mod 12;
      { The cells of a table are AnsiString. }
      oTable.AddRowData([AnsiString(IntToStr(i)),
        AnsiString(CS_PRODUCTS[vProduct]), AnsiString(CS_CATEGORIES[vProduct]),
        AnsiString(IntToStr(vQuantity)),
        AnsiString(FormatFloat('#,##0.00', vQuantity * CS_PRICES[vProduct]))]);
    end;

    oTable.SetDefaultFont(sfHelvetica, 10);
    oTable.SetHeaderFont(sfHelveticaBold, 10);
    { The header text colour comes from HeaderStyle (white by default). }
    oTable.SetHeaderBackground(0.13, 0.29, 0.53);
    oTable.SetAlternateRowColors(0.96, 0.97, 0.99);
    oTable.SetBorder(0.5, 0.6, 0.65, 0.75);

    { The top of the table is at Y = 740 and it may use 680 points of
      height on every page, so it never goes below Y = 60, where the
      footer is. The pages are A4, 595 x 842. }
    oTable.DrawMultiPage(PDF, 40, 740, 680, 595, 842);
  finally
    oTable.Free;
  end;
end;
```

When `DrawMultiPage` returns, the document has all its pages. `DrawTitleAndFooters` walks over them, and `FormCreate` puts it all together:

```
procedure TfrmPDFTablePages.DrawTitleAndFooters;
var
  i: Integer;
  vPage: TsgcPDFPage;
begin
  { The title, on the first page only. }
  vPage := PDF.Pages[0];
  vPage.DrawTextColor('Product List', 40, 775, sfHelveticaBold, 22, 0.13,
    0.29, 0.53);

  { A footer on every page: a thin line and "Page N of M". }
  for i := 0 to PDF.PageCount - 1 do
  begin
    vPage := PDF.Pages[i];
    vPage.DrawLineColor(40, 45, 555, 45, 0.5, 0.6, 0.65, 0.75);
    vPage.DrawText(AnsiString(Format('Page %d of %d', [i + 1, PDF.PageCount])),
      490, 30, sfHelvetica, 9);
    vPage.DrawText('sgcPDF QuickStart', 40, 30, sfHelveticaOblique, 9);
  end;
end;

  // ...

procedure TfrmPDFTablePages.FormCreate(Sender: TObject);
begin
  DrawTable;
  DrawTitleAndFooters;
  ShowDocument;
  Caption := Format('sgcPDF QuickStart: PDF Table Pages (%d pages)',
    [Viewer.PageCount]);
end;
```

`ShowDocument` is the same memory stream code as in demo 02.

How it works

1. `TsgcPDFTable.Create(5)` creates a table with five columns. `SetColumnWidths` gives them 515 points in total, the width of an A4 page less two margins of 40 points.
2. `SetHeaders` adds the header row and the loop adds 70 rows with `AddRowData`. Every cell is an `AnsiString`, so each value is cast.
3. The style calls apply to the whole table: Helvetica 10 for the rows, Helvetica Bold for the header, a dark blue header background (the header text is white by default), a light alternate row colour and a thin grey border.
4. `DrawMultiPage(PDF, 40, 740, 680, 595, 842)` adds an A4 page, draws the table with its top left corner at (40, 740) and uses at most 680 points of height, so it never goes below Y = 60. When the next row does not fit, it adds a page, draws the header row again and goes on.
5. `DrawTitleAndFooters` draws the title above the table on `Pages[0]`, then loops from 0 to `PageCount - 1` and draws a line and the page number at the bottom of each page, below Y = 60 where the table never reaches.
6. The document is shown through a memory stream, and the caption reads the page count back from `Viewer.PageCount`.

Try this

- Change the loop to 200 rows. The table simply takes more pages, and the footers still say the right “of M”.
- Set `ViewMode = pvmContinuous` and `ZoomMode = zmFitWidth` on the viewer to scroll through the pages one below the other.
- Add `oTable.SetPadding(6, 6, 6, 6);` before the draw call for more space inside the cells. Fewer rows fit on each page.

4.4 Demo 04: PDF Open View

Folder `04.PDFOpenView`, project `PDFOpenView.dpr`.

What you will build

A small PDF viewer application. A toolbar has an Open button, First, Prev, Next and Last buttons, zoom out and zoom in buttons and a label with the file name, the current page and the zoom. On the left, a thumbnail view shows all the pages; on the right, the viewer shows one page at a time. So that there is something to look at the first time, the demo writes a five page `SampleBook.pdf` next to the executable and opens it. The Open button opens any other PDF file.

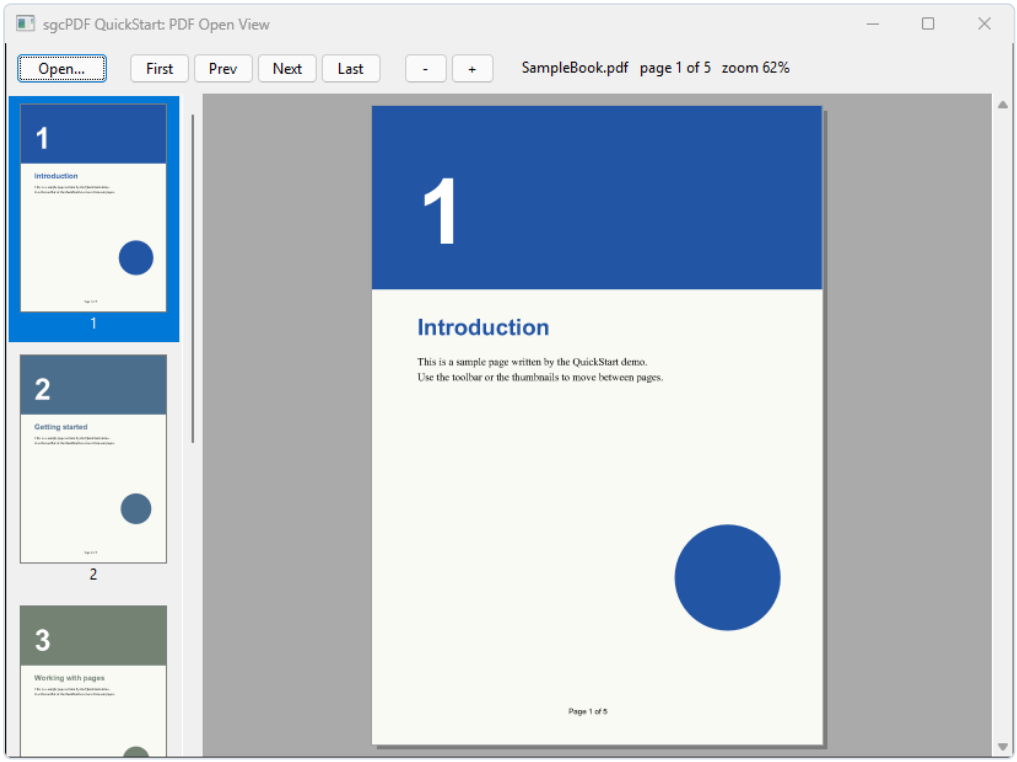


Figure 4.4. Demo 04. The label reads the file name, “page 1 of 5” and the current zoom.

Components on the form

Component	Class	Properties you set
`pnlToolbar`	`TPanel`	`Align = alTop`. Holds the seven buttons and `lblInfo`; each button has an `OnClick` handler.
`Thumbs`	`TsgcPDFThumbnailView`	`Align = alLeft`, `Viewer = Viewer`.
`Viewer`	`TsgcPDFViewerControl`	`Align = alClient`, `ViewMode = pvmSinglePage`, `ZoomMode = zmFitPage`, `OnPageChange` and `OnZoomChange` assigned.
`dlgOpen`	`TOpenDialog`	`Filter` for PDF files.

The only link between the thumbnails and the viewer is one property in the form file:

```
  object Thumbs: TsgcPDFThumbnailView
    Left = 0
    Top = 41
    Width = 160
    Height = 539
    Align = alLeft
    TabOrder = 1
    Viewer = Viewer
  end
  object Viewer: TsgcPDFViewerControl
    Left = 160
    Top = 41
    Width = 660
    Height = 539
    Align = alClient
    TabOrder = 2
    ZoomMode = zmFitPage
    ViewMode = pvmSinglePage
    OnPageChange = ViewerPageChange
    OnZoomChange = ViewerZoomChange
  end
```

The code

The sample file is written with a `TsgcPDFDocument` created in code, with no owner, and freed when it has been saved. Each page has its own colour:

```
const
  CS_CHAPTERS: array [0 .. 4] of string = ('Introduction', 'Getting started',
    'Working with pages', 'Images and fonts', 'Summary');

  // ...

{ Writes a five page PDF: one chapter per page, each with its own colour. }
procedure TfrmPDFOpenView.CreateSampleFile(const aFileName: string);
var
  oDocument: TsgcPDFDocument;
  vPage: TsgcPDFPage;
  i: Integer;
  r, g, b: Double;
begin
  oDocument := TsgcPDFDocument.Create(nil);
  try
    oDocument.Info.Title := 'sgcPDF Sample Book';
    for i := 0 to 4 do
    begin
      r := 0.15 + 0.15 * i;
      g := 0.35 + 0.08 * i;
      b := 0.65 - 0.1 * i;
      vPage := oDocument.AddPage(595, 842);
      vPage.DrawRectangleFill(0, 0, 595, 842, 0.98, 0.98, 0.96);
      vPage.DrawRectangleFill(0, 600, 595, 242, r, g, b);
      vPage.DrawTextColor(AnsiString(IntToStr(i + 1)), 60, 660, sfHelveticaBold,
        120, 1, 1, 1);
      vPage.DrawTextColor(AnsiString(CS_CHAPTERS[i]), 60, 540, sfHelveticaBold,
        30, r, g, b);
      vPage.DrawText('This is a sample page written by the QuickStart demo.',
        60, 500, sfTimesRoman, 14);
      vPage.DrawText('Use the toolbar or the thumbnails to move between pages.',
        60, 480, sfTimesRoman, 14);
```

```
      vPage.DrawCircleFill(470, 220, 70 - 8 * i, r, g, b);
      vPage.DrawText(AnsiString(Format('Page %d of 5', [i + 1])), 260, 40,
        sfHelvetica, 10);
    end;
    oDocument.SaveToFile(aFileName);
  finally
    oDocument.Free;
  end;
end;
```

Opening a file and showing the state of the viewer:

```
procedure TfrmPDFOpenView.OpenFile(const aFileName: string);
begin
  Viewer.LoadFromFile(aFileName);
  FFileName := aFileName;
  UpdateInfo;
end;

procedure TfrmPDFOpenView.UpdateInfo;
begin
  if Viewer.PageCount = 0 then
    lblInfo.Caption := 'No document'
  else
    { PageIndex starts at 0. Scale is the current zoom, 1 = 100%. }
    lblInfo.Caption := Format('%s   page %d of %d   zoom %d%%',
      [ExtractFileName(FFileName), Viewer.PageIndex + 1, Viewer.PageCount,
      Round(Viewer.Scale * 100)]);
end;

procedure TfrmPDFOpenView.FormCreate(Sender: TObject);
var
  vFileName: string;
begin
  vFileName := ExtractFilePath(ParamStr(0)) + 'SampleBook.pdf';
  if not FileExists(vFileName) then
    CreateSampleFile(vFileName);
  OpenFile(vFileName);
end;

procedure TfrmPDFOpenView.btnOpenClick(Sender: TObject);
begin
  if dlgOpen.Execute then
    OpenFile(dlgOpen.FileName);
end;
```

The toolbar buttons and the two viewer events are one line each:

```
procedure TfrmPDFOpenView.btnFirstClick(Sender: TObject);
begin
  Viewer.FirstPage;
end;

procedure TfrmPDFOpenView.btnPrevClick(Sender: TObject);
begin
  Viewer.PrevPage;
end;

  // ...

procedure TfrmPDFOpenView.btnZoomInClick(Sender: TObject);
begin
  Viewer.ZoomIn;
end;

procedure TfrmPDFOpenView.ViewerPageChange(Sender: TObject);
begin
  UpdateInfo;
end;

procedure TfrmPDFOpenView.ViewerZoomChange(Sender: TObject);
begin
  UpdateInfo;
end;
```

How it works

1. `FormCreate` writes `SampleBook.pdf` the first time, with the same drawing methods as the previous demos, then calls `OpenFile`.
2. `OpenFile` calls `Viewer.LoadFromFile`. The thumbnail view notices the new document through its `Viewer` property and draws the thumbnails; there is no code for that.
3. `UpdateInfo` reads three properties of the viewer: `PageIndex` (from 0, hence the `+ 1`), `PageCount` and `Scale` (1 means 100%).
4. The buttons call `FirstPage`, `PrevPage`, `NextPage`, `LastPage`, `ZoomOut` and `ZoomIn`. They do not update the label themselves: the viewer fires `OnPageChange` or `OnZoomChange`, and those handlers call `UpdateInfo`. The label therefore also follows a click on a thumbnail and the mouse wheel.
5. Scrolling, text selection and links are built into the viewer.

Try this

- Set `ViewMode = pvmContinuous` on the viewer. The pages now scroll one below the other, and `OnPageChange` fires as you scroll.
- Add a button that calls `Viewer.RotateRight`.
- Add a Print button that calls `Viewer.Print`. With its default parameters it prints all the pages.

4.5 Demo 05: PDF Invoice App

Folder `05.PDFInvoiceApp`, project `PDFInvoiceApp.dpr`.

What you will build

A small invoice generator that puts together everything from the previous demos. On the left you type the customer, the address, the invoice number and the invoice lines (description, quantity and price) in a grid. On the right the viewer shows the invoice: a blue header band with the company, a Bill to block, the number and the date, a table with the lines and three total rows (Subtotal, VAT 21% and TOTAL), and a footer note. **Update preview** rebuilds the invoice from the form, **Save PDF...** writes it to a file you choose.

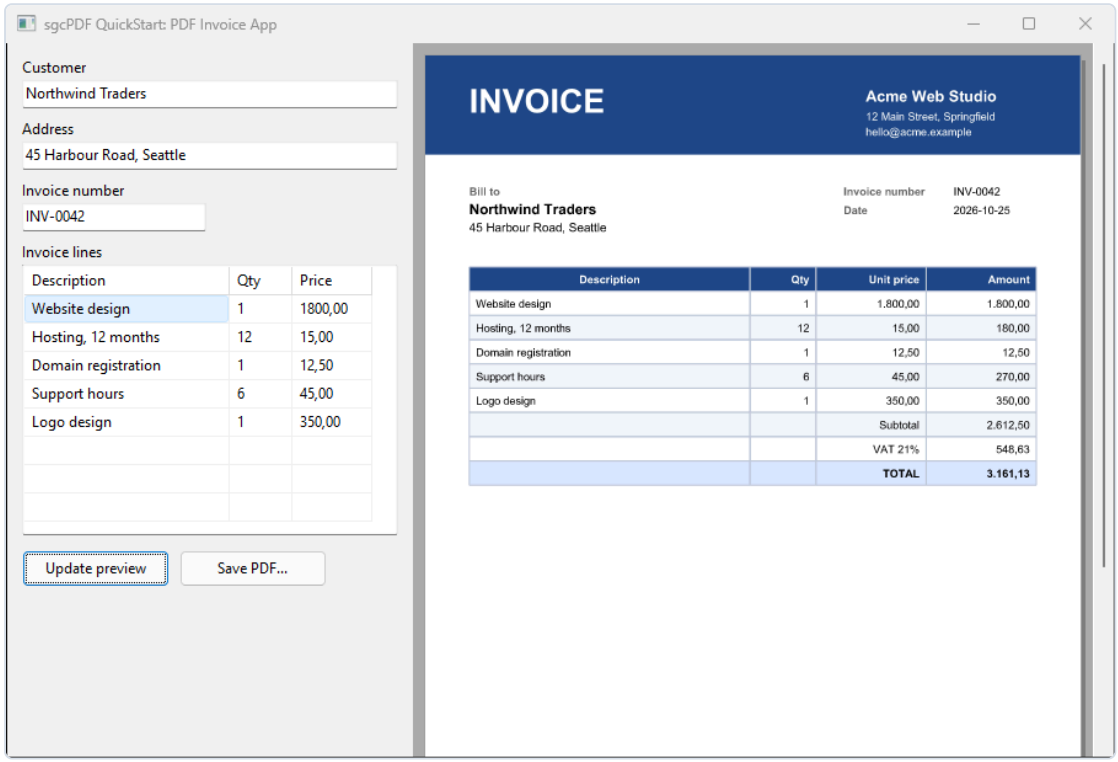


Figure 4.5. Demo 05. The amounts follow the regional settings of the machine; this screenshot uses a comma as the decimal separator.

Components on the form

Component	Class	Properties you set
`pnlInput`	`TPanel`	`Align = alLeft`. Holds the labels, edits, grid and buttons.
`edtCustomer`, `edtAddress`, `edtInvoice`	`TEdit`	`Text` with sample values.
`grdLines`	`TStringGrid`	`ColCount = 3`, `RowCount = 9`, `FixedCols = 0`, `goEditing` in `Options`.
`btnPreview`, `btnSave`	`TButton`	`OnClick` handlers.
`Viewer`	`TsgcPDFViewerControl`	`Align = alClient`. `ZoomMode` keeps its default, `zmFitWidth`.
`dlgSave`	`TSaveDialog`	`DefaultExt = 'pdf'`, `Filter` for PDF files, `ofOverwritePrompt`.

There is no `TsgcPDFDocument` on this form: every preview builds a new document in code, so it always starts empty.

The code

The header is drawn with filled rectangles and text, as in demo 02. Note the `AnsiString` casts around the values that come from the edits:

```
const
  CS_VAT = 0.21;

  // ...

procedure TfrmPDFInvoiceApp.DrawHeader(aPage: TsgcPDFPage);
begin
  aPage.DrawRectangleFill(0, 752, 595, 90, 0.13, 0.29, 0.53);
  aPage.DrawTextColor('INVOICE', 40, 790, sfHelveticaBold, 30, 1, 1, 1);
  aPage.DrawTextColor('Acme Web Studio', 400, 800, sfHelveticaBold, 14,
    1, 1, 1);
  aPage.DrawTextColor('12 Main Street, Springfield', 400, 783, sfHelvetica,
    10, 1, 1, 1);
  aPage.DrawTextColor('hello@acme.example', 400, 769, sfHelvetica, 10,
    1, 1, 1);

  aPage.DrawTextColor('Bill to', 40, 715, sfHelveticaBold, 10, 0.4, 0.4, 0.4);
  aPage.DrawText(AnsiString(edtCustomer.Text), 40, 698, sfHelveticaBold, 13);
  aPage.DrawText(AnsiString(edtAddress.Text), 40, 682, sfHelvetica, 11);

  aPage.DrawTextColor('Invoice number', 380, 715, sfHelveticaBold, 10,
    0.4, 0.4, 0.4);
  aPage.DrawText(AnsiString(edtInvoice.Text), 480, 715, sfHelvetica, 10);
  aPage.DrawTextColor('Date', 380, 698, sfHelveticaBold, 10, 0.4, 0.4, 0.4);
  aPage.DrawText(AnsiString(FormatDateTime('yyyy-mm-dd', Date)), 480, 698,
    sfHelvetica, 10);
end;
```

The lines are a `TsgcPDFTable`, as in demo 03, drawn with `DrawOnPage` because an invoice fits on one page. The totals are three more rows of the same table. The small procedure `StyleInvoiceRow` sets the `Style` of the cells of a row: it right aligns the Qty, price and amount columns, and for the last row (TOTAL) it also uses a bold font and a light blue background:

```
{ A cell Style overrides the table style for that cell only. This helper right
  aligns the number columns (1 to 3) of a row and, for the TOTAL row, also makes
  the text bold on a light background. }
procedure StyleInvoiceRow(aRow: TsgcPDFTableRow; aTotal: Boolean);
var
  i: Integer;
  vStyle: TsgcPDFTableCellStyle;
begin
  for i := 0 to aRow.CellCount - 1 do
  begin
    vStyle := aRow.GetCell(i).Style;
    if i > 0 then
      vStyle.Align := caRight;
    if aTotal then
    begin
      vStyle.Font := sfHelveticaBold;
      vStyle.HasBackground := True;
      vStyle.BackR := 0.85;
```

```
      vStyle.BackG := 0.9;
      vStyle.BackB := 1.0;
    end;
    aRow.GetCell(i).Style := vStyle;
  end;
end;

procedure TfrmPDFInvoiceApp.DrawLines(aPage: TsgcPDFPage);
var
  oTable: TsgcPDFTable;
  i, vQuantity: Integer;
  vPrice, vSubtotal: Double;
begin
  vSubtotal := 0;
  oTable := TsgcPDFTable.Create(4);
  try
    oTable.SetColumnWidths([255, 60, 100, 100]);
    oTable.SetHeaders(['Description', 'Qty', 'Unit price', 'Amount']);

    { One table row per grid row with a description. A value that is not a
      number counts as 0. }
    for i := 1 to grdLines.RowCount - 1 do
    begin
      if Trim(grdLines.Cells[0, i]) = '' then
        Continue;
      vQuantity := StrToIntDef(grdLines.Cells[1, i], 0);
      vPrice := StrToFloatDef(grdLines.Cells[2, i], 0);
      vSubtotal := vSubtotal + vQuantity * vPrice;
      oTable.AddRowData([AnsiString(grdLines.Cells[0, i]),
        AnsiString(IntToStr(vQuantity)),
        AnsiString(FormatFloat('#,##0.00', vPrice)),
        AnsiString(FormatFloat('#,##0.00', vQuantity * vPrice))]);
    end;

    { The totals are three more rows of the same table. }
    oTable.AddRowData(['', '', 'Subtotal',
      AnsiString(FormatFloat('#,##0.00', vSubtotal))]);
    oTable.AddRowData(['', '', AnsiString(Format('VAT %d%%',
      [Round(CS_VAT * 100)])), AnsiString(FormatFloat('#,##0.00',
      vSubtotal * CS_VAT))]);
    oTable.AddRowData(['', '', 'TOTAL',
      AnsiString(FormatFloat('#,##0.00', vSubtotal * (1 + CS_VAT)))]);

    { Right align the numbers of every row and highlight the last row. }
    for i := 0 to oTable.RowCount - 1 do
      StyleInvoiceRow(oTable.GetRow(i), i = oTable.RowCount - 1);

    oTable.SetDefaultFont(sfHelvetica, 10);
    oTable.SetHeaderFont(sfHelveticaBold, 10);
    oTable.SetHeaderBackground(0.13, 0.29, 0.53);
    oTable.SetAlternateRowColors(0.95, 0.96, 0.98);
    oTable.SetBorder(0.5, 0.75, 0.78, 0.85);
    oTable.SetPadding(6, 6, 5, 5);

    { An invoice fits on one page, so DrawOnPage is enough: the top left
      corner of the table is at (40, 650). }
    oTable.DrawOnPage(aPage, 40, 650);
  finally
    oTable.Free;
  end;
end;
```

`BuildInvoice` creates the document and calls the three drawing procedures. The caller owns the result:

```
procedure TfrmPDFInvoiceApp.DrawFooter(aPage: TsgcPDFPage);
begin
  aPage.DrawLineColor(40, 70, 555, 70, 0.5, 0.75, 0.78, 0.85);
  aPage.DrawText('Thank you for your business. Payment due within 30 days.',
    40, 52, sfHelveticaOblique, 10);
end;

function TfrmPDFInvoiceApp.BuildInvoice: TsgcPDFDocument;
var
  vPage: TsgcPDFPage;
begin
  Result := TsgcPDFDocument.Create(nil);
  try
    Result.Info.Title := AnsiString('Invoice ' + edtInvoice.Text);
    Result.Info.Author := 'Acme Web Studio';
    vPage := Result.AddPage(595, 842);
    DrawHeader(vPage);
    DrawLines(vPage);
    DrawFooter(vPage);
  except
    Result.Free;
    raise;
  end;
end;
```

The preview and the Save button use the same `BuildInvoice`, once to a stream for the viewer and once to a file:

```
procedure TfrmPDFInvoiceApp.ShowPreview;
var
  oDocument: TsgcPDFDocument;
  vStream: TMemoryStream;
begin
  oDocument := BuildInvoice;
  try
    vStream := TMemoryStream.Create;
    try
      oDocument.SaveToStream(vStream);
      vStream.Position := 0;
      Viewer.LoadFromStream(vStream);
    finally
      vStream.Free;
    end;
  finally
    oDocument.Free;
  end;
end;

procedure TfrmPDFInvoiceApp.FormCreate(Sender: TObject);
begin
  SeedLines;
  ShowPreview;
end;

procedure TfrmPDFInvoiceApp.btnPreviewClick(Sender: TObject);
begin
  ShowPreview;
end;
```

```
procedure TfrmPDFInvoiceApp.btnSaveClick(Sender: TObject);
var
  oDocument: TsgcPDFDocument;
begin
  dlgSave.FileName := 'Invoice ' + edtInvoice.Text + '.pdf';
  if not dlgSave.Execute then
    Exit;
  oDocument := BuildInvoice;
  try
    oDocument.SaveToFile(dlgSave.FileName);
  finally
    oDocument.Free;
  end;
  ShowPreview;
end;
```

How it works

1. `FormCreate` fills the grid with five sample lines (`SeedLines`) and shows the first preview.
2. `BuildInvoice` creates a new `TsgcPDFDocument` with one A4 page and draws the header, the lines and the footer on it. If anything raises an exception, the half built document is freed and the exception goes on.
3. `DrawLines` skips grid rows without a description, converts the quantity and the price with `StrToIntDef` and `StrToFloatDef` (a value that is not a number counts as 0), and adds up the subtotal while it adds the rows.
4. The three total rows use the same table. After the last row is added, a loop calls `StyleInvoiceRow` for every row, so the numbers are right aligned and the TOTAL row is bold on a light background. The VAT rate is the constant `CS_VAT`, and the label “VAT 21%” is built from it.
5. `ShowPreview` saves the document to a memory stream, loads the stream into the viewer and frees the document. The viewer keeps its own parsed copy.
6. `btnSaveClick` proposes a file name from the invoice number, builds the invoice again and saves it with `SaveToFile` . Then it refreshes the preview so that screen and file match.

Try this

- Change `CS_VAT` to 0.10. Both the label and the amounts follow.
- Add your logo to the header. In `BuildInvoice` , add the image to the document with `Result.AddImage('logo.png')` , which returns its index, and draw it inside the blue band with `vPage.DrawImageFromIndex` .
- Assign `grdLines.OnSetEditText` to a handler that calls `ShowPreview` , for a preview that follows every key stroke.

5. Main components

The sgcPDF palette page is long, but a first application needs only two of its components: the document and the viewer control. This chapter lists everything on the page, then introduces the components in four groups: creating documents, showing them, working with existing files, and security.

Every class has its own help topic, named after the class, such as `TsgcPDFDocument.htm` or `TsgcPDFViewerControl.htm`, with its properties, methods and events.

Reference table

Class	Unit	What it is for	Key members
`TsgcPDFDocument`	`sgcPDF_Document`	The document: pages, drawing, load and save	`AddPage`, `Pages`, `PageCount`, `Info`, `AddTrueTypeFont`, `AddImage`, `SaveToFile`, `SaveToStream`, `LoadFromFile`
`TsgcPDFPage`	`sgcPDF_Document`	One page (not on the palette)	`DrawText`, `DrawTextTT`, `DrawLine`, `DrawRectangleFill`, `DrawImageFromIndex`, `Width`, `Height`
`TsgcPDFTable`	`sgcPDF_Tables`	Tables with page breaks (not on the palette)	`SetColumnWidths`, `SetHeaders`, `AddRowData`, `DrawOnPage`, `DrawMultiPage`
`TsgcPDFViewerControl`	`sgcPDF_ViewerVCL`	The VCL viewer control	`LoadFromFile`, `LoadFromStream`, `ZoomMode`, `ViewMode`, `PageIndex`, `PageCount`, `Print`, `OnPageChange`
`TsgcPDFThumbnailView`	`sgcPDF_ViewerVCL`	Page thumbnails for a viewer control	`Viewer`
`TsgcPDFOutlineView`	`sgcPDF_ViewerVCL`	Bookmarks tree for a viewer control	`Viewer`
`TsgcPDFCommentsView`	`sgcPDF_ViewerVCL`	Comments list for a viewer control	`Viewer`
`TsgcPDFViewer`	`sgcPDF_Viewer`	A native viewer window	See its help topic
`TsgcPDFExporter`	`sgcPDF_Export`	Pages to BMP, PNG or JPEG files and thumbnails	See its help topic
`TsgcPDFEditor`	`sgcPDF_Editor`	Watermarks, page numbers, merge and forms on existing files	See its help topic
`TsgcPDFTextExtractor`	`sgcPDF_TextExtract`	Text with positions and fonts	See its help topic
`TsgcPDFImageExtractor`	`sgcPDF_ImageExtract`	Lists and saves the images of a PDF	See its help topic
`TsgcPDFOptimizer`	`sgcPDF_Optimize`	Smaller files	See its help topic

Class	Unit	What it is for	Key members
`TsgcPDFRedactor`	`sgcPDF_Redaction`	Removes content from areas of the pages	See its help topic
`TsgcPDFTextOutliner`	`sgcPDF_TextOutlines`	Text converted to vector outlines	See its help topic
`TsgcPDFHTMLRenderer`	`sgcPDF_HTML`	HTML and CSS to PDF pages	See its help topic
`TsgcPDFSignatureHandler`	`sgcPDF_Signatures`	Signs documents	See its help topic
`TsgcPDFSignatureVerifier`	`sgcPDF_Signatures`	Verifies signatures	See its help topic
`TsgcPDFBarcodeControl`	`sgcPDF_Barcode_VCL`	Barcode and QR Code control	See its help topic
`TsgcPDFViewerFMX`	`sgcPDF_ViewerFMX`	The FireMonkey viewer control	See its help topic

Creating documents

TsgcPDFDocument

The component you meet in every demo. It holds the document in memory and is both the writer and the reader: `AddPage` and the drawing methods of the pages for new documents, `LoadFromFile` and `LoadFromStream` for existing ones, `SaveToFile` and `SaveToStream` for both. Fonts and images are added to the document and referenced by index from the pages. The same component also sets the encryption and the passwords of a file (`SetEncryption` , `SetPassword`) and the PDF/A level (`SetPDFALevel`).

TsgcPDFTable

A helper class for rows and columns. It wraps the text inside the cells, draws the borders and the backgrounds, and with `DrawMultiPage` adds pages and repeats the header row. Demos 03 and 05 show the whole cycle.

TsgcPDFHTMLRenderer and the barcodes

Two other ways to fill pages. `TsgcPDFHTMLRenderer` turns HTML with CSS into PDF pages, which is handy when the content already exists as HTML. `TsgcPDFBarcodeControl` shows a barcode or a QR Code on a form and can add it to a PDF; the help topic on barcodes also describes drawing barcodes straight onto a page.

Showing documents

TsgcPDFViewerControl

The VCL viewer control used by every demo. It loads a file or a stream, shows the pages in one of three view modes with a zoom mode of your choice, and has methods for navigation, zoom, rotation, search (`FindText`), selection (`SelectAll` , `CopyToClipboard`) and printing (`Print`). `OnPageChange` and `OnZoomChange` tell your toolbar when to refresh. Beyond the QuickStart it can fill in forms and work with annotations; the `PDFViewerControl` demo shows those features.

TsgcPDFThumbnailView, TsgcPDFOutlineView and TsgcPDFCommentsView

Three companions for the viewer control. Set their `Viewer` property and they show, respectively, the page thumbnails, the bookmarks and the comments of the document in the viewer, and move the viewer when the user clicks an entry. Demo 04 uses the thumbnail view.

TsgcPDFViewer and TsgcPDFExporter

`TsgcPDFViewer` is a non visual component that opens a native window to display, search and print a PDF, for when you do not want to place a control on your own form. `TsgcPDFExporter` renders pages to BMP, PNG or JPEG files, for example to make previews for a web page.

Working with existing files

These components open a PDF file, do one job on it and save the result:

- **`TsgcPDFEditor`** adds watermarks, headers, footers and page numbers, merges and splits documents and fills forms.
- **`TsgcPDFTextExtractor`** returns the text of a page with the position, font and size of each character, for searching and indexing.
- **`TsgcPDFImageExtractor`** lists the images of a document and saves them as PNG, JPEG or BMP.
- **`TsgcPDFOptimizer`** compresses streams, removes duplicate and unused objects and downsamples images to make a file smaller.
- **`TsgcPDFRedactor`** really removes text, image pixels and graphics from marked areas or search results, instead of only covering them.
- **`TsgcPDFTextOutliner`** converts all the text to vector outlines, so the output needs no font to be printed.

Security and signatures

`TsgcPDFSignatureHandler` signs documents with a PFX file, the Windows certificate store or an external signer, including PAdES signatures and document time stamps. `TsgcPDFSignatureVerifier` checks the signatures of a document. Password and certificate encryption are set on `TsgcPDFDocument` itself. The `PDFSignature`, `PDFSignaturePAdES`, `PDFEncryption` and `PDFCertEncryption` demos show these features.

Beyond the basics

The document model goes much further than the QuickStart. Bookmarks (`Bookmarks`), annotations (`Annotations`), interactive forms (`AcroForm`) and optional content layers (`Layers`) are properties of `TsgcPDFDocument`. There are also embedded files, PDF/A output, tagged PDF for accessibility, a `TCanvas` style drawing class (`TsgcPDFCanvas`) and incremental saving that keeps existing signatures valid. Chapter 7 lists the demo for each of these, and the help has a guide topic for each.

6. Common problems

These are the problems that come up most often in a first sgcPDF application. Each entry says what you see, why it happens and how to fix it.

Text and fonts

The compiler warns about an implicit string cast

What you see. The warning W1058 "Implicit string cast with potential data loss from 'string' to 'AnsiString'" on a line with `DrawText`, `DrawTextColor` or `AddRowData`.

Why. The standard font methods and the table cells take an `AnsiString`. In Delphi 2009 and later, `string` is a Unicode string, so passing a `string` variable, a `Format` result or `Edit.Text` converts it implicitly. String literals are converted by the compiler and do not warn.

Fix. Cast explicitly: `vPage.DrawText(AnsiString(edtCustomer.Text), 40, 698, sfHelveticaBold, 13)`. Every QuickStart demo does this, so they compile without warnings.

Accented or non Latin characters come out wrong

What you see. Names with characters outside the Western character set, or Greek, Cyrillic or Asian text, are missing or show the wrong characters in the PDF.

Why. The 14 standard fonts only cover the WinAnsi character set, and the text of `DrawText` is an `AnsiString`, so characters outside it cannot be represented.

Fix. Load a TrueType font that contains the characters with `AddTrueTypeFont` and draw with `DrawTextTT` or `DrawTextTTColor`, which take a `WideString`. Section 3.4 has a compiled example.

The text is at the bottom of the page, or not on the page at all

What you see. A text you meant for the top of the page appears near the bottom, or you cannot find it.

Why. The origin of a PDF page is the bottom left corner and Y grows upwards. A Y of 50 is near the bottom, and a Y larger than the page height is above the page and invisible.

Fix. Count from the top: for a text 60 points below the top of an A4 page, use Y = 782, which is 842 minus 60. Remember that Y is the baseline of the text, so the letters rise above it by roughly their font size.

Layout and tables

The header row of a table looks empty

What you see. The table has a header row, but no text in it.

Why. The header text is white by default, so on a white or light header background it is invisible.

Fix. Either give the header a dark background, for example `oTable.SetHeaderBackground(0.13, 0.29, 0.53)` as in demos 03 and 05, or set a dark text colour in `HeaderStyle` (`TextR`, `TextG`, `TextB`).

The table runs into the footer

What you see. On a multi page table, the last rows of a page overlap the page number or the footer line.

Why. `DrawMultiPage` fills each page down to the height you give it. In demo 03 the table starts at Y = 740 with a maximum height of 680, so it stops at Y = 60; a larger height lets it go further down.

Fix. Choose the maximum height as the start Y minus the Y of the top of your footer, and draw the footer below that line.

Colours are wrong, or everything is white

What you see. A shape or a text that should be coloured is white or the wrong colour.

Why. The drawing methods take red, green and blue as numbers from 0 to 1. Values from 0 to 255, or a `TColor`, are out of range.

Fix. Divide each component by 255: a colour of 33, 74, 135 becomes 0.13, 0.29, 0.53.

The gradient goes the wrong way

What you see. You expected a gradient from left to right, and it runs from bottom to top.

Why. `DrawLinearGradient` runs from the first colour at the bottom edge of the rectangle to the second colour at the top edge.

Fix. Plan the gradient as vertical, and swap the two colours if you want the first one at the top.

Viewing and saving

The viewer does not show what you just drew

What you see. You draw more on the document after showing it, and the viewer does not change.

Why. `LoadFromFile` and `LoadFromStream` parse the PDF into a document that the viewer owns. Your `TsgcPDFDocument` is not connected to the viewer.

Fix. Save and load again after every change, as demo 05 does in `ShowPreview`. For a document that is rebuilt often, create a new one each time, as `BuildInvoice` does, so it always starts empty.

Loading from a stream fails

What you see. `Viewer.LoadFromStream` raises an error or shows nothing right after `SaveToStream`.

Why. After `SaveToStream` the position of the stream is at its end, so there is nothing left to read.

Fix. Set `vStream.Position := 0` between the save and the load, as every demo does.

Saving next to the executable fails

What you see. Demo 01 or 04 raises an error when it starts, after you copied it to a folder such as `Program Files`.

Why. These two demos write their PDF next to the executable (`ExtractFilePath(ParamStr(0))`), and normal users cannot write in protected folders.

Fix. Run the demos from a folder where you can write, such as their own demo folder. In your own applications, save to a user folder or to the file the user chooses with a `TSaveDialog`, as demo 05 does.

An invoice line counts as 0

What you see. In demo 05 a line shows a price of 0.00 although you typed one.

Why. `StrToFloatDef` uses the decimal separator of the Windows regional settings, and a value it cannot read becomes 0. Typing `15.00` on a machine that uses a comma, or the other way round, gives 0.

Fix. Type numbers with the decimal separator of your regional settings, as the sample lines do.

7. Where next

Once the five QuickStart demos feel familiar, you know the core of sgcPDF. This chapter points you to the larger demos, to the documentation and to support.

The other Delphi demos

The other folders in `Demos\PDF\Delphi` each show one feature. Open the project, run it and read the main unit.

Demo	What it shows
`PDFCreator`	Creating documents: the natural next step after demo 05.
`PDFCanvas`	Drawing with the `TCanvas` style class and converting EMF to PDF.
`PDFImages`	Embedding images.
`PDFTransparency`	Transparency and gradients.
`PDFUnicode`	Unicode text with TrueType fonts.
`PDFBarcodes2D`	2D barcodes drawn on PDF pages.
`PDFBarcodeControls`, `PDFBarcodeControlsFMX`	One barcode control per symbology, with rotation, colours, PNG and SVG export and Add to PDF, for the VCL and for FireMonkey.
`PDFViewer`	The `TsgcPDFViewer` component.
`PDFViewerControl`	The viewer control in depth.
`PDFViewerFMX`	The FireMonkey viewer.
`PDFEditor`	Editing existing documents with `TsgcPDFEditor`.
`PDFOrganizer`	Organising the pages of documents.
`PDFImposition`	Imposition: placing pages on larger sheets.
`PDFIncremental`	Incremental saving.
`PDFBookmarks`	Bookmarks (outlines).
`PDFAnnotations`	Annotations.
`PDFLayers`	Optional content layers.
`PDFAttachments`	Embedded files.
`PDFForms`, `PDFFormData`	Interactive forms, and form data exchange with FDF and XFDF.
`PDFTextExtract`, `PDFImageExtract`	Extracting the text and the images of a document.
`PDFTextOutlines`	Converting text to vector outlines.
`PDFRedaction`	Redaction.
`PDFEncryption`, `PDFCertEncryption`	Password encryption and certificate encryption.
`PDFSignature`, `PDFSignaturePlacement`, `PDFSignaturePAdES`	Digital signatures, placing a signature on the page, and PAdES signatures.

Demo	What it shows
`PDFPDFA`	PDF/A output.
`PDFTagged`	Tagged PDF and PDF/UA.
`MigrateFromQuickPDF`	The everyday jobs of a Quick PDF Library application done with sgcPDF, one button each.

A few of these demos also use other eSeGeCe libraries, for example sgcSign for some of the signature demos. Their source code says what they need.

Help topics

The sgcPDF help is part of the sgcUI help at https://www.esegece.com/help/sgcUI, in the PDF section. These topics are the best continuation of this guide:

Topic	What it covers
Introduction	What sgcPDF does and the full feature list.
Installation, Architecture	The packages and Delphi versions, and which unit holds which class.
Quick Start	Complete procedures from compiled console programs for the most common tasks.
Pages, Text and fonts, Graphics, Images, Tables	Everything chapter 3 introduced, in depth.
TsgcPDFViewerControl, TsgcPDFThumbnailView	The viewer control and its thumbnail view.
Error handling, FAQ	What can go wrong, and frequent questions.
One topic per class	`TsgcPDFDocument`, `TsgcPDFEditor`, `TsgcPDFSignatureHandler` and the others, with their properties, methods and events.

On the website

- https://www.esegece.com/products/sgcpdf/: the product page, with the features, the licences, the trial and the downloads.

Getting help

If you are stuck, write to info@esegece.com or open a ticket in the helpdesk of your account at https://www.esegece.com/. It helps to include your Delphi version, your sgcPDF version, and the demo or a small project that shows the problem. If the problem is with a particular PDF file, attach it if you can.

Enjoy making PDFs.

---

sgcPDF and sgcUI are products of eSeGeCe. https://www.esegece.com