# PDFRasterizer.NET 3.0

## Developer Guide

Version: 1.2

Date: January 5, 2011

## Legal Note

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, email addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of TallComponents BV.

TallComponents BV may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from TallComponents BV, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property. Acrobat and PDF are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

Windows NT, 2000, XP, Vista, 7, Server 2003 and Server 2008, .NET Framework and Internet Information Server are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are the property of their respective owners.

© 2001-2011 TallComponents BV. All rights reserved.

PDFRasterizer.NET is a trademark of TallComponents BV.

# Contents

1. Introduction ..... 6
1.1. What's new ..... 6
1.2. Conventions used in this document ..... 6
1.3. Please send us feedback! ..... 6
1.4. Online Resources ..... 6
1.5. Naming Conventions ..... 7
1.6. Licensing ..... 7
1.7. Restrictions ..... 7
2. Getting started ..... 8
2.1. Development environment ..... 8
2.2. Requirements ..... 8
2.3. Development environment ..... 8
2.4. IntelliSense ..... 8
2.5. Deployment ..... 8
2.6. Code Samples ..... 8
3. Upgrade from 2.1 ..... 9
3.1. LicenseAttribute.AddLicense ..... 9
3.2. RenderSettings.CMapFolder ..... 9
3.3. ColorRenderSettings.ColorProfile ..... 9
3.4. FontSubstitutionMap.SearchPath ..... 9
3.5. DocumentInfo ..... 9
3.6. Summary ..... 9
3.7. Page.Rotate ..... 9
3.8. PageCollection ..... 9
3.9. Rectangle constructor ..... 9
3.10. ConvertToImageOptions ..... 9
4. Convert to Raster Formats ..... 10
4.1. Render a single page to a bitmap ..... 10
4.2. Render multiple pages to a single bitmap ..... 10
4.3. Set the resolution ..... 11
4.4. Select a raster output format ..... 11
4.5. Code samples ..... 11
5. Convert to TIFF ..... 12
5.1. Basic code sample ..... 12
5.2. B&W and Color ..... 12
5.3. Compression ..... 12
6. Printing ..... 13
6.1. Basic code sample ..... 13
6.2. Select the output printer ..... 13


© TallComponents BV – www.tallcomponents.com

6.3. Select the output tray ..... 14
6.4. Scale and rotate to fit ..... 14
6.5. Control font substitution ..... 15
6.6. Code samples ..... 15
7. Convert PDF to WPF/XPS ..... 16
7.1. Save PDF as XPS ..... 16
7.2. Display PDF pages as WPF graphics ..... 16
7.3. Code samples ..... 16
8. Progressive Rendering ..... 17
8.1. Basic code sample ..... 17
8.2. Updated event ..... 17
8.3. Cancel drawing ..... 18
8.4. Code samples ..... 18
9. Text and Fonts ..... 19
9.1. Font search path ..... 19
9.2. Font substitution map ..... 19
9.3. ResolveFont event ..... 20
9.4. CMaps ..... 22
9.5. Font styles ..... 22
9.6. Code samples ..... 22
10. Color Profiles ..... 23
10.1. Color transformation ..... 23
10.2. PDF color spaces ..... 23
11. Layers ..... 25
11.1. Basic code sample ..... 25
11.2. Clone Layers ..... 26
11.3. Default behavior ..... 26
12. Diagnostics ..... 27
12.1. Basic code sample ..... 27


© TallComponents BV – www.tallcomponents.com

# List of Code Samples

Code sample 5-1 Convert a PDF document to a multipage TIFF document ..... 12
Code sample 7-1 Save a PDF document as XPS ..... 16
Code sample 8-1 Render a page asynchronously and save intermediate bitmaps ..... 17
Code sample 9-1 Implement font substitution through the FontSubstitutionMap ..... 19
Code sample 9-2 Use the ResolveFont event to dump the name and location of each font ..... 20
Code sample 9-3 Implement font substitution through the ResolveFont event ..... 21
Code sample 9-4 Render well-known fonts as system fonts instead of as curves ..... 21
Code sample 10-1 Load custom output device color profile ..... 23
Code sample 11-1 Control rendering of layers ..... 25
Code sample 11-2 Clone the LayerCollection ..... 26
Code sample 12-1 Dump render diagnostics to the output console ..... 27


© TallComponents BV – www.tallcomponents.com

# 1. Introduction

PDFRasterizer.NET is a 100% .NET (verifiable) solution for printing PDF documents unattended and converting PDF documents to a raster image, WPF graphics or an XPS document.

This developer guide will help you understand the PDFRasterizer.NET 3.0 component. It walks through the main concepts and illustrates them with code samples. This guide is not intended as a type reference. A full type reference is included in the download and can be viewed online at http://www.tallcomponents.com/pdfrasterizer3/help/.

## 1.1. What's new

- Only medium or high trust required (was full trust in 2.1)
- Progressive drawing
- Cancellable drawing
- Asynchronous drawing
- Convert to (multi-page) color TIFF (was only Black&White in 2.1)
- LZW compression for TIFF output
- More advanced font substitution
- Convert to XPS
- Convert to WPF graphics
- Extract embedded files (also known as attachements)
- Layer support
- Diagnostics

## 1.2. Conventions used in this document

Throughout this document, formatting is used to discern between different types of information.

A code sample is contained in between dashed yellow lines and uses a mono-space font.

```
// Code sample
```

Whenever a line in a code sample is too long to fit on the page, the character ↵ is inserted. When copy/pasting code from this document, make sure you remove that character and any whitespace following it.

Important side notes are formatted using a light-yellow background.

**Note** These blocks contain tips and warnings that may save you time and trouble.

## 1.3. Please send us feedback!

Please help us improve this document. If you have any questions, suggestions or find anything in this document is incorrect or missing then let us know at support@tallcomponents.com or send us a support request through our website at http://www.tallcomponents.com/question.aspx.

## 1.4. Online Resources

- Our web site:
  http://www.tallcomponents.com


© TallComponents BV – www.tallcomponents.com

- PDFRasterizer.NET 3.0 product page:
  http://www.tallcomponents.com/pdfrasterizer3.aspx

## 1.5. Naming Conventions

We try to adhere as much as possible to Microsoft's "Design Guidelines for Developing Class Libraries".

## 1.6. Licensing

PDFRasterizer.NET 3.0 runs either in licensed or unlicensed mode. It runs in licensed mode if a valid license key is present. How to obtain a valid license key is described in detail in our licensing guide.

In unlicensed mode, the component is fully functional and does not expire. It does however impose a few restrictions that makes it unsuitable for use in a production environment. Raster images that are rendered from a PDF page are stamped diagonally with an evaluation notice. Extracted images have a red cross imposed. When converting to WPF, only the first page is converted normally. All other pages are rendered as a single raster image so no vector graphics are preserved.

## 1.7. Restrictions

The following restrictions appy to PDFRasterizer.NET 3.0:

- Due to GDI+ and XPS restrictions, only the most basic form of transparency is supported.
- Overprint and rendering intents are ignored.
- Shading patterns *Function-based* shadings (type 1), *Free-form Gouraud-shaded triangle meshes* (type 4), *Lattice-form Gouraud-shaded triangle meshes* (type 5), *Coons patch meshes* (type 6) and *Tensor-product patch meshes* (type 7) are not supported.
- When printing, spool jobs may be very large due to conversion of glyphs to curves. This happens when there is no matching system font installed and no font substitution if configured. See also Chapter 9.


© TallComponents BV – www.tallcomponents.com

# 2. Getting started

## 2.1. Development environment

PDFRasterizer.NET 3.0 has been compiled and tested with Microsoft Visual Studio.NET 2003, 2005 and 2008 for Microsoft .NET Frameworks 1.1, 2.0, 3.0 and 3.5. PDFRasterizer.NET is a 100% managed .NET component and consists of just a single assembly: TallComponents.PDF.Rasterizer.dll.

## 2.2. Requirements

PDFRasterizer.NET 3.0 is compiled for versions 1.1, 2.0 and 3.0 of the Microsoft .NET framework. All three builds are included in the distribution.

## 2.3. Development environment

To develop applications using PDFRasterizer.NET 3.0 any development environment for Microsoft .NET 1.1, 2.0, 3.0 or 3.5 framework will do.

## 2.4. IntelliSense

The PDFRasterizer.NET 3.0 installation includes the XML documentation file TallComponents.PDF.Rasterizer.xml which documents all public types and members of PDFRasterizer.NET 3.0. If this XML file is in the same folder as the assembly, Visual Studio provides IntelliSense documentation as you type in your code.

## 2.5. Deployment

PDFRasterizer.NET 3.0 is fully compatible with the XCopy deployment principle of .NET.

## 2.6. Code Samples

All samples included with PDFRasterizer.NET are available as Visual Studio 2003, 2005 and 2008 projects. The component will work with any development environment that targets the .NET 1.1 framework or later.


© TallComponents BV – www.tallcomponents.com

# 3. Upgrade from 2.1

PDFRasterizer.NET 3.0 introduces a few breaking changes. This means that you may need to rewrite your client code. The following paragraphs discuss all the changes.

## 3.1. LicenseAttribute.AddLicense

Method `LicenseAttribute.AddLicense()` has been removed. You should now use `LicenseCollection.Add()` instead.

## 3.2. RenderSettings.CMapFolder

Static property `RenderSettings.CMapFolder` has been moved to `TextRenderSettings.CMapFolder`.

## 3.3. ColorRenderSettings.ColorProfile

Property `ColorRenderSettings.ColorProfile` has been renamed into `ColorRenderSettings.OutputColorProfile`.

## 3.4. FontSubstitutionMap.SearchPath

Property `FontSubstitutionMap.SearchPath` has been moved and renamed into `TextRenderSettings.FontSearchPath`. Notice that the Save and Load from the FontSubsitutionMap will now persist / restore that property.

## 3.5. DocumentInfo

Properties `Document.Author`, `CreationDate`, `Creator`, `Keywords`, `Producer`, `Subject` and `Title` have been moved into the new `DocumentInfo` class.

## 3.6. Summary

`Page.Draw()`, now returns null (previously a Summary object). Overloads have been introduced that accept a Summary argument.

Summary class is now located in namespace `TallComponents.PDF.Rasterizer.Diagnostics`.

## 3.7. Page.Rotate

Property `Page.Rotate (int)` has been replaced by `Page.Orientation (enum)`.

## 3.8. PageCollection

Class `Pages` has been renamed to `PageCollection`.

## 3.9. Rectangle constructor

`Rectangle` constructor removed, this was accidentially added.

## 3.10. ConvertToImageOptions

Class `ConvertToImageOptions` has been removed. Property `ConvertToImageOptions.Resolution` has been moved to the `ConvertToTiffOptions` class.


© TallComponents BV – www.tallcomponents.com

# 4. Convert to Raster Formats

Although TIFF is also a raster format, we handle convering PDF documents to TIFF somewhat differently. This is described in Chapter 5.

## 4.1. Render a single page to a bitmap

The following code renders the first page of a given document to a single bitmap (PNG) at 72 DPI:

```
using (FileStream file = new FileStream(
   @"document.pdf", FileMode.Open, FileAccess.Read))
{
   Document document = new Document(file);
   Page page = document.Pages[0];

   using (Bitmap bitmap = new Bitmap(
      (int)page.Width, (int)page.Height))
   {
      Graphics graphics = Graphics.FromImage(bitmap);
      graphics.SmoothingMode = SmoothingMode.AntiAlias;
      page.Draw(graphics);
      bitmap.Save(@"document_0.png", ImageFormat.Png);
   }
}
```

## 4.2. Render multiple pages to a single bitmap

The following code renders all pages of a given document to a horizontal strip at 72 DPI:

```
using (FileStream file = new FileStream(
   @"document.pdf", FileMode.Open, FileAccess.Read))
{
   Document document = new Document(file);
   PageCollection pages = document.Pages;

   // calculate strip height and width
   int stripHeight=0, stripWidth=0;
   for (int i = 0; i < pages.Count; i++)
   {
      stripHeight = Math.Max(stripHeight, (int)pages[i].Height);
      stripWidth += (int)pages[i].Width;
   }

   using (Bitmap bitmap = new Bitmap(stripWidth, stripHeight))
   {
      Graphics graphics = Graphics.FromImage(bitmap);
      for (int i = 0; i < pages.Count; i++)
      {
         pages[i].Draw(graphics);
         // translate the graphics to the right so the next page
         // is drawn next to to the previous
         graphics.TranslateTransform((int)pages[i].Width, 0);
      }
      bitmap.Save(@"document_strip.png", ImageFormat.Png);
   }
}
```

Note the call to `graphics.TranslateTransform`. This call makes sure that the next page is drawn next to the previous page by translating the graphics object to the right by the amount of the page width.


© TallComponents BV – www.tallcomponents.com

## 4.3. Set the resolution

The resolution is expressed as the number of pixels per inch. The physical dimension of a PDF page is a given and returned by `page.Width` and `page.Height` in points (1/72 inch). The vertical resolution follows from the number of rows that a rendered page spans and the horizontal resolution follows from the number of columns that a rendered page spans.

If you do not explicitly transform the `System.Drawing.Graphics` object, then the resolution is 72 DPI. This means that each point corresponds to a single pixel because an inch equals 72 points.

The set the resolution to 300 DPI, you need 300/72 times more rows and 300/72 more columns. To achieve this you need to scale the `Graphics` object in both directions with the same factor.

This is demonstrated in the following code:

```
using (FileStream file = new FileStream(
   @"document.pdf", FileMode.Open, FileAccess.Read))
{
   Document document = new Document(file);
   Page page = document.Pages[0];

   // calculate the scale from the DPI
   int dpi = 300;
   float scale = (float)dpi / 72f;

   using (Bitmap bitmap = new Bitmap(
      (int)(page.Width*scale), (int)(page.Height*scale)))
   {
      Graphics graphics = Graphics.FromImage(bitmap);
      graphics.SmoothingMode = SmoothingMode.AntiAlias;
      // scale the graphics object
      graphics.ScaleTransform(scale, scale);
      page.Draw(graphics);
      bitmap.Save(@"document_0.png", ImageFormat.Png);
   }
}
```

## 4.4. Select a raster output format

The raster format is selected when you write the `System.Drawing.Bitmap` to file using the following code:

```
bitmap.Save(@"document_0.png", ImageFormat.Png);
```

You can change the output format by passing another `ImageFormat`. E.g. the following line saves the bitmap to the lossy JPEG format:

```
bitmap.Save(@"document_0.png", ImageFormat.Jpeg);
```

## 4.5. Code samples

The following code samples are included in the distribution and are relevant to converting PDF to ratser formats:

- ConvertToImage
- ViewPDF


© TallComponents BV – www.tallcomponents.com

# 5. Convert to TIFF

Coversion of PDF to (multipage) TIFF is a typical usage of PDFRasterizer.NET. Because GDI+ is somewhat restricted we offer a dedicated conversion to single and multi-page TIFF.

## 5.1. Basic code sample

The following basic code sample converts a PDF document to a multi-page TIFF document.

```
using (FileStream pdfFile = new FileStream(
   @"document.pdf", FileMode.Open, FileAccess.Read))
using (FileStream tiffFile = new FileStream(
   @"document.tiff", FileMode.Create, FileAccess.Write))
{
   ConvertToTiffOptions options = new ConvertToTiffOptions();
   options.Compression = TiffCompression.CcittG3;
   options.Resolution = 300;

   Document document = new Document(pdfFile);
   document.ConvertToTiff(tiffFile, options);
}
```

Code sample 5-1 Convert a PDF document to a multipage TIFF document

A similar `ConvertToTiff` method exists on the Page class. This results in a single-page TIFF document.

## 5.2. B&W and Color

You can convert either to B&W or to color by setting the `ConvertToTiffOptions.PixelFormat` property. Set this property to `Bw1Bpp` to render to a B&W TIFF. Set this property to `Rgba32Bpp` to render to a color TIFF.

## 5.3. Compression

Per pixel format, a number of compressions are available. You select the compression by settings the `ConvertToTiffOptions.Compression` property. The following table shows the possible combinations:

| PixelFormat | Compression |
|---|---|
| `Bw1Bpp` | `None` |
| | `CcittG3` |
| | `CcittG4` |
| | `PackBits` |
| | `Lzw` |
| `Rgba32Bpp` | `None` |
| | `Lzw` |


© TallComponents BV – www.tallcomponents.com

# 6. Printing

Printing a PDF document comes down to rendering the PDF pages to a `System.Drawing.Graphics` object that represents a printer. So, basically the process is similar to rendering a PDF document to a raster image as described in Chapter 3. Nonetheless there are some points specif to printing PDF documents. These are discussed in this chapter.

## 6.1. Basic code sample

Here is the basic code sample of a console application that prints a PDF document to the default printer using default settings.

```
static int pageIndex = 0;
static Document document;

static void Main(string[] args)
{
   using (FileStream file = new FileStream(
      @"document.pdf", FileMode.Open, FileAccess.Read))
   {
      document = new Document(file);

      PrintDocument printDocument = new PrintDocument();
      printDocument.DocumentName = document.DocumentInfo.Title;

      printDocument.PrintPage += new PrintPageEventHandler(printPage);
      printDocument.Print();
   }
}

static void printPage(object sender, PrintPageEventArgs e)
{
   e.Graphics.PageUnit = GraphicsUnit.Point;
   Page page = document.Pages[pageIndex++];
   page.Draw(e.Graphics);
   e.HasMorePages = pageIndex < document.Pages.Count;
}
```

## 6.2. Select the output printer

You can only print to printers that have a printer driver installed. The installed printers can be retrieved and selected using the following code:


© TallComponents BV – www.tallcomponents.com

```
PrinterSettings.StringCollection printerNames =
   PrinterSettings.InstalledPrinters;
// select the first printer
string printerName = printerNames[0];

using (FileStream file = new FileStream(
   @"document.pdf", FileMode.Open, FileAccess.Read))
{
   document = new Document(file);

   PrintDocument printDocument = new PrintDocument();
   printDocument.PrinterSettings.PrinterName = printerName;

   printDocument.DocumentName = document.DocumentInfo.Title;

   printDocument.PrintPage += new PrintPageEventHandler(printPage);
   printDocument.Print();
}
```

## 6.3. Select the output tray

```
static void Main(string[] args)
{
   using (FileStream file = new FileStream(
      @"document.pdf", FileMode.Open, FileAccess.Read))
   {
      document = new Document(file);

      PrintDocument printDocument = new PrintDocument();

      PrinterSettings.PaperSourceCollection paperSources =
         printDocument.PrinterSettings.PaperSources;
      printDocument.PrinterSettings.DefaultPageSettings.PaperSource =
         paperSources[0];

      printDocument.DocumentName = document.DocumentInfo.Title;

      printDocument.PrintPage += new PrintPageEventHandler(printPage);
      printDocument.Print();
   }
}

static void printPage(object sender, PrintPageEventArgs e)
{
   e.Graphics.PageUnit = GraphicsUnit.Point;
   Page page = document.Pages[pageIndex++];
   page.Draw(e.Graphics);
   e.HasMorePages = pageIndex < document.Pages.Count;
}
```

## 6.4. Scale and rotate to fit

A typical operation is to scale and rotate the source page to fit the output paper. This is done by applying a geometric transformation to the `System.Drawing.Graphics` object before rendering the PDF page. The following PrintPahge event handler shows how to do this:


© TallComponents BV – www.tallcomponents.com

```
void printPage( object sender, PrintPageEventArgs e )
{
   e.Graphics.PageUnit = GraphicsUnit.Point;

   Page page = document.Pages[ pageIndex++ ];

   autoPosition(
      e.Graphics,
      ( e.PageSettings.PaperSize.Width * 72 ) / 100,
      ( e.PageSettings.PaperSize.Height * 72 ) / 100,
      // As of .NET 2.0, you can use the printable area
      // instead of the papersize.
      // ( e.PageSettings.PrintableArea.Width * 72 ) / 100,
      // ( e.PageSettings.PrintableArea.Height * 72 ) / 100,
      page.Width, page.Height );

   page.Draw( e.Graphics )

   e.HasMorePages = pageIndex < document.Pages.Count;
}

void autoPosition( Graphics graphics, double printerWidth,
   double printerHeight, double pdfWidth, double pdfHeight )
{
   double scaleX = printerWidth / pdfWidth;
   double scaleY = printerHeight / pdfHeight;
   double scale = Math.Min( scaleX, scaleY );

   graphics.ScaleTransform( (float) scale, (float) scale );
   graphics.TranslateTransform(
      (float) ( printerWidth - scale * pdfWidth ) / 2,
      (float) ( printerHeight - scale * pdfHeight ) / 2 );
}
```

## 6.5. Control font substitution

Font substitution is discussed in detail in Chapter 9.

## 6.6. Code samples

The following code samples are included in the distribution and are relevant to printing PDF documents:

- PrintPDF
- ScaledPrinting


© TallComponents BV – www.tallcomponents.com

# 7. Convert PDF to WPF/XPS

**Note** This chapter only applies to the .NET 3.0 build.

In addtion to rendering PDF pages to raster images such as BMP, PNG and JPG, PDFRasterizer,NET 3.0 also lets you convert PDF documents to WPF graphics or XPS documents. All vector graphics are preserved. This is new as of version 3.0.

## 7.1. Save PDF as XPS

Central to the conversion to WPF/XPS are the `Document.ConvertToWpf` and `Page.ConvertToWpf` methods. These methods create WPF `FixedDocument` and WPF `FixedPage` instances, respectively.

The following basic code sample demonstrates how to create a `FixedDocument` from a PDF document:

```
[STAThread]
static void Main(string[] args)
{
   using (FileStream pdfFile = new FileStream(
      @"document.pdf", FileMode.Open, FileAccess.Read))
   {
      RenderSettings renderSettings = new RenderSettings();
      ConvertToWpfOptions options = new ConvertToWpfOptions();

      Document document = new Document(pdfFile);
      FixedDocument fixedDocument = document.ConvertToWpf(
         renderSettings, options);

      XpsDocument xpsDocument= new XpsDocument(
         @"document.xps", FileAccess.ReadWrite);
      XpsDocumentWriter xpsWriter =
         XpsDocument.CreateXpsDocumentWriter(xpsDocument);
      xpsWriter.Write(fixedDocument);
   }
}
```

Code sample 7-1 Save a PDF document as XPS

Note that the calling threat must live in an STA. Unfortunately COM still haunts us today.

## 7.2. Display PDF pages as WPF graphics

This is demonstrated by code sample ViewPDF_WPF which is included in the product download.

## 7.3. Code samples

The following code samples are included in the distribution and are relevant to converting PDF documents to WPF/XPS:

- ConvertToXPS
- ViewPDF_WPF


© TallComponents BV – www.tallcomponents.com

# 8. Progressive Rendering

As of version 3.0 it is possible to render PDF documents aynchronously and retrieve intermediate bitmaps. This allows you to render a bitmap progressively. The central class is `DrawJob`.

## 8.1. Basic code sample

The following code sample starts an asynchronous draw job and renders intermediate results.

```
static void Main(string[] args)
{
   using (FileStream file = new FileStream(
      @"docment.pdf", FileMode.Open, FileAccess.Read))
   {
      Document document = new Document(file);
      Page page = document.Pages[10];

      using (Bitmap bitmap = new Bitmap((int)page.Width, (int)page.Height))
      {
         DrawJob drawJob = new DrawJob(page);
         drawJob.Updated += new EventHandler(drawJob_Updated);
         drawJob.Start(bitmap, new RenderSettings(),
            PageBoundary.Default, new Matrix());

         // this call blocks until the job has finished
         drawJob.Wait();

         bitmap.Save(@"bitmap_ready.png", ImageFormat.Png);
      }
   }
}

static void drawJob_Updated(object sender, EventArgs e)
{
   DrawJob drawJob = sender as DrawJob;
   using (Bitmap copy = drawJob.GetBitmap())
   {
      string path = string.Format(@"bitmap_{0}.png",
         DateTime.Now.Ticks);
      copy.Save(path, ImageFormat.Png);
   }
}
```

Code sample 8-1 Render a page asynchronously and save intermediate bitmaps

## 8.2. Updated event

As shown in the previous code sample, it is possible to subscribe to the `Updated` event.

This event is raised regularly while the job is running and immediately after drawing has finished. This event is provided as a convenience so that the generated bitmap can be inspected regularly, without needing to implement this functionality oneself. As a rule of thumb, this event fires at a higher frequency for simple drawing jobs, so that results can be shown quickly, and at a lower frequency for complex jobs, so that the drawing job itself will not be slowed down too much by frequent updates.

The sender argument of the event handler can be cast to a `DrawJob` object. Inside this handler the `JobStatus` property indicates the status of the drawing job.


© TallComponents BV – www.tallcomponents.com

## 8.3. Cancel drawing

You may cancel a drawing job while in progress by calling the `DrawJob.Stop` method. You typically call this method from an event handler of a UI control such as a Cancel button. The job may not stop immediately, but at a moment that permits it.

## 8.4. Code samples

The following code samples are included in the distribution and are relevant to rendering PDF documents asynchronously:

- ViewPDF_Progressive


© TallComponents BV – www.tallcomponents.com

# 9. Text and Fonts

The appearance of a fragment of text in a PDF document is defined by its font. The PDF format allows embedding the entire or partial font program so that the appearance can always be reproduced on any machine independent of what fonts are installed on the client machine. It is however also possible to omit the font program. In this case, it is up to the PDF consumer to decide how to render the text. In general this is done by selecting its own font that matches the parameters (most importantly the name) of the referenced font. This process is called font substitution. PDFRasterizer.NET 3.1 has its own default substitution scheme but it also allows you to hook into this process.

Central to controlling the rendering of text is the `TextRenderSettings` class which can be accessed through the `RenderSettings.TextSettings` property. The `RenderSettings` class has been discussed in Chapter **Error! Reference source not found.**.

## 9.1. Font search path

The TextRenderSettings has a property `FontSearchPath` which is a semi-colon separated list of folders to search for font files that are included in the substitution map and in the ResolveFont event. The folders are searched from left to right until the font file is found. The default value is `".;%FONTDIR%"`. This means that we will first look in the current folder and then in the Windows fonts folder.

## 9.2. Font substitution map

One way to control font substitution for non-embedded fonts is to add entries to the `FontSubstitutionMap` property of the `TextRenderSettings` class. This is demonstrated in the following code:

```
using (FileStream file = new FileStream(
   @"FontDemo.pdf", FileMode.Open, FileAccess.Read))
{
   Document document = new Document(file);
   Page page = document.Pages[0];

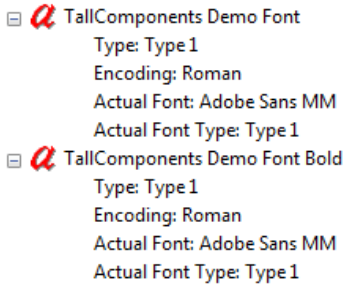
   using (Bitmap bitmap = new Bitmap((int)page.Width, (int)page.Height))
   {
      RenderSettings renderSettings = new RenderSettings();
      renderSettings.TextSettings.FontSubstitutionMap.Add(
         "TallComponents Demo Font", "ariali.ttf" );
      renderSettings.TextSettings.FontSubstitutionMap.Add(
         "TallComponents Demo Font Bold", "arialbi.ttf" );

      Graphics graphics = Graphics.FromImage(bitmap);
      page.Draw(graphics, renderSettings);
      bitmap.Save(@"FontDemo.png", ImageFormat.Png);
   }
}
```

Code sample 9-1 Implement font substitution through the FontSubstitutionMap

If you open the PDF document using the Adobe PDF Reader and hit CTRL+D and then go to the Fonts tab, you will see the following fonts listed:


© TallComponents BV – www.tallcomponents.com




The first argument to the `FontSubstitutionMap.Add` method call is the name of the font in the PDF document. The second argument is the file name of the font program that should be used to render the font.

### 9.2.1. Default substitution font

If a font is not embedded and no entry is found in the map, then the property `FontSubstitutionMap. DefaultSubstitutionFont` indicates what font to use. The value of this property is again the file name of a font program that is found with the help of the SearchPath property. The default value is `"times.ttf"`.

## 9.3. ResolveFont event

An alternative to adding entries to the `FontSubstitutionMap` is to handle the `ResolveFont` event. In contrast to the font substitution map, which only plays a role for non-embedded fonts, this event will be raised for all fonts.

The first time that a particular PDF font is encountered, the system will try to resolve the font, and then raise this event to inform the client which device font it has found. The `Location` property of the `ResolveFontEventArgs` object specifies whether a font definition has been found, and if so where it has been found.

The following code sample does not change anything but dumps the name of each font and the value of the Location property:

```
static void Main(string[] args)
{
   using (FileStream file = new FileStream(
      @"FontDemo.pdf", FileMode.Open, FileAccess.Read))
   {
      Document document = new Document(file);
      Page page = document.Pages[0];

      using (Bitmap bitmap = new Bitmap((int)page.Width, (int)page.Height))
      {
         RenderSettings renderSettings = new RenderSettings();
         renderSettings.TextSettings.ResolveFont +=
            new ResolveFontEventHandler(resolveFont);

         Graphics graphics = Graphics.FromImage(bitmap);
         page.Draw(graphics, renderSettings);
      }
   }
}

static void resolveFont(object sender, ResolveFontEventArgs args)
{
   Console.WriteLine("{0} ({1})", args.PdfFontName, args.Location);
}
```

Code sample 9-2 Use the ResolveFont event to dump the name and location of each font


© TallComponents BV – www.tallcomponents.com

If the event handler does not change any properties of this event, the system will use the definition as indicated by the `Location` propery. If the font has been resolved this is often the right thing to do. If the font is unresolved the system will use the default font as specified in the `FontSubstitutionMap`.

By changing one or more event properties, the event handler can change this behavior. Not only is it possible to map unresolved fonts to different fonts than the default, but it is also possible to map resolved fonts to different font definitions.

The following code does the exact same thing as Code sample 9-1 that uses the `FontSubstitutionMap` to map fonts. For brevity, only the event handler is listed.

```
static void resolveFont(object sender, ResolveFontEventArgs args)
{
   if (args.Location == FontLocation.Unresolved)
   {
      switch (args.PdfFontName)
      {
         case "TallComponents Demo Font":
            args.FontPath = "ariali.ttf";
            break;
         case "TallComponents Demo Font Bold":
            args.FontPath = "arialbi.ttf";
            break;
      }
   }
}
```

Code sample 9-3 Implement font substitution through the ResolveFont event

Changing the font handling for fonts that have been resolved may make sense in some cases. Fonts that have their `FontRenderMode` set to `RenderAsCurves` increase the size of print jobs and the size of WPF output. Therefore, one may choose to have these fonts rendered by an appropriate system font. This is merely a matter of setting the `SystemFontName` of the event to the appropriate font name. This is demonstrated in the following code sample:

```
void resolveFont(object sender, ResolveFontEventArgs args)
{
   if (args.FontRenderMode == FontRenderMode.RenderAsCurves)
   {
      switch (args.PdfFontName)
      {
         case "Times-Roman":
            args.SystemFontName = "Times New Roman";
            break;
         case "Times-Bold":
            args.SystemFontName = "Times New Roman";
            args.Bold = true;
            break;
         case "Times-Italic":
            args.SystemFontName = "Times New Roman";
            args.Italic = true;
            break;
      }
   }
}
```

Code sample 9-4 Render well-known fonts as system fonts instead of as curves

If the system is able to resolve the font, based on the information provided by the event handler, it will simply use that font. If for some reason the font cannot be resolved, or if it cannot be used as requested, a new `ResolveFont` event will be raised.


© TallComponents BV – www.tallcomponents.com

## 9.4. CMaps

CMaps are used when displaying some fonts, mostly Chinese, Japanese or Korean (also known as CJK fonts). The CMaps are part of the zip achive that can be downloaded from our website. You should copy the folder \Support\CMaps to your deployment environment and tell PDFRasterizer.NET 3.0 where they can be found. You can do this by assigning the path of folder CMaps to the static property `TextRenderSettings.CMapFolder` or by handling the event `TextRenderSettings.ResolveCMap`. The event allows you to provide each individual CMap as a `System.IO.Stream` so that you may pull the CMap from e.g. a database or load a resource from an assembly.

## 9.5. Font styles

Windows has the concept of font styles. So a font may be regular, bold, italic or bold-italic. The PDF format does not share this concept. A PDF document may ofcourse use bold or italic fonts but if the font description inside the PDF document does not explicitly contain this information other than that the name bold or italic is part of the PDF font name by convention.

## 9.6. Code samples

The following code samples asre included in the distribution and are relevant to font handling:

- FontMapping
- UseOSFont


© TallComponents BV – www.tallcomponents.com

# 10. Color Profiles

PDFRasterizer.NET 3.0 allows you to specify an output color profile that defines how colors are transformed to RGB. Currently, RGB is the only supported output color space.

## 10.1. Color transformation

Conceptually, colors are rendered to RGB in two steps. First the color is transformed to an absolute color, next the absolute color is transformed to the output RGB color. This process is controlled by properties `ColorRenderSettings.TransformationMode` and `ColorRenderSettings.OutputColorProfile`.

Property `ColorRenderSettings.OutputColorProfile` has a default color profile assigned. It can be overridden with your own profile using the following code:

```
using (FileStream file = new FileStream("sRGB.icm", FileMode.Open,
FileAccess.Read))
{
   ColorProfile profile = new ColorProfile(file);
   RenderSettings renderSettings = new RenderSettings();
   renderSettings.ColorSettings.ColorProfile = profile;
}
```

Code sample 10-1 Load custom output device color profile

## 10.2. PDF color spaces

In PDF, each color is defined by a combination of the current *color space* and the current *color operands*. The number of operands depends on the current color space. E.g. if DeviceRGB is the current color space, then the color is defined by 3 operands that each specifiy the intensity of the R, G and B color channels, respectively.

There are three groups of color spaces: Device color spaces, CIE-based color spaces and Special color spaces.

### 10.2.1. CIE-based color spaces

The CIE-based color spaces are ICCBased, Lab, CalGray and CalRGB. A color specified in one of these color spaces is already an *absolute color*. The color space includes parameters that fully define how to transform the color operands in the CIE-based color space to the absolute color.

If the TransformationMode is set to HighQuality, the absolute color is transformed to the output RGB color as defined by the output color profile. Otherwise, an internal fast algorithm is used.

### 10.2.2. Device color spaces

The Device color spaces are DeviceRGB, DeviceGray and DeviceCMYK. These color spaces are named device color spaces because they correspond to the color space of a typical output device.

Color operands in the DeviceRGB color space are simply copied to the output RGB color operands so no transformation is applied. Propeties `OutputColorProfile` and `TransformationMode` are ignored.

The single operand (gray value) in the DeviceGray color space is copied to all three RGB color operands. Propeties `OutputColorProfile` and `TransformationMode` are ignored.

If the TransformationMode is set to HighQuality, then first the CMYK color is transformed to an absolute color using an internal CMKY color profile. Next the absolute color is transformed to the output RGB color as defined by the output color profile. If the TransformationMode is set to HighSpeed, then an internal fast algorithm is used.


© TallComponents BV – www.tallcomponents.com

### 10.2.3. Special color spaces

The Special color spaces are Indexed, Pattern, Separation and DeviceN. These color spaces have an underlying color space that is either a Device color space or a CIE-based color space. The transformation from the color operands in the Special color space to the equivalent operands in the underlying color space are fully defined by the parameters that define the Special color space. E.g. If the color space is Indexed, a single operand selects a set of color operands from a palette. These operands specify a color in the underlying color space. Next, the transformation of the underlying color space is applied.


© TallComponents BV – www.tallcomponents.com

# 11. Layers

The PDF specification allows you to group graphics objects together in layers and turn visiblity selectively on and off in any combination. This is typically used in e..g CAD drawing to show/hide levels of details. The following snapshot from the Adobe PDF Reader shows how the user can turn on/off visibility of layers.

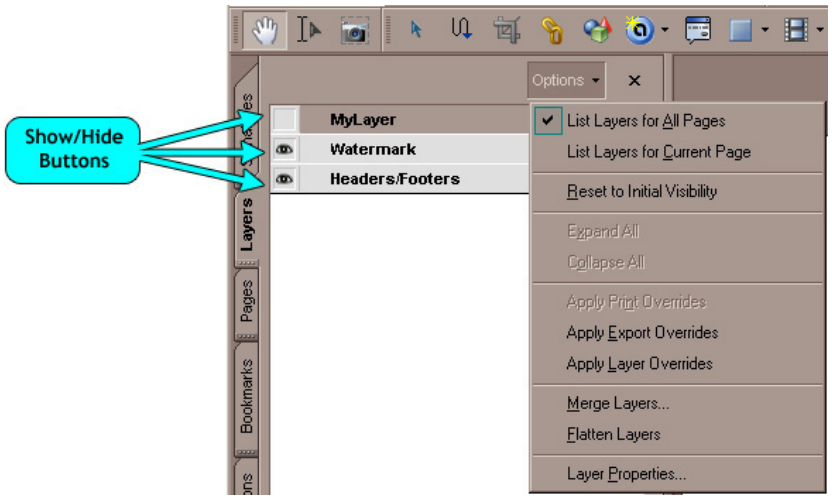



Figure 1 Visibility of layers can be turned on and off

## 11.1. Basic code sample

PDFRasterizer.NET 3.0 lets you control how to render the different layers. Here is a basic code sample that shows you how to do this:

```
using (FileStream file = new FileStream(
   @"layers.pdf", FileMode.Open, FileAccess.Read))
{
   Document document = new Document(file);
   Page page = document.Pages[0];

   LayerCollection layers = document.Layers;
   // render all layers
   foreach (LayerInfo layer in layers)
   {
      layer.Visible = true;
   }

   using (Bitmap bitmap = new Bitmap((int)page.Width, (int)page.Height))
   {
      Graphics graphics = Graphics.FromImage(bitmap);

      RenderSettings renderSettings = new RenderSettings();
      renderSettings.LayerSettings.Layers = layers;

      page.Draw(graphics, renderSettings);
      bitmap.Save(@"layers.png", ImageFormat.Png);
   }
}
```

Code sample 11-1 Control rendering of layers


© TallComponents BV – www.tallcomponents.com

## 11.2. Clone Layers

The code sample above assigns `document.Layers` to `LayerSetitings.Layers` by reference. This means that in principal the settings may change afterwards if a second thread modifies the `document.Layers` collection. To avoid this you should make clone as follows:

```
LayerCollection layers = document.Layers;
RenderSettings renderSettings = new RenderSettings();
renderSettings.LayerSettings.Layers = layers.Clone();
```

Code sample 11-2 Clone the LayerCollection

## 11.3. Default behavior

If `LayerSettings.Layers` equals `null`, the layer visibility settings as stored in the document will be used. In other words, the document will be rendered equal to how e.g. the Adobe PDF Reader will show the document initially.


© TallComponents BV – www.tallcomponents.com

# 12. Diagnostics

It is possible to collect diagnostics about the render process by passing a Summary instance to the various render/conversion methods. This class builds a list of messages that can be inspected after the render process has finished.

## 12.1. Basic code sample

The following code sample demonstrates how the Summary class provides diagnostics:

```
Summary summary = new Summary();
page.Draw(graphics, new RenderSettings(), summary);

foreach (Message message in summary.Messages)
{
   switch (message.Severity)
   {
      case MessageSeverity.Information:
         Console.WriteLine("Info: {0}", message.Text);
         break;
      case MessageSeverity.Warning:
         Console.WriteLine("Warning: {0}", message.Text);
         break;
      case MessageSeverity.Error:
         Console.WriteLine("Error: {0} | {1}", message.Text,
            message.Exception);
         break;
   }
}
```

Code sample 12-1 Dump render diagnostics to the output console


© TallComponents BV – www.tallcomponents.com