

Table of Contents

Document Solutions for Imaging Overview	3
Key Features	4
Getting Started	5-7
Quick Start	8-10
License Information	11
Technical Support	12
Contacting Sales	13
Redistribution	14
End-User License Agreement	15
Product Architecture	16-21
Features	22
Create Image	23-29
Load Image	30-32
Save Image	33
Work with GIF files	34-36
Work with TIFF Images	37-39
Work with ICO files	40-41
Work with SVG Files	42-46
Work with WebP Files	47
Process Image	48-59
Apply Effects	60-63
Layouts	64-81
Complex Graphic Layouts	82-93
Tables	94-119
Work with Image Colors	120-122
Transparency Mask	123-129
Work with Graphics	130
Draw and Fill Shapes	131-135
Clip Region	136-137
Align Image	138-139
Apply Matrix Transformation	140-141
Add Transparency Layer	142-143
Interpolation Mode	144-147
Add Shadow	148-152

Add Glow and Soft Edges	153-155
Work with Text	156-170
Draw Rotated Text	171-180
Work with Exif Metadata	181-182
Render HTML to Image	183-187
Render Using Skia Library	188-189
Document Solutions Image Viewer	190
Samples	191
API Reference	192
Release Notes	193
Breaking Changes	194
Version 7.1.0	195
Version 7.0.0	196
Version 6.2.0	197
Version 6.1.0	198-199
Version 6.0.0	200
Version 5.2.0.800	201
Version 5.1.0.790	202
Version 5.0.0.762	203
Version 4.2.0.715	204
Version 4.1.0.658	205
Version 4.0.0.616	206
Version 3.1.0.508	207
Version 3. 0. 0. 414	208
Version 2.2.0.310	209-210

Document Solutions for Imaging Overview

Document Solutions is a cross-platform solution for document management which provides a universal document, editor and viewer solution for all popular document formats.

Document Solutions for Imaging (Dslmaging, previously Gclmaging), is a part of Document Solutions product line, that offers imaging API for image processing without using any external image editor. The library can create, load, modify, and save images programmatically. The library supports Windows, macOS, and Linux and can also be deployed as FaaS with AWS Lambda, Azure Functions, etc. The library offers a feature-rich API that can be used to create and load popular image formats, such as JPEG, PNG, BMP, GIF, TIFF, ICO, SVG, WebP and apply advanced image processing techniques and save them. In addition to reading and writing images, the library also allows developers to rotate, crop, resize, convert images, draw and fill graphics on images, draw text on images, apply dithering and thresholding effects on grayscale images, apply effects on RGB images, apply advanced TIFF features and much more.

Key Features

Dslmaging provides different features that enable the developers to build intuitive and professional-looking applications. The main features for Dslmaging Library are as follows:

- **Fast and efficient library**
Dslmaging saves memory and time with its lightweight API architecture. It allows you to apply advanced imaging effects in less time for yielding high-performance results.
- **Create, load, modify, convert and save images programmatically**
Using Dslmaging, you can programmatically create images in .NET Standard applications, with full support on Windows, macOS and Linux, without the help of an external image processor. You can also load, modify, convert the popular image formats, such as JPEG, PNG, BMP, GIF, and TIFF, and save them again.
- **Process images with advanced imaging effects**
Dslmaging lets you rotate, flip, crop, resize, composite, blend, apply dithering, thresholding and RGB effects on images.
- **Process GIF files**
Dslmaging allows you to read individual frames from a GIF file and save them as images in different formats supported by Dslmaging. It also supports the creation of a GIF file by using multiple frames.
- **Create thumbnails**
Dslmaging allows you to downscale the images and apply various interpolation algorithms for creating image thumbnails.
- **Draw and fill graphics**
Using Dslmaging, you can draw and fill graphics like lines, polygons, rectangles, rounded rectangles, ellipses, paths on the graphics.
- **Advanced processing of image colors**
Dslmaging allows you to adjust color intensity and histogram levels of an image. Additionally, it lets you perform advanced imaging operations with color channels and color quantization.
- **Draw text on images**
Dslmaging lets you draw text with advanced font and allows paragraph formatting on images. It also supports RTL text and Kashida on Arabic text, and bitmap glyphs in OpenType CJK fonts.
- **Advanced TIFF processing**
Dslmaging supports reading and writing TIFF frames, apply TIFF compression and color spaces, tiled images and other advanced processing on TIFF images.
- **Work with EXIF (Exchangeable Image File Format) Metadata**
Dslmaging allows you to extract the EXIF metadata, such as the shutter speed, flash use, focal length, light value, location, title, creator, date, description, copyright etc. from the JPEG, PNG, TIFF images and save EXIF profile to the same image formats.
- **Seamless HTML to Image rendering**
Dslmaging library along with DsHtml library, allows you to render HTML text or files to images.

For additional information about the supported features in Dslmaging, see [Features](#) topic.

Getting Started

System Requirements


The Dslmaging packages are fully supported on Visual Studio 2017 or later for Windows, Visual Studio for MAC, and Visual Studio Code for Linux and are compatible with the following:

- .NET 5, [.NET 6](#), and [.NET 7](#)
- .NET Core 2.x and 3.x
- .NET Standard 2.x
- .NET Framework 4.6.1 or higher

Setting up an Application

Dslmaging references are available through NuGet, a Visual Studio extension that adds the required libraries and references to your project automatically. To work with Dslmaging, you need to have following references in your application:

Reference	Purpose
DS.Documents.Imaging	To use Dslmaging in an application, you need to reference (install) just the DS.Documents.Imaging package. It will pull in the required infrastructure packages.
DS.Documents.Imaging.Windows	DS.Documents.Imaging.Windows provides support for font linking specified in the Windows registry, and access to native Windows imaging APIs, improving performance and adding some features (e.g. reading TIFF-JPEG frames).
DS.Documents.DX.Windows	DS.Documents.DX.Windows is an infrastructure package used by DS.Documents.Imaging.Windows. You do not need to reference it directly.
DS.Documents.Imaging.Skia	Skia represents a rendering engine based on SkiaSharp and is used for drawing text and graphics. You can optionally install this package for rendering quality graphics across various platforms. For more information, see Render using Skia Library .

 **Note:** With v7.0, GrapeCity.Documents.Imaging (Gclmaging) package is renamed to DS.Documents.Imaging (Dslmaging). The namespaces and classes within DS.Documents.Imaging remain the same, which provide the same functionality and are backwards compatible with GrapeCity.Documents.Imaging, ensuring minimal impact on your existing projects.

To upgrade Gclmaging package to Dslmaging package in your existing projects, follow one of the below options:

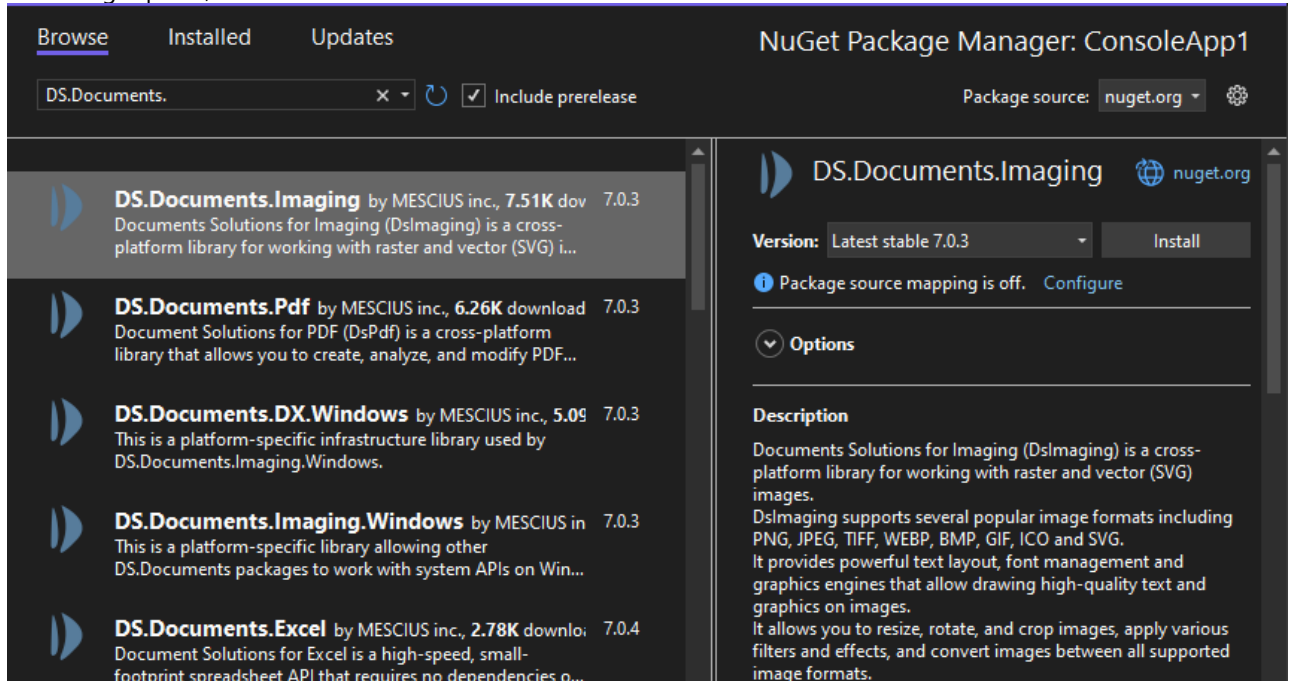
- Update package using Migration tool:
 1. The migration tool is present in the package downloaded from the website. Follow the instructions on the tool for a seamless migration from Gclmaging to Dslmaging.
- Update package manually from NuGet package manager:
 1. In **Solution Explorer**, right-click either **Dependencies** or a project and select **Manage NuGet Packages**.
 2. In **Installed** tab, click on **GrapeCity.Documents.Imaging** package and click **Uninstall** to remove it and its dependencies from the project.
 3. In **Browse** tab, type "ds.documents" or "DS.Documents" in the search text box at the top and find the package "DS.Documents.Imaging".
 4. Click Install to add the **DS.Documents.Imaging** package and its dependencies into the project.

Add reference to Dslmaging in your application from NuGet.org

In order to use Dslmaging in a .NET Core, ASP.NET Core, .NET Framework application (any target that supports .NET Standard 2.0), install the NuGet packages in your application using the following steps:

Visual Studio for Windows

1. Open Visual Studio.
2. Create any application (any target that supports .NET Standard 2.0).
3. Right-click the project in Solution Explorer and choose **Manage NuGet Packages**.
4. In the **Package source** on top right, select **nuget.org**.
5. Click **Browse** tab on top left and search for "DS.Documents".
6. On the left panel, select **DS.Documents.Imaging**
7. On the right panel, click **Install**.



8. In the **Preview Changes** dialog, click **OK** and choose **I Accept** in the next screen.

This adds all the required references of the package to your application. After this step, follow the steps in the [Quick Start](#) section.

Visual Studio for Mac

1. Open Visual Studio for Mac.
2. Create any application (any target that supports .NET Standard 2.0).
3. In tree view on the left, right-click **Dependencies** and choose **Add Packages**.
4. In the Search panel, type "DS.Documents".
5. From the list of packages displayed in the left panel, select **DS.Documents.Imaging** and click **Add Packages**.
6. Click **Accept**.

This automatically adds references of the package and its dependencies to your application. After this step, follow the steps in the [Quick Start](#) section.

Visual Studio Code for Linux

1. Open Visual Studio Code.
2. Install **Nuget Package Manager** from **Extensions**.
3. Create a folder "MyApp" in your **Home** folder.
4. In the Terminal in Visual Studio Code, type "cd MyApp"
5. Type command "dotnet new console"
Observe: This creates a .NETCore application with MyApp.csproj file and Program.cs.
6. Press **Ctrl+P**. A command line opens at the top.
7. Type command: ">"
Observe: "**Nuget Package Manager: Add Package**" option appears.

8. Click the above option.
9. Type "**DS**" and press Enter.
Observe: DS packages get displayed in the dropdown.
10. Choose **DS.Documents.Imaging**.
11. Type following command in the Terminal window: `"dotnet restore"`

This adds references of the package to your application. After this step, follow the steps in the [Quick Start](#) section.

Quick Start

The following quick start sections help you in getting started with the DslImaging library:

- **Create and Save an Image**
- **Load and Modify an Image**

Create and Save an Image

This quick start covers how to create an image and draw string on it in a specified font using a .NET Core or .NET Standard application. Follow the steps below to get started:

1. **Create an instance of GcBitmap class**
2. **Draw and fill a rectangle**
3. **Save the image**



Step 1: Create an instance of GcBitmap class

1. Create a new application (.NET Core Console App\Windows Forms App) and add the references.
2. Include the following namespaces
 - using GrapeCity.Documents.Imaging;
3. Create a new image using an instance of **GcBitmap** class, through code.

```
C#  
  
//Create GcBitmap  
var bmp = new GcBitmap(1024, 1024, true, 96, 96);  
//Create a graphics for drawing  
GcBitmapGraphics g = bmp.CreateGraphics();
```

Back to Top

Step 2: Draw and fill a rectangle

Add the following code to draw a rectangle using the **RectangleF** class, and then add text to it using the **DrawString** method of **GcBitmapGraphics** class.

C#

```
//Add a radial gradient
RadialGradientBrush r= new RadialGradientBrush(Color.Beige,
        Color.RosyBrown, new PointF(0.5f, 0.5f), true);

//Draw a rectangle
var rc = new RectangleF(0, 0, bmp.Width, bmp.Height);

//Fill the rectangle using specified brush
g.FillRectangle(rc, r);

// Create a text format for the "Hello World!" string:
TextFormat tf = new TextFormat();

//Pick a font size, color and style
tf.FontSize = 80;
tf.FontStyle = FontStyle.BoldItalic;
tf.ForeColor = Color.Chocolate;

//Draw the string (text)
g.DrawString("Hello World!", tf, rc, TextAlignment.Center,
        ParagraphAlignment.Center, false);
```

[Back to Top](#)

Step 3: Save the image

Save the image using **SaveAsJpeg** method of the **GcBitmap** class.

C#

```
//Save bitmap as JPEG image
bmp.SaveAsJpeg("HelloWorld.jpg");
```

[Back to Top](#)

Load and Modify an Image

This quick start covers how to load an existing image, modify and save it using a .NET Core or .NET Standard application. Follow the steps below to get started:

1. **Load an existing image**
2. **Modify the image**
3. **Save the image**

Step 1: Load an existing image

1. Create a new application (.NET Core Console App\Windows Forms App) and add the references.
2. Include the following namespace
 - o using GrapeCity.Documents.Imaging;
3. Load an existing image using **Load** method of the **GcBitmap** class.

C#

```
//Create GcBitmap
```

```
var bmp = new GcBitmap();
var fs = new FileStream(Path.Combine("puffins-small.jpg"), FileMode.Open,
    FileAccess.ReadWrite);
//Load image
bmp.Load(fs);
```

[Back to Top](#)

Step 2: Modify the image

1. Add the following code that to add a text using the **DrawString** method of **GcBitmapGraphics** class to draw string.

```
C#
//Create a graphics for drawing
GcBitmapGraphics g = bmp.CreateGraphics();

// Create a text format for the string:
TextFormat tf = new TextFormat();

// Pick a font size, color and style
tf.FontSize = 10;
tf.ForeColor = Color.Red;
tf.FontStyle = FontStyle.BoldItalic;

//Draw the string (text)
g.DrawString("Penguins", tf, new PointF(10, 10));
```

[Back to Top](#)

Step 3: Save the image

Save the image using **SaveAsJpeg** method of the **GcBitmap** class.

```
C#
//Save bitmap
bmp.SaveAsJpeg("NewImage.jpg");
```

[Back to Top](#)

License Information

Document Solutions for Imaging supports the following types of license:

- **Unlicensed**
- **Evaluation License**
- **Licensed**

Unlicensed

After downloading the product, the product works in the unlicensed mode. However, not more than 10 instances of `GcBitmapGraphics` and `GcWicBitmapGraphics` (combined) can be created when the product is used without license.

If you have already created 10 instances of `GcBitmapGraphics` (`BitmapRenderer`) and `GcWicBitmapGraphics` (`RenderTarget`), following exception is thrown on creating the next instance:

'Unlicensed copy of Document Solutions for Imaging. The number of GcBitmapGraphics (BitmapRenderer) and GcWicBitmapGraphics (RenderTarget) instances is limited to 10. Contact us.sales@mescius.com to get your 30-day evaluation key.'

Evaluation License

Dslmaging evaluation license is available to users for 30 days to evaluate the product. If you want to evaluate the product, you can ask for evaluation license key by sending an email to us.sales@mescius.com.

The evaluation version has an expiration date that is determined when an evaluation key is generated. After applying the evaluation license key, you can use the complete product until the license expiry date.

After the expiry date, following exception is thrown:

'This evaluation copy of Document Solutions for Imaging has expired. Contact us.sales@mescius.com to purchase your license. To continue using Dslmaging with limitations, remove the expired evaluation license key.'

Licensed

Dslmaging production license is issued at the time of purchase of the product. If you have a production license, you can access all the features of Dslmaging without any limitations.

Apply License

To apply evaluation/production license in Dslmaging, the long string key needs to be copied to the code in one of the following two ways.

- Pass it as an argument to the `GcBitmap`'s ctor:

```
var bmp = new GcBitmap();
bmp.ApplyLicenseKey("Key");
```

This licenses the instance being created.
- Call a static method on `GcBitmap`:

```
GcBitmap.SetLicenseKey("key");
```

This licenses all the instances while the program is running.

GcWicBitmap and **GcSvgGraphics** are licensed just like **GcBitmap**, using the same keys and instance counts. Also, please note that if a `GcBitmap` is converted to `GcWicBitmap` or vice versa, the converted object also gets licensed if the original was.

Technical Support

If you have a technical question about this product, consult the following sources:

- Product forum: <https://developer.mescius.com/forums>
- Email: us.sales@mescius.com


Contacting Sales

If you would like to find out more about our products, contact our Sales department using one of these methods:

World Wide Web site	https://developer.mescius.com/
E-mail	us.sales@mescius.com
Phone	(800) 858-2739 or (412) 681-4343 outside the U.S.A.
Fax	(412) 681-4384

Redistribution

To distribute an application containing the DsImaging API, it is necessary to have a valid Distribution License and meet all [System Requirements](#).

 DsImaging makes it easy to deploy your application to your local servers or cloud offerings such as Azure.

For more information about Distribution License, contact our Sales department using one of these methods:

World Wide Web site	https://developer.mescius.com/
E-mail	us.sales@mescius.com
Phone	1.800.858.2739 or 412.681.4343
Fax	(412) 681-4384

End-User License Agreement

The MESCIUS licensing information, including the MESCIUS end-user license agreement, frequently asked licensing questions, and the MESCIUS licensing model, is available online. For detailed information on licensing, see [MESCIUS Licensing](#). For MESCIUS end-user license agreement, see [End-User License Agreement For MESCIUS Software](#).

Product Architecture

Packaging

DslImaging is a collection of cross-platform .NET class libraries written in C#, providing an API that allows to create image from scratch and to load, analyze and modify existing images.

DslImaging is compatible with .NET Core 2.x/3.x, .NET Standard 2.x, .NET Framework 4.6.1 or higher, and .NET 6 or higher.

DslImaging and supporting packages are available on nuget.org:

- **DS.Documents.Imaging**
- **DS.Documents.Imaging.Windows**
- **DS.Documents.Imaging.Skia**
- **DS.Documents.DX.Windows**

To use DslImaging in an application, you need to reference just the **DS.Documents.Imaging** package. It pulls in the required infrastructure packages.

On a Windows system, you can optionally install **DS.Documents.Imaging.Windows**. It contains GcWicBitmap class that works efficiently with various image formats and allows drawing text and graphics using DirectWrite/Direct2D-based functionality. Also, it provides support for font linking specified in the Windows registry. This library can be referenced on a non-Windows system, but does nothing.

DslImaging API Overview

Namespaces

Namespaces	Description
GrapeCity.Documents.Drawing	Framework for drawing on the abstract GcGraphics surface.
GrapeCity.Documents.Imaging	Types used to create, process and modify images. Nested namespaces contain types supporting specific image spec areas: <ul style="list-style-type: none">• GrapeCity.Documents.Drawing• GrapeCity.Documents.Imaging
GrapeCity.Documents.Text	Text processing sub-system.
GrapeCity.Documents.Imaging.Skia	Types used for drawing text and graphics using a highly optimized library SKIA.

DslImaging provides classes for three main purposes.

- Creating new images or loading images from various formats including multi-frame GIFs and TIFFs
- Drawing graphics and text on the in-memory bitmaps, applying various effects and transformation to the bitmaps
- Saving the resulting images as JPEG, PNG, BMP, multipage TIFF, multi-frame GIF or WebP.

DslImaging classes can be used efficiently in a multi-threaded environment. They don't depend on system handles or UI threads.

Image Containers

There are several containers in the DsImaging package (DS.Documents.Imaging) and in the related Windows specific package (DS.Documents.Imaging.Windows).

- **GcBitmap** is a platform-independent storage for raster images. You can access individual pixels as 32-bit unsigned integer values in the ARGB format where alpha component is the most significant byte. The Alpha channel is either pre-multiplied to Red, Green, Blue color channels or it is not pre-multiplied at all. GcBitmap can be encoded and saved to BMP, PNG, JPEG, GIF, TIFF, WebP or decoded and loaded from the same set of image formats.
- **GrapeCity.Documents.Imaging.Windows.GcWicBitmap** which resides in the DS.Documents.Imaging.Windows package, is very similar to GcBitmap but uses the Windows Imaging Component (WIC) subsystem for storing a raster image. GcWicBitmap supports various pixel formats and conversion between the formats. Usually, it works with 32-bit ARGB pixels and pre-multiplied Alpha channel. It is easy to copy such an image between GcWicBitmap and GcBitmap classes. GcWicBitmap can be saved to BMP, PNG, JPEG, GIF, TIFF, WebP, JPEGXR and loaded from BMP, PNG, JPEG, GIF, TIFF, WebP, JPEGXR, ICO image formats. It works faster than GcBitmap but is available only on the Windows platform and lacks some of the functionalities of GcBitmap such as direct pixel access.
- **GrayscaleBitmap** is a platform-independent storage for a grayscale image with 8 bits per pixel or an 8-bit transparency mask. It is four times more compact than GcBitmap. A full-color GcBitmap can be transformed to grayscale using GrayscaleEffect, and can easily be converted to GrayscaleBitmap. It is possible to save a GrayscaleBitmap to TIFF and load it from a TIFF file. Working with other image formats requires conversion to GcBitmap. GrayscaleBitmap is handy to use as a transparency mask to be applied to GcBitmap.
- **BilevelBitmap** is a compact storage for a bi-level transparency mask or an image containing two colors, such as black and white. To convert a full-color GcBitmap to BilevelBitmap, you need to apply the GrayscaleEffect, then apply one of the dithering or thresholding effects to make the image bi-level. The result can be stored as a BilevelBitmap. It supports saving to TIFF and loading from TIFF. You can read or modify individual pixels in BilevelBitmap and apply it to GcBitmap as a transparency mask.
- **Indexed4bppBitmap** and **Indexed8bppBitmap** are palette-based containers with 4-bit or 8-bit pixels containing indices of corresponding palette entries. These images can be saved to TIFF and loaded from TIFF. Indexed8bppBitmap can also be loaded from GIF and both Indexed4bppBitmap and Indexed8bppBitmap can be saved to GIF. It is easy to convert full-color GcBitmap to an indexed bitmap using one of the dithering algorithms. The palette entries and pixels are accessible for modifications.
- **Image** is a lightweight class representing the image in a file, stream, or array of bytes. You can draw the Image on GcGraphics, convert it to GcBitmap, or save to a MemoryStream in the original format.

Graphics

GrapeCity.Documents.Drawing.GcGraphics is an abstract base class for implementing graphics functionality for different targets. It allows to draw graphics primitives and text on various media, including **GcBitmap**, **GcWicBitmap**, and **GcPdfDocument**. The **GcGraphics** class offers roughly the same functionality as **System.Drawing.Graphics** class in **WinForms** but is platform-independent and provides implementations for different targets.

The **GcBitmapGraphics** class is derived from GcGraphics and allows to draw on a GcBitmap. Use **GcBitmap.CreateGraphics()** method to create an instance of GcBitmapGraphics. Likewise, **GcWicBitmap.CreateGraphics()** method creates an instance of GcWicBitmapGraphics that can be used to draw on a **GcWicBitmap**. Please note that you need to dispose the graphics objects after use.

Classes like **GcBitmapGraphics** and **GcWicBitmapGraphics** obey the universal object model for drawing with **GcGraphics**. Internally, both classes are based on more specific implementations targeting the actual media, such as **GcBitmap** or **GcWicBitmap**.

Renderer Classes

The target-specific renderer classes like **BitmapRenderer** for **GcBitmap** and **GrapeCity.Documents.DX.Direct2D.RenderTarget** for **GcWicBitmap** provide access to various fine-tuning settings and to methods not supported by the universal GcGraphics abstract class.

For example, you must work with `BitmapRenderer` to update anti aliasing setting or to enable multi threading during the rendering phase. An instance of **BitmapRenderer** is available through the **GcBitmap.Renderer** and **GcBitmapGraphics.Renderer** properties. An important feature provided by `BitmapRenderer` is the capability to work with lightweight objects called regions, that can be created from simple figures and graphics paths. Regions can be combined using various logical operations, then filled with brushes or used for clipping.

TIFF Reader/Writer

DsImaging has special support for multi page TIFF format. **GcTiffReader** allows to read individual images from a multi page TIFF file or stream. After the proper initialization, the user can access **GcTiffReader.Frames** property, which is a list of **TiffFrame** class instances. `TiffFrame` is a lightweight reference to the actual image data. It allows to read the frame image into one of the container classes, such as **BilevelBitmap** or **GcBitmap**. `GcTiffReader` works on any platform but has some limitations. For example, it does not currently support TIFF-JPEG compression scheme.

The **GcWicTiffReader** from **GrapeCity.Documents.Imaging.Windows** namespace in **DS.Documents.Imaging.Windows** package is a platform-dependent counterpart for `GcTiffReader`. It is based on the Windows Imaging Component subsystem and supports a few color spaces and compression schemes which are currently not available with platform-independent **GcTiffReader**. The `Frames` collection in **GcWicTiffReader** contains instances of the **WicTiffFrame** class. It allows to read frame images into the `GcWicBitmap` image container.

GcTiffWriter is a platform-independent class making it possible to create a multi page TIFF file or stream from a set of individual images. You can append images, such as **GrayscaleBitmap**, **Indexed8bppBitmap** and so on, to a `GcTiffWriter` and specify the detailed settings controlling the frame storage format and metadata using the **TiffFrameSettings** class. `GcTiffWriter` fully supports the Baseline TIFF specification and several TIFF extensions, such as tiled images, the Deflate compression scheme, associated and unassociated Alpha and other features. **GcWicTiffWriter** is a Windows-specific WIC-based class that allows to write `GcWicBitmaps` to TIFF as separate frames. It does not offer much functionality beyond `GcTiffWriter`, but may be handy when drawing images to `GcWicBitmap` and saving them as a bunch.

GIF Reader/Writer

DsImaging has special support for multi-frame GIF format. **GcGifReader** allows to read individual frames. After the proper initialization, the user can access **GcGifReader.Frames** property, which is a list of `GifFrame` class instances. **GifFrame** is a lightweight reference to the actual image data. It allows to read the frame image into one of the container classes, such as **Indexed8bppBitmap** or **GcBitmap**.

GcGifWriter is a platform-independent class making it possible to create a multi-frame GIF file or stream from a set of individual images. You can append images, such as **GrayscaleBitmap**, **Indexed8bppBitmap** and so on, to a **GcGifWriter** and specify the detailed settings controlling the frame storage format and the playback (animation) properties.

DsHtml API Overview

DsHtml is a utility library that renders HTML to PDF file or an image in PNG, JPEG, and WebP format. DsHtml uses a Chrome or Edge browser (already installed in the current system, or downloaded from a public web site) in headless mode. Also, it doesn't matter whether your .NET application is built for x64, x86 or AnyCPU platform target. The browser is continuously working in a separate process.

The **DS.Documents.Html** library consists of a platform-independent main package that exposes the HTML rendering functionality. The main package contains the following namespaces:

Namespaces	Description
GrapeCity.Documents.Drawing	It provides extension methods and formatting attributes for rendering HTML to image. The namespace comprises the following classes:

	<ul style="list-style-type: none"> ● GcBitmapGraphicsExt ● HtmlToImageFormat
GrapeCity.Documents.Html	<p>It provides methods for converting HTML to PDF or images and defines parameters for the PDF or image.</p> <p>The namespace comprises the following classes:</p> <ul style="list-style-type: none"> ● BrowserFetcher ● GcHtmlBrowser ● HtmlPage ● ImageOptions ● JpegOptions ● LaunchOptions ● PageOptions ● PdfMargins ● PdfOptions ● PngOptions ● TimeOutOptions ● WebpOptions
GrapeCity.Documents.Pdf	<p>It provides the extension methods for rendering HTML to image and represents the formatting attributes for rendering HTML to image.</p> <p>The namespace comprises the following classes:</p> <ul style="list-style-type: none"> ● GcPdfGraphicsExt ● HtmlToPdfFormat

GrapeCity.Documents.Html.BrowserFetcher

The **BrowserFetcher** class has two static methods: `GetSystemChromePath()` and `GetSystemEdgePath()`. The methods return the path to an executable file of Chrome or Edge browsers correspondingly. Another option is to download and install Chromium into a local folder. You can create an instance of `BrowserFetcher` and pass the information such as host, platform, revision, and the destination folder, if needed. Then, execute the `BrowserFetcher.GetDownloadedPath()` method which downloads Chromium, if required, and returns the path to an executable file for running the Chromium.

GrapeCity.Documents.Html.GcHtmlBrowser

The **GcHtmlBrowser** class provides methods for converting HTML to PDF and images. With path to executable file for running the Chromium fetched using `BrowserFetcher` class, we can create an instance of `GcHtmlBrowser` class which effectively runs the browser process in the background. `GcHtmlBrowser` also accepts another parameter of `LaunchOptions` type. The `LaunchOptions` class provides various settings specific to launching the browser.

The class has two important methods: **NewPage(Uri uri)** and **NewPage(string html)**. Both methods return an instance of **HtmlPage** class which represents a browser tab after navigating to the specified web address, file, or the arbitrary HTML content. The second parameter of `PageOptions` type provides various properties to be applied to the new browser page such as username, password for HTTP authentication, disabling JavaScript, lazy loading etc.

Note:

- We recommend using Chrome browser with `GcHtmlBrowser` class as Edge has some differences in the

implementation of some DevTools features.

- It is important to dispose every instance of the GcHtmlBrowser and HtmlPage classes after use.

GrapeCity.Documents.Html.HtmlPage

The **HtmlPage** class represents a browser tab after navigating to the specified web address, file, or the arbitrary HTML content. The class has methods such as `SaveAsPng`, `SaveAsJpeg`, and `SaveAsWebp` to save the current page as a raster image of PNG, JPEG, or WebP formats respectively. The first parameter of these methods specifies the destination file or stream. The second parameter passes the additional options for rendering HTML page as single image, scaling or cropping the image, or setting the image quality.

The `HtmlPage` class contains the additional methods that help to interact with HTML page content. For example, you can obtain the full HTML content of the page using the `GetContent` method. The `SetContent` method updates the HTML markup. You can reload the web page with the `Reload` method or even execute a script in the browser context using the `EvaluateExpression` method. The `WaitForNetworkIdle` method helps with loading asynchronous web content.

GrapeCity.Documents.Html.ImageOptions

ImageOptions is the base abstract class for three specific classes: `PngOptions`, `WebpOptions` and `JpegOptions`. As compared to `PngOptions` and `WebpOptions` classes, the `JpegOptions` class has an additional property (`CompressionQuality`) for providing the JPEG compression quality (from 0% to 100%).

The **FullPage** property allows to capture the whole scrollable page. While the `Clip` property specifies the region to capture (if `FullPage` is false). `Clip` and `Scale` properties work with the result of layout. They allow to extract and scale some rectangular area and are applied before the rasterization stage. So, any graphics remains crisp with any scale factor. When exporting HTML to images the Dots Per Inch (DPI) is not set in the resulting image file. It requires some post-processing in order to set DPI.

GrapeCity.Documents.Drawing.HtmlToImageFormat

The **HtmlToImageFormat** class represents the formatting attributes for rendering HTML to `GcGraphics` as an image. Also, it helps converting HTML to a `GcBitmap`.

MaxTopMargin, **MaxBottomMargin**, **MaxLeftMargin**, **MaxRightMargin** properties specify the maximum allowable margins of the resulting image (larger margins will be trimmed), in pixels. Setting these properties to a negative value prevents trimming the margins. All those properties are equal to 0 by default which means no margins.

Other properties of `HtmlToImageFormat` are mapped to the corresponding properties of the `ImageOptions/PageOptions` class:

HtmlToImageFormat Property	ImageOptions/PageOptions Property
<code>DefaultBackgroundColor</code>	<code>PageOptions.DefaultBackgroundColor</code>
<code>WindowSize</code>	<code>PageOptions.WindowSize</code>
<code>MaxWindowWidth</code>	<code>PageOptions.WindowSize.Width</code>
<code>MaxWindowHeight</code>	<code>PageOptions.WindowSize.Height</code>
<code>FullPage</code>	<code>ImageOptions.FullPage</code>
<code>Scale</code>	<code>ImageOptions.Scale</code>
<code>Clip</code>	<code>ImageOption.Clip</code>

GcBitmapGraphics Extension Methods

`DsHtml` provides 2 methods that extend `GcBitmapGraphics` and allow to render an HTML text or page as an image:

- Draws an HTML string on this GcBitmapGraphics at a specified position.
bool GcBitmapGraphics.**DrawHtml**(GcHtmlBrowser browser, string html, float x, float y, HtmlToImageFormat format, out SizeF size, bool loadLazyImages = false)
- Draws an HTML page provided by a URI on this GcBitmapGraphics at a specified position.
bool GcBitmapGraphics.**DrawHtml**(GcHtmlBrowser browser, Uri htmlUri, float x, float y, HtmlToImageFormat format, out SizeF size, bool loadLazyImages = false)

Skia API Overview


Namespaces

Namespaces	Description
Grapecity.Documents.Imaging.Skia	Types used for drawing text and graphics using a highly optimized library SKIA.

Skia comprises the following main classes:

- **GcSkiaBitmap**: It represents a bitmap in CPU memory. It works similar to GcBitmap and GcWicBitmap but internally encapsulates an instance of SKBitmap object from SkiaSharp. GcSkiaBitmap can load images in JPEG, PNG, and WEBP formats and save images in the same formats. Also, you can convert GcSkiaBitmap to a GcBitmap or GcSkialmage and vice versa. It is possible to draw text and graphics on GcSkiaBitmap after executing the CreateGraphics method which returns an instance of the associated GcSkiaGraphics.
- **GcSkialmage**: It is an immutable image based on SKImage. It looks like a lightweight version of GcSkiaBitmap which does not support any modifications. You can load and save GcSkialmage to the same formats as GcSkiaBitmap, and convert it to GcBitmap or GcSkiaBitmap. Both GcSkialmage and GcSkiaBitmap implement the Image interface and hence can be drawn to any GcGraphics derived class.
- **GcSkiaGraphics**: It is the main drawing class which is derived from GcGraphics. You can create an instance of GcSkiaGraphics from either GcSkiaBitmap or directly. When the drawing is done you can simply dispose the graphics object in case of drawing to GcSkiaBitmap. If the GcSkiaGraphics object was created directly you can execute ToSkialmage() or ToGcBitmap() methods to get a snapshot of the current drawing. If you draw text to multiple instances of GcSkiaGraphics please make sure that you created and assigned the same SkiaFontCache object to the FontCache property of all those instances.

For more information about Skia library, see [Render using Skia Library](#).

 **Note:** In DslImaging release version 6.0.0, the **GcHtmlRenderer** class has been marked **obsolete** and has been replaced by the new **GcHtmlBrowser** class. This is done to avoid GPL or LGPL licensed software that had to be used in the custom chromium build. To know tips about migration from obsolete GcHtmlRenderer class, see [Tips to Migrate from Obsolete GcHtmlRenderer class](#).

Features

This section comprises the features available in Dslmaging.

Create Image

Create images and thumbnails in Dslmaging.

Load Image

Load images from file, stream, and byte array in Dslmaging.

Save Image

Save images to different formats in Dslmaging.

Work with GIF Files

Create a GIF file and read a GIF file to save the frames as separate images in Dslmaging.

Work with TIFF Images

Create a multi-framed TIFF, save TIFF frames as separate images, and create tiled images in Dslmaging.

Work with ICO Files

Create an ICO image file and read images from an ICO file in Dslmaging.

Work with SVG Files

Create an SVG image file and render SVG images to PNG formats in Dslmaging.

Process Image

Resize, crop, rotate, flip, clear, and combine images, convert an image to indexed image and change its resolution in Dslmaging.

Apply Effects

Apply dithering, thresholding, gray scaling, and RGB effects on an image in Dslmaging.

Layouts

Place multiple elements on a PDF page or image without having to calculate positions of each element relative to other ones.

Complex Graphic Layouts

Draw complex graphics, text, and images.

Tables

Create and work with tables easily and straightforwardly without having to think much about the size of table columns, merged cells, or the layout of rotated text.

Work with Image colors

Adjust color intensity and image histogram levels, work with color channels and color quantization in Dslmaging.

Apply Transparency Mask

Set transparency and set the background color for semi-transparent images in Dslmaging.

Work with Graphics

Draw and fill shapes, clip region, align image, and apply matrix transformation in Dslmaging.

Work with Text

Render and trim text, add watermark text on an image, draw text with anti-aliasing and different font types, add complex bitmap glyphs, draw text around images, use RTL, and format paragraphs in Dslmaging.

Work with EXIF Metadata

Extract and modify the EXIF metadata of an image using Dslmaging.

Create Image

An image is a visual representation of information that can be created using a combination of graphics and text. DsImaging allows you to create image(s) programmatically using such graphics, and text. It allows you to create and save images in various image formats such as, JPEG, PNG, BMP, TIFF, SVG, ICO, GIF and WebP.

DsImaging provides two main classes, namely **GcBitmap** and **GcBitmapGraphics**, that can be used to create image(s). The **GcBitmap** class represents an uncompressed in-memory bitmap in 32-bit ARGB format. This class provides **CreateGraphics** method to create graphics for **GcBitmap**. The **CreateGraphics** method creates an instance of **GcBitmapGraphics** class, which lets you draw shapes, graphics, and text to an image. On the other hand, the **GcBitmapGraphics** class derives from the **GcGraphics** class and implements a drawing surface for **GcBitmap**.

Create Image

To create an image:

1. Initialize the **GcBitmap** class.
2. Create a drawing surface to draw shapes and render text for an image using **CreateGraphics** method of the **GcBitmap** class which returns an instance of the **GcBitmapGraphics** class.
3. Draw rounded rectangles and connecting lines in the image using **DrawRoundRect** and **DrawLine** methods of the **GcBitmapGraphics** class respectively.
4. Apply the background color to the rectangles using **FillRoundRect** method of the **GcBitmapGraphics** class.
5. Initialize the **TextFormat** class to define the style used to render text in the image.
6. Add text to the rectangles using **DrawString** method of the **GcBitmapGraphics** class.

```
C#  
  
public void CreateImage(int pixelWidth = 550, int pixelHeight = 350,  
    bool opaque = true, float dpiX = 96, float dpiY = 96)  
{  
    //Initialize GcBitmap with the expected height/width  
    var bmp = new GcBitmap(pixelWidth, pixelHeight, true, dpiX, dpiY);  
  
    //Create graphics for GcBitmap  
    using (var g = bmp.CreateGraphics(Color.LightBlue))  
    {  
        // Rounded rectangle's radii:  
        float rx = 36, ry = 24;  
  
        //Define text format used to render text in shapes  
        var tf = new TextFormat()  
        {  
            Font = Font.FromFile(Path.Combine("Resources", "Fonts", "times.ttf")),  
            FontSize = 18  
        };  
  
        // Using dedicated methods to draw and fill round rectangles:  
        var rec1 = new RectangleF(30, 110, 150, 100);  
        g.FillRoundRect(rec1, rx, ry, Color.PaleGreen);  
        g.DrawRoundRect(rec1, rx, ry, Color.Blue, 4);  
  
        //Draw string within the rectangle  
        g.DrawString("Image", tf, rec1, TextAlignment.Center,  
            ParagraphAlignment.Center, false);  
  
        var rec2 = new RectangleF(300, 30, 150, 100);  
        g.FillRoundRect(rec2, rx, ry, Color.PaleGreen);  
        g.DrawRoundRect(rec2, rx, ry, Color.Blue, 4);  
  
        //Draw string within the rectangle  
        g.DrawString("Text", tf, rec2, TextAlignment.Center,
```

```

        ParagraphAlignment.Center, false);

var rec3 = new RectangleF(300, 230, 220, 100);
g.FillRoundRect(rec3, rx, ry, Color.PaleGreen);
g.DrawRoundRect(rec3, rx, ry, Color.Blue, 4);

//Draw string within the rectangle
g.DrawString("Graphics", tf, rec3, TextAlignment.Center,
    ParagraphAlignment.Center, false);

//Draw lines between the rectangles
g.DrawLine(183, 160, 299, 80, Color.Red, 5, DashStyle.Solid);
g.DrawLine(183, 160, 299, 280, Color.Red, 5, DashStyle.Solid);
}

//Save GcBitmap to jpeg format
bmp.SaveAsJpeg("Image.jpeg");
}

```

[Back to Top](#)

Create SVG Image using Code

To create an SVG image using code:

1. Create a new SVG document by creating an instance of **GcSvgDocument**.
2. Create an instance of **SvgPathBuilder** class. This class provides methods to execute the path commands.
3. Define the path to draw the outline of shape to be drawn on SVG using methods such as **AddMoveTo** and **AddCurveTo**.
4. Add these elements into root collection of 'svg' element using the **Add** method.
5. Provide the **SvgPathData** using **ToPathData** method of the **SvgPathBuilder** class which represents sequence of instructions for drawing the path.
6. Define other properties of each path such as, Fill, Stroke etc.
7. Save the document as SVG by using **Save** method of the **GcSvgDocument** class.
8. To save the SVG as image, use the **DrawSvg** method of the **GcBitmapGraphics** class.

C#

```

public static GcSvgDocument DrawCarrot()
{
    // Create a new SVG document
    var doc = new GcSvgDocument();
    var svg = doc.RootSvg;
    svg.ViewBox = new SvgViewBox(0, 0, 313.666f, 164.519f);

    //Create an instance of SvgPathBuilder class.
    var pb = new SvgPathBuilder();

    //Define the path
    pb.AddMoveTo(false, 29.649f, 9.683f);
    pb.AddCurveTo(true, -2.389f, -0.468f, -4.797f, 2.57f, -6.137f, 5.697f);
    pb.AddCurveTo(true, 2.075f, -2.255f, 3.596f, -1.051f, 4.915f, -0.675f);
    pb.AddCurveTo(true, -2.122f, 2.795f, -4f, 5.877f, -7.746f, 5.568f);
    pb.AddCurveTo(true, 2.384f, -6.014f, 2.963f, -12.977f, 0.394f, -17.78f);
    pb.AddCurveTo(true, -1.296f, 2.591f, -1.854f, 6.054f, -5.204f, 7.395f);
    pb.AddCurveTo(true, 3.575f, 2.455f, 0.986f, 7.637f, 1.208f, 11.437f);
    pb.AddCurveTo(false, 11.967f, 21.17f, 6.428f, 16.391f, 9.058f, 10.67f);
    pb.AddCurveTo(true, -3.922f, 8.312f, -2.715f, 19.745f, 4.363f, 22.224f);
    pb.AddCurveTo(true, -3.86f, 4.265f, -2.204f, 10.343f, 0.209f, 13.781f);
    pb.AddCurveTo(true, -0.96f, 1.808f, -1.83f, 2.546f, -3.774f, 3.195f);
    pb.AddCurveTo(true, 3.376f, 1.628f, 6.612f, 4.866f, 11.326f, 3.366f);
}

```

```

pb.AddCurveTo(true, -1.005f, 2.345f, -12.389f, 9.499f, -15.16f, 10.35f);
pb.AddCurveTo(true, 3.216f, 0.267f, 14.492f, -2.308f, 16.903f, -5.349f);
pb.AddCurveTo(true, -1.583f, 2.84f, 1.431f, 2.28f, 2.86f, 4.56f);
pb.AddCurveTo(true, 1.877f, -3.088f, 3.978f, -2.374f, 5.677f, -3.311f);
pb.AddCurveTo(true, -0.406f, 4.826f, -2.12f, 9.27f, -5.447f, 13.582f);
pb.AddCurveTo(true, 2.834f, -4.894f, 6.922f, -5.367f, 10.474f, -5.879f);
pb.AddCurveTo(true, -0.893f, 4.245f, -3.146f, 8.646f, -7.077f, 10.479f);
pb.AddCurveTo(true, 5.359f, 0.445f, 11.123f, -3.934f, 13.509f, -9.944f);
pb.AddCurveTo(true, 12.688f, 3.209f, 28.763f, -1.932f, 39.894f, 7.084f);
pb.AddCurveTo(true, 1.024f, 0.625f, 1.761f, -4.98f, 1.023f, -5.852f);
pb.AddCurveTo(false, 72.823f, 55.357f, 69.273f, 68.83f, 52.651f, 54.498f);
pb.AddCurveTo(true, -0.492f, -0.584f, 1.563f, -5.81f, 1f, -8.825f);
pb.AddCurveTo(true, -1.048f, -3.596f, -3.799f, -6.249f, -7.594f, -6.027f);
pb.AddCurveTo(true, -2.191f, 0.361f, -5.448f, 0.631f, -7.84f, 0.159f);
pb.AddCurveTo(true, 2.923f, -5.961f, 9.848f, -4.849f, 12.28f, -11.396f);
pb.AddCurveTo(true, -4.759f, 2.039f, -7.864f, -2.808f, -12.329f, -1.018f);
pb.AddCurveTo(true, 1.63f, -3.377f, 4.557f, -2.863f, 6.786f, -3.755f);
pb.AddCurveTo(true, -3.817f, -2.746f, -9.295f, -5.091f, -14.56f, -0.129f);
pb.AddCurveTo(false, 33.228f, 18.615f, 32.064f, 13.119f, 29.649f, 9.683f);

//Add elements into Children collection of SVG
svg.Children.Add(new SvgPathElement()
{
    FillRule = SvgFillRule.EvenOdd,
    Fill = new SvgPaint(Color.FromArgb(0x43, 0x95, 0x39)),
    PathData = pb.ToPathData(),
});

pb.Reset();
pb.AddMoveTo(false, 29.649f, 9.683f);
pb.AddCurveTo(true, -2.389f, -0.468f, -4.797f, 2.57f, -6.137f, 5.697f);
pb.AddCurveTo(true, 2.075f, -2.255f, 3.596f, -1.051f, 4.915f, -0.675f);
pb.AddCurveTo(true, -2.122f, 2.795f, -4f, 5.877f, -7.746f, 5.568f);
pb.AddCurveTo(true, 2.384f, -6.014f, 2.963f, -12.977f, 0.394f, -17.78f);
pb.AddCurveTo(true, -1.296f, 2.591f, -1.854f, 6.054f, -5.204f, 7.395f);
pb.AddCurveTo(true, 3.575f, 2.455f, 0.986f, 7.637f, 1.208f, 11.437f);
pb.AddCurveTo(false, 11.967f, 21.17f, 6.428f, 16.391f, 9.058f, 10.67f);
pb.AddCurveTo(true, -3.922f, 8.312f, -2.715f, 19.745f, 4.363f, 22.224f);
pb.AddCurveTo(true, -3.86f, 4.265f, -2.204f, 10.343f, 0.209f, 13.781f);
pb.AddCurveTo(true, -0.96f, 1.808f, -1.83f, 2.546f, -3.774f, 3.195f);
pb.AddCurveTo(true, 3.376f, 1.628f, 6.612f, 4.866f, 11.326f, 3.366f);
pb.AddCurveTo(true, -1.005f, 2.345f, -12.389f, 9.499f, -15.16f, 10.35f);
pb.AddCurveTo(true, 3.216f, 0.267f, 14.492f, -2.308f, 16.903f, -5.349f);
pb.AddCurveTo(true, -1.583f, 2.84f, 1.431f, 2.28f, 2.86f, 4.56f);
pb.AddCurveTo(true, 1.877f, -3.088f, 3.978f, -2.374f, 5.677f, -3.311f);
pb.AddCurveTo(true, -0.406f, 4.826f, -2.12f, 9.27f, -5.447f, 13.582f);
pb.AddCurveTo(true, 2.834f, -4.894f, 6.922f, -5.367f, 10.474f, -5.879f);
pb.AddCurveTo(true, -0.893f, 4.245f, -3.146f, 8.646f, -7.077f, 10.479f);
pb.AddCurveTo(true, 5.359f, 0.445f, 11.123f, -3.934f, 13.509f, -9.944f);
pb.AddCurveTo(true, 12.688f, 3.209f, 28.763f, -1.932f, 39.894f, 7.084f);
pb.AddCurveTo(true, 1.024f, 0.625f, 1.761f, -4.98f, 1.023f, -5.852f);
pb.AddCurveTo(false, 72.823f, 55.357f, 69.273f, 68.83f, 52.651f, 54.498f);
pb.AddCurveTo(true, -0.492f, -0.584f, 1.563f, -5.81f, 1f, -8.825f);
pb.AddCurveTo(true, -1.048f, -3.596f, -3.799f, -6.249f, -7.594f, -6.027f);
pb.AddCurveTo(true, -2.191f, 0.361f, -5.448f, 0.631f, -7.84f, 0.159f);
pb.AddCurveTo(true, 2.923f, -5.961f, 9.848f, -4.849f, 12.28f, -11.396f);
pb.AddCurveTo(true, -4.759f, 2.039f, -7.864f, -2.808f, -12.329f, -1.018f);
pb.AddCurveTo(true, 1.63f, -3.377f, 4.557f, -2.863f, 6.786f, -3.755f);
pb.AddCurveTo(true, -3.817f, -2.746f, -9.295f, -5.091f, -14.56f, -0.129f);

```

```
pb.AddCurveTo(false, 33.228f, 18.615f, 32.064f, 13.119f, 29.649f, 9.683f);
pb.AddClosePath();
//Add elements into Children collection of SVG
svg.Children.Add(new SvgPathElement()
{
    Fill = SvgPaint.None,
    Stroke = new SvgPaint(Color.Black),
    StrokeWidth = new SvgLength(2.292f),
    StrokeMiterLimit = 14.3f,
    PathData = pb.ToPathData(),
});

pb.Reset();
pb.AddMoveTo(false, 85.989f, 101.047f);
pb.AddCurveTo(true, 0f, 0f, 3.202f, 3.67f, 8.536f, 4.673f);
pb.AddCurveTo(true, 7.828f, 1.472f, 17.269f, 0.936f, 17.269f, 0.936f);
pb.AddCurveTo(true, 0f, 0f, 2.546f, 5.166f, 10.787f, 7.338f);
pb.AddCurveTo(true, 8.248f, 2.168f, 17.802f, 0.484f, 17.802f, 0.484f);
pb.AddCurveTo(true, 0f, 0f, 8.781f, 1.722f, 19.654f, 8.074f);
pb.AddCurveTo(true, 10.871f, 6.353f, 20.142f, 2.163f, 20.142f, 2.163f);
pb.AddCurveTo(true, 0f, 0f, 1.722f, 3.118f, 14.11f, 9.102f);
pb.AddCurveTo(true, 12.39f, 5.982f, 14.152f, 2.658f, 28.387f, 4.339f);
pb.AddCurveTo(true, 14.232f, 1.672f, 19.36f, 5.568f, 30.108f, 7.449f);
pb.AddCurveTo(true, 10.747f, 1.886f, 25.801f, 5.607f, 25.801f, 5.607f);
pb.AddCurveTo(true, 0f, 0f, 4.925f, 0.409f, 12.313f, 6.967f);
pb.AddCurveTo(true, 7.381f, 6.564f, 18.453f, 4.506f, 18.453f, 4.506f);
pb.AddCurveTo(true, 0f, 0f, -10.869f, -6.352f, -15.467f, -10.702f);
pb.AddCurveTo(true, -4.594f, -4.342f, -16.901f, -11.309f, -24.984f, -15.448f);
pb.AddCurveTo(true, -8.079f, -4.14f, -18.215f, -7.46f, -30.233f, -11.924f);
pb.AddCurveTo(true, -12.018f, -4.468f, -6.934f, -6.029f, -23.632f, -13.855f);
pb.AddCurveTo(true, -16.695f, -7.822f, -13.662f, -8.565f, -28.347f, -10.776f);
pb.AddCurveTo(true, -14.686f, -2.208f, -6.444f, -11.933f, -23.917f, -16.356f);
pb.AddCurveTo(true, -17.479f, -4.423f, -11.037f, -4.382f, -26.016f, -9.093f);
pb.AddCurveTo(true, -14.97f, -4.715f, -10.638f, -10.104f, -26.665f, -13.116f);
pb.AddCurveTo(true, -14.149f, -2.66f, -21.318f, 0.468f, -27.722f, 11.581f);
pb.AddCurveTo(false, 73.104f, 89.075f, 85.989f, 101.047f, 85.989f, 101.047f);
// Add elements into Children collection of SVG
svg.Children.Add(new SvgPathElement()
{
    FillRule = SvgFillRule.EvenOdd,
    Fill = new SvgPaint(Color.FromArgb(0xFF, 0xC2, 0x22)),
    PathData = pb.ToPathData(),
});

pb.Reset();
pb.AddMoveTo(false, 221.771f, 126.738f);
pb.AddCurveTo(true, 0f, 0f, 1.874f, -4.211f, 4.215f, -6.087f);
pb.AddCurveTo(true, 2.347f, -1.868f, 2.812f, -2.339f, 2.812f, -2.339f);
pb.AddMoveTo(false, 147.11f, 105.122f);
pb.AddCurveTo(true, 0f, 0f, 0.882f, -11.047f, 6.765f, -15.793f);
pb.AddCurveTo(true, 5.879f, -4.745f, 10.882f, -5.568f, 10.882f, -5.568f);
pb.AddMoveTo(false, 125.391f, 86.008f);
pb.AddCurveTo(true, 0f, 0f, 2.797f, -6.289f, 6.291f, -9.081f);
pb.AddCurveTo(true, 3.495f, -2.791f, 4.194f, -3.49f, 4.194f, -3.49f);
pb.AddMoveTo(false, 181.153f, 124.8f);
pb.AddCurveTo(true, 0f, 0f, -1.206f, -4.014f, -0.709f, -6.671f);
pb.AddCurveTo(true, 0.493f, -2.66f, 0.539f, -3.256f, 0.539f, -3.256f);
pb.AddMoveTo(false, 111.704f, 107.641f);
pb.AddCurveTo(true, 0f, 0f, -1.935f, -6.604f, -1.076f, -10.991f);
```

```

pb.AddCurveTo(true, 0.862f, -4.389f, 0.942f, -5.376f, 0.942f, -5.376f);
pb.AddMoveTo(false, 85.989f, 101.047f);
pb.AddCurveTo(true, 0f, 0f, 3.202f, 3.67f, 8.536f, 4.673f);
pb.AddCurveTo(true, 7.828f, 1.472f, 17.269f, 0.936f, 17.269f, 0.936f);
pb.AddCurveTo(true, 0f, 0f, 2.546f, 5.166f, 10.787f, 7.338f);
pb.AddCurveTo(true, 8.248f, 2.168f, 17.802f, 0.484f, 17.802f, 0.484f);
pb.AddCurveTo(true, 0f, 0f, 8.781f, 1.722f, 19.654f, 8.074f);
pb.AddCurveTo(true, 10.871f, 6.353f, 20.142f, 2.163f, 20.142f, 2.163f);
pb.AddCurveTo(true, 0f, 0f, 1.722f, 3.118f, 14.11f, 9.102f);
pb.AddCurveTo(true, 12.39f, 5.982f, 14.152f, 2.658f, 28.387f, 4.339f);
pb.AddCurveTo(true, 14.232f, 1.672f, 19.36f, 5.568f, 30.108f, 7.449f);
pb.AddCurveTo(true, 10.747f, 1.886f, 25.801f, 5.607f, 25.801f, 5.607f);
pb.AddCurveTo(true, 0f, 0f, 4.925f, 0.409f, 12.313f, 6.967f);
pb.AddCurveTo(true, 7.381f, 6.564f, 18.453f, 4.506f, 18.453f, 4.506f);
pb.AddCurveTo(true, 0f, 0f, -10.869f, -6.352f, -15.467f, -10.702f);
pb.AddCurveTo(true, -4.594f, -4.342f, -16.901f, -11.309f, -24.984f, -15.448f);
pb.AddCurveTo(true, -8.079f, -4.14f, -18.215f, -7.46f, -30.233f, -11.924f);
pb.AddCurveTo(true, -12.018f, -4.468f, -6.934f, -6.029f, -23.632f, -13.855f);
pb.AddCurveTo(true, -16.695f, -7.822f, -13.662f, -8.565f, -28.347f, -10.776f);
pb.AddCurveTo(true, -14.686f, -2.208f, -6.444f, -11.933f, -23.917f, -16.356f);
pb.AddCurveTo(true, -17.479f, -4.423f, -11.037f, -4.382f, -26.016f, -9.093f);
pb.AddCurveTo(true, -14.97f, -4.715f, -10.638f, -10.104f, -26.665f, -13.116f);
pb.AddCurveTo(true, -14.149f, -2.66f, -21.318f, 0.468f, -27.722f, 11.581f);
pb.AddCurveTo(false, 73.104f, 89.075f, 85.989f, 101.047f, 85.989f, 101.047f);
pb.AddClosePath();

//Add elements into Children collection of SVG
svg.Children.Add(new SvgPathElement()
{
    Fill = SvgPaint.None,
    Stroke = new SvgPaint(Color.Black),
    StrokeWidth = new SvgLength(3.056f),
    StrokeMiterLimit = 11.5f,
    PathData = pb.ToPathData(),
});
//Save the document as svg
doc.Save("demo.svg");
return doc;
}
public static void CreateAndRenderSvgToImage()
{
    int factor = 2;
    using (var bmp = new GcBitmap(320 * factor, 170 * factor, true, 96f * factor, 96f * factor))
    ;
    using (var gr = bmp.CreateGraphics(Color.White))
    {
        gr.DrawSvg(DrawCarrot(), PointF.Empty);
    }

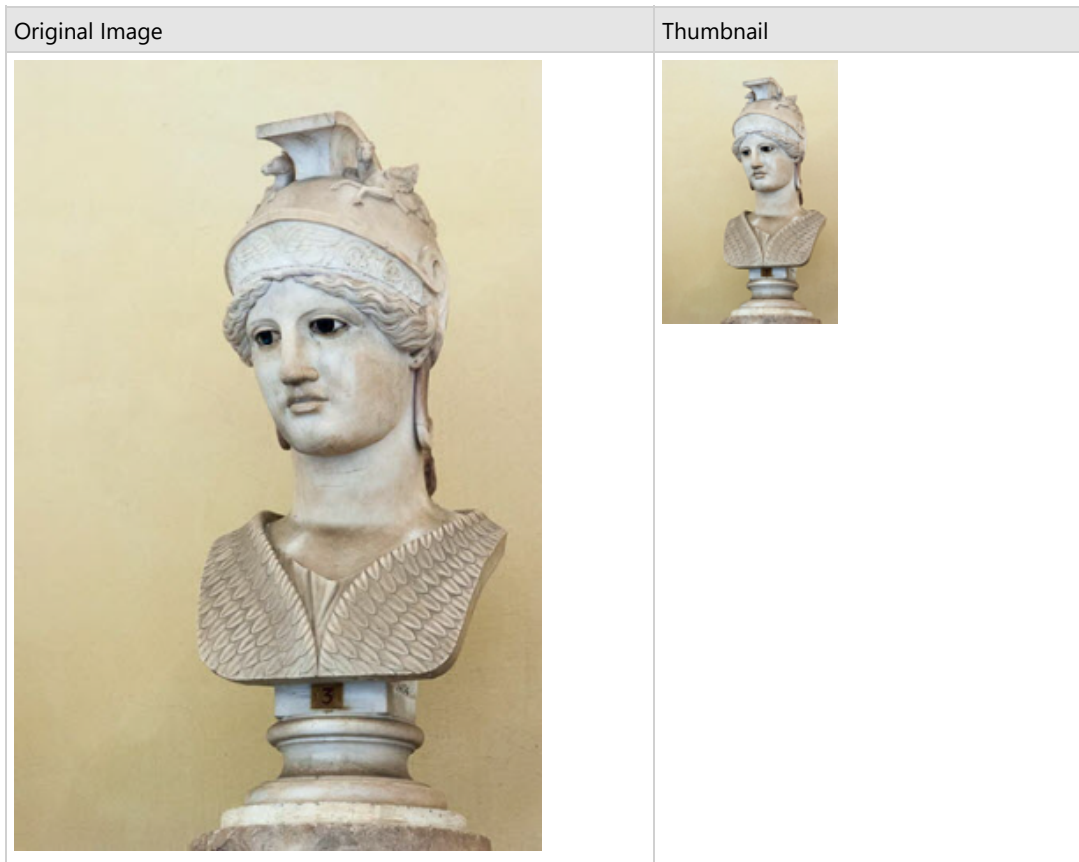
    // Save the SVG as image
    bmp.SaveAsPng("carrot.png");
    Console.WriteLine("CreateAndRenderSvgToImage");
}

```

Create Thumbnail

Dsmaging allows you to create thumbnails of images using **Resize** method of the **GcBitmap** class. The Resize method takes **InterpolationMode** as a parameter to generate the transformed image which is stored as a **GcBitmap** instance. The interpolation

parameter can be set using the **InterpolationMode** enumeration which specifies the algorithm used to scale images. This affects the way an image stretches or shrinks.




To create a thumbnail of an image:

1. Load an image in a GcBitmap instance.
2. Determine the height and width for the thumbnail.
3. Invoke the **Resize** method of GcBitmap class with thumbnail height, width, and interpolation mode as its parameters.

```
C#  
  
public void CreateThumbnail(string origImagePath, string thumbImagePath, int thumbWidth, int  
thumbHeight)  
{  
    using (var origBmp = new GcBitmap(origImagePath, null))  
    using (var thumbBmp = new GcBitmap(thumbWidth, thumbHeight, true))  
    {  
        thumbBmp.Clear(Color.White);  
        float k = Math.Min((float)thumbWidth / origBmp.PixelWidth, (float)thumbHeight /  
origBmp.PixelHeight);  
        var interpolationMode = k < 0.5f ? InterpolationMode.Downscale :  
InterpolationMode.Cubic;  
        int bmpWidth = (int)(k * origBmp.PixelWidth + 0.5f);  
        int bmpHeight = (int)(k * origBmp.PixelHeight + 0.5f);  
        using (var bmp = origBmp.Resize(bmpWidth, bmpHeight, interpolationMode))  
        {  
            thumbBmp.BitBlt(bmp, (thumbWidth - bmpWidth) / 2, thumbHeight - bmpHeight);  
        }  
  
        thumbBmp.SaveAsJpeg(thumbImagePath);  
    }  
}
```

For more information about creating images using Dslmaging, see [Dslmaging sample browser](#).

 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Load Image

DsImaging allows you to load images using **Load** method of the GcBitmap class. You can load images from file, stream, and byte arrays.

Purpose	Method
Load image from a file	Load (string, System.Drawing.Rectangle?)
Load image from a stream	Load (System.IO.Stream, System.Drawing.Rectangle?)
Load image from a byte array	Load (byte[], System.Drawing.Rectangle?)

Load Image from File

To load an image from file, get the image path, store it in a variable and load the file in **GcBitmap** object using the **Load** method with the variable as its parameter.

C#

```
public void LoadSaveFile()
{
    //Get the image path
    var origImagePath = Path.Combine("Resources", "Images",
        "color-woman-postits.jpg");

    //Initialize GcBitmap
    GcBitmap fileBmp = new GcBitmap();

    //Load image from file
    fileBmp.Load(origImagePath);

    //Add title to image
    using (var g = fileBmp.CreateGraphics())
    {
        var rc = new RectangleF(512, 0, 100, 100);

        var tf = new TextFormat
        {
            Font = Font.FromFile(Path.Combine("Resources", "Fonts",
                "times.ttf")),
            FontSize = 40
        };
        g.DrawString("Hello World!", tf, rc, TextAlignment.Center,
            ParagraphAlignment.Center, false);
    }

    //Save image to file
    fileBmp.SaveAsJpeg("color-woman-postits-file.jpg");
}
```

[Back to Top](#)

Load Image from Stream

To load an image from stream, instantiate the **FileStream** class to read the image in the stream and load the file in GcBitmap object using the **Load** method with FileStream object as its parameter.

```
C#
public void LoadSaveStream()
{
    //Get the image path
    var origImagePath = Path.Combine("Resources", "Images",
        "color-woman-postits.jpg");

    //Initialize GcBitmap
    GcBitmap streamBmp = new GcBitmap();

    //Load image from stream
    FileStream stm = new FileStream(origImagePath, FileMode.Open);
    streamBmp.Load(stm);
    stm.Close();

    //Add title to image
    using (var g = streamBmp.CreateGraphics())
    {
        var rc = new RectangleF(512, 0, 100, 100);

        var tf = new TextFormat
        {
            Font = Font.FromFile(Path.Combine("Resources", "Fonts",
                "times.ttf")),
            FontSize = 40
        };
        g.DrawString("Hello World!", tf, rc, TextAlignment.Center,
            ParagraphAlignment.Center, false);
    }

    //Save GcBitmap to stream
    MemoryStream outStream = new MemoryStream();
    streamBmp.SaveAsJpeg(outStream);
}
```

[Back to Top](#)

Load Image from Byte Array

To load an image from byte array, you need to read all the bytes of an image using the **ReadAllBytes** method and load the created byte array in GcBitmap using the **Load** method.

```
C#
public void LoadSaveByteArray()
{
```

```
//Get the image path
var origImagePath = Path.Combine("Resources", "Images",
    "color-woman-postits.jpg");


//Initialize GcBitmap
GcBitmap byteArrayBmp = new GcBitmap();

//Load image from byte array
byte[] imgArray = File.ReadAllBytes(origImagePath);
byteArrayBmp.Load(imgArray);

//Add title to image
using (var g = byteArrayBmp.CreateGraphics())
{
    var rc = new RectangleF(512, 0, 100, 100);

    var tf = new TextFormat
    {
        Font = Font.FromFile(Path.Combine("Resources", "Fonts",
            "times.ttf")),
        FontSize = 40
    };
    g.DrawString("Hello World!", tf, rc, TextAlignment.Center,
        ParagraphAlignment.Center, false);
}

//Save image to file
byteArrayBmp.SaveAsJpeg("color-woman-postits-byteArray.jpg");
}
```

 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Save Image

DsImaging allows you to save images in various formats, such as JPEG, PNG, BMP etc. Each of these formats have a dedicated method as shown below:


Format	Method
JPEG (with specified quality)	SaveAsJpeg
PNG	SaveAsPng
BMP	SaveAsBmp
TIFF	SaveAsTiff
GIF	SaveAsGif
SVG	ToSvgDocument (For more information, see Work with SVG Files)
ICO	Save (For more information, see Work with ICO Files)
WebP	SaveAsWebP (For more information, see Work with WebP Files)

Each of these methods has two overloads, one saves the image in a file and other saves the image in a stream.

C#

```
// Save image
bmp.SaveAsJpeg("color-postits.jpg");

// Save image using stream
bmp.SaveAsJpeg(stream);
```

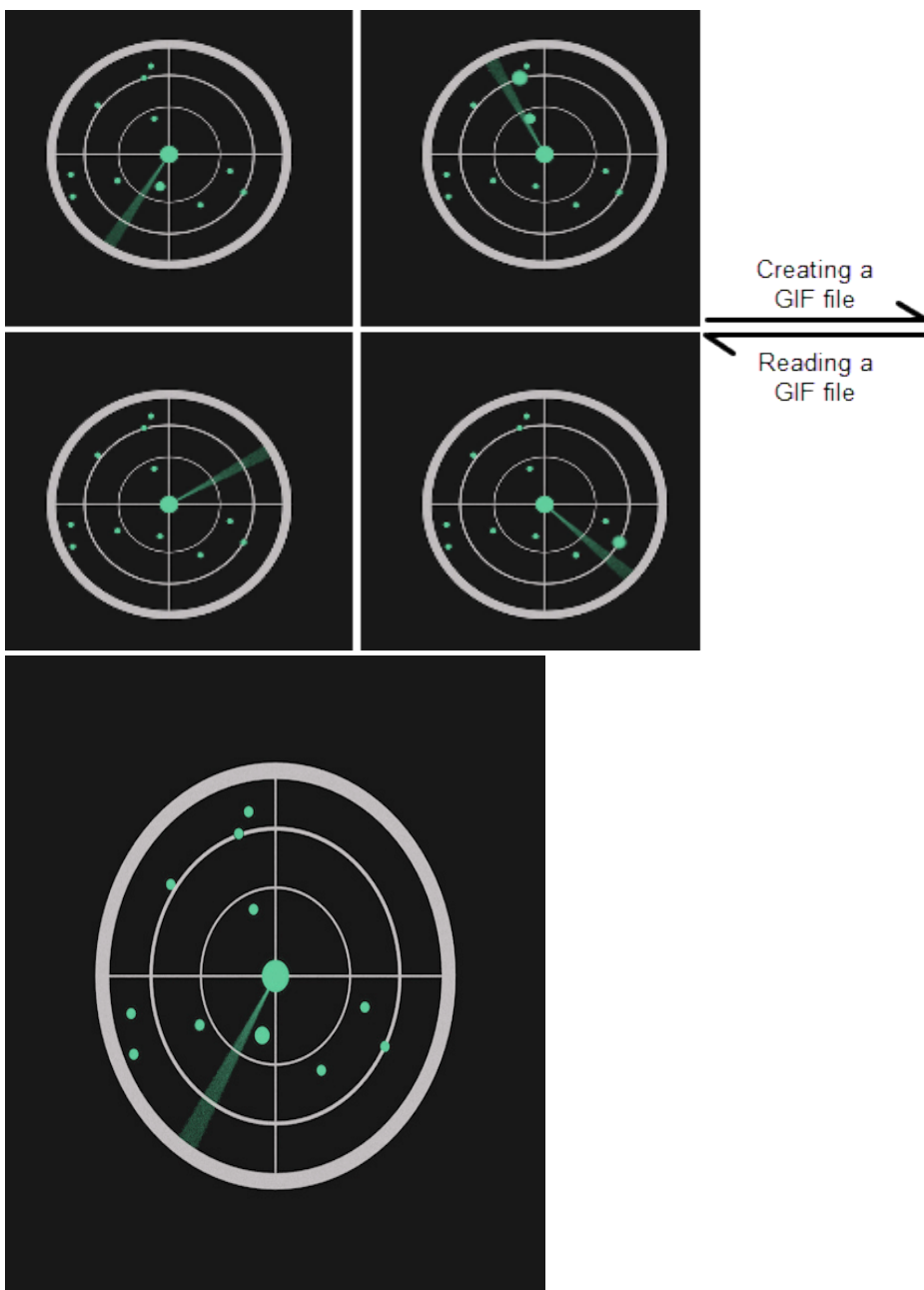
 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Work with GIF files

Graphic Interchange Format (GIF) is a commonly used web image format to create animated graphics. GIF file is created by combining multiple images into a single file. Unlike the JPEG image format, GIF file uses lossless data compression technique to reduce the file size without degrading the visual quality. The image data in a GIF file is stored using indexed color which implies that a standard GIF image can include a maximum of 256 colors.

Apart from reading and creating a GIF file, Dslmaging provides control over various features of GIF files. It allows you to set comments for a GIF file. The comment string can be encoded in various formats supported by Dslmaging. While creating a multiframed GIF file by appending frames, you can use either an indexed image, bitmap, bilevel bitmap or a grayscale image. It also lets you set the number of iterations that should be executed by the animated GIF file.

The below image represents the creation of a GIF file using different frames and the extraction of different frames as images while reading a GIF file.



Reading Frames from a GIF File

DsImaging provides **GcGifReader** class that helps you to read a GIF file and save the frames as separate images. The constructor of this class accepts the GIF file name or stream as a parameter and loads the contents of GIF file. The information about the individual GIF frames is collected in the **Frames** property of the GcGifReader class. While extracting the frames, you can process them in a number of ways, store them in different formats or add them as input frames to GcGifWriter to create a GIF.

To read a multiframe GIF file and save its frames as separate images:

1. Initialize the **GcGifReader** class and pass the GIF file name as a parameter to the constructor.
2. Access the GIF frames from the GIF file using **Frames** property of the GcGifReader class.
3. Load the frame using **ToGcBitmap** method of GcWicBitmap and save it as an image using the **SaveAsJpeg** method of GcBitmap class.

```
C#  
  
//Read frames form the GIF image  
GcGifReader reader = new GcGifReader("Images/radar.gif");  
var frames = reader.Frames;  
  
using (var bmp = new GcBitmap())  
{  
    //Saving GIF frames as individual images  
    for (var i = 0; i < frames.Count; i++)  
    {  
        frames[i].ToGcBitmap(bmp, i - 1);  
        bmp.SaveAsJpeg("Images/Frames/Radar/fr" + (i + 1).ToString() + ".jpg");  
    }  
}
```

[Back to Top](#)

Creating a GIF File

The DsImaging library provides **GcGifWriter** class which helps you to create a GIF file using multiple images. The **AppendFrame** method of **GcGifWriter** class appends an image as a frame to the GIF file. You can invoke this method multiple time to append multiple frames and create a GIF file.

To create a GIF file using multiple images:

1. Initialize the **GcGifWriter** class and pass the GIF file name as a parameter to the constructor.
2. Instantiate **GcBitmap** class to load the images which will serve as frames for the multiframe GIF file.
3. Invoke the **AppendFrame** method of GcGifWriter class to append frames to the GIF file.

```
C#  
  
//Creating GIF image using set of images  
GcGifWriter writer = new GcGifWriter("Images/newradar.gif");  
  
GcBitmap bmp = new GcBitmap();  
bmp.Load("Images/Frames/fr1.jpg");  
writer.AppendFrame(bmp, 255, 0, 0, GifDisposalMethod.DoNotDispose, 20, false);  
  
bmp.Load("Images/Frames/fr2.jpg");  
writer.AppendFrame(bmp, 255, 0, 0, GifDisposalMethod.DoNotDispose, 20, false);
```

```
bmp.Load("Images/Frames/fr3.jpg");  
writer.AppendFrame(bmp, 255, 0, 0, GifDisposalMethod.DoNotDispose, 20, false);  
  
bmp.Load("Images/Frames/fr4.jpg");  
writer.AppendFrame(bmp, 255, 0, 0, GifDisposalMethod.DoNotDispose, 20, false);
```

Back to Top

For more information about working with GIF images using Dslmaging, see [Dslmaging sample browser](#).

Work with TIFF Images

Tagged Image File Format (TIFF) is a widely used file format for storing raster images. A primary goal of TIFF is to provide a rich environment within which applications can exchange image data. TIFF can describe bi-level, grayscale, palette-color, and full-color images with optional transparency and Exif metadata. It supports several compression schemes that allow developers to choose the best space or time tradeoff for their applications. In general, TIFF can store lossless and lossy (JPEG-based) image data. Dslmaging supports only lossless compression for TIFF frames. PNG format usually offers slightly better compression ratio, but it is limited to one image per file. TIFF can store multiple images in the same file. For more info see the [Adobe TIFF specifications](#).

Reading Images from TIFF

Dslmaging provides two main classes that help extracting images from a multi-frame TIFF: **GcTiffReader** and **TiffFrame**. To read an image from a single-frame TIFF, just load the image into a GcBitmap as other supported image formats, like JPEG or BMP. Also, when a TIFF file contains JPEG-based frames, you can use the platform-dependent **GcWicTiffReader** and **WicTiffFrame** classes from GrapeCity.Documents.Imaging.Windows namespace. However, there is no such option available for non-Windows systems.

GcTiffReader accepts a file name or stream as the constructor argument and immediately loads the contents of TIFF without loading the actual image data. The information about TIFF frames is collected in the **Frames** property of the GcTiffReader class. The list contains objects of type **TiffFrame** providing the detailed information about the specific frame, including its size, format, and various metadata. Also, TiffFrame allows to read the frame image into the regular image storing classes of Dslmaging, such as GcBitmap, BilevelBitmap, GrayscaleBitmap, and palette-based bitmaps. These images can be processed in a number of ways, stored in different formats or added as frames to a GcTiffWriter.

To read a multiframe TIFF and save its frames as separate images:

1. Initialize the **GcTiffReader** class and pass the multi frame TIFF as a parameter to the constructor.
2. Access the list of frames from the TIFF image using **Frames** property of the GcTiffReader class.
3. Invoke the **ReadAsGcBitmap** method to get the frame image as GcBitmap object.
4. Save the image to a file in PNG format using **SaveAsPng** method.

```
C#  
  
//Initialize TiffReader class and load the Tiff image  
string tiffFilePath = Path.Combine("Resources", "Images", "Test.tif");  
GcTiffReader tr = new GcTiffReader(tiffFilePath);  
  
string pngName = "FrameImage";  
  
//Save separate images for each Tiff frame  
for (int i = 0; i < tr.Frames.Count; i++)  
{  
    using (var bmp = tr.Frames[i].ReadAsGcBitmap())  
    {  
        bmp.SaveAsPng($"{pngName}_{(i + 1)}.png");  
    }  
}
```

[Back to Top](#)

Creating a Multiframe TIFF

To create a single-frame TIFF, you can use the **GcBitmap.SaveAsTiff()** method which accepts either file path or the output stream as an argument. Now, you can create a multi-frame TIFF by creating an instance of the **GcTiffWriter**

class with a specified file path or stream. Then, you can add various bitmaps to the output TIFF using the **AppendFrame** method of **GcTiffWriter**. Further, you can pass an instance of the **TiffFrameSettings** class to the **GcBitmap.SaveAsTiff()** method as well as to the **AppendFrame()** method. Also, **DefaultFrameSettings** property of the **GcTiffWriter** class allows you to create the common settings for all the frames. For more information on TIFF frame settings, see **TIFF Configuration Options**.

To create a multiframe TIFF by combining four images:

1. Create an instance of the **GcBitmap** class to load the images which will serve as frames for the multiframe TIFF.
2. Initialize the **GcTiffWriter** class by passing the output file name as its parameter.
3. Invoke the **AppendFrame** method of **GcTiffWriter** class for each frame to write frames to the output stream.
4. Optionally, set the compression and orientation of the frame using **Compression** and **Orientation** properties of the **TiffFrameSettings** class through **TiffCompression** and **TiffOrientation** enumerations respectively.

```
C#  
  
string imagePath = Path.Combine("Resources", "Images", "MultiFrameTiff.tif");  
  
//Initialize TiffWriter class to generate multi-frame TIFF  
GcTiffWriter tiffWriter = new GcTiffWriter(imagePath);  
  
//Define Tiff frame settings  
TiffFrameSettings settings = new TiffFrameSettings();  
settings.Compression = TiffCompression.PackBits;  
settings.Orientation = TiffOrientation.TopLeft;  
  
//Initialize GcBitmap to load images for frames  
GcBitmap origbmp = new GcBitmap();  
  
//Load image and append first frame  
imagePath = Path.Combine("Resources", "Images", "TiffFrames", "Img1.png");  
settings.ImageDescription = "Frame1";  
origbmp.Load(imagePath);  
tiffWriter.AppendFrame(origbmp, settings);  
  
//Load image and append second frame  
imagePath = Path.Combine("Resources", "Images", "TiffFrames", "Img2.png");  
origbmp.Load(imagePath);  
settings.ImageDescription = "Frame2";  
tiffWriter.AppendFrame(origbmp, settings);  
  
tiffWriter.Dispose();
```

[Back to Top](#)

TIFF Configuration Options

DslImaging gives full control over the format and settings of an output TIFF frame with the **TiffFrameSettings** class. The frame settings include various metadata, such as the image description, the date of image creation and so on. Also, there are some important properties controlling the compression scheme of the frame image. For the best compression of a full-color image, you can set the **Compression** property to **TiffCompression.Deflate** or **LZW**. The **Differencing** and **Planar** properties also can help in better compression results. In the case of bilevel and grayscale images, the other compression schemes can also fit well. With **GcBitmap** it is possible to shrink the color channels (Red, Green, Blue, Alpha) from 8 bits to some lower value using one of the error-diffusion algorithms (see **GcBitmap.ShrinkARGBFormat** and **GrayscaleBitmap.ShrinkPixelFormat** methods). Then, you can save such an image

as TIFF frame specifying the exact number of bits per channel using the **BitsPer[Color]Channel** or **BitsPerGrayscale** properties of **TiffFrameSettings**. Before doing that please make sure that, just like **GcTiffReader**, your TIFF viewer application supports TIFF frames with variable bits per channel.

Creating Tiled image

Tiled TIFF frames are, generally preferred over stripped frames in case of large images as well as for images where the color areas change more frequently in the horizontal direction than in vertical. For more information, see ["Tiled Images" section in the TIFF specification](#). In DslImaging, you can create tiled images by setting the **TileWidth** and **TileHeight** properties to some positive values. Please note that it might affect the compression ratio.

To create a tiled TIFF image consisting of four frames:

1. Create an instance of the **GcBitmap** class to load the images which will serve as frames for the multiframe TIFF.
2. Initialize the **GcTiffWriter** class by passing the output file name as its parameter.
3. Also, set the tile height and tile width using the **TileHeight** and **TileWidth** properties of the **TiffFrameSettings** class.
4. Invoke the **AppendFrame** method of **GcTiffWriter** class for each frame to write frames to the output stream.

```
C#  
  
string imagePath = Path.Combine("Resources", "Images", "TiledTiff.tif");  
  
//Initialize TiffWriter class to generate multi-frame TIFF  
GcTiffWriter tiffWriter = new GcTiffWriter(imagePath);  
  
//Define Tiff frame settings  
TiffFrameSettings settings = new TiffFrameSettings();  
settings.TileHeight = 200;  
settings.TileWidth = 200;  
  
//Initialize GcBitmap to load images for frames  
GcBitmap origbmp = new GcBitmap();  
  
//Load image and append first frame  
imagePath = Path.Combine("Resources", "Images", "TiffFrames", "Img1.png");  
settings.ImageDescription = "Frame1";  
origbmp.Load(imagePath);  
tiffWriter.AppendFrame(origbmp, settings);  
  
//Load image and append second frame  
imagePath = Path.Combine("Resources", "Images", "TiffFrames", "Img2.png");  
origbmp.Load(imagePath);  
settings.ImageDescription = "Frame2";  
tiffWriter.AppendFrame(origbmp, settings);  
  
tiffWriter.Dispose();
```

Back to Top

For more information about working with TIFF images using DslImaging, see [DslImaging sample browser](#).

Work with ICO files

DslImaging supports ICO file format which is a widely used image file format for computer icons. It stores a collection of small images of different sizes and color sets. The images can be saved in ICO file format by using **GcIco** class. You can work with different frames of an ICO file by using the methods of **IcoFrame** class.

You can also load and save icons in various encodings by using **IcoFrameEncoding** enumeration which sets the encoding of an ICO frame image. For example, a frame can be stored in PNG format or as indexed image with color palette and transparency mask.

Create an ICO File

DslImaging lets you create the frames of an ICO image file from scratch or load from an existing ICO file. These frames can also be converted to GcBitmap or created from existing GcBitmap instances. The whole collection can then be saved to an ICO file. The frames in a multiframe ICO image file can be appended, removed, modified, or reordered.

To create an ICO file from a PNG image:

1. Instantiate **GcBitmap** class and load the PNG file in GcBitmap instance.
2. Initialize **GcIco** class and add the bitmap instance as an ICO frame.
3. Save the ICO image file using **Save** method.

C#

```
//Load a png file
var srcPath = System.IO.Path.Combine("gcd-hex-logo.png");
var srcBmp = new GcBitmap(srcPath);

//Resize the image
var bmp256 = srcBmp.Resize(256, 256);

var ico = new GcIco();
//Add ico file frame
ico.Frames.Add(new IcoFrame(bmp256, IcoFrameEncoding.Png));

//Save ico image file
ico.Save("GcDocs.ico");
```

Read Images from ICO File

You can load the image data in ICO format from a file, stream, or an array of bytes. It can then be saved to a stream or file. The **GcIco** class must be disposed off after use, to prevent memory loss in image frames. Also, dispose off any removed frames from the collection.

To read a multiframe ICO file and save its frames as separate PNG images:

1. Load an ICO file by instantiating the **GcIco** class.
2. Convert the ICO frames to GcBitmap and save them as separate PNG files.

C#

```
//Load an ico file
using (var ico = new GcIco("Windows.ico"))
{
```

```
for (int i = 0; i < ico.Frames.Count; i++)
{
    //Save png file for every ico frame
    using (var bmp = ico.Frames[i].ToGcBitmap())
    {
        bmp.SaveAsPng($"image{i}.png");
    }
}
}
```

Limitations

- Some rare frame encodings, such as indexed images with 2 bits per pixel, are not supported.
- Bitmap compressions other than BI_RGB, are not supported.
- CUR format has limited support (Gclco does not distinguish it from ICO and cannot save images in CUR format).

Work with SVG Files

DslImaging supports SVG (Scalable Vector Graphics) image file format which allows you to render vector images at any size without loss of quality.

The **GcSvgDocument** class is provided in the GrapeCity.Documents.Svg namespace in the DslImaging library. This class allows you to create, load, inspect, modify and save the internal structure of an SVG image.

SVG graphics can be loaded from files or strings into the object model of **GcSvgDocument** class, further drawn to the **GcGraphics** class to effectively output the result to GcPdfDocument, GcBitmap or GcWicBitmap classes. SVG documents can be drawn to objects derived from GcGraphics, such as GcPdfGraphics or GcBitmapGraphics using the **GcGraphics.DrawSvg** method overloads.

Render SVG to PNG

To render an SVG image to a PNG image and output the result to GcBitmap class:

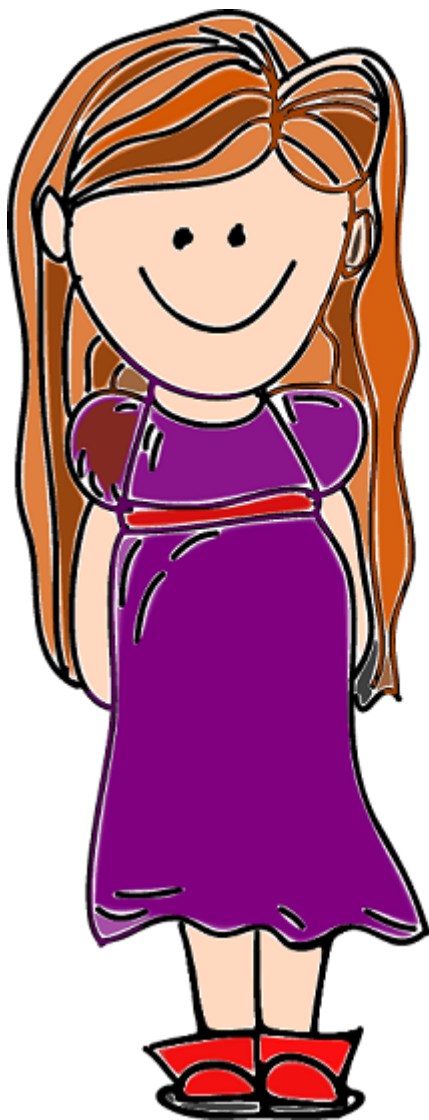
1. Load an SVG image by using the **FromFile** method of **GcSvgDocument** class.
2. Initialize the GcBitmap class and create a drawing surface using CreateGraphics method of the GcBitmap class.
3. Draw the specified SVG document at a location in PDF document by using **DrawSvg** method of **GcGraphics** class.
4. Save the image to a file in PNG format using **SaveAsPng** method.

```
C#  
  
using var svg = GcSvgDocument.FromFile("Smiling-Girl.svg");  
var rect = svg.Measure(PointF.Empty);  
float factor = 1.5f;  
using var bmp = new GcBitmap((int)(rect.Width * factor + 0.95f), (int)  
(rect.Height * factor + 0.95f), true, 96f * factor, 96f * factor);  
using (var g = bmp.CreateGraphics(Color.White))  
{  
    g.DrawSvg(svg, new PointF(-rect.X, -rect.Y));  
}  
bmp.SaveAsPng("Smiling-Girl.png");
```

Similarly as above, use the following code to render an SVG image to a PNG image and output the result to GcWicBitmap class:

```
C#  
  
using var svg = GcSvgDocument.FromFile("Smiling-Girl.svg");  
var rect = svg.Measure(PointF.Empty);  
float factor = 1.5f;  
using var bmp = new GcWicBitmap((int)(rect.Width * factor + 0.95f), (int)(rect.Height  
* factor + 0.95f), true, 96f * factor, 96f * factor);  
using (var g = bmp.CreateGraphics(Color.White))  
{  
    g.DrawSvg(svg, new PointF(-rect.X, -rect.Y));  
}  
bmp.SaveAsPng("Smiling-Girl.png");
```

The output of above code snippets will look like below:



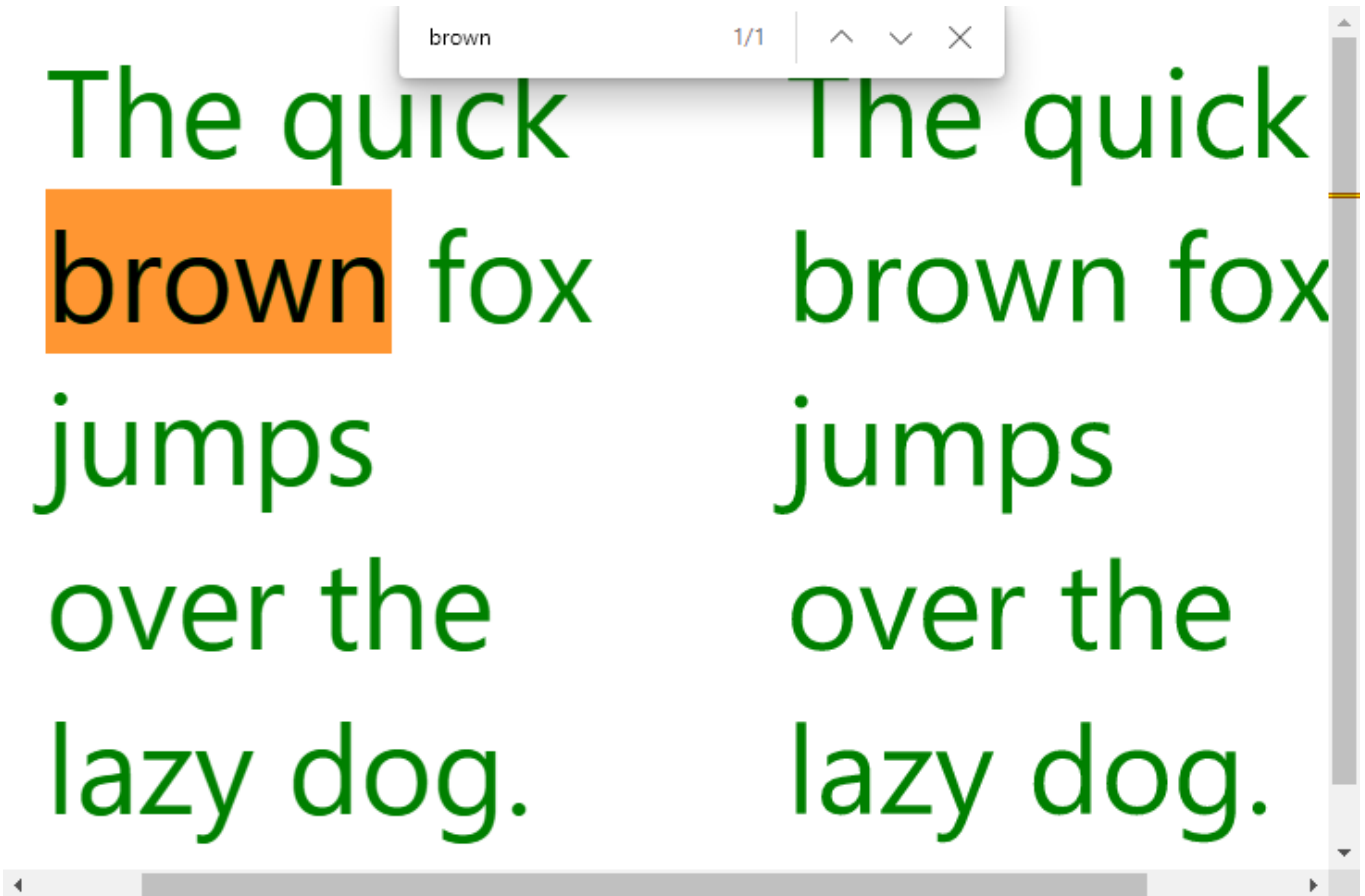
You can also render an SVG image to a PDF document. For more information, refer [Images](#) topic in DsPdf docs.

Render Graphics and Text to SVG

DsImaging lets you render graphics and text on a `GcSvgDocument` by using `ToSvgDocument` method of the `GcSvgGraphics` class. This class is derived from the `GcGraphics` class.

By default, the method saves text as paths in an SVG document. However, by setting `GcGraphics.DrawTextAsPath` property to **false**, you can save the text using the standard SVG text elements. While drawing strings or `TextLayout` objects using paths ensures that the resulting SVG image looks as expected, rendering them as text elements lets you select, copy or even search text fragments. However, the text layout depends on the specific fonts. Hence, DsImaging lets you embed fonts to the output SVG file by setting the boolean property `GcSvgGraphics.EmbedFonts` to **true**. The property is especially useful in rendering text with rare fonts or fonts unavailable on the client machine.

DsImaging also supports specifying the positions of each individual character within the text element by setting `PreciseCharPositions` property of the `GcSvgGraphics` class.



The code below shows how you can render the same text as path elements and text elements.


```
C#  
  
var g = new GcSvgGraphics(900, 500);  
var tl = g.CreateTextLayout();  
  
var fmt = new TextFormat()  
{  
    FontName = "Segoe UI",  
    FontSize = 50,  
    ForeColor = Color.Green  
};  
tl.MaxWidth = 300;  
tl.Append("The quick brown fox jumps over the lazy dog.", fmt);  
  
// the text at the left is drawn with the text elements  
g.DrawTextAsPath = false;  
g.DrawTextLayout(tl, new PointF(100f, 10f));  
  
// the text at the right is drawn with paths  
g.DrawTextAsPath = true;  
g.DrawTextLayout(tl, new PointF(500f, 10f));  
  
var svg = g.ToSvgDocument();
```



```
svg.Save("BrownFox.svg", new XmlWriterSettings() { Indent = true });
```

Limitation

- All text effects of the TextLayout class are not supported when using the SVG text elements. Always check the output SVG to make sure that the text elements are rendered correctly.
- Vertical text is always drawn using path elements.
- In GcSvgDocument class, SVG files having embedded fonts are rendered without embedded fonts.

 **Note:** You need to apply a license key to use the **ToSvgDocument()** method. Without a license key, only a few calls of ToSvgDocument() are allowed, after which an exception is thrown. For more information about applying license, see [Apply License](#).

Render PDF Page as SVG

DsImaging lets you save a PDF page or an instance of GcPdfDocument as SVG format using the **SaveAsImageOptions** class. The class is used for passing options to the methods used for saving a PDF page in the SVG format. By default, the class renders strings and TextLayoutObjects as path elements. However, to handle the text related operations such as select, copy, search etc, you can also render SVG with the text as text elements by setting the **DrawTextAsPath** property to **false**. To handle fonts while working with text, the SaveAsImageOptions class provides **EmbedSvgFonts** property to embed font subsets to the output SVG file. The property is set to false by default which means fonts are not embedded. You can set this property to true to cater to some rare fonts or fonts that might not be available on the client machine.

The SVG format has the ability to specify positions of each individual characters within the text element. This mode can be enabled using the **PreciseCharPositions** properties of the SaveAsImageOptions classes. The PreciseCharPositions property is set to **true** by default because the fonts embedded in PDF documents do not often contain the positioning tables. In some cases, setting character positions also helps when the proposed font is not available on the client machine and the SVG text element is rendered using a fallback font.

The code below shows how you can render a PDF page as SVG using path elements and using text elements.

C#

```
var pdfDoc = new GcPdfDocument();
    using (var fs = new FileStream(@"DOC_2317_MS.pdf", FileMode.Open,
FileAccess.Read, FileShare.Read))
    {
        pdfDoc.Load(fs);
        var page = pdfDoc.Pages[0];

        // save the SVG with text elements and embedded fonts
        page.SaveAsSvg("DOC_2317_MS_1.svg", null,
            options: new SaveAsImageOptions() { Zoom = 2f, DrawSvgTextAsPath =
false, EmbedSvgFonts = true },
            new XmlWriterSettings() { Indent = true });

        // save the SVG with all text drawn as paths
        page.SaveAsSvg("DOC_2317_MS_2.svg", null,
            new SaveAsImageOptions() { Zoom = 2f },
            new XmlWriterSettings() { Indent = true });
    }
```

Limitations:

- Vertical text is always drawn using path elements.
- A PDF page saved as an SVG file renders all text fragments drawn with CFF, Type1, and incomplete OpenType fonts as paths.
- In GcSvgDocument class, SVG files having embedded fonts are rendered without embedded fonts.

Save SVG to File or Stream

DslImaging lets you save a newly created SVG document or a modified document as a file or a stream. You can use the **GcSvgDocument.Save** method to serialize the new or modified SVG document to a file or a stream.

C#

```
using var svg = GcSvgDocument.FromFile("cerdito.svg");

var paint = new SvgPaint(Color.Bisque);
foreach (var elem in svg.GetElementsByClass("rose"))
{
    elem.Fill = paint;
}

svg.Save("cerdito2.svg", new XmlWriterSettings() { Indent = true });

File.WriteAllBytes("cerdito2.svgz", svg.ToSvgz());
```

Limitations

- The supported elements in SVG files are svg, g, defs, style, use, symbol, image, path, circle, ellipse, line, polygon, polyline, rect, clipPath, marker, pattern, radialGradient, linearGradient, stop, title, metadata and desc. When rendering SVG content that contains an unsupported element or attribute, the unsupported entity is ignored. The remainder of the content is rendered as faithfully as possible.
- The image element is only supported if its href attribute is set to a base64-encoded image. File and remote references are not supported.

Work with WebP Files

WebP is a modern and widespread image file format to showcase high-quality images without affecting website performance. This format is supported by most of the web browsers.

In DslImaging, you can load WebP images using **Load** method of the **GcBitmap** class wherein you can load images from file, stream and byte arrays. You can also load an image by using constructor of the GcBitmap class. For details about loading images, see [Load Image](#).

To save an image to the WebP format, you can use **SaveAsWebp** method of the **GcBitmap** class.

```
C#  
  
// Converting a JPG image to WEBP format  
using var bmp = new GcBitmap();  
bmp.Load("image.jpg");  
bmp.SaveAsWebp("image.webp", null, false, 50);
```

Limitations

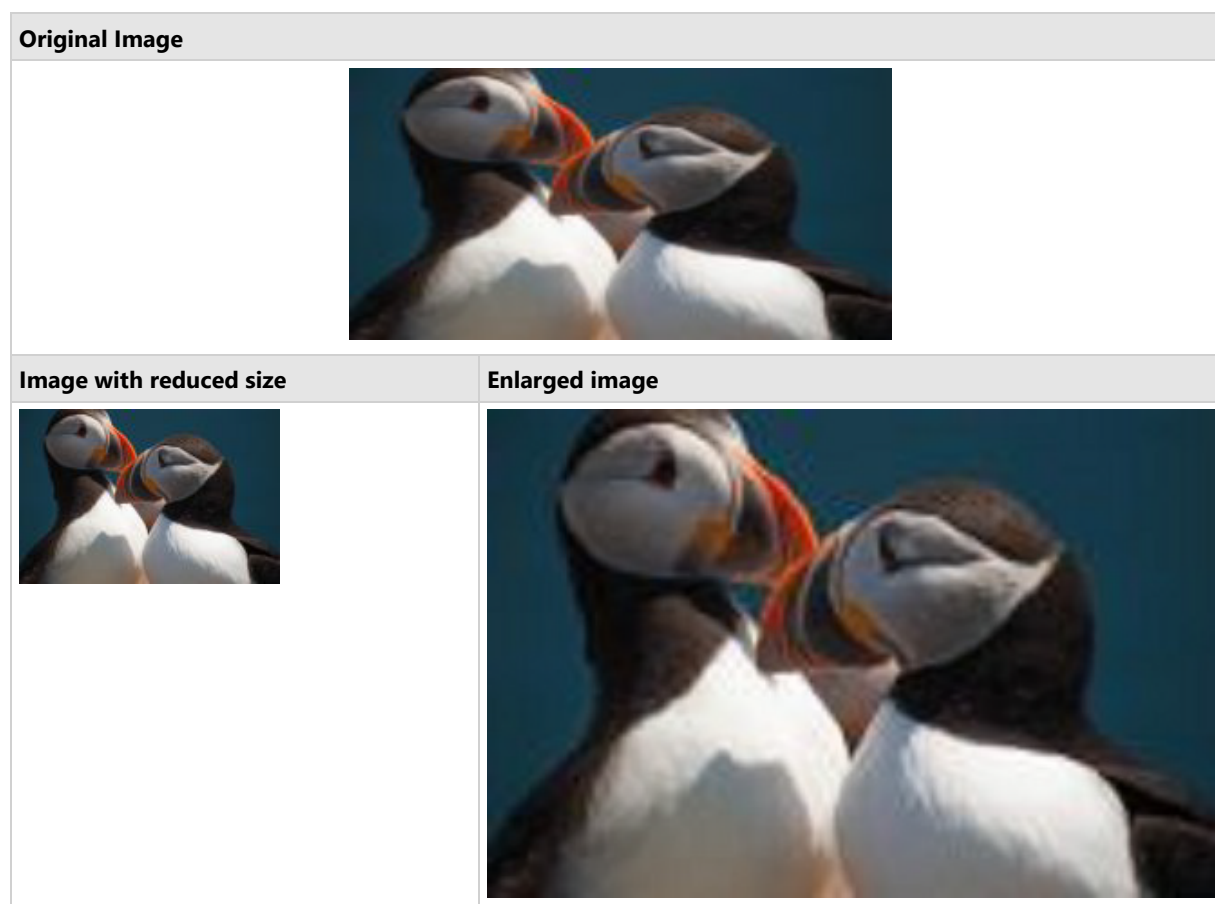
- Saving an image with transparency in lossy WebP format may result in a relatively large image file.
- Saving an image to lossless WebP format using high quality encoding may result in slow performance.

Process Image

DsImaging allows you to process images in different ways, such as alter the image size, crop, rotate, flip image, and change image resolution. It provides various properties and methods, such as `Resize`, `FlipRotate`, etc. in the **GcBitmap** class to handle such type of processing.

Resize Image

DsImaging lets you reduce or enlarge an image using **Resize** method of the **GcBitmap** class. The **Resize** method takes **InterpolationMode** as a parameter to generate the transformed image which is stored as a `GcBitmap` instance. The interpolation parameter can be set using the **InterpolationMode** enumeration which specifies the algorithm used to scale images.



To resize an image:

1. Initialize the **GcBitmap** class.
2. Load an image in the `GcBitmap` instance.
3. Calculate the new height and width of the image for scaling the image.
4. Invoke the **Resize** method of `GcBitmap` class with new height, width, and interpolation mode as its parameters.

C#

```
//Get the image path
var origSmallImagePath = Path.Combine("Resources", "Images",
    "puffins-small.jpg");

//Initialize GcBitmap
GcBitmap origLargeBmp = new GcBitmap();
GcBitmap origSmallBmp = new GcBitmap();

//Load image from file
```

```
origLargeBmp.Load(origSmallImagePath);
origSmallBmp.Load(origSmallImagePath);

//Reduce image
int rwidth = origLargeBmp.PixelWidth - 564;
int rheight = origLargeBmp.PixelHeight - 376;
GcBitmap smallBmp = origLargeBmp.Resize(rwidth, rheight,
    InterpolationMode.Linear);

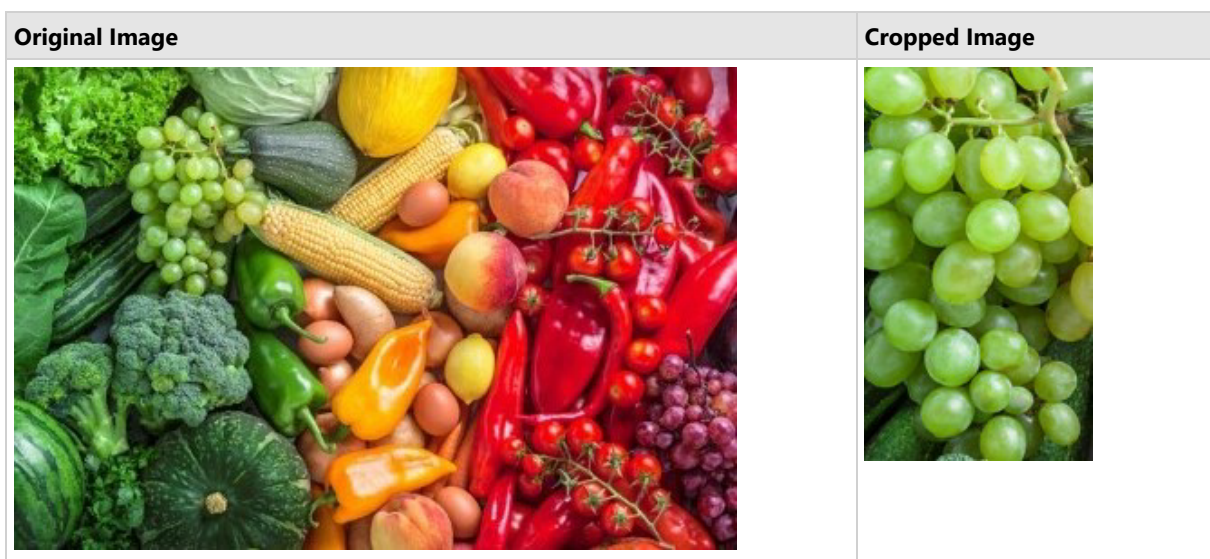
//Enlarge image
int ewidth = origSmallBmp.PixelWidth + 156;
int eheight = origSmallBmp.PixelHeight + 54;
GcBitmap largeBmp = origSmallBmp.Resize(ewidth, eheight,
    InterpolationMode.Linear);

//Save scaled image to file
smallBmp.SaveAsJpeg("puffins-scale-small.jpg");
largeBmp.SaveAsJpeg("puffins-scale-large.jpg");
```

[Back to Top](#)

Crop Image

Image cropping is usually done to remove the extraneous part of an image in order to improve its framing, to change the aspect ratio and to isolate a particular object from its background. Dslmaging allows you to crop an image using **Clip** method of the GcBitmap class. This method creates new GcBitmap instance that stores the cropped fragment of the original image.



To crop an image:

1. Load an image in the GcBitmap instance.
2. Define a rectangle with specified location and size which is to be cropped.
3. Invoke the **Clip** method of GcBitmap class while specifying the rectangle to separate the required image fragment from the original image.

```
C#
//Get the image path
var origImagePath = Path.Combine("Resources", "Images",
    "color-vegetables.jpg");

//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap();
```

```
//Load image from file
origBmp.Load(origImagePath);

//Crop image
Rectangle clipRec = new Rectangle(661, 327, 508, 878);
GcBitmap clipbmp = origBmp.Clip(clipRec);

//Save cropped image to file
clipbmp.SaveAsJpeg("color-vegetables-crop.jpg");
```

[Back to Top](#)

Rotate and Flip Image

An image can be rotated at different angles and flipped to create its mirror image. DsImaging supports both rotation and flipping of an image through **FlipRotate** method of the GcBitmap class. This method accepts a parameter of type **FlipRotateAction** enumeration which specifies flip and rotation transformations. Using **FlipRotateAction** enumeration, an image can be rotated clockwise at 90, 180, or 270 degrees and flipped horizontally or vertically. The enumeration also provides an option to flip an image horizontally with a clockwise rotation of 90 or 270 degrees.

Original Image



Rotated Image

Flipped Image



To rotate an image clockwise at 90 degree:

1. Load an image in a GcBitmap instance.
2. Call the **FlipRotate** method of GcBitmap class while specifying the FlipRotateAction to produce an image rotated clockwise at 90 degrees.

```
C#  
  
//Get the image path  
var origImagePath = Path.Combine("Resources", "Images",  
                                "color-vegetables.jpg");  
  
//Initialize GcBitmap  
GcBitmap origBmp = new GcBitmap();  
  
//Load image from file  
origBmp.Load(origImagePath);  
  
//Rotate image by 90 degree  
GcBitmap rotatebmp = origBmp.FlipRotate(FlipRotateAction.Rotate90);  
  
//Save rotated image to file  
rotatebmp.SaveAsJpeg("color-vegetables-rotate.jpg");
```

To flip an image horizontally:

1. Load an image in a GcBitmap instance.
2. Call the **FlipRotate** method of GcBitmap class while specifying the FlipRotateAction to flip the pixels around the vertical y-axis which produces a mirror image.

```
C#  
  
//Get the image path  
var origImagePath = Path.Combine("Resources", "Images",  
                                "color-vegetables.jpg");  
  
//Initialize GcBitmap
```

```
GcBitmap origBmp = new GcBitmap();

//Load image from file
origBmp.Load(origImagePath);

//Flip image horizontally
GcBitmap flipbmp = origBmp.FlipRotate(FlipRotateAction.FlipHorizontal);

//Save image to file
flipbmp.SaveAsJpeg("color-vegetables-flip.jpg");
```

[Back to Top](#)

Clear Image

In Dslmaging, you can remove text and graphics from GcBitmap using **Clear** method of the GcBitmap class. It leaves a specified color on the surface.

```
C#

//Initialize GcBitmap with the expected height/width
var origBmp = new GcBitmap(pixelWidth, pixelHeight, true, dpiX, dpiY);

//Clear image
origBmp.Clear(Color.LightBlue);

//Save image to file
origBmp.SaveAsJpeg("color-vegetables-clear.jpg");
```

[Back to Top](#)

Change Resolution

Resolution of an image refers to the measurement of its output quality. Dslmaging allows you to change the resolution of an image using **SetDpi** method of the GcBitmap class. The **SetDpi** method has following two overloads, **SetDpi(float dpi)** and **SetDpi(float dpiX, float dpiY)**. The **SetDpi(float dpi)** method allows you to change the physical resolution of an image by accepting a single value for the horizontal and vertical resolution. On the other hand, the **SetDpi(float dpiX, float dpiY)** method lets you change the physical resolution of an image by accepting separate values for the horizontal and the vertical resolution.

Additionally, GcBitmap class provides two properties, namely DpiX and DpiY, using which you can fetch the horizontal and vertical resolution of the bitmap, respectively.

To change the resolution of an image:

1. Load an image from file in the GcBitmap instance.
2. Invoke the **SetDpi** method of GcBitmap class which accepts the new horizontal and vertical resolution as its parameters.

```
C#

//Get the image path
var origImagePath = Path.Combine("Resources", "Images",
    "color-vegetables.jpg");

//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap();

//Load image from file
origBmp.Load(origImagePath);

//Change image resolution
int newDpiX = 200, newDpiY = 400;
origBmp.SetDpi(newDpiX, newDpiY);
```

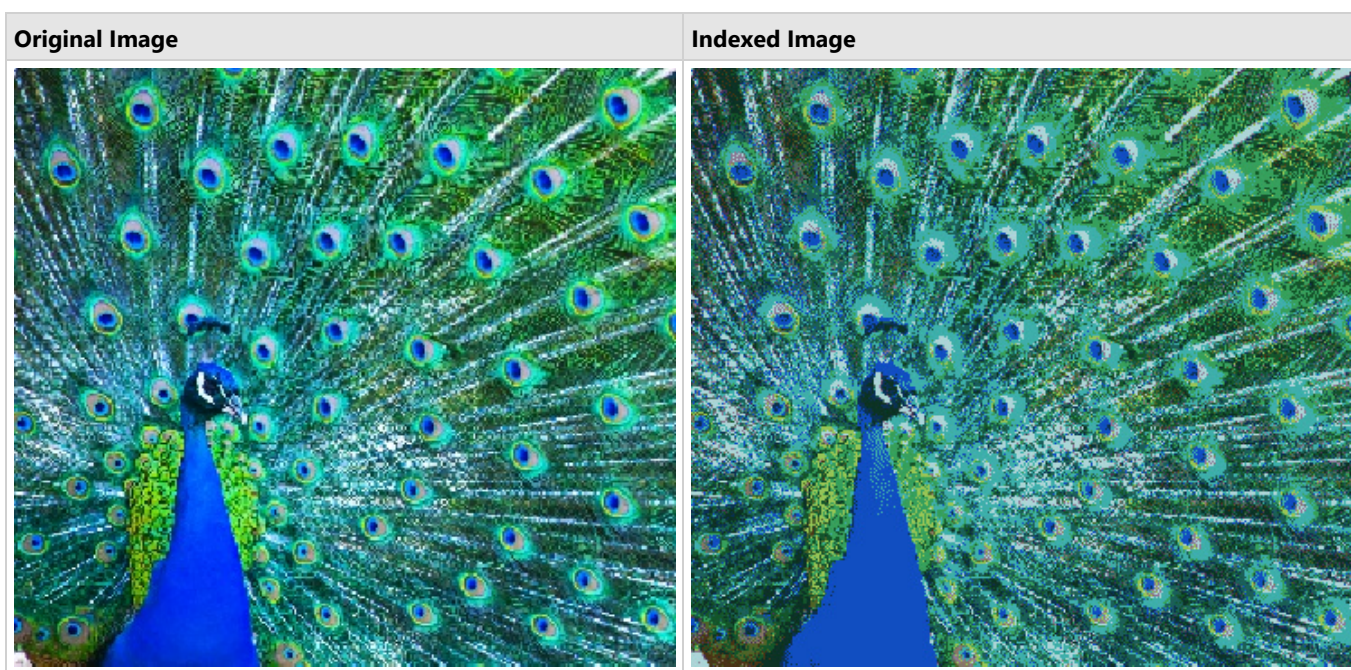


```
//Save image to file
origBmp.SaveAsJpeg("color-vegetables-resolution.jpg");
```

Back to Top

Convert to Indexed Image

DsImaging supports high quality ARGB images. However, such high quality images take more memory than the indexed images. Hence, you can convert the ARGB images to indexed images to store them compactly. DsImaging provides two methods to convert ARGB images to indexed images, which are **ToIndexed4bppBitmap** and **ToIndexed8bppBitmap** of the GcBitmap class. The **ToIndexed4bppBitmap** method converts an image to 4 bpp (bits per pixel) indexed image which returns an instance of the **Indexed4bppBitmap** class. Similarly, **ToIndexed8bppBitmap** method converts an image to 8 bpp indexed image which returns an instance of **Indexed8bppBitmap** class. The ToIndexed4bppBitmap and ToIndexed8bppBitmap methods can take any custom palette as a parameter while converting an image to the indexed image.



To convert an image to a 4bpp indexed image using the octree color palette based on the Octree color quantization algorithm:

1. Load an image in the GcBitmap instance.
2. Generate the Octree color palette by using **GenerateOctreePalette** method of GcBitmap class.
3. Convert the image to 4 bpp using **ToIndexed4bppBitmap** method of GcBitmap class and pass the octree color palette as its parameter.
4. Save the indexed image using the **SaveAsJpeg** method.

```
C#
//Load an image to generate a custom palette
GcBitmap bmpSrc = new GcBitmap();
bmpSrc.Load("Images/peacock_small.jpg");

//Generate color palette using Octree quantizer and dithering
var pal = bmpSrc.GenerateOctreePalette(16);

//Use octree palette generated above as a custom palette to create an Indexed image
Indexed4bppBitmap ind = bmpSrc.ToIndexed4bppBitmap(pal, DitheringMethod.FloydSteinberg);
ind.ToGcBitmap().SaveAsJpeg("Images/IndexedPeacockpal1.jpg");
```

Back to Top

Combine Images

DsImaging allows you to combine multiple images with different formats to generate a new image. You can combine multiple images and place them on one GcBitmap using **BitBlt** method of the GcBitmap class. The BitBlt method performs a bit-block transfer of the color data corresponding to pixels from the specified source bitmap into the current bitmap.

To combine multiple images, say four images, with different formats into a new image:

1. Create GcBitmap instances for each image.
2. Load an image in each GcBitmap instance.
3. Initialize a new GcBitmap instance with specified width and height, in pixel, to combine all the four images into one.
4. Place all the images one by one with specified coordinates on this GcBitmap by performing bit-block transfer using **BitBlt** method of the GcBitmap class.

```
C#
//Get the images paths
var jpgImagePath = Path.Combine("Resources", "Images",
                                "gray-puffins-small.jpg");
var pngImagePath = Path.Combine("Resources", "Images",
                                "gray-dog-small.png");
var bmpImagePath = Path.Combine("Resources", "Images",
                                "color-goldfish-small.bmp");
var gifImagePath = Path.Combine("Resources", "Images",
                                "peacock-small.gif");

//Initialize GcBitmap instances and load an image in each instance
GcBitmap jpgBmp = new GcBitmap();
jpgBmp.Load(jpgImagePath);
jpgBmp.Opaque = true;

GcBitmap pngBmp = new GcBitmap();
pngBmp.Load(pngImagePath);
pngBmp.Opaque = true;

GcBitmap bmpBmp = new GcBitmap();
bmpBmp.Load(bmpImagePath);
bmpBmp.Opaque = true;

GcBitmap gifBmp = new GcBitmap();
gifBmp.Load(gifImagePath);
gifBmp.Opaque = true;

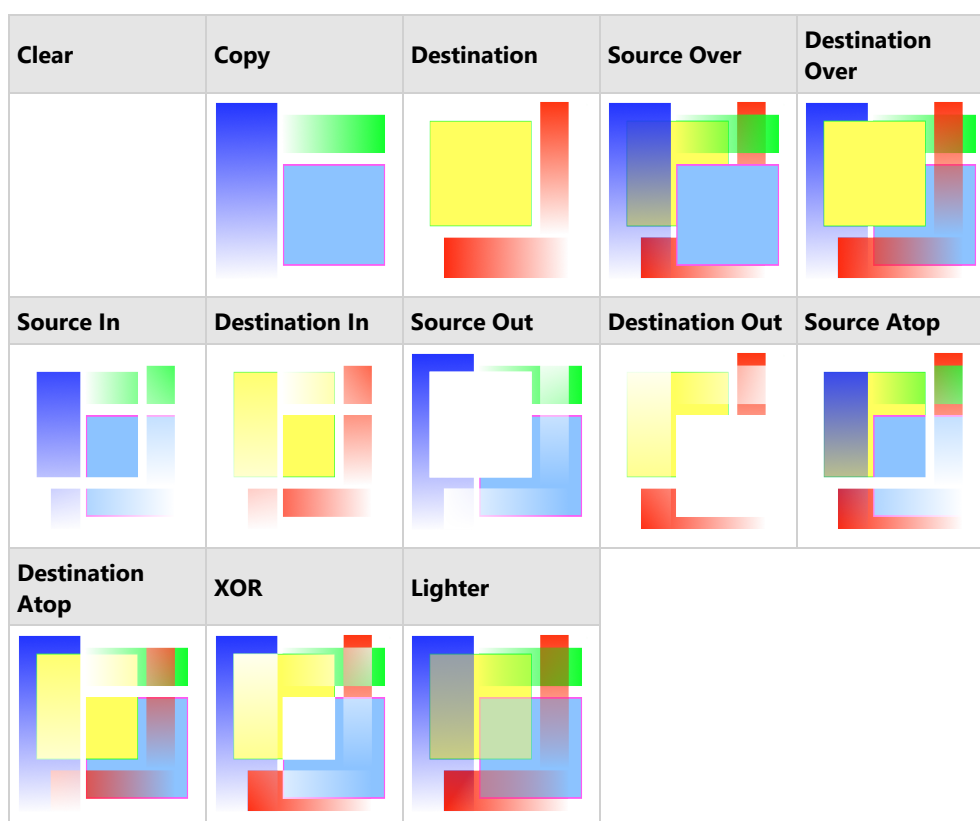
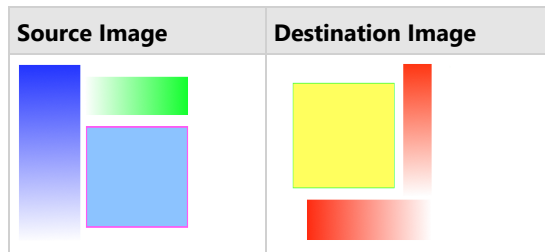
//Concatenate the images with different formats to
//generate a new image
int w = jpgBmp.PixelWidth + 1;
int h = jpgBmp.PixelHeight + 1;
GcBitmap outBmp = new GcBitmap(w * 2, h * 2, true);
outBmp.BitBlt(jpgBmp, 0, 0);
outBmp.BitBlt(pngBmp, w, 0);
outBmp.BitBlt(bmpBmp, 0, h);
outBmp.BitBlt(gifBmp, w, h);

//Save concatenated image to file
outBmp.SaveAsJpeg("color-concatenate.jpg");
```

[Back to Top](#)

Compositing Images

Compositing defines various ways in which two bitmaps can be combined into a single image. Dslmaging allows you to composite images using Porter-Duff compositing algorithm by providing **CompositeAndBlend** method in the GcBitmap class. The method takes values from **CompositeMode** enumeration as a parameter to generate the resultant image by compositing the source and destination bitmap. There are 13 composite modes which can be implemented through the **CompositeMode** enumeration as displayed below:



To perform Porter-Duff compositing on two bitmaps using DestinationOver composite mode :

1. Create GcBitmap instances to load the source and destination images.
2. Invoke the **CompositeAndBlend** method of GcBitmap class, and pass the **DestinationOver** composite mode as the parameter to combine the two images.

```
C#
//Load the two images to be combined
using (var dst = new GcBitmap(@"in\dst.png"))
using (var src = new GcBitmap(@"in/src.png"))
//Combine the two images using various compositing and blending modes
{
    var tmp = dst.Clone();
    tmp.CompositeAndBlend(src, 0, 0, CompositeMode.DestinationOver);
    tmp.SaveAsPng(@"out\res_DestinationOver.png");
}
```

```
}
```

[Back to Top](#)

Blend Modes

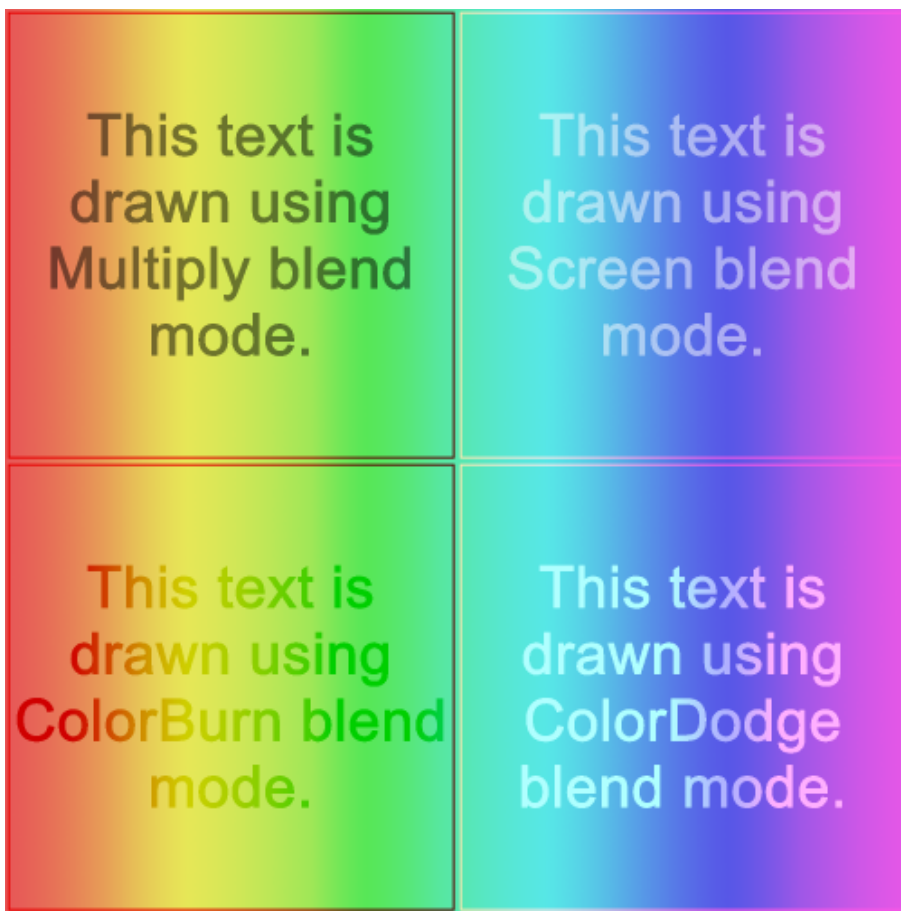
Blend mode determines how the colors of the target image and the colors of graphic primitives or images that are drawn on the target are mixed (blended) with each other. The **BlendMode** enumeration is used to specify the blend mode. In DsImaging, the blend mode can be specified in two ways:

- By setting the **BlendMode** property on the current instance of the **GcBitmapGraphics** class or directly on the **GcBitmap.Renderer** (the two properties are associated with the same value internally). In this case the specified blend mode will affect all subsequent drawing on the bitmap until changed to a different value.
- By specifying a blend mode value as a parameter of the **CompositeAndBlend** method of GcBitmap. In this case the specified blend mode will only apply to the current method call. This approach is preferable if you only need to overlay two images, and also provides other useful options.

The following example shows how the BlendMode property can be used to affect all drawing on a GcBitmapGraphics:

```
C#  
  
// Use the spectrum image as the background to draw on:  
using var bmp = new GcBitmap("spectrum-pastel-500x500.png");  
using var g = bmp.CreateGraphics();  
  
// Draw text on the spectrum background using a few blend modes:  
var rc = new RectangleF(0, 0, bmp.PixelWidth / 2, bmp.PixelHeight / 2);  
var tf = new TextFormat() { FontSize = 24, FontBold = true, ForeColor = Color.Gray };  
var modes = new BlendMode[]  
    { BlendMode.Multiply, BlendMode.Screen, BlendMode.ColorBurn, BlendMode.ColorDodge };  
var pts = new PointF[]  
    { new PointF(0, 0), new PointF(250, 0), new PointF(-250, 250), new PointF(250, 0) };  
int i = 0;  
foreach (var mode in modes)  
{  
    g.BlendMode = mode;  
    rc.Offset(pts[i++]);  
    g.DrawString($"This text is drawn using {g.BlendMode} blend mode.",  
        tf, rc, TextAlignment.Center, ParagraphAlignment.Center);  
    var rcb = rc;  
    rcb.Inflate(-2, -2);  
    g.DrawRectangle(rcb, Color.Red);  
}  
bmp.SaveAsPng("blend-modes.png");  
}
```

The output of the above code will look like below:

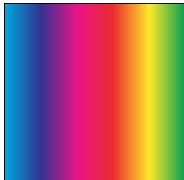


To use the **CompositeAndBlend** method, you need to create two instances of **GcBitmap**. One will be the target of the operation containing the backdrop on which to draw. The second bitmap (source) should contain the image that will be blended with the target. You will also need to also specify the **CompositeMode** and other parameters. The following code shows an example:

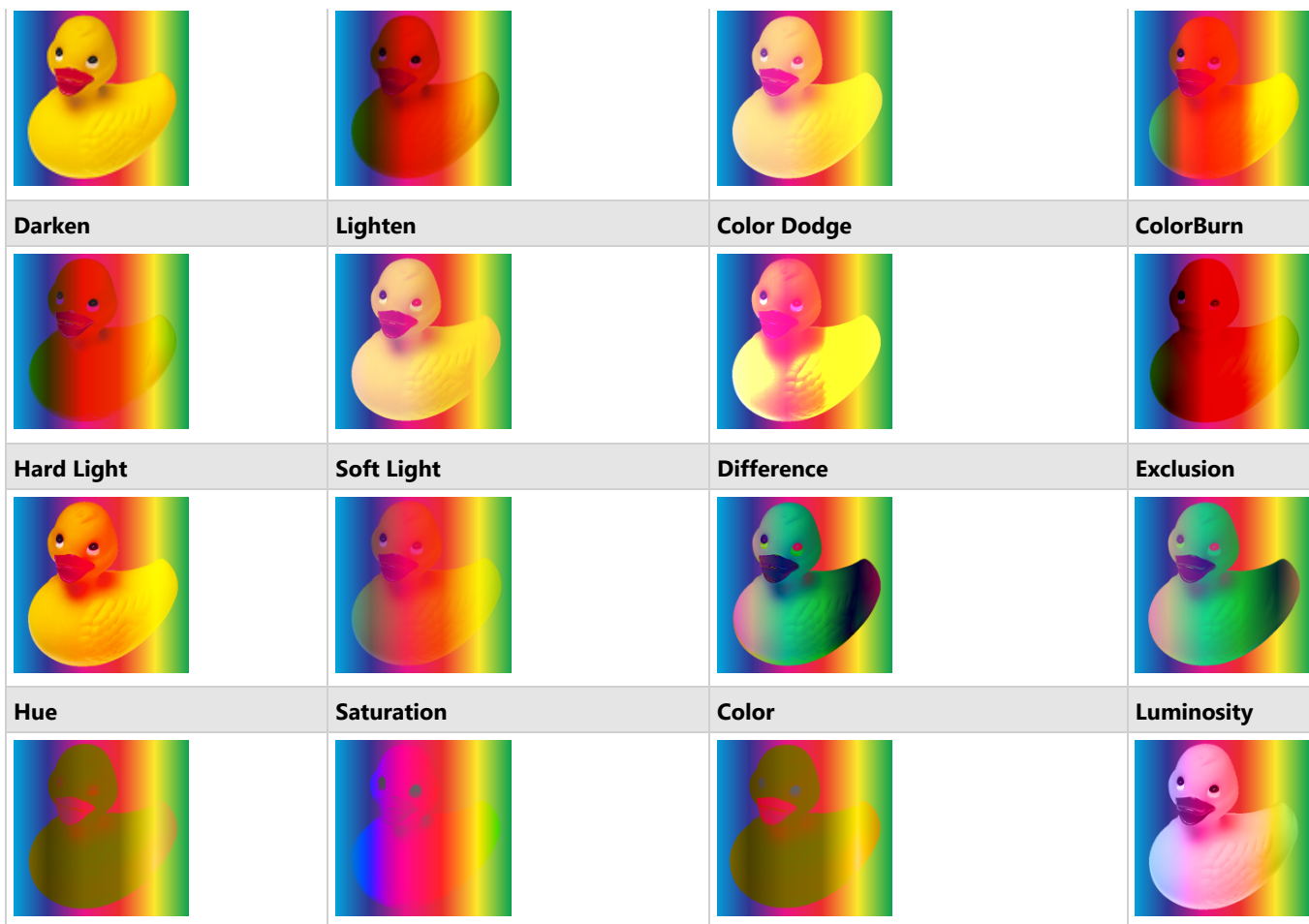
C#

```
//Load the two images to be combined
GcBitmap ducky = new GcBitmap("Images/ducky.png");
GcBitmap spectrum = new GcBitmap("Images/spectrum.png");

//Combine the two images using various compositing and blending modes
spectrum.CompositeAndBlend(ducky, 0, 0, CompositeMode.SourceOver, BlendMode.ColorDodge);
spectrum.SaveAsPng("BlendDucky.png");
```

Source Image	Destination Image
	

Normal	Multiply	Screen	Overlay
--------	----------	--------	---------



Support for ICC Profiles

ICC profile is a color management standard for specifying the color attributes of imaging devices. It ensures that the colors of an image are correctly displayed over different devices. In DsImaging library, the ICC profile is handled as binary data and can be extracted or embedded using **IccProfileData** property of **GcBitmap** class. The ICC profile is supported for various image formats such as, JPEG, PNG, TIFF and GIF.

To extract ICC profile of an image and embed it to another image:


1. Load an image in the GcBitmap instance.
2. Get the ICC profile of an image from the **IccProfileData** property of GcBitmap class.
3. Load another image in the GcBitmap instance to which you want to apply the ICC profile.
4. Assign the ICC profile of first image to this image using the **IccProfileData** property of GcBitmap class.

C#

```
//Get the ICCProfileData for an image and set it to another image
GcBitmap bmp = new GcBitmap();
bmp.Load("Images/peacock-small.jpg");
var peacockICC_Data = bmp.IccProfileData;
Console.WriteLine($"ICC Profile of peacock image consists of {bmp.IccProfileData.Length} bytes");

bmp.Load("Images/puffins-small.jpg");
bmp.IccProfileData = peacockICC_Data;
Console.WriteLine($"ICC Profile of peacock image copied to puffins image which now consists of {bmp.IccProfileData.Length} bytes");
```

For more information about processing images using DsImaging, see [DsImaging sample browser](#).

 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Apply Effects

Advanced imaging effects are helpful in a lot of scenarios such as low-color depth environment, image transmission, medical imaging, remote-sensing, acoustic imagery and forensic surveillance imagery.

DsImaging library offers great flexibility while working with these advanced effects which includes dithering, thresholding, gray scaling, Gaussian blur, and various RGB effects. DsImaging provides the **ApplyEffect** method in the **GcBitmap** class which takes the instance of class representing the effect as a parameter. These effects and the corresponding classes are described in detail in the table below. Please note that the **ApplyEffect** method applies a graphic effect to an image or a portion in-place, which means it stores the result back in the existing Bitmap object instead of storing it in a new instance.

Grayscale	BrightnessContrastEffect
	
TemperatureAndTintEffect	Gaussian Blur
	
Thresholding	Dithering
	

Effects	Classes	Descriptions
Dithering	DitheringEffect	<p>Allows you to apply dithering effect through 9 different algorithms which are provided by the DitheringMethod enumeration.</p> <ul style="list-style-type: none"> • Atkinson • Burks • FloydSteinberg • JarvisJudiceNinke • Sierra • SierraLite • Stucki • TwoRowSierra • NoDithering
Thresholding	BradleyThresholdingEffect OtsuThresholdingEffect	<p>Allows you to apply two types of thresholding effects, Bradley's thresholding and Otsu's thresholding, through BradleyThresholdingEffect and OtsuThresholdingEffect class respectively.</p>
Grayscale	GrayscaleEffect	<p>Allows you to apply grayscale effect as per the three grayscale standards provided by the GrayscaleStandard enumeration.</p> <ul style="list-style-type: none"> • BT709 • BT601 • BT2100
Gaussian Blur	GaussianBlurEffect	<p>Allows you to create a blur effect based on the Gaussian function over the entire input image or a part of the image using the Get method of GaussianBlurEffect class.</p>
RGB effects	OpacityEffect HueRotationEffect SaturationEffect SepiaEffect TemperatureAndTintEffect LuminanceToAlphaEffect BrightnessContrastEffect GammaCorrectionEffect	<p>Allows you to apply various RGB effects using their corresponding classes mentioned in the column on left hand side.</p>

To apply a graphic effect, say dithering, on an image:

1. Initialize the GcBitmap class.
2. Invoke **Get** method of the **DitheringEffect** class to define the dithering effect that specifies the method to be used for dithering.
3. Apply dithering effect to an image using the **ApplyEffect** method which accepts the defined dithering effect as its parameter.

```
C#  
  
var imagePath = Path.Combine("Resources", "Images",  
                             "color-vegetables-small.jpg");  
  
//Initialize GcBitmap  
GcBitmap origBmp = new GcBitmap(imagePath,  
                                new Rectangle(50, 50, 1024, 1024));  
  
//Apply Dithering effect FloydSteinberg  
origBmp.ApplyEffect(DitheringEffect.Get(DitheringMethod.FloydSteinberg),  
                   new Rectangle(0, 0, 1024, 1024));  
  
//Save Dithering effect image  
origBmp.SaveAsJpeg("Dithering.jpg");
```

Similarly, you can apply any other effect on images as mentioned in the table above.

DslImaging library also provides **IsBlackAndWhite** and **IsGrayscale** methods in the **GcBitmap** class to check whether the image is already black and white or grayscale. Both methods work very quickly, as GcBitmap makes it easy to convert a colorful image to a grayscale or bi-level black and white image. These methods also skip unnecessary conversions if the original image is already grayscale or black and white. However, if the image is colorful, these methods just check a few pixels and return the result immediately.

The **IsBlackAndWhite** method checks whether all the pixels of the image are either opaque black (0xFF000000) or opaque white (0xFFFFFFFF). Transparent and semi-transparent pixels are neither black nor white.

The **IsGrayscale** method checks whether all pixels of the image are shades of gray, i.e., their alpha channel is set to 0xFF (fully opaque) and their red, green, and blue channels have the same value.

Refer to the following example code in order to check whether the image is already black and white or grayscale:

```
C#  
  
// Initialize GcBitmap and load the image.  
using var bmp = new GcBitmap("qrcode.png");  
  
// Check if black and white is applied.  
if (bmp.IsBlackAndWhite())  
{  
    Console.WriteLine("The image is black and white.");  
}  
  
// Check if grayscale is applied.  
if (bmp.IsGrayscale())  
{  
    Console.WriteLine("The image is grayscale.");  
}
```

Back to Top

For more information about implementation of different effects using Dslmaging, see [Dslmaging sample browser](#).

Layouts

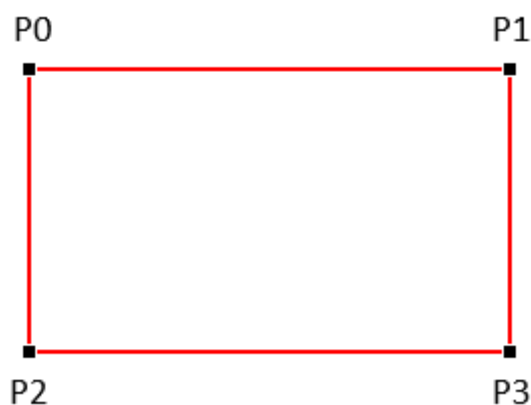
DslImaging provides **LayoutRect** and other related classes in the **GrapeCity.Documents.Layout** namespace to place multiple elements on a PDF page or image without having to calculate positions of each element relative to other ones.

The **LayoutRect** and other related classes implement the flat layout model. There are no chains, barriers, guidelines, biases, or other complications. Certain features of the layout model are:

- Rectangles can be rotated by a multiple of 90 degrees.
- Constraints can reference anchor points from other views (with different transformation matrices).
- Rectangle sides can be bound to arbitrary contours.
- Views can be nested, and the inner view's transformation is automatically recalculated when the outer view's transformation changes.

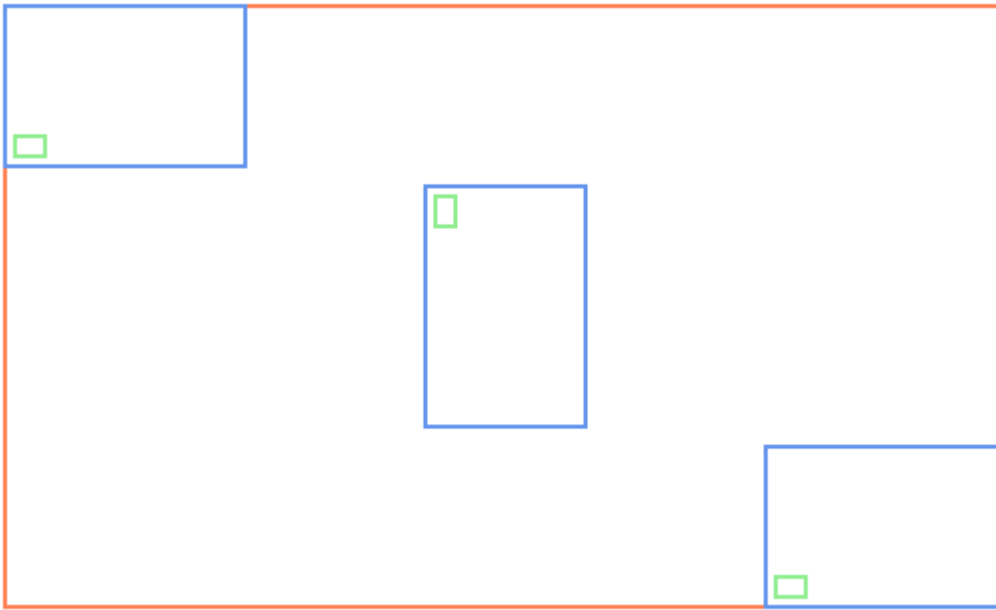
LayoutHost is the main object, which defines the origin of the coordinate system. Also, this object performs layout when all other objects are prepared and linked. **LayoutHost** can create views. **LayoutView** defines a rectangular region with some width, height, and transformation, and the units of all sizes and coordinates are floats and can be of arbitrary dimension.

A **LayoutHost** can create multiple **LayoutViews** with different sizes and transformations, and each **LayoutView** can create multiple **LayoutRects**. A **LayoutRect** is a rectangle whose sides are parallel to the **LayoutView** sides. **LayoutRect** is defined by four points: P0, P1, P2, and P3.



The layout engine calculates the exact positions of the P0, P1, and P2 points for each **LayoutRect** of each **LayoutView** within a **LayoutHost**. The size and position of a **LayoutRect** can be determined if some of the following parameters are known: Width, Height, AspectRatio, Angle (as a multiple of 90 degrees), Left, Top, Right, Bottom, HorizontalCenter, VerticalCenter. The Width, Height and AspectRatio parameters are assigned directly; however, other parameters are usually defined as an offset or delta from the **LayoutView**, other rectangles, or the special anchor points.

The transformation matrix is **Matrix** from the **GrapeCity.Documents.Common** namespace. It has double precision vs. single precision **Matrix3x2** from **System.Numerics**. The **Matrix** can easily be converted to **Matrix3x2**, or it can be multiplied by a **Matrix3x2**.



Refer to the following example code to draw a simple layout:

C#

```
// Initialize LayoutHost. This defines the origin of the coordinate system.
var host = new LayoutHost();

// Create LayoutView. This defines a rectangular region with some width, height, and
// transformation.
LayoutView view = host.CreateView(500, 300, Matrix.Identity);

// Create lists for blue and green rectangles.
var blueList = new List<LayoutRect>();
var greenList = new List<LayoutRect>();

// Create LayoutRect. LayoutRect is a rectangle whose sides are parallel to the owner
// LayoutView sides.
LayoutRect rect = view.CreateRect();

// Set a constraint on the rotation angle of the LayoutRect.
rect.SetAngle(null, 90);

// Set width, height, and center point.
rect.SetWidth(120);
rect.SetHeight(80);
rect.SetHorizontalCenter(null, AnchorParam.VerticalCenter);
rect.SetVerticalCenter(null, AnchorParam.HorizontalCenter);

// Add the LayoutRect to the blue list.
blueList.Add(rect);

// Add a green LayoutRect to the blue LayoutRect.
AddGreenRect(rect);
```

```
// Add a tiny green rectangle at the bottom left corner of the blue rectangle.
void AddGreenRect(LayoutRect r0)
{
    var r1 = view.CreateRect();
    r1.SetWidth(r0, AnchorParam.Width, 0, 0.125f);
    r1.SetHeight(r0, AnchorParam.Height, 0, 0.125f);
    r1.AnchorBottomLeft(r0, 5, 5);
    greenList.Add(r1);
}

// Create two more blue rectangles and place them at different places in the
// LayoutView.
// Add green rectangles to the blue rectangles.
rect = view.CreateRect();
rect.AnchorTopLeft(null, 0, 0, 120, 80);
blueList.Add(rect);
AddGreenRect(rect);

rect = view.CreateRect();
rect.AnchorBottomRight(null, 0, 0, 120, 80);
blueList.Add(rect);
AddGreenRect(rect);

// Calculate all rectangle coordinates based on the constraints provided.
host.PerformLayout();

// Draw the rectangles on a bitmap.
using var bmp = new GcBitmap(540, 340, true);
using (var g = bmp.CreateGraphics(Color.White))
{
    var pen = new Pen(Color.Coral, 2);

    // Set the transformation matrix of the LayoutView when creating the view.
    var m = Matrix3x2.CreateTranslation(20, 20);
    g.Transform = view.Transform.Multiply(m);

    // Draw a rectangle with the corresponding values of the LayoutView.
    g.DrawRectangle(view.AsRectF(), pen);

    // Draw blue rectangles.
    pen.Color = Color.CornflowerBlue;
    for (int i = 0; i < blueList.Count; i++)
    {
        rect = blueList[i];
        g.Transform = rect.Transform.Multiply(m);
        g.DrawRectangle(rect.AsRectF(), pen);
    }

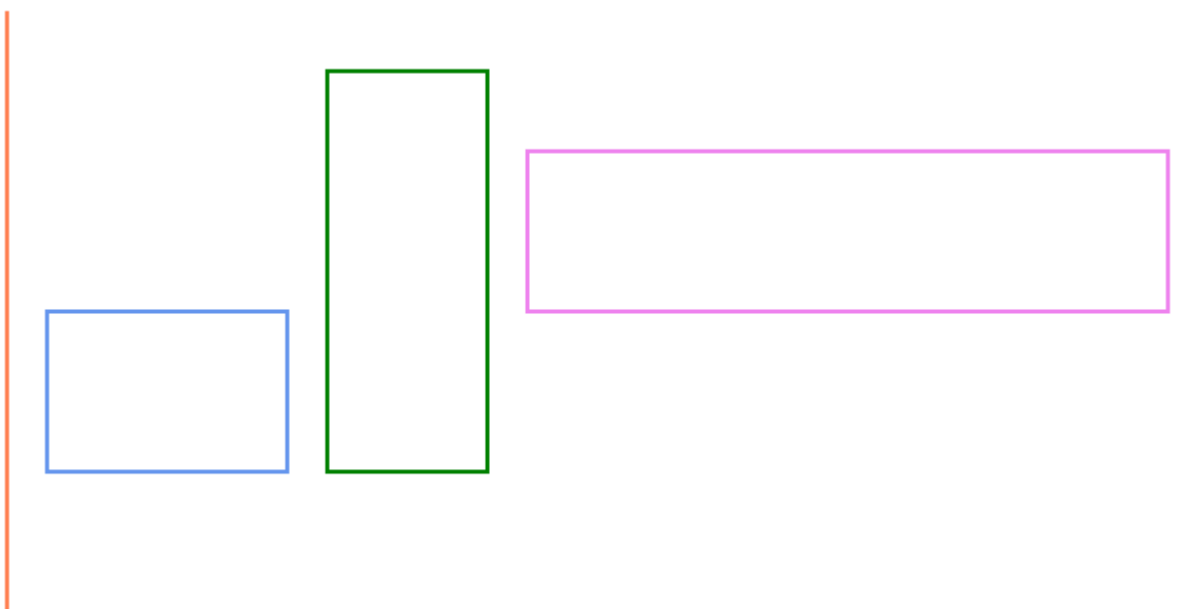
    // Draw green rectangles.
    pen.Color = Color.LightGreen;
```

```
for (int i = 0; i < greenList.Count; i++)
{
    rect = greenList[i];
    g.Transform = rect.Transform.Multiply(m);
    g.DrawRectangle(rect.AsRectF(), pen);
}

// Save the image.
bmp.SaveAsPng("test1.png");
```

Simple Position Constraints

Simple position constraints change one of the following `LayoutRect` parameters: `Left`, `Right`, `Top`, `Bottom`, `HorizontalCenter`, and `VerticalCenter`.



Refer to the following example code to draw a layout with simple position constraints:

```
C#
// Initialize LayoutHost. This defines the origin of the coordinate system.
var host = new LayoutHost();

// Create LayoutView. This defines a rectangular region with width and height.
LayoutView view = host.CreateView(600, 300);

// Create a left vertical line by setting AnchorParam to left.
var bL = view.CreateRect();
bL.AnchorVerticalLine(null);
bL.SetLeft(null, AnchorParam.Left);

// Create a right vertical line by setting AnchorParam to right.
var bR = view.CreateRect();
```

```
bR.AnchorVerticalLine(null);
bR.SetRight(null, AnchorParam.Right);

// Create a blue rectangle and set its AnchorParam.
var r1 = view.CreateRect();
r1.SetTop(null, AnchorParam.VerticalCenter);
r1.SetWidth(120);
r1.SetHeight(80);
r1.SetLeft(bL, AnchorParam.Right, 20);

// Create a green rectangle (rotated 270 degrees or 90 degrees counterclockwise) and
set its AnchorParam.
var r2 = view.CreateRect();
r2.SetAngle(r1, 270);
r2.SetLeft(r1, AnchorParam.Bottom);
r2.SetWidth(200);
r2.SetHeight(80);
r2.SetTop(r1, AnchorParam.Right, 20);

// Create a violet rectangle and set its AnchorParam.
var r3 = view.CreateRect();
r3.SetAngle(r1, 0);
r3.SetBottom(r1, AnchorParam.Top);
r3.SetHeight(80);
r3.SetLeft(r2, AnchorParam.Bottom, 20);
r3.SetRight(bR, AnchorParam.Left, -20);

// Calculate all rectangle coordinates based on the constraints provided.
host.PerformLayout();

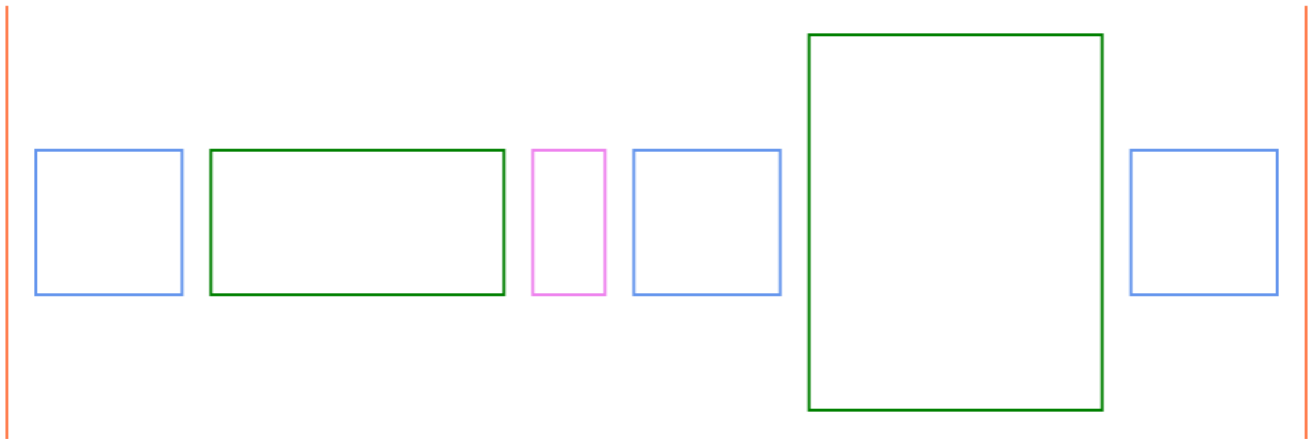
// Draw the rectangles on a bitmap.
using var bmp = new GcBitmap((int)(view.Width + 40), (int)(view.Height + 40), true);
using var g = bmp.CreateGraphics(Color.White);
var m = Matrix3x2.CreateTranslation(20, 20);

// Draw the vertical lines and rectangles.
DrawRect(bL, Color.Coral);
DrawRect(bR, Color.Coral);
DrawRect(r1, Color.CornflowerBlue);
DrawRect(r2, Color.Green);
DrawRect(r3, Color.Violet);
void DrawRect(LayoutRect r, Color c)
{
    g.Transform = r.Transform.Multiply(m);
    g.DrawRectangle(r.AsRectF(), new Pen(c, 2));
}

// Save the image.
bmp.SaveAsPng("test2.png");
```

Chained Position Constraints

Chained position constraints set the parameters of a `LayoutRect` to fill the whole width or height of some area with multiple rectangles having different proportional sizes (measured in stars). The widths or heights of the rectangles will be proportional to their weights (number of stars). The **SetLeftAndOpposite** and **SetBottomAndOpposite** methods are used to set the chained constraints. These methods create two constraints at once.



Refer to the following example code to draw a layout with chained position constraints:

```
C#  
  
// Initialize LayoutHost. This defines the origin of the coordinate system.  
var host = new LayoutHost();  
  
// Create LayoutView. This defines a rectangular region with some width and height.  
LayoutView view = host.CreateView(900, 300);  
  
// Create a left vertical line by setting AnchorParam to left.  
var bL = view.CreateRect();  
bL.AnchorVerticalLine(null);  
bL.SetLeft(null, AnchorParam.Left);  
  
// Create a right vertical line by setting AnchorParam to right.  
var bR = view.CreateRect();  
bR.AnchorVerticalLine(null);  
bR.SetRight(null, AnchorParam.Right);  
  
// Create blue, green, and violet rectangles.  
var r1 = view.CreateRect();  
r1.AnchorTopBottom(null, 100, 100);  
  
var r2 = view.CreateRect();  
r2.AnchorTopBottom(null, 100, 100);  
  
var r3 = view.CreateRect();  
r3.AnchorTopBottom(null, 100, 100);  
  
var r4 = view.CreateRect();  
r4.AnchorTopBottom(null, 100, 100);
```

```
var r6 = view.CreateRect();
r6.AnchorTopBottom(null, 100, 100);

// Create a green rectangle rotated 90 degrees.
var r5 = view.CreateRect();
r5.SetAngle(null, 90);
r5.SetLeft(null, AnchorParam.Top, 20);
r5.SetRight(null, AnchorParam.Bottom, -20);

// Create a chain of rectangles by setting the AnchorParam. The SetLeftAndOpposite
and SetBottomAndOpposite methods are used to set chained position constraints.
r1.SetLeft(bL, AnchorParam.Right, 20);
r2.SetLeftAndOpposite(r1, AnchorParam.Right, 20);
r3.SetLeftAndOpposite(r2, AnchorParam.Right, 20);
r4.SetLeftAndOpposite(r3, AnchorParam.Right, 20);
r5.SetBottomAndOpposite(r4, AnchorParam.Right, -20);
r6.SetLeftAndOpposite(r5, AnchorParam.Top, 20);
r6.SetRight(bR, AnchorParam.Left, -20);

// Set the star and fixed widths.
r1.SetStarWidth(2);
r2.SetStarWidth(4);
r3.SetWidth(50);
r4.SetStarWidth(2);
r6.SetStarWidth(2);

// Set the star height as this rectangle is rotated.
r5.SetStarHeight(4);

// Calculate all rectangle coordinates based on the constraints provided.
host.PerformLayout();

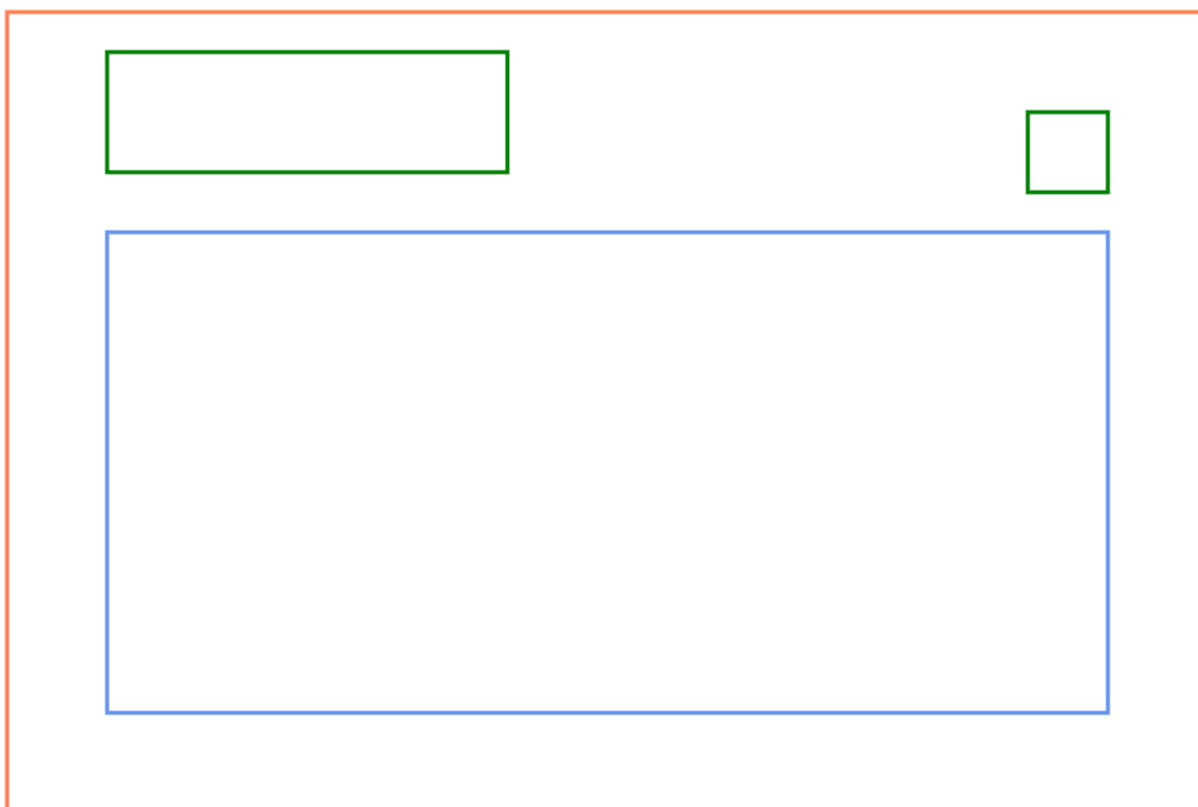
// Draw the rectangles on a bitmap.
using var bmp = new GcBitmap((int)(view.Width + 40), (int)(view.Height + 40), true);
using var g = bmp.CreateGraphics(Color.White);
var m = Matrix3x2.CreateTranslation(20, 20);

// Draw the vertical lines and rectangles.
DrawRect(bL, Color.Coral);
DrawRect(bR, Color.Coral);
DrawRect(r1, Color.CornflowerBlue);
DrawRect(r2, Color.Green);
DrawRect(r3, Color.Violet);
DrawRect(r4, Color.CornflowerBlue);
DrawRect(r5, Color.Green);
DrawRect(r6, Color.CornflowerBlue);
void DrawRect(LayoutRect r, Color c)
{
    g.Transform = r.Transform.Multiply(m);
    g.DrawRectangle(r.AsRectF(), new Pen(c, 2));
}
```

```
// Save the image.  
bmp.SaveAsPng("test3.png");
```

Minimum or Maximum Position Constraints

Min/Max position constraints bind a single `LayoutRect` parameter to one or several other `LayoutRects`. The **AppendMinTop** method is used in the following case to define the minimum top gap between the `LayoutRect` and other `LayoutRects`.



Refer to the following example code to draw a layout with minimum position constraints:

```
C#  
  
// Initialize LayoutHost. This defines the origin of the coordinate system.  
var host = new LayoutHost();  
  
// Create LayoutView. This defines a rectangular region with some width and height.  
LayoutView view = host.CreateView(600, 400);  
  
// Create an outer rectangle.  
var rOuter = view.CreateRect();  
rOuter.AnchorExact(null);  
  
// Create an inner rectangle.  
var rInner = view.CreateRect();  
rInner.AnchorBottomLeftRight(rOuter, 50, 50, 50);
```

```
// Create rectangles.
var r1 = view.CreateRect();
r1.AnchorTopLeft(rOuter, 20, 50, 200, 60);
var r2 = view.CreateRect();
r2.AnchorTopRight(rOuter, 50, 50, 40, 40);

// Set the position of the inner rectangle according to the other rectangles.
rInner.AppendMinTop(rOuter, AnchorParam.Top, 50);
rInner.AppendMinTop(r1, AnchorParam.Bottom, 20);
rInner.AppendMinTop(r2, AnchorParam.Bottom, 20);

// Calculate all rectangle coordinates based on the constraints provided.
host.PerformLayout();

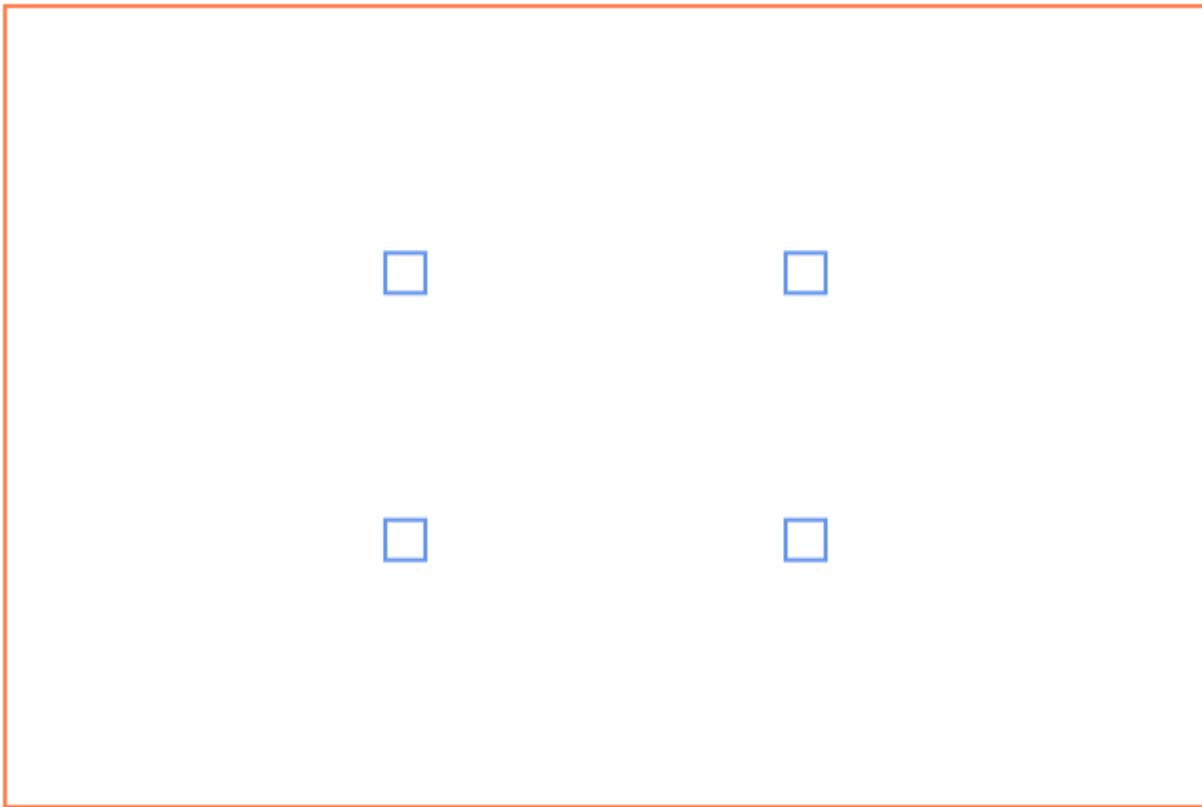
// Draw the rectangles on a bitmap.
using var bmp = new GcBitmap((int)(view.Width + 40), (int)(view.Height + 40), true);
using var g = bmp.CreateGraphics(Color.White);
var m = Matrix3x2.CreateTranslation(20, 20);

// Draw the rectangles.
DrawRect(rOuter, Color.Coral);
DrawRect(rInner, Color.CornflowerBlue);
DrawRect(r1, Color.Green);
DrawRect(r2, Color.Green);
void DrawRect(LayoutRect r, Color c)
{
    g.Transform = r.Transform.Multiply(m);
    g.DrawRectangle(r.AsRectF(), new Pen(c, 2));
}

// Save the image.
bmp.SaveAsPng("test4.png");
```

Anchor Points

Anchor points set the parameters of a `LayoutRect` relative to a `LayoutView`, other `LayoutRects`, or the special anchor points. The anchor points can be created with the **CreatePoint** method of a `LayoutView` or `LayoutRect` object.



Refer to the following example code to draw a layout with anchor points:

C#

```
// Initialize LayoutHost. This defines the origin of the coordinate system.
var host = new LayoutHost();

// Create LayoutView. This defines a rectangular region with some width and height.
LayoutView view = host.CreateView(600, 400);

// Create main rectangle.
var rMain = view.CreateRect();
rMain.AnchorExact(null);

// Create anchor points.
var ap1 = rMain.CreatePoint(1f / 3, 1f / 3);
var ap2 = rMain.CreatePoint(2f / 3, 1f / 3);
var ap3 = rMain.CreatePoint(1f / 3, 2f / 3);
var ap4 = rMain.CreatePoint(2f / 3, 2f / 3);

// Create four rectangles and position them as per the anchor points defined.
var r1 = view.CreateRect();
AnchorCenter(r1, ap1);
var r2 = view.CreateRect();
AnchorCenter(r2, ap2);
var r3 = view.CreateRect();
AnchorCenter(r3, ap3);
```

```
var r4 = view.CreateRect();
AnchorCenter(r4, ap4);
void AnchorCenter(LayoutRect r, AnchorPoint ap)
{
    r.SetHorizontalCenter(ap);
    r.SetVerticalCenter(ap);
    r.SetWidth(20);
    r.SetHeight(20);
}

// Calculate all rectangle coordinates based on the constraints provided.
host.PerformLayout();

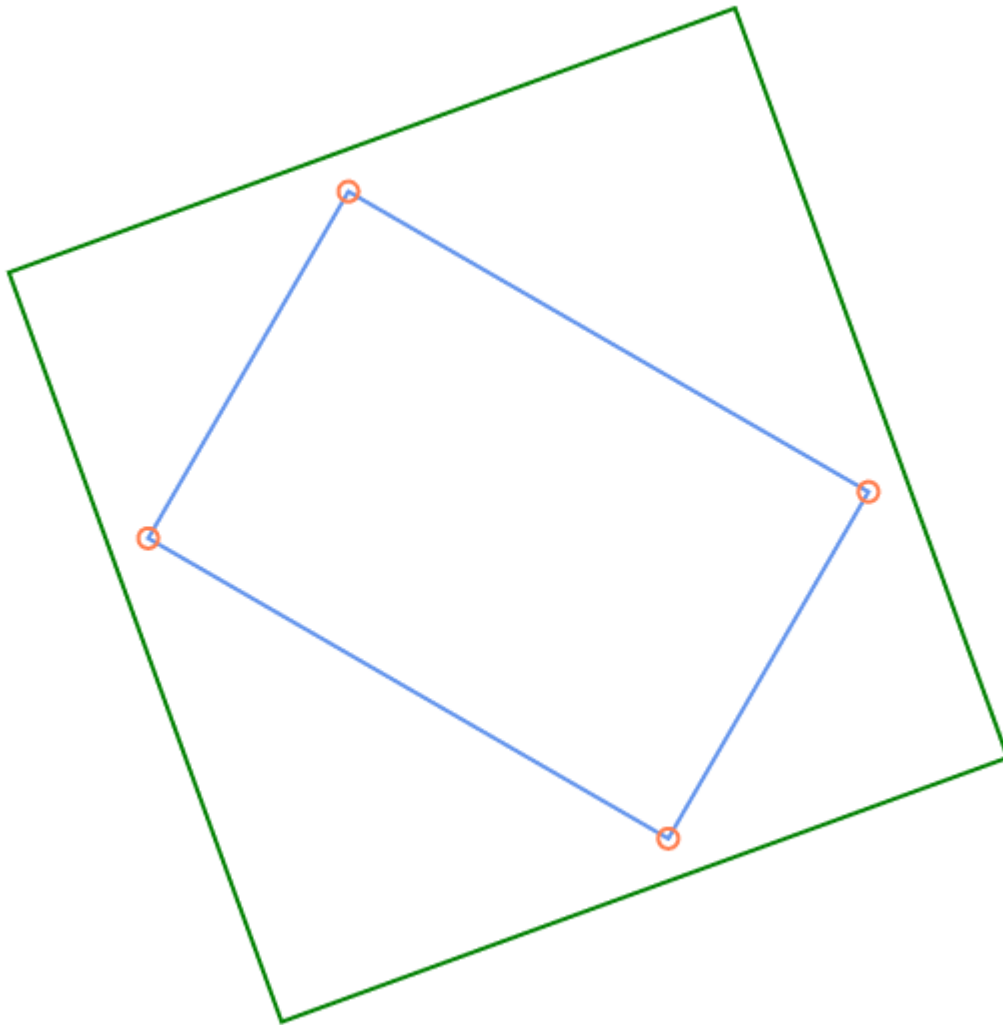
// Draw the rectangles on a bitmap.
using var bmp = new GcBitmap((int)(view.Width + 40), (int)(view.Height + 40), true);
using var g = bmp.CreateGraphics(Color.White);
var m = Matrix3x2.CreateTranslation(20, 20);

// Draw the rectangles.
DrawRect(rMain, Color.Coral);
DrawRect(r1, Color.CornflowerBlue);
DrawRect(r2, Color.CornflowerBlue);
DrawRect(r3, Color.CornflowerBlue);
DrawRect(r4, Color.CornflowerBlue);
void DrawRect(LayoutRect r, Color c)
{
    g.Transform = r.Transform.Multiply(m);
    g.DrawRectangle(r.AsRectF(), new Pen(c, 2));
}

// Save the image.
bmp.SaveAsPng("test5.png");
```

Constraints based on other LayoutView

The `LayoutRect` parameters cannot be bound to a `LayoutRect` from another `LayoutView`. However, it is possible to bind parameters to any anchor point within the same `LayoutHost`.



Refer to the following example code to draw a layout circumscribed in a layout from another LayoutView:

C#

```
// Initialize LayoutHost. This defines the origin of the coordinate system.
var host = new LayoutHost();

//Create rotation.
const double DegToRad = Math.PI / 180;
var m1 = Matrix.CreateRotation(DegToRad * 30);
m1 = m1.Translate(190, -50);

// Create first view and rectangle.
var view1 = host.CreateView(10, 10, m1);
var rc1 = view1.CreateRect();
rc1.AnchorTopLeft(null, 30, 30, 300, 200);

// Create second view and rectangle.
var m2 = Matrix.CreateRotation(DegToRad * -20);
var view2 = host.CreateView(10, 10, m2);
var rc2 = view2.CreateRect();
```

```
// Create anchor points.
var ap1 = rc1.CreatePoint(0, 0);
var ap2 = rc1.CreatePoint(1, 0);
var ap3 = rc1.CreatePoint(1, 1);
var ap4 = rc1.CreatePoint(0, 1);

// Add constraints relative to the anchor points.
rc2.SetTop(ap1, -20);
rc2.SetBottom(ap3, 20);
rc2.SetLeft(ap4, -20);
rc2.SetRight(ap2, 20);

// Calculate all rectangle coordinates based on the constraints provided.
host.PerformLayout();

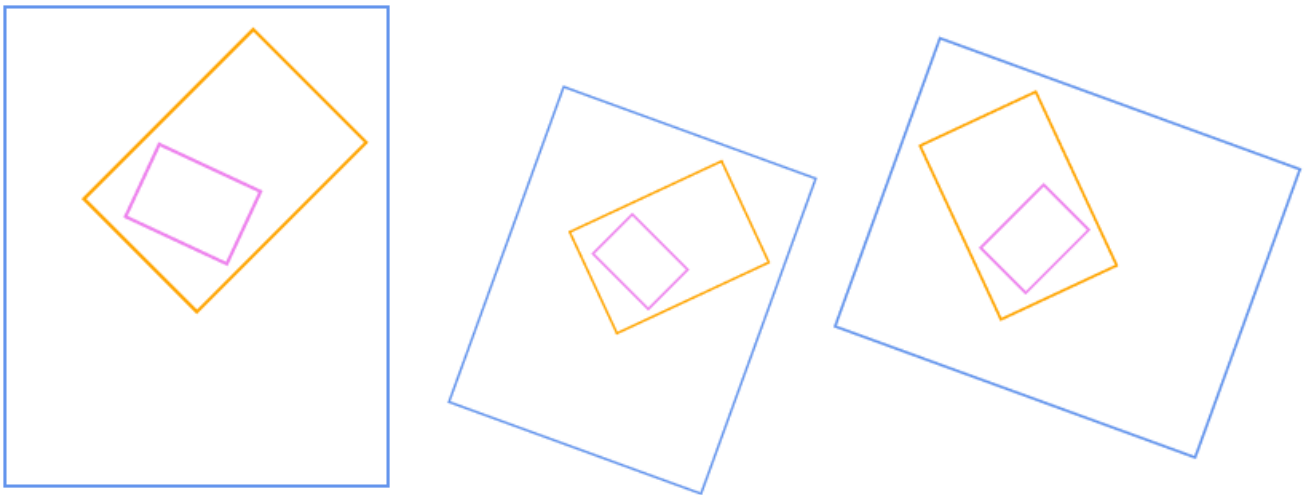
// Draw the rectangles and ellipses on a bitmap.
using var bmp = new GcBitmap(600, 550, true);
using var g = bmp.CreateGraphics(Color.White);
var m = Matrix3x2.CreateTranslation(20, 20);

// Draw the rectangles and ellipses.
DrawRect(rc1, Color.CornflowerBlue);
DrawRect(rc2, Color.Green);
DrawPoint(ap1);
DrawPoint(ap2);
DrawPoint(ap3);
DrawPoint(ap4);
void DrawRect(LayoutRect r, Color c)
{
    g.Transform = r.Transform.Multiply(m);
    g.DrawRectangle(r.AsRectF(), new Pen(c, 2));
}
void DrawPoint(AnchorPoint ap)
{
    g.Transform = ap.Transform.Multiply(m);
    g.DrawEllipse(new RectangleF(-5, -5, 10, 10), new Pen(Color.Coral, 2));
}

// Save the image.
bmp.SaveAsPng("test6.png");
```

Dependent Views and Transformations

The hierarchy of `LayoutViews` is not necessarily flat within the same `LayoutHost`. Some views can be nested in other views. When the transformation matrix of the parent `LayoutView` is updated, all child view transformations are recalculated.



C#

```
// Initialize LayoutHost. This defines the origin of the coordinate system.
var host = new LayoutHost();

// Create first view and rectangle.
var view1 = host.CreateView(240, 300);
var rc1 = view1.CreateRect();
rc1.AnchorExact(null);

// Create second view and rectangle.
var view2 = host.CreateView(100, 150);
var rc2 = view2.CreateRect();
rc2.AnchorExact(null);

// Create third view and rectangle.
var view3 = host.CreateView(70, 50);
var rc3 = view3.CreateRect();
rc3.AnchorExact(null);

//Create rotation.
const double DegToRad = Math.PI / 180;
var m2 = Matrix.CreateRotation(DegToRad * 45);
view2.SetRelativeTransform(view1, m2.Translate(120, -100));
var m3 = Matrix.CreateRotation(DegToRad * -20);
view3.SetRelativeTransform(view2, m3.Translate(-23, 90));

// Calculate all rectangle coordinates based on the constraints provided.
host.PerformLayout();

// Draw the rectangles on a bitmap.
using var bmp = new GcBitmap(850, 350, true);
using var g = bmp.CreateGraphics(Color.White);

// Draw the first set of rectangles according to the first transformation matrix.
```

```
var m = Matrix3x2.CreateTranslation(20, 20);
DrawRect(rc1, Color.CornflowerBlue);
DrawRect(rc2, Color.Orange);
DrawRect(rc3, Color.Violet);

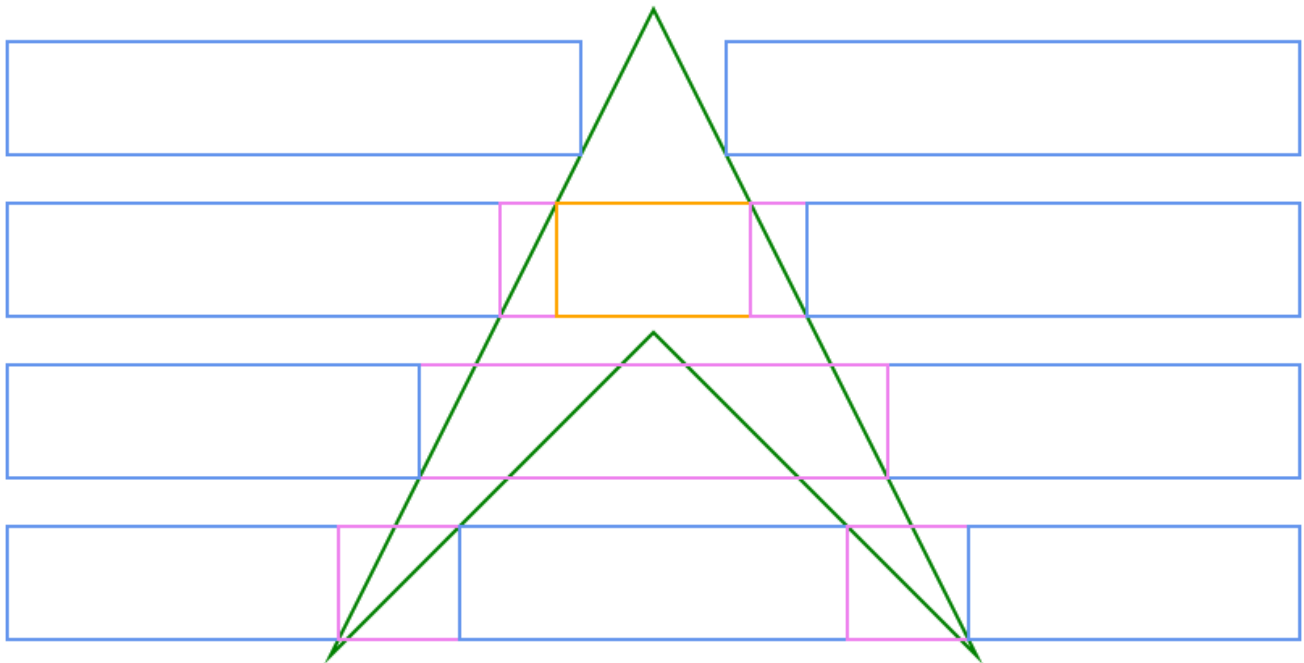
// Draw the second set of rectangles according to the second transformation matrix.
view1.Transform = Matrix.CreateTranslation(350, 50).Scale(0.7).Rotate(DegToRad * 20);
host.PerformLayout();
DrawRect(rc1, Color.CornflowerBlue);
DrawRect(rc2, Color.Orange);
DrawRect(rc3, Color.Violet);

// Draw the third set of rectangles according to the third transformation matrix.
view1.Transform = Matrix.CreateTranslation(520, 200).Scale(0.8).Rotate(DegToRad * -
70);
host.PerformLayout();
DrawRect(rc1, Color.CornflowerBlue);
DrawRect(rc2, Color.Orange);
DrawRect(rc3, Color.Violet);
void DrawRect(LayoutRect r, Color c)
{
    g.Transform = r.Transform.Multiply(m);
    g.DrawRectangle(r.AsRectF(), new Pen(c, 2));
}

// Save the image.
bmp.SaveAsPng("test7.png");
```

Contours

Contour is a closed figure drawn through anchor points. One side of a `LayoutRect` can be bound to one or several contours. From `LayoutRect`'s point of view, contours consist of the outer area, the inner area, and the border area. Rectangles can be bound to positions where one area changes to another. The **CreateContour** method of `LayoutView` class creates an object of **Contour** class, which is used to draw a contour.



C#

```
// Initialize LayoutHost. This defines the origin of the coordinate system.
var host = new LayoutHost();

// Create LayoutView. This defines a rectangular region with some width and height.
LayoutView view = host.CreateView(800, 400);

// Create a contour.
var contour = view.CreateContour();
contour.AddPoints(new AnchorPoint[]
{
    view.CreatePoint(0, 0, 400, 0),
    view.CreatePoint(0, 0, 600, 400),
    view.CreatePoint(0, 0, 400, 200),
    view.CreatePoint(0, 0, 200, 400)
});

// Create first row of rectangles.
var rc11 = view.CreateRect();
rc11.AnchorLeftTopBottom(null, 0, 20, 310);
rc11.AppendMaxRight(contour, ContourPosition.FirstInOutside);
var rc12 = view.CreateRect();
rc12.AnchorRightTopBottom(null, 0, 20, 310);
rc12.AppendMinLeft(contour, ContourPosition.FirstInOutside);

// Create second row of rectangles.
var rc21 = view.CreateRect();
rc21.AnchorLeftTopBottom(null, 0, 120, 210);
rc21.AppendMaxRight(contour, ContourPosition.FirstInOutside);
var rc22 = view.CreateRect();
```

```
rc22.AnchorTopBottom(null, 120, 210);
rc22.SetLeft(rc21, AnchorParam.Right);
rc22.AppendMaxRight(contour, ContourPosition.FirstInInside);
var rc23 = view.CreateRect();
rc23.AnchorTopBottom(null, 120, 210);
rc23.SetLeft(rc22, AnchorParam.Right);
rc23.AppendMaxRight(contour, ContourPosition.NextOutInside);
var rc24 = view.CreateRect();
rc24.AnchorTopBottom(null, 120, 210);
rc24.SetLeft(rc23, AnchorParam.Right);
rc24.AppendMaxRight(contour, ContourPosition.NextOutOutside);
var rc25 = view.CreateRect();
rc25.SetLeft(rc24, AnchorParam.Right);
rc25.AnchorRightTopBottom(null, 0, 120, 210);

// Create third row of rectangles.
var rc31 = view.CreateRect();
rc31.AnchorRightTopBottom(null, 0, 220, 110);
rc31.AppendMinLeft(contour, ContourPosition.FirstInOutside);
var rc32 = view.CreateRect();
rc32.AnchorTopBottom(null, 220, 110);
rc32.SetRight(rc31, AnchorParam.Left);
rc32.AppendMinLeft(contour, ContourPosition.LastOutOutside);
var rc33 = view.CreateRect();
rc33.SetRight(rc32, AnchorParam.Left);
rc33.AnchorLeftTopBottom(null, 0, 220, 110);

// Create fourth row of rectangles.
var rc41 = view.CreateRect();
rc41.AnchorLeftTopBottom(null, 0, 320, 10);
rc41.AppendMaxRight(contour, ContourPosition.FirstInOutside);
var rc42 = view.CreateRect();
rc42.AnchorTopBottom(null, 320, 10);
rc42.SetLeft(rc41, AnchorParam.Right);
rc42.AppendMaxRight(contour, ContourPosition.NextOutOutside);
var rc43 = view.CreateRect();
rc43.AnchorTopBottom(null, 320, 10);
rc43.SetLeft(rc42, AnchorParam.Right);
rc43.AppendMaxRight(contour, ContourPosition.FirstInOutside);
var rc44 = view.CreateRect();
rc44.AnchorTopBottom(null, 320, 10);
rc44.SetLeft(rc43, AnchorParam.Right);
rc44.AppendMaxRight(contour, ContourPosition.NextOutOutside);
var rc45 = view.CreateRect();
rc45.SetLeft(rc44, AnchorParam.Right);
rc45.AnchorRightTopBottom(null, 0, 320, 10);

// Calculate all rectangle coordinates based on the constraints provided.
host.PerformLayout();

// Draw the rectangles and contour on a bitmap.
```

```
using var bmp = new GcBitmap((int)(view.Width + 40), (int)(view.Height + 40), true);
using var g = bmp.CreateGraphics(Color.White);
var m = Matrix3x2.CreateTranslation(20, 20);
g.Transform = m;

// Draw the rectangles and contour.
DrawContour(contour);
DrawRect(rc11, Color.CornflowerBlue);
DrawRect(rc12, Color.CornflowerBlue);
DrawRect(rc21, Color.CornflowerBlue);
DrawRect(rc22, Color.Violet);
DrawRect(rc23, Color.Orange);
DrawRect(rc24, Color.Violet);
DrawRect(rc25, Color.CornflowerBlue);
DrawRect(rc31, Color.CornflowerBlue);
DrawRect(rc32, Color.Violet);
DrawRect(rc33, Color.CornflowerBlue);
DrawRect(rc41, Color.CornflowerBlue);
DrawRect(rc42, Color.Violet);
DrawRect(rc43, Color.CornflowerBlue);
DrawRect(rc44, Color.Violet);
DrawRect(rc45, Color.CornflowerBlue);
void DrawContour(Contour co)
{
    var pts = co.Points.Select(ap => ap.TransformedLocation).ToArray();
    g.DrawPolygon(pts, new Pen(Color.Green, 2));
}
void DrawRect(LayoutRect r, Color c)
{
    g.Transform = r.Transform.Multiply(m);
    g.DrawRectangle(r.AsRectF(), new Pen(c, 2));
}

// Save the image.
bmp.SaveAsPng("test8.png");
```

Complex Graphic Layouts

DslImaging provides **Surface**, **Layer**, **View**, **Space**, and **Visual** classes in **GrapeCity.Documents.Layout.Composition** namespace that acts as a medium between the **layout engine** and the drawing surface, allowing you to draw complex graphics, text, and images. Furthermore, these classes also enable you to customize the z-order and clipping of the drawn graphics.

Surface is the main class in the Composition engine. Surface class contains a **LayoutHost** (the layout engine's root object) and one or more views (layers). Layers consist of visuals (drawable elements) and nested layers. The **Render** method of Surface class calls **PerformLayout** method of LayoutHost class to calculate the surface layout and then it draws all the layers, including nested ones, from the bottom to the top layer on the specified **GcGraphics** object

Layers are of two types: Layer and View class objects (derived from Layer objects). The View object encapsulates the **LayoutView** object, which represents a separate coordinate system with its own transformation matrix. The Layer object functions as a lightweight View with its own list of visuals, nested layers, and possible clipping area. The Surface object can only create Views, not Layers. However, each View object (as well as the Layer object) can create both nested Layers and nested Views. You must create at least one View on the Surface then use that View to create nested Layers (with the same transformation) or nested Views (with different transformation matrices).

Layers contain Visuals and Spaces. The Space object represents a **LayoutRect** that may affect the layout of other elements but is never drawn itself. Spaces are not part of the z-hierarchy of visual elements. The Visual class derives from the Space class. Visual class represents an element that will be drawn on the target GcGraphics. The Render method of the Surface class calls the special **Draw** delegate of the Visual and Layer classes (with the **Visible** property set to True) and passes the GcGraphics object and the current item (Layer or Visual) as parameters to the Draw delegate.

Refer to the following example code to draw a complex graphic with some text:

```
C#  
  
// Set text format.  
var fmt = new TextFormat  
{  
    FontName = "Segoe UI",  
    FontSize = 12f,  
    ForeColor = Color.White  
};  
  
// Initialize Surface.  
var sf = new Surface();  
  
// Create LayoutView.  
var view = sf.CreateView(10, 10);  
  
// Create first figure.  
var fig1 = view.CreateVisual();  
fig1.LayoutRect.AnchorTopLeft(null, 10, 10, 300, 200);  
fig1.Draw = (g, v) => {  
    g.FillEllipse(v.AsRectF(), Color.LightSalmon);  
    g.DrawString("1", fmt, new PointF(50, 50));  
};  
  
// Create second figure.  
var fig2 = view.CreateVisual((g, v) => {  
    g.FillRoundRect(v.AsRectF(), 20, Color.MediumAquamarine);
```

```
        g.DrawString("2", fmt, new PointF(v.Width - 35, v.Height - 45));
    });
    fig2.LayoutRect.AnchorTopLeft(null, 50, 50, 300, 200);

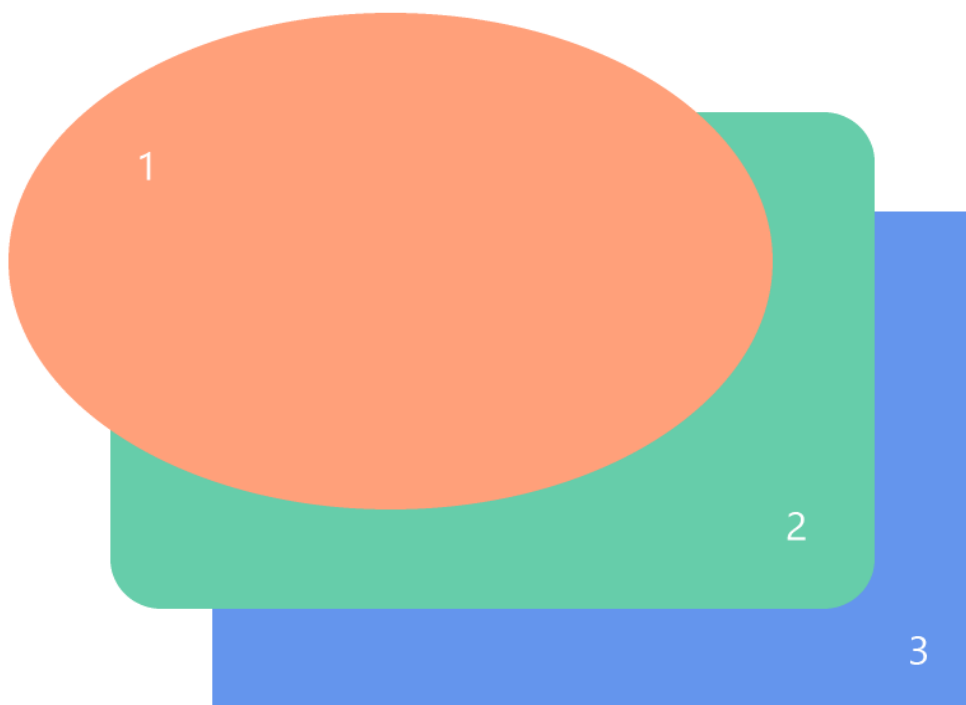
    // Create third figure.
    view.CreateVisual((g, v) => {
        g.FillRectangle(v.AsRectF(), Color.CornflowerBlue);
        g.DrawString("3", fmt, new PointF(v.Width - 27, v.Height - 35));
    }).LayoutRect.AnchorTopLeft(null, 90, 90, 300, 200);

    // Bring the first and second figures to the front.
    fig2.BringToFront();
    fig1.BringToFront();

    // Initialize GcBitmap.
    using var bmp = new GcBitmap(400 * 2, 300 * 2, true);
    using (var g = bmp.CreateGraphics(Color.White))
    {
        g.Transform = Matrix3x2.CreateScale(2);

        // Render the surface.
        sf.Render(g);
    }

    // Save the image.
    bmp.SaveAsPng("Composition.png");
```



Clipping

Any clipping specified on a Layer object applies to the layer's visuals and the nested layers. The Layer class provides

two properties that allow you to define clipping: **ClipRect** and **CreateClipRegion**. You can specify just one of these two properties or both. The behavior is different in the three cases:

1. If only **ClipRect** is specified, then **LayoutRect** value of that property defines the clipping. Note that it can be a **LayoutRect** in any **View** on the same **Surface** and can have its transformation applied to the corresponding **View**.

```
C#
// Initialize Surface.
var sf = new Surface();

// Create first LayoutView.
var view = sf.CreateView(1, 1);

// Create first sub-layer.
var nestedLayer1 = view.CreateSubLayer();

// Create first figure.
var rect = nestedLayer1.CreateVisual((g, v) => {
    g.DrawRectangle(v.AsRectF(), new Pen(Color.Magenta, 1));
    g.DrawString("Rectangle 1", new TextFormat
    {
        FontName = "Segoe UI",
        FontSize = 16f,
        ForeColor = Color.Magenta
    }, new PointF(120, 90));
});
rect.LayoutRect.AnchorTopLeft(null, 20, 20, 300, 200);

// Create second sub-layer.
var nestedLayer2 = view.CreateSubLayer();

// Create second figure.
nestedLayer2.CreateVisual((g, v) => {
    g.FillRectangle(v.AsRectF(), new HatchBrush(HatchStyle.Weave)
    {
        BackColor = Color.White,
        ForeColor = Color.RoyalBlue
    });
}).LayoutRect.AnchorExact(rect.LayoutRect);

// Create second LayoutView.
var view2 = sf.CreateView(1, 1).Translate(120, 30).Rotate(30);

// Create clipping region.
var clipRect = view2.CreateVisual((g, v) => {
    g.DrawRectangle(v.AsRectF(), Color.Green, 1, DashStyle.Dash);
}).LayoutRect;
clipRect.AnchorTopLeft(null, 0, 0, 300, 100);

// Set clipping region.
nestedLayer2.ClipRect = clipRect;
```

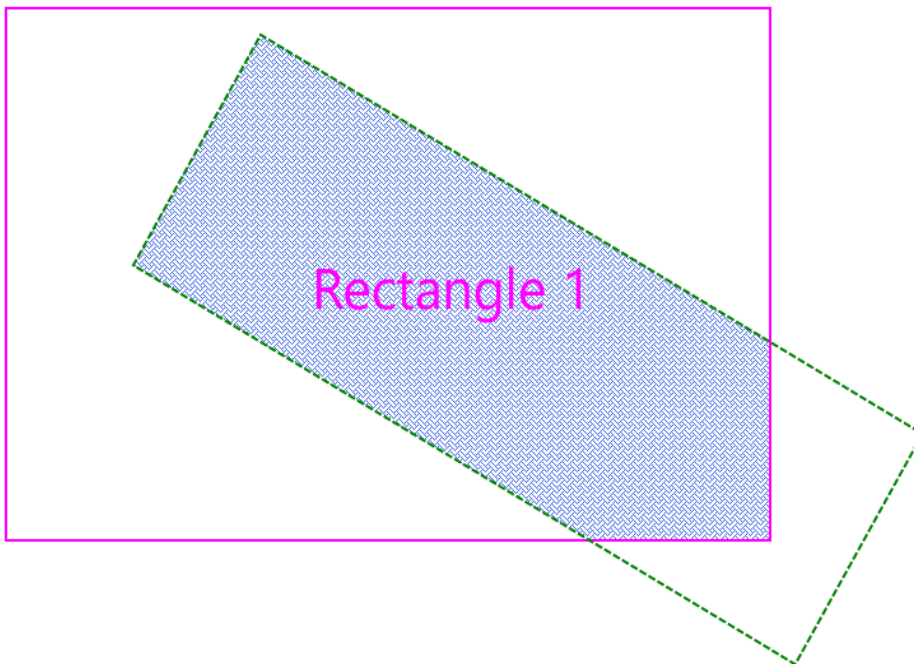


```
// Bring first sub-layer to front.
nestedLayer1.BringToFront();

// Initialize GcBitmap.
using var bmp = new GcBitmap(400 * 2, 280 * 2, true);
using (var g = bmp.CreateGraphics(Color.White))
{
    g.Transform = Matrix3x2.CreateScale(2);

    // Render the surface.
    sf.Render(g);
}

// Save the image.
bmp.SaveAsPng("Clipping.png");
```



2. If only `CreateClipRegion` delegate is specified, then `GcGraphics.PushClip(clipRegion)` applies the clip region returned by the delegate to the graphics before drawing the layer. In this case, the clip region is defined in the layer's own coordinate system without additional transformations. Using `CreateClipRegion` delegate allows you to set a non-rectangular clipping area. You can create an arbitrary path, then a clipping region based on that path, and return it from the delegate.
3. If both `ClipRect` and `CreateClipRegion` properties are specified, then the clip region is defined in the coordinate system of the `LayoutRect` specified by the `ClipRect` property. The top left corner of `LayoutRect` becomes the origin, with axes directed right and down along its sides. Similar to the first case, `LayoutRect` can be from any `View`, and its transformation does not depend on the transformation of the layer to be clipped. Then the returned clip region is applied in the coordinate system defined by `ClipRect` by calling `CreateClipRegion` delegate. After applying the clip region, objects on the layer are drawn in the layer's coordinate system, while the clipping remains transformed by the `ClipRect` and `CreateClipRegion`. This approach simplifies creating complex clipping scenarios. For example, to create a rotated elliptical clipping, you can return an unrotated elliptical region from the `CreateClipRegion` delegate and rotate it using the `ClipRect` defined transformation.

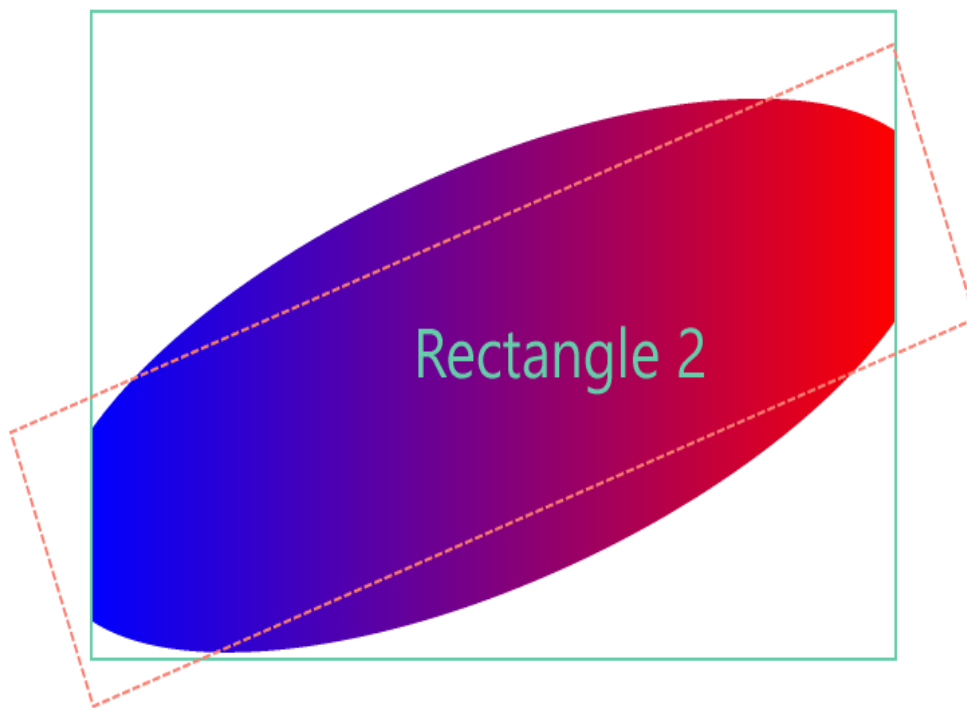
```
C#  
  
// Initialize Surface.  
var sf = new Surface();  
  
// Create first LayoutView.  
var view = sf.CreateView(1, 1);  
  
// Create first sub-layer.  
var nestedLayer1 = view.CreateSubLayer();  
  
// Create first figure.  
var rect = nestedLayer1.CreateVisual((g, v) => {  
    g.DrawRectangle(v.AsRectF(), new Pen(Color.MediumAquaMarine, 1));  
    g.DrawString("Rectangle 2", new TextFormat  
    {  
        FontName = "Segoe UI",  
        FontSize = 16f,  
        ForeColor = Color.MediumAquaMarine  
    }, new PointF(120, 90));  
});  
rect.LayoutRect.AnchorTopLeft(null, 10, 40, 300, 200);  
  
// Create second sub-layer.  
var nestedLayer2 = view.CreateSubLayer();  
  
// Create second figure.  
nestedLayer2.CreateVisual((g, v) => {  
    var lgb = new LinearGradientBrush(Color.Blue, Color.Red);  
    g.FillRectangle(v.AsRectF(), lgb);  
}).LayoutRect.AnchorExact(rect.LayoutRect);  
  
// Create second LayoutView.  
var view2 = sf.CreateView(1, 1).Translate(10, 140).Rotate(-20);  
  
// Create clipping region.  
var clipRect = view2.CreateVisual((g, v) => {  
    g.DrawRectangle(v.AsRectF(), Color.Salmon, 1, DashStyle.Dash);  
}).LayoutRect;  
clipRect.AnchorTopLeft(null, 0, 0, 350, 90);  
  
// Set clipping region.  
nestedLayer2.ClipRect = clipRect;  
nestedLayer2.CreateClipRegion = (g, layer) =>  
{  
    var path = (GcBitmapGraphics.Path)g.CreatePath();  
    var rc = layer.ClipRect.AsRectF();  
    rc.Inflate(0, 20);  
    path.PathBuilder.AddFigure(new EllipticFigure(rc));  
    return g.CreateClipRegion(path);  
};
```

```
// Send second sub-layer to back.
nestedLayer2.SendToBack();

// Initialize GcBitmap.
using var bmp = new GcBitmap(380 * 2, 230 * 2, true);
using (var g = bmp.CreateGraphics(Color.White))
{
    g.Transform = Matrix3x2.CreateScale(2);

    // Render the surface.
    sf.Render(g);
}

// Save the image.
bmp.SaveAsPng("ClippingEllipticalRegion.png");
```



Anchor Points

Layer class provides **CreateVisual** method that creates a Visual that is not associated with a LayoutRect. The position and size of the Visual are calculated based on one or several anchor points.

The anchor points assigned to **AnchorPoints** property of Visual class are recalculated to the View coordinate system and saved to **Points** property of Visual class before executing the Draw delegate of Visual class.

Refer to the following example code to draw a rectangle relative to anchor points:

C#

```
const int pageWidth = 500;
const int pageHeight = 300;
```

```
// Initialize Surface.
var sf = new Surface();

// Create LayoutView.
var view = sf.CreateView(pageWidth, pageHeight);

// Create margin rectangle.
var marginRect = view.CreateVisual((g, v) => {
    g.DrawRectangle(v.AsRectF(), new Pen(Color.Green));
}).LayoutRect;
marginRect.AnchorDeflate(null, 10);

// Create points.
var ap1 = marginRect.CreatePoint(0.25f, 0.25f);
var ap2 = marginRect.CreatePoint(0.75f, 0.25f);
var ap3 = marginRect.CreatePoint(0.75f, 0.75f);
var ap4 = marginRect.CreatePoint(0.25f, 0.75f);

var bluePen = new Pen(Color.CornflowerBlue);

// Create anchor points.
view.CreateVisual(new AnchorPoint[] { ap1 }, DrawAP);
view.CreateVisual(new AnchorPoint[] { ap2 }, DrawAP);
view.CreateVisual(new AnchorPoint[] { ap3 }, DrawAP);
view.CreateVisual(new AnchorPoint[] { ap4 }, DrawAP);

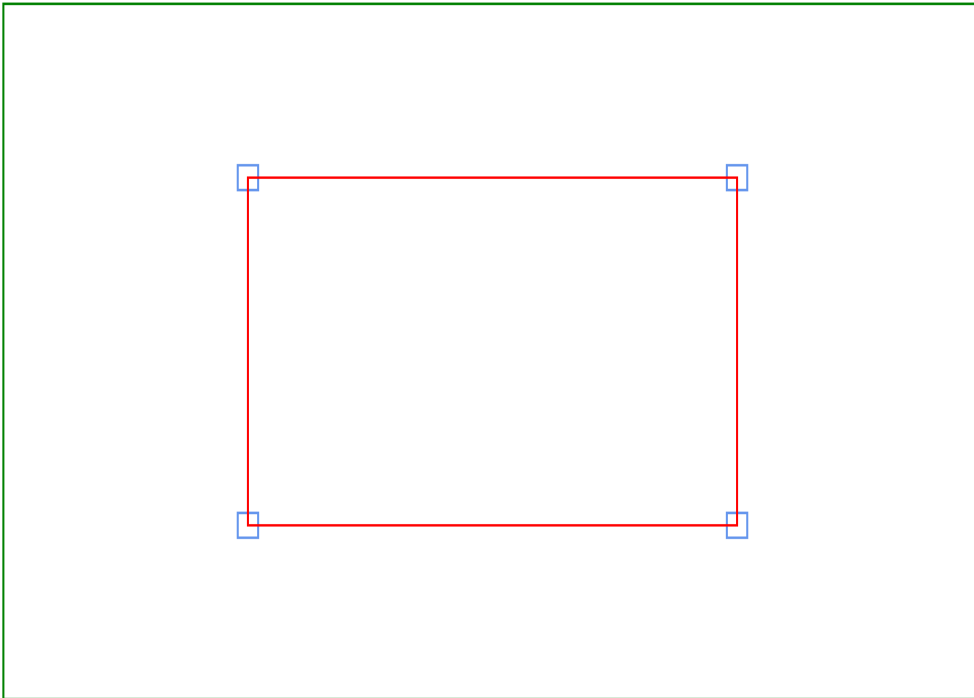
// Draw anchor points.
void DrawAP(GcGraphics g, Visual v)
{
    var pt = v.Points[0];
    g.DrawRectangle(new RectangleF(pt.X - 5, pt.Y - 5, 10, 10), bluePen);
}

// Draw polygon through the anchor points.
view.CreateVisual(new AnchorPoint[] { ap1, ap2, ap3, ap4 },
(g, v) => {
    g.DrawPolygon(v.Points, new Pen(Color.Red));
});

// Initialize GcBitmap.
using var bmp = new GcBitmap(pageWidth * 2, pageHeight * 2, true);
using (var g = bmp.CreateGraphics(Color.White))
{
    g.Transform = Matrix3x2.CreateScale(2);

    // Render the surface.
    sf.Render(g);
}

// Save the image.
bmp.SaveAsPng("AnchorPoints.png");
```



Contours

A Contour can be visualized similarly to anchor points. The **Contour** and **AnchorPoints** properties of Visual class are mutually exclusive; assigning both properties to non-empty values causes an exception when drawing the Surface. The Contour points are converted to regular points via Points property of Visual class before executing the Draw delegate. Then, you can use methods such as **DrawPolygon** of GcGraphics class to render the contour from the Draw delegate.

You may encounter a situation where there are several curves and you need to fill the space between them. The Draw delegate of Layer class enables you to fill the space between several contours. Each Contour can be associated with a separate Visual object on the same View or Layer. You can obtain an array of all visuals from the Draw delegate using **GetVisuals** method of Layer class. You can create a graphics path and add multiple contours as separate figures using the values of Points property of Visual class.

Refer to the following example code to draw multiple contours and fill the space between them by drawing multiple rectangles:

```
C#  
  
// Initialize Surface.  
var sf = new Surface();  
  
// Create bar contour.  
var c1 = CreateBarContour(sf);  
  
// Create donut contour.  
var (c2, c3) = CreateDonutContours(sf);  
  
// Create first LayoutView.  
var view = sf.CreateView(1, 1, (g, 1) =>  
{  
    using var path = g.CreatePath();  
    path.SetFillMode(FillMode.Winding);  
});
```

```
var figures = l.GetVisuals();
for (int i = 0; i < figures.Length; i++)
{
    var pts = figures[i].Points;
    path.BeginFigure(pts[0]);
    for (int j = 1; j < pts.Length; j++)
        path.AddLine(pts[j]);
    path.EndFigure(FigureEnd.Closed);
}
g.FillPath(path, Color.LemonChiffon);
g.DrawPath(path, Color.Tomato, 1f);
});

// Create Visuals.
view.CreateVisual(c1, false);
view.CreateVisual(c2, false);
view.CreateVisual(c3, false);

// Create second LayoutView.
var view2 = sf.CreateView(1, 1).Translate(-90, -20).Skew(30, 0).Rotate(20);

// Draw rectangles.
float top = 0f;
var pen = new Pen(Color.LightSeaGreen);
for (int i = 0; i < 22; i++)
{
    DrawRects(view2, pen, top, c1, c2, c3);
    top += 20f;
}

// Initialize GcBitmap.
using var bmp = new GcBitmap(600 * 2, 570 * 2, true);
using (var g = bmp.CreateGraphics(Color.White))
{
    g.Transform = Matrix3x2.CreateScale(2);

    // Render the surface.
    sf.Render(g);
}

// Save the image.
bmp.SaveAsPng("Contours.png");

// Create bar contour.
static Contour CreateBarContour(Surface surf)
{
    // Create layout view for bar contour.
    var fig = surf.CreateView(1, 1).Translate(160, 80).Rotate(-30);

    // Create rectangular space.
    var sp = fig.CreateSpace().LayoutRect;
```

```
sp.AnchorTopLeft(null, 0f, 0f, 30, 500);

// Create contour.
var c = fig.LayoutView.CreateContour();

// Create points to anchor.
c.AddPoints(new AnchorPoint[]
{
sp.CreatePoint(0, 0),
sp.CreatePoint(1, 0),
sp.CreatePoint(1, 1),
sp.CreatePoint(0, 1)
});
return c;
}

// Create donut contour.
static (Contour, Contour) CreateDonutContours(Surface surf)
{
// Create layout view for donut contour.
var fig = surf.CreateView(1, 1).Translate(30, 100).Skew(20, 0);

// Set dimensions of the donut contour.
float rMax = 150;
float xOffsetMax = 200;
float yOffsetMax = 200;
float rMin = 100;
float xOffsetMin = 230;
float yOffsetMin = 210;
int nMax = 100;
int nMin = 70;

var maxPoints = new List<AnchorPoint>(nMax);
var minPoints = new List<AnchorPoint>(nMin);
double deltaMax = Math.PI * 2 / nMax;
double deltaMin = Math.PI * (-2) / nMin;
var lv = fig.LayoutView;

for (int i = 0; i < nMax; i++)
{
double alpha = deltaMax * i;
float x = (float)(Math.Cos(alpha) * rMax + xOffsetMax);
float y = (float)(Math.Sin(alpha) * rMax + yOffsetMax);
maxPoints.Add(lv.CreatePoint(0f, 0f, x, y));
}
for (int i = 0; i < nMin; i++)
{
double alpha = deltaMin * i;
float x = (float)(Math.Cos(alpha) * rMin + xOffsetMin);
float y = (float)(Math.Sin(alpha) * rMin + yOffsetMin);
minPoints.Add(lv.CreatePoint(0f, 0f, x, y));
}
```

```
    }

    // Create contours.
    var c1 = lv.CreateContour();
    c1.AddPoints(maxPoints);

    var c2 = lv.CreateContour();
    c2.AddPoints(minPoints);

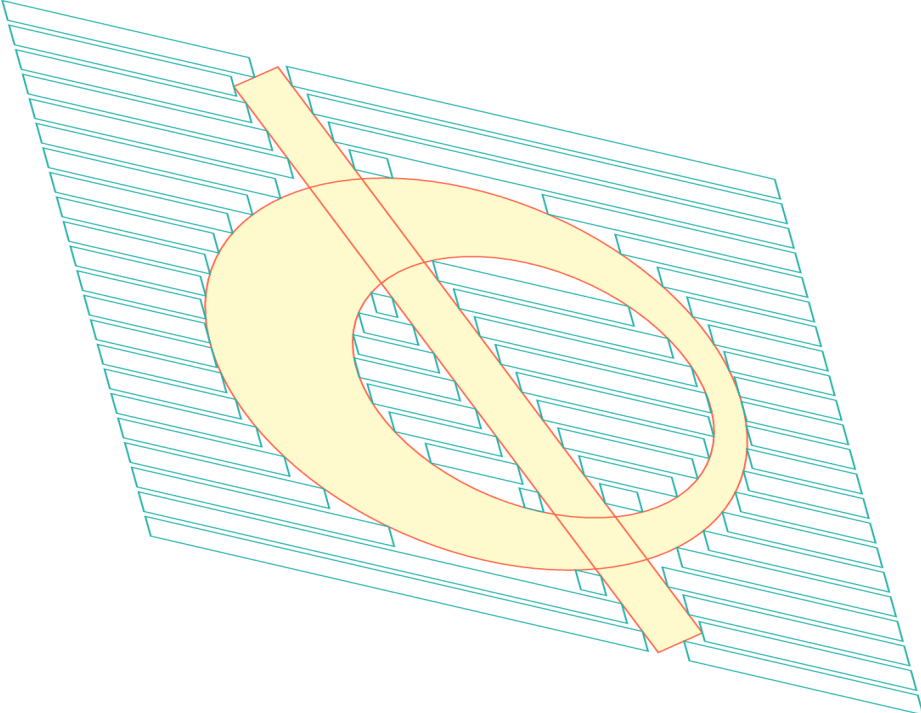
    return (c1, c2);
}

// Draw rectangles.
static void DrawRects(View view, Pen pen, float top, params Contour[] contours)
{
    LayoutRect? rPrev = null;
    while (true)
    {
        var r0 = view.CreateVisual((g, v) => {
            g.DrawRectangle(v.AsRectF(), pen);
        }).LayoutRect;
        if (rPrev is null)
            r0.AnchorTopLeft(null, top, 100);
        else
        {
            r0.SetTop(null, AnchorParam.Top, top);
            r0.SetLeft(rPrev, AnchorParam.Right);
        }
        r0.SetHeight(16);
        r0.AppendMaxRight(null, AnchorParam.Left, 500);

        var r1 = view.CreateSpace().LayoutRect;
        r1.SetTop(null, AnchorParam.Top, top);
        r1.SetHeight(16);
        r1.SetLeft(r0, AnchorParam.Right);
        r1.AppendMaxRight(null, AnchorParam.Left, 500);

        foreach (var c in contours)
        {
            r0.AppendMaxRight(c, ContourPosition.FirstInOutside);
            r1.AppendMaxRight(c, ContourPosition.NextOutOutside);
        }
        view.Surface.PerformLayout();
        if (r1.Width > 0f)
            rPrev = r1;
        else
        {
            ((Space)r1.Tag).Detach();
            break;
        }
    }
}
```


}



Tables

Drawing tables is a common task when creating documents in PDF, JPEG, SVG, and other formats. Creating a table requires calculating the position of the cells, size of the cells, position of the table, size of the table, etc., but calculating all these parameters manually consumes a lot of effort and time. DslImaging provides a **TableRenderer** class in **GrapeCity.Documents.Drawing namespace** that allows you to draw the table without having to think much about the size of table columns, merged cells, the layout of rotated text auto-fitting, etc.

DslImaging's [layout engine](#) handles the task of automatically calculating all the complex details of cell and table resizing and positioning; you just need to provide information about the desired layout, style, and content. The **LayoutHost** is the layout engine's core object. A host is always required to instantiate views and calculate the layout. A LayoutHost creates **LayoutView**. A LayoutView object is a fixed-width and fixed-height rectangle. Each view has its own transformation matrix. The LayoutView can be thought of as a transformed surface with an origin (zero point), two axes (X and Y), and base dimensions (Width and Height). LayoutView places rectangles (**LayoutRect** objects) whose sides are always parallel to the LayoutView's X or Y axis. Based on the constraints specified, the layout engine calculates the size and position of those rectangles relative to the owner LayoutView.

A LayoutView is created to place a table in it. A table is always contained within a rectangle (LayoutRect). The exact size of that rectangle is unknown. The size may vary depending on the contents of the table. However, we must repair at least one table corner (or two or four corners if necessary).

The TableRenderer class is built on top of the layout engine described in [Layouts](#). All table rows, columns, vertical and horizontal grid lines, cells, and cell text have the associated LayoutRect objects available through the object model. The grid lines are individual rectangles with their own width and height. A table cell is also more than just the intersection of a table column and row. Table cells are added to the grid as separate objects on top of the columns and rows. The same grid cell may contain regular, background, and foreground table cells. TableRenderer is a useful tool for drawing tables of any complexity because it combines this flexibility with the power of the layout engine. TableRenderer represents an immutable table. You cannot add more rows to an existing table or split it into two parts; however, you can create a new TableRenderer instance or more instances with the desired number of rows. You can use an instance of TableRenderer for measuring a table without actually drawing it.

Create Table

The TableRenderer constructor accepts multiple parameters. The following table lists the parameters accepted by TableRenderer constructor:

Parameter	Description
graphics	This parameter specifies the GcGraphics object that will be used to draw the table after it has been constructed.
tableRect	This parameter is the table's LayoutRect .
fixedSides	This parameter specifies which sides of a table are fixed.
rowCount	This parameter specifies the overall number of rows in the table. The table will have at least one row.
columnCount	This parameter specifies the overall number of columns in the table. The table will have at least one column.
gridLineColor	This parameter specifies the color of grid lines.
gridLineWidth	This parameter specifies the thickness of the table grid lines by default. Use the SetVerticalGridLineWidth and SetHorizontalGridLineWidth methods for applying custom thickness to individual grid lines).

The constructor also has optional parameters to specify row minimum height and column minimum width, in case

they are not defined manually. The constructor can also set padding for all sides relative to the outer table frame.

The column width and row height do not have to be integers. You can apply star sizes to table columns if both the left and right sides of the table are fixed. It is possible to mix star width, fixed width, and auto width columns in the same table. To set the explicit column width, use the **SetWidth** method of `LayoutRect`. Alternatively, you can add a "minimal width" constraint for auto-sized columns by using the **AppendMinWidth** method of `LayoutRect`. Applying minimal width constraints is optional for auto-sized columns with horizontal text (and **FixedWidth** of **CellStyle** is set to false). Similar constraints can be applied to table rows. You can assign star heights to table rows if the top and bottom table sides are both fixed.

To create a simple table:

1. Initialize `LayoutHost`, `LayoutView`, and `LayoutRect` class instances to define table size and position. The layout engine automatically calculates the position of the table and cells.
2. Create an instance of the `TableRenderer` class and set the parameters of the table.
3. Set the star width (proportional width) of the columns using **SetStarWidth** method.
4. Use **PerformLayout** method on the `LayoutHost`, which will calculate the coordinates based on the constraints provided.
5. Use **Render** method on `TableRenderer` object, which will draw the table.

C#

```
// Initialize GcBitmap.
var bmp = new GcBitmap(440, 270, true);

// Create a graphic.
var g = bmp.CreateGraphics(Color.White);

// Initialize LayoutHost.
var host = new LayoutHost();

// Create LayoutView.
var view = host.CreateView(400, 230, Matrix.Identity);

// Create LayoutRect. Add anchor points.
var rt = view.CreateRect();
rt.AnchorTopLeftRight(null, 36, 36, 36);

// Create an instance of TableRenderer.
var ta = new TableRenderer(
    g,
    rt, FixedTableSides.TopLeftRight,
    rowCount: 5,
    columnCount: 4,
    gridLineColor: Color.Coral,
    gridLineWidth: 3,
    rowMinHeight: 30);

// Set the star width (proportional width) of the columns.
var columns = ta.ColumnRects;
columns[0].SetStarWidth(1);
columns[1].SetStarWidth(5);
columns[2].SetStarWidth(2);
columns[3].SetStarWidth(3);
```

```
// Calculate all coordinates based on the constraints provided.
host.PerformLayout();

// Set the transformation matrix of the LayoutView when creating the view.
var m = Matrix3x2.CreateTranslation(20, 20);
g.Transform = view.Transform.Multiply(m);

// Draw the table.
ta.Render();

// Save the image.
bmp.SaveAsPng("simple-table.png");
```

The output of the above-mentioned example code is shown in the image below:

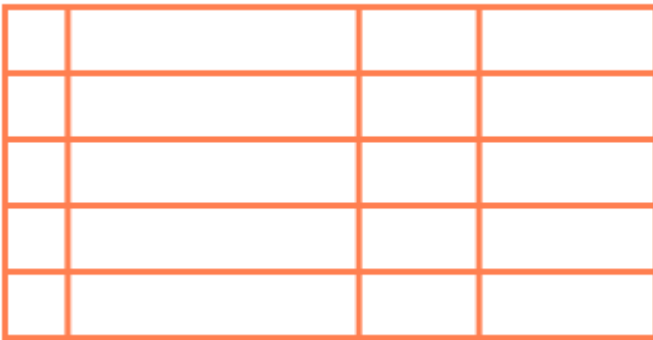


Table Cells

The table cells are separate objects added to the table grid, and one cell can spread to several rows and/or columns. The **AddCell** method adds a table cell containing text or custom content in the specified position with the default or specified style. The AddCell method also allows you to specify row and column spans. There are three types of table cells: normal (regular), background, and foreground.

The regular cells appear on top of the grid and cannot overlap. For example, if such a cell merges two rows, there will be no grid line between the rows in the cell's column. These cells are added at the specified row or column index using the TableRenderer class's indexer property. If there is at least one regular cell, the grid lines will only be drawn around such cells. The grid lines are not drawn around the gaps that are not covered by regular cells.

The **AddMissingCells** method ensures that there are no gaps in the grid.

The background cells always appear behind the grid and behind the filling of the regular and foreground cells (if such a fill exists). Background cells can be overlapped by other cells. You can use background cells to highlight some regions in the table. For example, the odd rows may have a different background color. Also, sometimes you may want to display two or more **TextLayout** objects in the same table cell. One TextLayout will belong to a regular cell, and the others will belong to background cells. If background cells have some text content, their size can be adjusted automatically, as for the regular cells.

The foreground cells appear on top of the background and regular cells. You can use such cells to draw additional elements on top of the grid. Foreground cells can overlap each other and other cells.

Add Cells to Table

To add cells to the table:

1. Add cells to the table using **AddCell** method which takes **CellStyle**, **row**, and **column** indexes as its arguments.
2. Use **AddMissingCells** method to fill the gaps in the table with empty regular cells.

C#

```
// Initialize GcBitmap.
var bmp = new GcBitmap(440, 270, true);

// Create a graphic.
var g = bmp.CreateGraphics(Color.White);

// Initialize LayoutHost.
var host = new LayoutHost();

// Create LayoutView.
var view = host.CreateView(400, 230, Matrix.Identity);

// Create LayoutRect. Add anchor points.
var rt = view.CreateRect();
rt.AnchorTopLeftRight(null, 36, 36, 36);

/* Create an instance of TableRenderer.
Pass paddingAll paramater to avoid overlapping of grid lines with the outer table
frame. */
var ta = new TableRenderer(
    g,
    rt, FixedTableSides.TopLeftRight,
    rowCount: 5,
    columnCount: 4,
    gridLineColor: Color.Coral,
    gridLineWidth: 3,
    rowMinHeight: 30,
    paddingAll: 5);

// Set the star width (proportional width) of the columns.
var columns = ta.ColumnRects;
columns[0].SetStarWidth(1);
columns[1].SetStarWidth(5);
columns[2].SetStarWidth(2);
columns[3].SetStarWidth(3);

// Set style of the new cell.
var csBlue = new CellStyle
{
    LineColor = Color.LightSkyBlue,
    LineWidth = 3f,
    LinePaddingAll = 2f
};

// Add cell to the table.
ta.AddCell(csBlue, 2, 1);
```

```
// Set style of the second new cell.
var csGreen = new CellStyle(csBlue)
{
    LineColor = Color.MediumAquaMarine
};

// Add the second cell to the table.
ta.AddCell(csGreen, 0, 3);

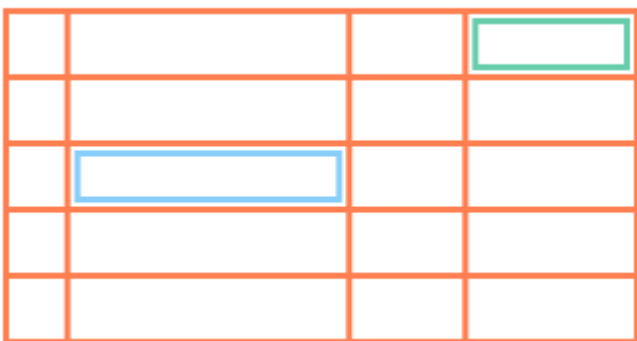
// Fill gaps in the table with empty regular cells.
ta.AddMissingCells();

// Set the transformation matrix of the LayoutView when creating the view.
var m = Matrix3x2.CreateTranslation(20, 20);
g.Transform = view.Transform.Multiply(m);

// Draw the table.
ta.Render();

// Save the image.
bmp.SaveAsPng("add-cells.png");
```

The output of the above-mentioned example code is shown in the image below:



Add Data to Cells

DsImaging allows you to add data to the cells by defining the **TextFormat** property. To add data to the cells:

1. Create a text format using the `TextFormat` class, then a cell style for normal cells using the `CellStyle` class. Create a cell style for header text using the `CellStyle` class, and a header format using the `TextFormat` class.
2. To add data to cells, use `AddCell` method and pass the data to be added to cells as one of its parameters.

```
C#
// Initialize GcBitmap.
var bmp = new GcBitmap(470, 270, true);

// Create a graphic.
var g = bmp.CreateGraphics(Color.White);

// Initialize LayoutHost.
```

```
var host = new LayoutHost();

// Create LayoutView.
var view = host.CreateView(450, 230, Matrix.Identity);

// Create LayoutRect. Add anchor points.
var rt = view.CreateRect();
rt.AnchorTopLeftRight(null, 36, 36, 36);

// Create an instance of TableRenderer.
var ta = new TableRenderer(g,
    rt, FixedTableSides.TopLeftRight,
    rowCount: 5,
    columnCount: 4,
    gridLineColor: Color.Empty,
    gridLineWidth: 1,
    rowMinHeight: 30,
    paddingAll: 3)

// Add table frame style.
{
    TableFrameStyle = new FrameStyle
    {
        FillColor = Color.AliceBlue,
        LineColor = Color.CornflowerBlue,
        LineWidth = 1,
        CornerRadius = 5
    }
};

// Set the star width (proportional width) of the columns.
var columns = ta.ColumnRects;
columns[0].SetStarWidth(1);
columns[1].SetStarWidth(5);
columns[2].SetStarWidth(2);
columns[3].SetStarWidth(3);

// Set text format.
var fmt = new TextFormat
{
    FontName = "Calibri",
    ForeColor = Color.CornflowerBlue,
    FontSize = 16
};

// Set cell style for normal text.
var csNormal = new CellStyle
{
    TextFormat = fmt,
    ParagraphAlignment = ParagraphAlignment.Center,
    PaddingLeftRight = 10,
```

```
        FillColor = Color.MistyRose,
        LineColor = Color.CornflowerBlue,
        LinePaddingAll = 2,
        LineWidth = 1,
        CornerRadius = 5
};

// Set text alignment to center.
var csCenter = new CellStyle(csNormal)
{
    TextAlignment = TextAlignment.Center,
    PaddingLeftRight = 0,
};

// Set cell style for table header.
var csHeader = new CellStyle(csCenter)
{
    TextFormat = new TextFormat(fmt)
    {
        ForeColor = Color.White,
        FontBold = true
    },
    FillColor = Color.LightBlue
};

// Add cells to the table with data and cell style.
ta.AddCell(csHeader, 0, 0, "#");
ta.AddCell(csHeader, 0, 1, "Name");
ta.AddCell(csHeader, 0, 2, "Age");
ta.AddCell(csHeader, 0, 3, "Country");

ta.AddCell(csCenter, 1, 0, "1.");
ta.AddCell(csNormal, 1, 1, "Alice");
ta.AddCell(csCenter, 1, 2, "25");
ta.AddCell(csNormal, 1, 3, "Spain");

ta.AddCell(csCenter, 2, 0, "2.");
ta.AddCell(csNormal, 2, 1, "Bob");
ta.AddCell(csCenter, 2, 2, "36");
ta.AddCell(csNormal, 2, 3, "Germany");

ta.AddCell(csCenter, 3, 0, "3.");
ta.AddCell(csNormal, 3, 1, "Ken");
ta.AddCell(csCenter, 3, 2, "5");
ta.AddCell(csNormal, 3, 3, "Brazil");

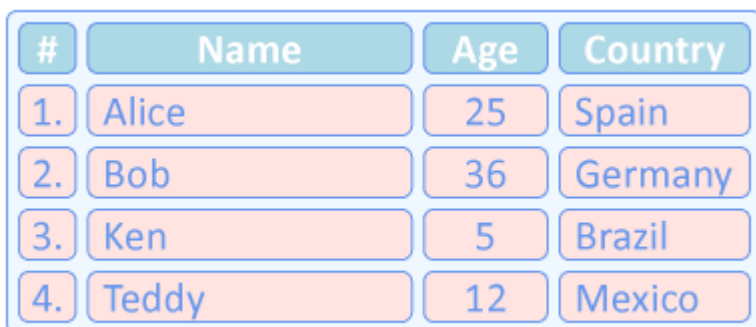
ta.AddCell(csCenter, 4, 0, "4.");
ta.AddCell(csNormal, 4, 1, "Teddy");
ta.AddCell(csCenter, 4, 2, "12");
ta.AddCell(csNormal, 4, 3, "Mexico");
```



```
// Draw the table.
ta.Render();

// Save the image.
bmp.SaveAsPng("add-data-to-cells.png");
```

The output of the above-mentioned example code is shown in the image below:



#	Name	Age	Country
1.	Alice	25	Spain
2.	Bob	36	Germany
3.	Ken	5	Brazil
4.	Teddy	12	Mexico

Merge Cells

DsImaging allows you to merge cells by defining the column and row span in the AddCell method. To merge cells:

1. Use **DefaultCellStyle** property to set the default cell style.
2. Use AddCell method to add the cells to the table and also set the column and row spans while adding cells.
3. Use AddMissingCells method to fill the gaps in the table with empty regular cells.
4. Use ApplyCellConstraints method to calculate the layout of the table and cells.

```
C#
// Initialize GcBitmap.
var bmp = new GcBitmap(440, 270, true);

// Create a graphic.
var g = bmp.CreateGraphics(Color.White);

// Initialize LayoutHost.
var host = new LayoutHost();

// Create LayoutView.
var view = host.CreateView(400, 230, Matrix.Identity);

// Create LayoutRect. Add anchor points.
var rt = view.CreateRect();
rt.AnchorTopLeftRight(null, 36, 36, 36);

// Create an instance of TableRenderer.
var ta = new TableRenderer(
    g,
    rt, FixedTableSides.TopLeftRight,
    rowCount: 5,
    columnCount: 4,
```

```
        gridLineColor: Color.Empty,
        gridLineWidth: 1,
        rowMinHeight: 30,
        paddingAll: 3)

// Add table frame style.
{
    TableFrameStyle = new FrameStyle
    {
        LineColor = Color.CornflowerBlue,
        LineWidth = 1,
        CornerRadius = 5,
        FillColor = Color.AliceBlue
    }
};

// Set the star width (proportional width) of the columns.
var columns = ta.ColumnRects;
columns[0].SetStarWidth(1);
columns[1].SetStarWidth(5);
columns[2].SetStarWidth(2);
columns[3].SetStarWidth(3);

// Set default cell style.
ta.DefaultCellStyle = new CellStyle
{
    LinePaddingAll = 2,
    LineColor = Color.CornflowerBlue,
    LineWidth = 1,
    CornerRadius = 5,
    FillColor = Color.MistyRose
};

// Add cells and set row and column spans.
ta.AddCell(0, 1, 3, 1);
ta.AddCell(3, 0, 1, 4);
ta.AddCell(1, 2, 2, 2);

// Fill gaps in the table with empty regular cells.
ta.AddMissingCells();

// Calculate layout of the grid and cells.
ta.ApplyCellConstraints();

// Set the transformation matrix of the LayoutView when creating the view.
var m = Matrix3x2.CreateTranslation(20, 20);
g.Transform = view.Transform.Multiply(m);

// Draw the table.
ta.Render();
```

```
// Save the image.  
bmp.SaveAsPng("merge-cells.png");
```

The output of the above-mentioned example code is shown in the image below:

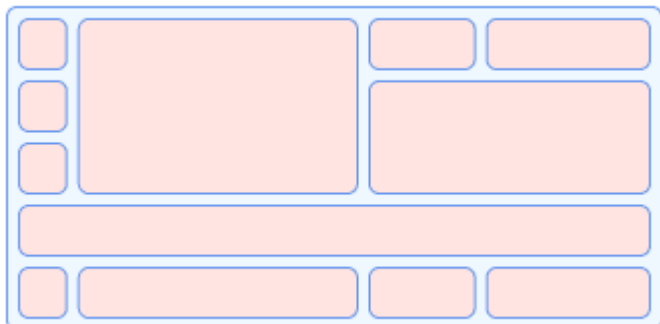


Table and Cell Styling

Each table cell has an associated style (CellStyle) that describes the appearance and behavior of the cell. For example, the **Background** property in the CellStyle class defines whether the cell is in the background. The **FrameStyle** class provides several appearance properties, such as **FillColor**, **LineColor**, **LineWidth**, **LinePadding**, **CornerRadius**, etc., and it is also used to describe the appearance of the outer table frame.

GcImaging allows you to customize the table and cells by adding an outer table frame and inner cell borders. To customize the table and cells:

1. Create an outer table frame using **TableFrameStyle** property.
2. Use **AddMissingCells** method to add empty cells to the table. Set the padding of the empty cells in the table to make spaces equal using **LinePaddingAll** property, and also set **LineColor** and **LineWidth** to draw inner cell borders.
3. Use **ApplyCellConstraints** method to calculate the layout of the table and the cells.

C#

```
// Initialize GcBitmap.  
var bmp = new GcBitmap(440, 270, true);  
  
// Create a graphic.  
var g = bmp.CreateGraphics(Color.White);  
  
// Initialize LayoutHost.  
var host = new LayoutHost();  
  
// Create LayoutView.  
var view = host.CreateView(400, 230, Matrix.Identity);  
  
// Create LayoutRect. Add anchor points.  
var rt = view.CreateRect();  
rt.AnchorTopLeftRight(null, 36, 36, 36);  
  
// Create an instance of TableRenderer.  
var ta = new TableRenderer(  
    g,
```

```
rt, FixedTableSides.TopLeftRight,
rowCount: 5,
columnCount: 4,
gridLineColor: Color.Empty,
gridLineWidth: 1,
rowMinHeight: 30,
paddingAll: 3)

// Add table frame style.
{
    TableFrameStyle = new FrameStyle
    {
        LineColor = Color.CornflowerBlue,
        LineWidth = 1,
        CornerRadius = 5,
        FillColor = Color.AliceBlue
    }
};

// Set the star width (proportional width) of the columns.
var columns = ta.ColumnRects;
columns[0].SetStarWidth(1);
columns[1].SetStarWidth(5);
columns[2].SetStarWidth(2);
columns[3].SetStarWidth(3);

// Create empty cells.
ta.AddMissingCells(new CellStyle
{
    LinePaddingAll = 2,
    LineColor = Color.CornflowerBlue,
    LineWidth = 1,
    CornerRadius = 5,
    FillColor = Color.MistyRose
});

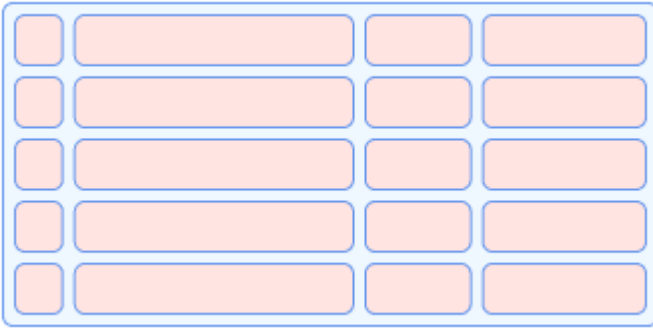
// Calculate layout of the grid and cells.
ta.ApplyCellConstraints();

// Set the transformation matrix of the LayoutView when creating the view.
var m = Matrix3x2.CreateTranslation(20, 20);
g.Transform = view.Transform.Multiply(m);

// Draw the table.
ta.Render();

// Save the image.
bmp.SaveAsPng("table-cell-customization.png");
```

The output of the above-mentioned example code is shown in the image below:



Text Customizations in Cells

The **RightToLeft**, **TextAlignment**, **ParagraphAlignment**, **MaxWidth**, and **MaxHeight** properties of **CellStyle** class resemble the properties of the **TextLayout** class that set the style for text in a cell. The **CellStyle** class also has **RotationAngle** property, which specifies the flow direction of the cell text. If cell content is rotated, then other properties of **CellStyle** are also defined relative to the current text direction.

The **FixedWidth** and **FixedHeight** properties of **CellStyle** class fix the width and height of the cell. The **FixedWidth** property is set to true by default, while the **FixedHeight** property is set to false. These properties work for merged and rotated cells as well.

Position	Nation	Games				Points				Table points
		Played	Won	Drawn	Lost	For	Against	Difference	Tries	
1	England	38	36	26	26	27	40	8	19	53
2	France	20	49	42	32	28	48	35	42	91
3	Ireland	14	39	45	29	18	36	35	40	148
4	Italy	45	37	45	38	29	2	25	5	140
5	Scotland	28	24	49	41	16	5	36	40	116
6	Wales	37	40	36	7	26	49	10	38	85

Refer to the following example code to add text customizations in the cells:

```
C#
// Initialize Team.
var teams = new Team[]
{
    new Team("England"),
    new Team("France"),
    new Team("Ireland"),
    new Team("Italy"),
    new Team("Scotland"),
    new Team("Wales"),
}
```

```
};

int imgW = 410;
int imgH = 320;
int scale = 2;

// Initialize GcBitmap.
using var bmp = new GcBitmap(imgW * scale, imgH * scale, true);

// Create a graphic.
using var g = bmp.CreateGraphics(Color.White);

// Set the transformation matrix of the LayoutView when creating the view.
g.Transform = Matrix3x2.CreateScale(scale);

// Initialize LayoutHost.
var host = new LayoutHost();

// Create LayoutView.
var view = host.CreateView(imgW, imgH);

// Create LayoutRect. Add anchor points.
var rt = view.CreateRect();
rt.AnchorTopRight(null, 10, 10);

// Create an instance of TableRenderer.
var ta = new TableRenderer(g,
    rt, FixedTableSides.TopRight,
    rowCount: teams.Length + 2,
    columnCount: 11,
    gridLineColor: Color.FromArgb(173, 223, 252),
    gridLineWidth: 1,
    rowMinHeight: 10,
    columnMinWidth: 10);

// Set text format.
var fmt = new TextFormat
{
    FontName = "Tahoma",
    FontSize = 12
};

// Set cell style.
var cs = new CellStyle
{
    TextFormat = fmt,
    FixedWidth = false,
    PaddingAll = 4
};

// Set horizontal cell style for table header.
```

```
var csHeaderH = new CellStyle(cs)
{
    TextFormat = new TextFormat(fmt)
    {
        ForeColor = Color.White,
        FontBold = true
    },
    FillColor = Color.FromArgb(17, 93, 140),
    TextAlignment = TextAlignment.Center,
    ParagraphAlignment = ParagraphAlignment.Center,
};

// Set vertical cell style for table header.
var csHeaderV = new CellStyle(csHeaderH)
{
    RotationAngle = 270,
    TextAlignment = TextAlignment.Leading,
    PaddingLeft = 3
};

// Set cell style for numbers.
var csNumber = new CellStyle(cs)
{
    TextAlignment = TextAlignment.Center
};

// Set cell style for Nation.
var csNation = new CellStyle(cs)
{
    TextFormat = new TextFormat(fmt)
    {
        ForeColor = Color.FromArgb(50, 123, 197)
    },
};

// Add the header cells.
ta.AddCell(csHeaderV, 0, 0, 2, 1, "Position");
ta.AddCell(csHeaderH, 0, 1, 2, 1, "Nation");
ta.AddCell(csHeaderH, 0, 2, 1, 4, "Games");
ta.AddCell(csHeaderV, 1, 2, "Played");
ta.AddCell(csHeaderV, 1, 3, "Won");
ta.AddCell(csHeaderV, 1, 4, "Drawn");
ta.AddCell(csHeaderV, 1, 5, "Lost");
ta.AddCell(csHeaderH, 0, 6, 1, 4, "Points");
ta.AddCell(csHeaderV, 1, 6, "For");
ta.AddCell(csHeaderV, 1, 7, "Against");
ta.AddCell(csHeaderV, 1, 8, "Difference");
ta.AddCell(csHeaderV, 1, 9, "Tries");
ta.AddCell(csHeaderH, 0, 10, 2, 1, "Table\npoints");

// Add the data cells.
```

```
for (int i = 0; i < teams.Length; i++)
{
    var team = teams[i];
    int rowIndex = i + 2;
    ta.AddCell(csNumber, rowIndex, 0, $"{i + 1}");
    ta.AddCell(csNation, rowIndex, 1, team.Nation);
    ta.AddCell(csNumber, rowIndex, 2, $"{team.Played}");
    ta.AddCell(csNumber, rowIndex, 3, $"{team.Won}");
    ta.AddCell(csNumber, rowIndex, 4, $"{team.Drawn}");
    ta.AddCell(csNumber, rowIndex, 5, $"{team.Lost}");
    ta.AddCell(csNumber, rowIndex, 6, $"{team.For}");
    ta.AddCell(csNumber, rowIndex, 7, $"{team.Against}");
    ta.AddCell(csNumber, rowIndex, 8, $"{team.Diff}");
    ta.AddCell(csNumber, rowIndex, 9, $"{team.Tries}");
    ta.AddCell(csNumber, rowIndex, 10, $"{team.TablePoints}");
}

// Change background for odd rows.
ta.DefaultCellStyle = new CellStyle
{
    Background = true,
    FillColor = Color.FromArgb(238, 238, 238)
};
for (int i = 0; i < teams.Length; i += 2)
{
    ta.AddCell(i + 2, 0, 1, 11);
}

// Draw the table.
ta.Render();

// Save the image.
bmp.SaveAsPng("text-customizations.png");

// Create the class.
class Team
{
    static readonly Random _rnd = new(24323429);

    public string Nation;
    public int Played, Won, Drawn, Lost;
    public int For, Against, Diff, Tries;
    public int TablePoints;

    internal Team(string nation)
    {
        Nation = nation;
        Played = _rnd.Next(0, 50);
        Won = _rnd.Next(0, 50);
        Drawn = _rnd.Next(0, 50);
        Lost = _rnd.Next(0, 50);
    }
}
```




























```

        For = _rnd.Next(0, 50);
        Against = _rnd.Next(0, 50);
        Diff = _rnd.Next(0, 50);
        Tries = _rnd.Next(0, 50);
        TablePoints = _rnd.Next(0, 150);
    }
}

```

Draw Custom Content in Cells

DslImaging allows you to draw custom content in a cell using **CustomDraw** delegate property of CellStyle class and it is executed from the Render method of TableRenderer class. The CustomDraw property accepts two parameters: a GcGraphics object (passed to the Render method) and a TableCell object.

Shape \ Color	Circle	Triangle	Rectangle	Oval	Square
Red	Red Circle 	Red Triangle 	Red Rectangle 	Red Oval 	Red Square 
Green	Green Circle 	Green Triangle 	Green Rectangle 	Green Oval 	Green Square 
Blue	Blue Circle 	Blue Triangle 	Blue Rectangle 	Blue Oval 	Blue Square 
Orange	Orange Circle 	Orange Triangle 	Orange Rectangle 	Orange Oval 	Orange Square 
Purple	Purple Circle 	Purple Triangle 	Purple Rectangle 	Purple Oval 	Purple Square 

Refer to the following example code to add custom content in the cells:

```

C#
// Initialize GcBitmap.
var bmp = new GcBitmap(960, 530, true);

// Create a graphic.
var g = bmp.CreateGraphics(Color.White);

// Initialize LayoutHost.
var host = new LayoutHost();

// Create LayoutView.
var view = host.CreateView(800, 500);

// Create LayoutRect. Add anchor points.
var rt = view.CreateRect();
rt.AnchorTopLeft(null, 36, 36);

```

```
// Create an instance of TableRenderer.
var ta = new TableRenderer(g,
    rt, FixedTableSides.TopLeft,
    rowCount: 6, columnCount: 6,
    gridLineColor: Color.Black,
    gridLineWidth: 1);

// Set height and width of the the rows and columns.
ta.RowRects[0].SetHeight(50);
ta.ColumnRects[0].SetWidth(120);

// Set the cell style.
var cs = new CellStyle
{
    TextAlignment = TextAlignment.Center,
    ParagraphAlignment = ParagraphAlignment.Center,

    // Set text format.
    TextFormat = new TextFormat
    {
        FontName = "Calibri",
        FontSize = 16,
        FontSizeInGraphicUnits = true,
        FontBold = true
    }
};

// Add a background style for displaying shapes with a custom drawn element
(diagonal line).
var csCornerTopRight = new CellStyle(cs)
{
    Background = true,
    LineWidth = 1f,
    Borders = FrameBorders.MainDiagonal,
    TextAlignment = TextAlignment.Trailing,
    ParagraphAlignment = ParagraphAlignment.Near,
    PaddingRight = 10,
    PaddingTop = 5
};

// Add a regular style for displaying color at the bottom left corner of the
same cell.
var csCornerBottomLeft = new CellStyle(cs)
{
    TextAlignment = TextAlignment.Leading,
    ParagraphAlignment = ParagraphAlignment.Far,
    PaddingLeft = 10,
    PaddingBottom = 5
};
```

```
// Add a background cell at the top left corner.
ta.AddCell(csCornerTopRight, 0, 0, "Shape");

// Add row header cells.
ta.AddCell(cs, 0, 1, "Circle");
ta.AddCell(cs, 0, 2, "Triangle");
ta.AddCell(cs, 0, 3, "Rectangle");
ta.AddCell(cs, 0, 4, "Oval");
ta.AddCell(cs, 0, 5, "Square");

// Add a regular cell at the top left corner.
ta.AddCell(csCornerBottomLeft, 0, 0, "Color");

// Add column header cells.
ta.AddCell(cs, 1, 0, "Red");
ta.AddCell(cs, 2, 0, "Green");
ta.AddCell(cs, 3, 0, "Blue");
ta.AddCell(cs, 4, 0, "Orange");
ta.AddCell(cs, 5, 0, "Purple");

// Add default cell style.
ta.DefaultCellStyle = new CellStyle
{
    PaddingTop = 3,
    PaddingLeftRight = 20,
    PaddingBottom = 55,
    FixedWidth = false,
    TextAlignment = TextAlignment.Center,
    TextFormat = new TextFormat
    {
        FontName = "Calibri",
        FontSize = 14
    },

    // Set text layout.
    CreateTextLayout = (g, cs, data) =>
    {
        var tl = g.CreateTextLayout();
        tl.Append(((Figure)data).Title, cs.TextFormat);
        return tl;
    },

    // Draw the custom content into the cells.
    CustomDraw = (g, tc) =>
    {
        ((Figure)tc.Data).Draw(g, tc.Width, tc.Height);
    }
};

// Add data cells.
ta.AddCell(1, 1, new Figure("Red Circle", Shape.Circle, Color.Red));
```

```
ta.AddCell(1, 2, new Figure("Red Triangle", Shape.Triangle, Color.Red));
ta.AddCell(1, 3, new Figure("Red Rectangle", Shape.Rectangle, Color.Red));
ta.AddCell(1, 4, new Figure("Red Oval", Shape.Oval, Color.Red));
ta.AddCell(1, 5, new Figure("Red Square", Shape.Square, Color.Red));

ta.AddCell(2, 1, new Figure("Green Circle", Shape.Circle, Color.Green));
ta.AddCell(2, 2, new Figure("Green Triangle", Shape.Triangle, Color.Green));
ta.AddCell(2, 3, new Figure("Green Rectangle", Shape.Rectangle,
Color.Green));
ta.AddCell(2, 4, new Figure("Green Oval", Shape.Oval, Color.Green));
ta.AddCell(2, 5, new Figure("Green Square", Shape.Square, Color.Green));

ta.AddCell(3, 1, new Figure("Blue Circle", Shape.Circle, Color.Blue));
ta.AddCell(3, 2, new Figure("Blue Triangle", Shape.Triangle, Color.Blue));
ta.AddCell(3, 3, new Figure("Blue Rectangle", Shape.Rectangle, Color.Blue));
ta.AddCell(3, 4, new Figure("Blue Oval", Shape.Oval, Color.Blue));
ta.AddCell(3, 5, new Figure("Blue Square", Shape.Square, Color.Blue));

ta.AddCell(4, 1, new Figure("Orange Circle", Shape.Circle, Color.Orange));
ta.AddCell(4, 2, new Figure("Orange Triangle", Shape.Triangle,
Color.Orange));
ta.AddCell(4, 3, new Figure("Orange Rectangle", Shape.Rectangle,
Color.Orange));
ta.AddCell(4, 4, new Figure("Orange Oval", Shape.Oval, Color.Orange));
ta.AddCell(4, 5, new Figure("Orange Square", Shape.Square, Color.Orange));

ta.AddCell(5, 1, new Figure("Purple Circle", Shape.Circle, Color.Purple));
ta.AddCell(5, 2, new Figure("Purple Triangle", Shape.Triangle,
Color.Purple));
ta.AddCell(5, 3, new Figure("Purple Rectangle", Shape.Rectangle,
Color.Purple));
ta.AddCell(5, 4, new Figure("Purple Oval", Shape.Oval, Color.Purple));
ta.AddCell(5, 5, new Figure("Purple Square", Shape.Square, Color.Purple));

// Draw the table.
ta.Render();

// Save the image.
bmp.SaveAsPng("custom-content.png");
}
// Create enum for Shape.
enum Shape
{
    Circle,
    Triangle,
    Rectangle,
    Oval,
    Square
}

// Create a class for Figure.
```

```
class Figure
{
    public string Title;
    public Shape Shape;
    public Color Color;

    public Figure(string title, Shape shape, Color color)
    {
        Title = title;
        Shape = shape;
        Color = color;
    }


    public void Draw(GcGraphics g, float w, float h)
    {
        RectangleF rc;
        var pen = new Pen(Color.Black, 1);
        switch (Shape)
        {
            case Shape.Circle:
                rc = new RectangleF(w / 2 - 20, h - 50, 40, 40);
                g.FillEllipse(rc, Color);
                g.DrawEllipse(rc, pen);
                break;
            case Shape.Triangle:
                var points = new PointF[]
                {
                    new(w / 2, h - 50),
                    new(w / 2 + 25, h - 10),
                    new(w / 2 - 25, h - 10)
                };
                g.FillPolygon(points, Color);
                g.DrawPolygon(points, pen);
                break;
            case Shape.Rectangle:
                rc = new RectangleF(w / 2 - 35, h - 50, 70, 40);
                g.FillRectangle(rc, Color);
                g.DrawRectangle(rc, pen);
                break;
            case Shape.Oval:
                rc = new RectangleF(w / 2 - 35, h - 50, 70, 40);
                g.FillEllipse(rc, Color);
                g.DrawEllipse(rc, pen);
                break;
            case Shape.Square:
                rc = new RectangleF(w / 2 - 20, h - 50, 40, 40);
                g.FillRectangle(rc, Color);
                g.DrawRectangle(rc, pen);
                break;
        }
    }
}
```

```
}
}
```

Nested Tables

Nested tables are those in which a table (or tables) is drawn inside another table, where the larger table works as a container for the smaller table. The nested tables work as a way for you to organize images or text in evenly spaced cells. The nested tables can also be helpful in grouping different sets of data. DsImaging allows you to create nested tables and add and customize the data in the nested tables by creating different objects of `LayoutView` and `LayoutRect`.

Fire Diamond

Standard Representation	Tabular Representation								
	<p>Risk levels of hazardous materials in this facility</p> <table border="1"> <thead> <tr> <th>Health Risk</th> <th>Flammability</th> <th>Reactivity</th> <th>Special</th> </tr> </thead> <tbody> <tr> <td>Level 3</td> <td>Level 2</td> <td>Level 1</td> <td></td> </tr> </tbody> </table>	Health Risk	Flammability	Reactivity	Special	Level 3	Level 2	Level 1	
Health Risk	Flammability	Reactivity	Special						
Level 3	Level 2	Level 1							

Refer to the following example code to draw nested tables:

```
C#
float scale = 1.4f;

// Initialize GcBitmap.
using var bmp = new GcBitmap((int)(800 * scale), (int)(300 * scale), true);

// Create a graphic.
using var g = bmp.CreateGraphics(Color.White);
g.Transform = Matrix3x2.CreateScale(scale);

// Initialize LayoutHost.
var host = new LayoutHost();

// Create first table. The view is rotated 45 degrees counterclockwise.
var view1 = host.CreateView(100, 100, Matrix.CreateRotation(-Math.PI / 4));
var rect1 = view1.CreateRect();

// Coincide the rect1 with the view.
rect1.AnchorExact(null);

// Create an instance of TableRenderer for the first table.
var tal = new TableRenderer(g, rect1, FixedTableSides.All,
    rowCount: 2, columnCount: 2, Color.Black, gridLineWidth: 2);
```

```
// Set height and width of the the rows and columns.
var columns = tal.ColumnRects;
columns[0].SetStarWidth(1);
columns[1].SetStarWidth(1);

tal.RowRects[0].SetStarHeight(1);
tal.RowRects[1].SetStarHeight(1);

// Set text format.
var fmt1 = new TextFormat
{
    FontName = "Arial",
    FontSize = 32,
    FontSizeInGraphicUnits = true
};

// Set cell style.
var cs1 = new CellStyle
{
    CustomDraw = (g, tc) =>
    {
        var tl = g.CreateTextLayout();
        tl.Append((string)tc.Data, fmt1);
        tl.PerformLayout();
        var rc = tl.ContentRectangle;
        var m = g.Transform;

        // display the number centered and rotated 45 degrees clockwise
        m = Matrix3x2.CreateTranslation(tc.Width * 0.5f, tc.Height * 0.5f) * m;
        g.Transform = Matrix3x2.CreateRotation((float)Math.PI / 4) * m;
        g.DrawTextLayout(tl, new PointF(-rc.Width * 0.5f, -rc.Height * 0.5f));
    }
};

// Add the cells.
tal.AddCell(new CellStyle(cs1)
{
    FillColor = Color.FromArgb(102, 145, 255)
}, 0, 0).Data = "3";

tal.AddCell(new CellStyle(cs1)
{
    FillColor = Color.FromArgb(255, 102, 102)
}, 0, 1).Data = "2";

tal.AddCell(new CellStyle(cs1)
{
    FillColor = Color.FromArgb(252, 255, 102)
}, 1, 1).Data = "1";
```

```
ta1.AddCell(new CellStyle { FillColor = Color.White }, 1, 0);

// Calculate the layout of the table and the cells.
ta1.ApplyCellConstraints();

/* Shift the view down. The Y-coordinate of the top right point (P1)
is zero, and all other coordinates are not negative.*/
var p = view1.Transform.Transform(rect1.P1);
view1.ApplyOffset(null, 0f, -p.Y);

/* The bottom right point (P3) is at the far right,
which can be used to calculate the maximum width.*/
p = view1.Transform.Transform(rect1.P3);
float w1 = p.X;

/* The bottom left point (P2) is at the bottommost position,
which can be used to calculate the maximum height.*/
p = view1.Transform.Transform(rect1.P2);
float h1 = p.Y;

// Create second table.
var view2 = host.CreateView(0, 0);
var rect2 = view2.CreateRect();

// Anchor the second table to top left corner.
rect2.AnchorTopLeft(null, 0, 0);

// Create an instance of TableRenderer for the second table.
var ta2 = new TableRenderer(g, rect2, FixedTableSides.TopLeft,
    rowCount: 2, columnCount: 4,
    gridLineColor: Color.FromArgb(162, 169, 177),
    gridLineWidth: 2);

// Set text format.
var fmt = new TextFormat
{
    FontName = "Calibri",
    FontSize = 24,
    FontSizeInGraphicUnits = true
};
var fmtBold = new TextFormat(fmt)
{
    FontBold = true
};

// Set cell style.
var cs = new CellStyle
{
    TextFormat = fmt,
    PaddingAll = 6,
    TextAlignment = TextAlignment.Center,
```



```
        FixedWidth = false
    };

    // Set default cell style.
    ta2.DefaultCellStyle = new CellStyle(cs)
    {
        TextFormat = fmtBold,
        FillColor = Color.FromArgb(234, 236, 240)
    };

    // Add cells to the second table with data.
    ta2.AddCell(0, 0, "Health Risk");
    ta2.AddCell(0, 1, "Flammability");
    ta2.AddCell(0, 2, "Reactivity");
    ta2.AddCell(0, 3, "Special");

    ta2.AddCell(cs, 1, 0, "Level 3");
    ta2.AddCell(cs, 1, 1, "Level 2");
    ta2.AddCell(cs, 1, 2, "Level 1");
    ta2.AddCell(cs, 1, 3);

    // Calculate the layout of the table and the cells.
    ta2.ApplyCellConstraints();

    float w2 = rect2.Width;
    float h2 = rect2.Height;

    // Create a third table, which will be larger and the outer table.
    var view3 = host.CreateView(1, 1);
    var rect3 = view3.CreateRect();

    // Anchor the third table to top left corner.
    rect3.AnchorTopLeft(null, 5, 20);

    // Create an instance of TableRenderer for the third table.
    var ta3 = new TableRenderer(g, rect3, FixedTableSides.TopLeft,
        rowCount: 3, columnCount: 3,
        gridLineColor: Color.FromArgb(162, 169, 177),
        gridLineWidth: 2);

    const float CellPaddingX = 6f;
    const float CellPaddingY = 20f;
    const float TextTableGap = 6f;

    // Set the width of the columns.
    columns = ta3.ColumnRects;
    columns[0].AppendMinWidth(CellPaddingX * 2 + w1);
    columns[1].SetWidth(12);
    columns[2].AppendMinWidth(CellPaddingX * 2 + w2);

    // Set the height of the row.
```

```
ta3.RowRects[2].AppendMinHeight(CellPaddingY * 2 + h1);

// Add cells to the table with data.
ta3.AddCell(new CellStyle(cs)
{
    TextFormat = fmtBold,
    Background = true
}, 0, 0, 1, 3, "Fire Diamond");

ta3.AddCell(cs, 1, 0, "Standard Representation");
ta3.AddCell(cs, 1, 2, "Tabular Representation");

// Add the first table to the outer table.
ta3.AddCell(new CellStyle
{
    // Draw the first table.
    CustomDraw = (g, tc) =>
    {
        float x = (tc.Width - w1) * 0.5f;
        float y = (tc.Height - h1) * 0.5f;
        g.Transform = Matrix3x2.CreateTranslation(x, y) * g.Transform;
        ta1.Render(g);
    }
}, 2, 0);

// Add the second table to the outer table.
ta3.AddCell(new CellStyle(cs)
{
    TextFormat = fmtBold,
    ParagraphAlignment = ParagraphAlignment.Center,
    PaddingTop = CellPaddingY,
    PaddingBottom = TextTableGap + h2 + CellPaddingY,
    // Draw the second table.
    CustomDraw = (g, tc) =>
    {
        var tl = tc.TextLayout;
        float y = CellPaddingY + tl.ContentY + tl.ContentHeight + TextTableGap;
        float x = (tc.Width - w2) * 0.5f;
        g.Transform = Matrix3x2.CreateTranslation(x, y) * g.Transform;
        ta2.Render(g);
    }
}, 2, 2, "Risk levels of hazardous materials in this facility");

// Fill gaps in the table with empty regular cells.
ta3.AddMissingCells(1, 0, 2, 3);

ta3.AddCell(new CellStyle
{
    FillColor = Color.FromArgb(248, 249, 250),
    Background = true
}, 1, 0, 2, 3);
```

```
// Draw the third table.
ta3.Render();

// Save the image.
bmp.SaveAsPng("nested-table.png");
```

Limitations

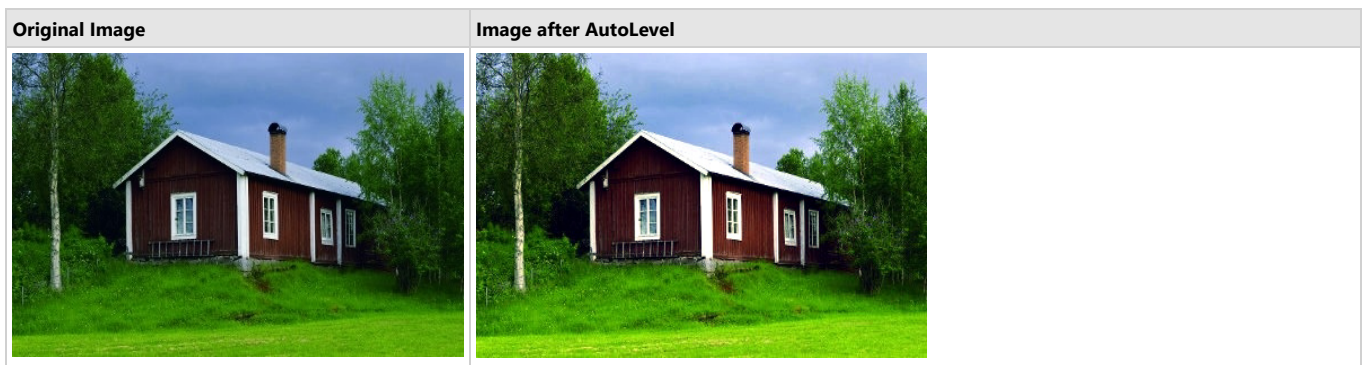
A table created with TableRenderer is immutable. The number of rows and columns must be known before calling the TableRenderer constructor. To split the table between multiple pages, calculate the layout for the huge table first, then recreate the layout for a subset of rows to fit the available space on each page. Moreover, text having an "East Asian" font and aligned vertically can only be displayed in cells with both FixedWidth and FixedHeight set to true in the CellStyle.

Work with Image Colors

DsImaging provides a powerful API to handle various operations on image colors, such as setting the contrast level, adjusting levels of an image histogram, working with color channels and color quantization. These features help to manipulate the color attributes of an image.

Adjust Color Intensity of an Image

DsImaging provides **AutoContrast** and **AutoLevel** methods in **GrayscaleBitmap** class and **GcBitmap** class respectively to improve the colors of an image. These methods modify color intensities such that the maximum range of values (0-255) is fully covered. They also clip the extremely high and low values, and correct the highlights and shadows of an image.



To improve the colors of an image:

1. Initialize a new instance of GcBitmap class and load the image in it.
2. Invoke the **AutoLevel** method of GcBitmap class.
3. Save the image with adjusted contrast.

```
C#  
  
public void SetContrast()  
{  
    //Adjust contrast/level for GcBitmap  
    GcBitmap bmp = new GcBitmap("Images/house.jpg");  
    bmp.AutoLevel(2f, 2f);  
    bmp.SaveAsJpeg("autolevel_house.jpg");  
}
```

[Back to Top](#)

Adjust Image Histogram Levels

Levels adjustments are used to improve the tonal range and brightness levels of an image histogram. For this purpose, DsImaging library provides **AdjustLevels** method in both **GcBitmap** class and **GrayscaleBitmap** class.



To adjust levels of an image histogram:

1. Initialize a new instance of GcBitmap class and load the image in it.
2. Invoke the **AdjustLevels** method.
3. Save the image with adjusted histogram in the desired format.

```
C#  
  
public void SetBrightness()  
{  
    GcBitmap bmp = new GcBitmap("Images/house.jpg");
```

```

bmp.AdjustLevels(0, 0x00646464, 0x00969696, 0x00FAFAFA);
bmp.SaveAsJpeg("briighthouse.jpg");
}

```

[Back to Top](#)

Work with Color Channels

The basic elements of a digital image are the pixels, which in turn are made up of color channels, or the primary colors. For example, the RGB color model has three separate color channels; one for red, one for green and one for blue. DsImaging provides two methods, **ExportColorChannel** and **ImportColorChannel** in the **GcBitmap** class. The **ExportColorChannel** method exports the specific color channel data from an image, whereas the **ImportColorChannel** method creates a colored image based on the specified color channel data.



To export Blue and Green color channels of a colored image:

1. Create an instance of GcBitmap class and load a colored image in it.
2. To create a grayscale image for one of the color channel of a colored image, either invoke **ToGrayscaleBitmap** method or **ExportColorChannel** method of GcBitmap class and pass the color channel as a parameter.
3. Save the image using the **SaveAsJpeg** method.

```

C#
using (var bmp = new GcBitmap("Images/tudor.jpg"))
{
    var gbmp = bmp.ToGrayscaleBitmap(ColorChannel.Blue);
    var outBmp = gbmp.ToGcBitmap();
    outBmp.SaveAsJpeg("Images/blue.jpg");

    bmp.ExportColorChannel(gbmp, ColorChannel.Green);
    gbmp.ToGcBitmap(outBmp, false);
    outBmp.SaveAsJpeg("Images/green.jpg");

    outBmp.Dispose();
    gbmp.Dispose();
}

```





To create an image based on its Red color channel:

1. Create an instance of GcBitmap class and load a colored image in it.
2. Invoke the **ToGrayscaleBitmap** method of GcBitmap class to create a grayscale image based on Red color channel of the colored image.
3. Clear the GcBitmap object representing the colored image by invoking the **Clear** method of GcBitmap class.
4. Invoke the **ImportColorChannel** method of GcBitmap class to copy the red color channel data from the GrayScaleBitmap to colored image's bitmap.
5. Save the image using the **SaveAsJpeg** method.

```
C#  
using (var bmp = new GcBitmap(Images/tudor.jpg))  
using (var gbmpRed = bmp.ToGrayscaleBitmap(ColorChannel.Red))  
{  
    bmp.Clear(Color.Black);  
    //Use the ImportColorChannel method for creating a color image from one of its grayscale channel  
    bmp.ImportColorChannel(gbmpRed, ColorChannel.Red);  
    bmp.SaveAsJpeg(Images/red.jpg);  
}
```

[Back to Top](#)

Work with Color Quantization

Octree color quantization algorithm achieves color quantization by reducing the number of distinct colors used in an image while trying to retain the visual appearance of the original image. The **GenerateOctreePalette** method of the GcBitmap class applies the octree color quantization algorithm to a colored image for generating an octree color palette. This color palette is very useful in scenarios where the device only supports limited number of colors, or when there is a need to reduce the color information of an image due to memory limitations.

To apply Octree color quantization:

1. Load an image by instantiating the GcBitmap class.
2. Generate the Octree color palette using the **GenerateOctreePalette** method of GcBitmap class.
3. Create a new GIF image using the **AppendFrame** method of **GcGifWriter** class which accepts the octree palette as a parameter.

```
C#  
using (GcBitmap bmp = new GcBitmap("Images/tudor.jpg"))  
{  
    uint[] pal = bmp.GenerateOctreePalette(10);  
    using (GcGifWriter gw = new GcGifWriter("Images/test.gif"))  
    {  
        gw.AppendFrame(bmp, pal, DitheringMethod.FloydSteinberg);  
    }  
}
```

[Back to Top](#)

For more information about working with image colors using DslImaging, see [DslImaging sample browser](#).


Transparency Mask

DsImaging allows you to use transparency masks for all drawing and filling operations.

Apply Transparency Mask

Transparency masks are used in imaging to hide some portion of the image while retaining rest of the image. The mask is either an image that already has transparency set on it or it is a bilevel/grayscale image which can serve the purpose because in that case, the black or white pixels are used as a mask.

In DsImaging, the transparency mask can be defined using **BilevelBitmap** or **GrayscaleBitmap** class. The image to be used as a transparency mask is loaded in a **GcBitmap** instance and converted to a **BilevelBitmap** or **GrayscaleBitmap** by using the **ToBilevelBitmap** or **ToGrayscaleBitmap** methods of the **GcBitmap** class. To use the defined mask, you need to draw the image on which the mask is to be applied on the target **GcBitmap** and then apply a mask using the **ApplyTransparencyMask** method of the **GcBitmap** class.

Base Image	Mask
	
Output Image	



To set the transparency mask:

1. Initialize an instance of the GcBitmap class to load the semi-transparent image which is to be applied as a mask.
2. Convert this **GcBitmap** to **GrayscaleBitmap** which will be used as the image mask, using the **ToGrayscaleBitmap** method of the GcBitmap class.
3. Initialize another instance of the GcBitmap class to load the image on which the transparency mask is to be applied.
4. Apply the transparency mask to the resulting bitmap using the **ApplyTransparencyMask** method of the GcBitmap class.
5. Convert the resulting bitmap to an opaque image with specified background color using the **ConvertToOpaque** method of the GcBitmap class.

C#

```
//Initialize bitmap for generating mask image
GcBitmap mask = new GcBitmap("logo.png");

//Draw image to which the transparency mask has to be applied
GcBitmap bmp = new GcBitmap("tudor.jpg");

//Define the transparency mask using mask image
GrayscaleBitmap grayscaleMask = mask.ToGrayscaleBitmap(ColorChannel.Blue, true);


//Apply the transparency mask to the result bitmap
bmp.ApplyTransparencyMask(grayscaleMask);

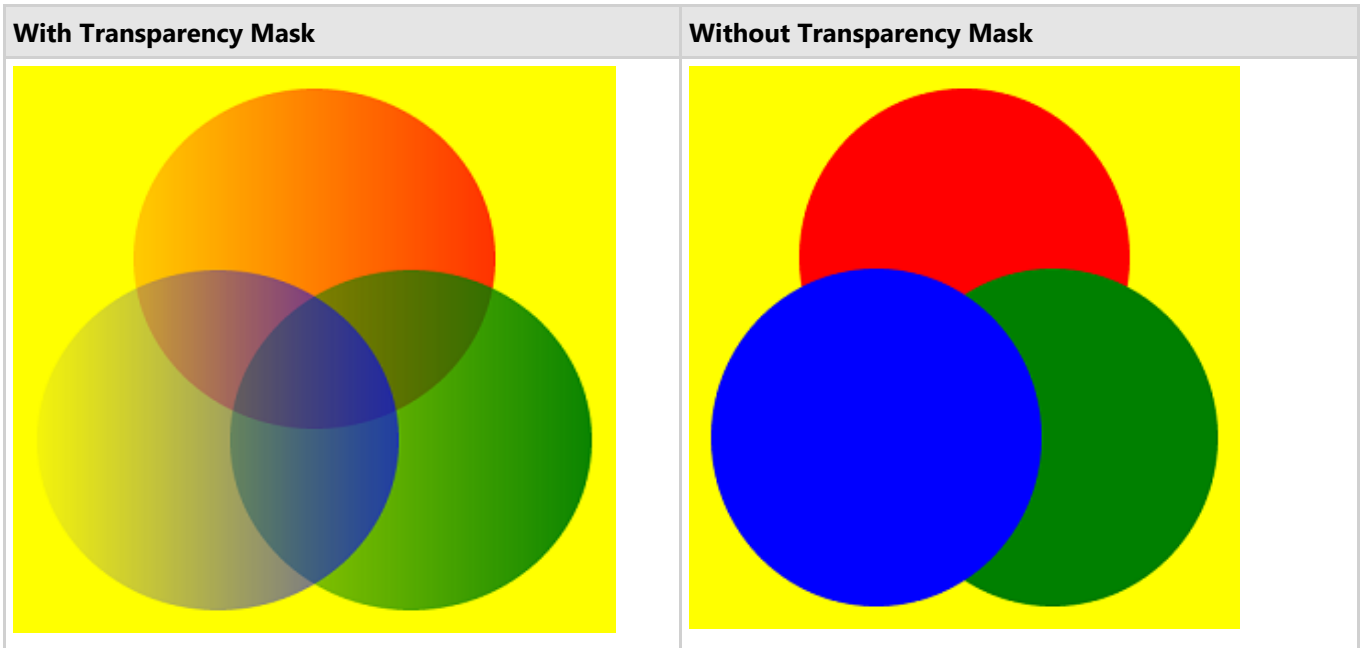
//Convert the result bitmap to opaque
bmp.ConvertToOpaque(Color.Beige);

//Save the result bitmap to save transparent image
bmp.SaveAsJpeg("TransparentImg.jpg");
```

[Back to Top](#)

You can use the **TransparencyMaskBitmap** property of **BitmapRenderer** class to specify a transparency mask that will be used for any subsequent drawing on the bitmap. Pixels in the mask with value 0 are fully opaque and will completely mask any drawing (meaning, the pixels of the target bitmap will remain unchanged). Pixels in the mask with value 255 are fully transparent, meaning that any drawing will have the same effect as if there was no mask. Pixels with values between 0 and 255 will modify the transparency of the pixels being drawn according to their value.

 **Note:** The transparency mask bitmap must be of the same pixel size as the target bitmap.



The following example shows how to set the transparency mask and prevent opacity while working with overlapping images:

C#

```
// Prepare a linear gradient transparency mask,
// from 0 (transparent) to 255 (opaque):
using var mask = new GcBitmap(500, 500, true);
using var gmask = mask.CreateGraphics();
var grad = new LinearGradientBrush(Color.Black, Color.White);
gmask.FillRectangle(new RectangleF(0, 0, mask.Width, mask.Height), grad);
// Convert to GrayscaleBitmap to be used as Renderer.TransparencyMaskBitmap:
using var gsb = mask.ToGrayscaleBitmap();

// Fill target bitmap with yellow background:
using var bmp = new GcBitmap(500, 500, false);
using var g = bmp.CreateGraphics(Color.Yellow);

// Apply the transparency mask (comment out to see the results without the mask):
g.Renderer.TransparencyMaskBitmap = gsb;

// Fill 3 circles, note how the fill gradually changes
// from transparent to opaque (left to right) along with the gradient:
g.FillEllipse(new RectangleF(100, 20, 300, 300), Color.Red);
g.FillEllipse(new RectangleF(180, 180, 300, 300), Color.Green);
g.FillEllipse(new RectangleF(20, 180, 300, 300), Color.Blue);
```

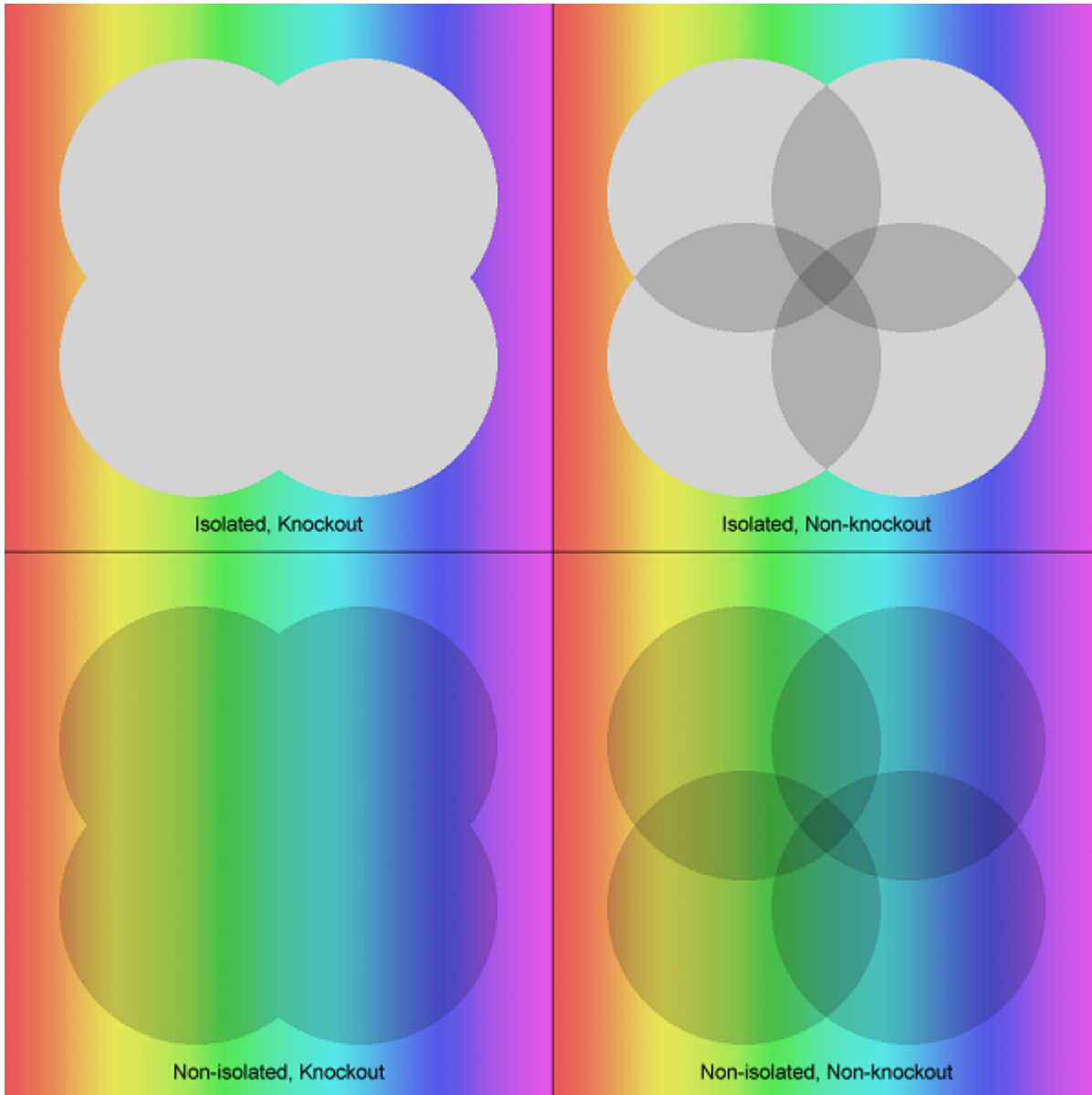
```
bmp.SaveAsPng("transpmask.png");
```

Remove Transparency

You can check whether an image contains any transparent pixels by using the **HasTransparentPixels** method of **GcBitmap** class. It scans the image and returns true if there are any pixels with the alpha channel value different from 255. You can also convert images with transparent pixels to opaque with a specified solid background color by using the **ConvertToOpaque** method of the **GcBitmap** class, as shown in the below example code:

```
C#  
  
// Initialize a GcBitmap and load a transparent image into it:  
using GcBitmap origBmp = new GcBitmap("Transparent.png");  
  
// Check for transparent pixels and convert to opaque if any:  
if (origBmp.HasTransparentPixels())  
{  
    origBmp.ConvertToOpaque(Color.LightBlue);  
    origBmp.SaveAsJpeg("NotTransparent.jpg");  
}  
else  
    Console.WriteLine("No transparent pixels");
```

When drawing semi-transparent graphic objects, usually the resulting color of a pixel is a combination of the target bitmap pixel's color and the color of the graphic object's pixel. But if the **BackgroundBitmap** is set on the **BitmapRenderer**, pixels of that bitmap will be used instead of the target bitmap's pixels when determining the resulting color (the **BackgroundBitmap** must have the same pixel size as the target bitmap). Background bitmaps are used to support Isolated and Knockout groups when rendering PDFs to images.



The following example shows the use of **BackgroundBitmap** to reproduce isolated and knockout groups rendering from the PDF specification:

Example Title

```
// The target bitmap will be 1000x10000 pixels containing 4 demo quadrants:
using var bmp = new GcBitmap(500 * 2, 500 * 2, false, 96f, 96f);
// The spectrum image used for the backdrop:
using var bmp1 = new GcBitmap("spectrum-pastel-500x500.png");

using var bmpBackdrop = new GcBitmap(500, 500, false, 96f, 96f);
using var bmpInitial = new GcBitmap(500, 500, false, 96f, 96f);
using var gB = bmpBackdrop.CreateGraphics();
using var gI = bmpInitial.CreateGraphics();

gB.Renderer.Aliased = true;
gB.Renderer.BlendMode = BlendMode.Multiply;
gI.Renderer.Aliased = true;
```

```
gI.Renderer.BlendMode = BlendMode.Multiply;

// Isolated, Knockout
bmpBackdrop.BitBlt(bmp1, 0, 0);
bmpInitial.Clear(Color.Transparent);
gB.Renderer.BackgroundBitmap = bmpInitial;
gB.FillEllipse(new RectangleF(50, 50, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(200, 50, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(50, 200, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(200, 200, 250, 250), Color.LightGray);
bmp.BitBlt(bmpBackdrop, 0, 0);

// Isolated, Non-knockout
gI.FillEllipse(new RectangleF(50, 50, 250, 250), Color.LightGray);
gI.FillEllipse(new RectangleF(200, 50, 250, 250), Color.LightGray);
gI.FillEllipse(new RectangleF(50, 200, 250, 250), Color.LightGray);
gI.FillEllipse(new RectangleF(200, 200, 250, 250), Color.LightGray);
bmpBackdrop.BitBlt(bmp1, 0, 0);
bmpBackdrop.AlphaBlend(bmpInitial, 0, 0);
bmp.BitBlt(bmpBackdrop, 500, 0);


// Non-isolated, Knockout
bmpBackdrop.BitBlt(bmp1, 0, 0);
bmpInitial.BitBlt(bmp1, 0, 0);
gB.FillEllipse(new RectangleF(50, 50, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(200, 50, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(50, 200, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(200, 200, 250, 250), Color.LightGray);
bmp.BitBlt(bmpBackdrop, 0, 500);

// Non-isolated, Non-knockout:
bmpBackdrop.BitBlt(bmp1, 0, 0);
gB.Renderer.BackgroundBitmap = null;
gB.FillEllipse(new RectangleF(50, 50, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(200, 50, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(50, 200, 250, 250), Color.LightGray);
gB.FillEllipse(new RectangleF(200, 200, 250, 250), Color.LightGray);
bmp.BitBlt(bmpBackdrop, 500, 500);

// Adornments:
using var g = bmp.CreateGraphics();
g.DrawLine(0, 500, 1000, 500, new Pen(Color.Black));
g.DrawLine(500, 0, 500, 1000, new Pen(Color.Black));
var tf = new TextFormat() { FontSize = 14 };
g.DrawString("Isolated, Knockout", tf, new RectangleF(0, 460, 500, 30),
TextAlignment.Center, ParagraphAlignment.Center, false);
g.DrawString("Isolated, Non-knockout", tf, new RectangleF(500, 460, 500, 30),
TextAlignment.Center, ParagraphAlignment.Center, false);
g.DrawString("Non-isolated, Knockout", tf, new RectangleF(0, 960, 500, 30),
TextAlignment.Center, ParagraphAlignment.Center, false);
g.DrawString("Non-isolated, Non-knockout", tf, new RectangleF(500, 960, 500, 30),
```

```
TextAlignment.Center, ParagraphAlignment.Center, false);  
  
// Done:  
bmp.SaveAsPng("isolated-knockout.png");
```

Back to Top

 **Note:** Please dispose off the **BackgroundBitmap** and **TransparencyMaskBitmap** bitmaps after using them.

Limitation

The bitmaps assigned to the **BackgroundBitmap** or **TransparencyMaskBitmap** properties must be of the same pixel size as the target bitmap of the current **GcBitmapGraphics**.

Work with Graphics

Graphics are visual elements that can be displayed in the form of different shapes, lines, curves or images. Additionally, graphics can be composed of paths as well. A graphic path is a sequence of connected lines and curves which work as a single graphics object. DslImaging allows you to draw these shapes and graphics path using GcGraphics class methods.

In this section, you learn how to work with the following:

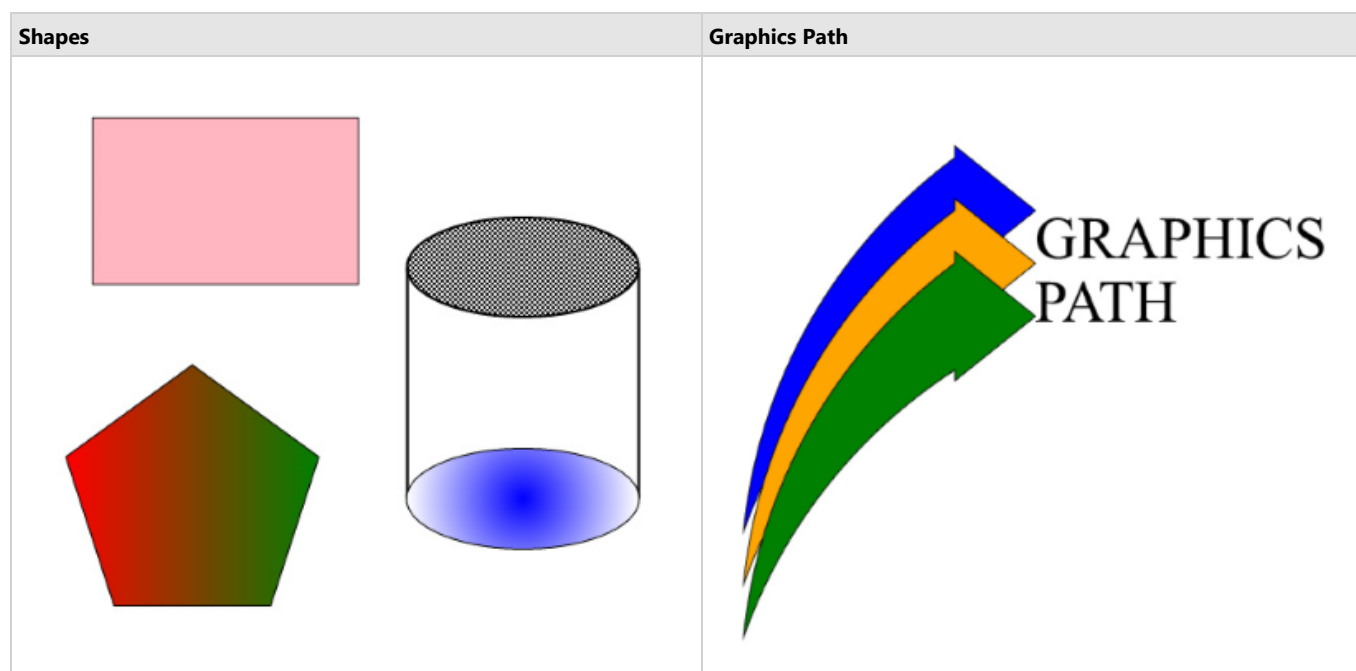
- [Draw and Fill Shapes](#)
- [Clip Region](#)
- [Align Image](#)
- [Add Matrix Transformation](#)
- [Add Transparency Layer](#)
- [Interpolation Mode](#)
- [Add Shadow](#)
- [Add Glow and Soft Edges](#)

Draw and Fill Shapes

DsImaging provides various drawing methods in **GcGraphics** class to draw graphic elements (shapes) on a drawing surface using an object of **GcBitmapGraphics** class. These shapes can be simple shapes, such as line, rectangle, etc. or complex shapes, such as graphics path, which can be any shape created using a sequence of connected lines and curves. All these shapes are drawn using draw methods available in the **GcGraphics** class. These draw methods accept a color or a Pen object as a parameter.

Moreover, DsImaging allows you to fill the shapes using fill methods available in the **GcGraphics** class. These methods fill the shapes using a color or a brush, which can be either **SolidBrush**, **LinearGradientBrush**, **RadialGradientBrush**, or **HatchBrush**. An instance of a required brush can be passed as a parameter to the fill methods.

Shape	Draw methods	Fill methods
Line	DrawLine	-
Rectangle	DrawRectangle	FillRectangle
Rounded rectangle	DrawRoundRect	FillRoundRect
Ellipse	DrawEllipse	FillEllipse
Polygon	DrawPolygon	FillPolygon
Path	DrawPath	FillPath



Draw Shapes

To draw a rectangle, polygon, and cylinder:

1. Initialize the GcBitmap class.
2. Create a drawing surface to draw shapes using **CreateGraphics** method of the GcBitmap class which returns an instance of the GcBitmapGraphics class.
3. Define Pen for drawing shapes using the **Pen** class.
4. Draw a rectangle and a pentagon using **DrawRectangle** and **DrawPolygon** methods of GcGraphics class.
5. Draw a cylinder with the help of line and ellipse using **DrawLine** and **DrawEllipse** methods of GcGraphics class.

C#

```
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(500, 500, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);
```

```
//Define the start point and pen for drawing shapes
Pen shapePen = new Pen(Color.Black, 2);

//Draw rectangle
//Start point for rectangle i.e. the upper left corner
PointF startPoint = new PointF(50, 50);
RectangleF rectangleBounds = new RectangleF(startPoint,
    new SizeF(200, 125));
g.DrawRectangle(rectangleBounds, shapePen);

//Draw Pentagon
PointF center_Pent = new PointF(125, 337);

//Defining distance of side from center and angle to start at
float radius = 100, startAngle = (float)-Math.PI / 2;

//Number of sides for polygon
int n = 5;
PointF[] pts = new PointF[n];

//Defining the connecting points for the sides calculated
//using the radius and start angle
for (int i = 0; i < 5; ++i)
    pts[i] = new PointF(center_Pent.X +
        (float)(radius * Math.Cos(startAngle + 2 * Math.PI * i / n)),
        center_Pent.Y +
        (float)(radius * Math.Sin(startAngle + 2 * Math.PI * i / n)));
g.DrawPolygon(pts, shapePen);

//Draw Cylinder
// Horizontal radius for ellipse
float radX = 87.5f;

// Vertical radius for ellipse
float radY = 37.5f;

//Cylinder Height
float height = 250;

//Center point for cylinder shape
PointF center_cyl = new PointF(375, 250);

//Rendering two ellipses and two lines to render cylinder shape
//Rectangle bounds/startpoint/end point are calculated based
//on the center point of the shape
g.DrawEllipse(new RectangleF(center_cyl.X - radX,
    center_cyl.Y - height / 2, radX * 2, radY * 2), shapePen);
g.DrawEllipse(new RectangleF(center_cyl.X - radX,
    center_cyl.Y + height / 2 - radY * 2, radX * 2,
    radY * 2), shapePen);
g.DrawLine(new PointF(center_cyl.X - radX,
    center_cyl.Y - height / 2 + radY),
    new PointF(center_cyl.X - radX, center_cyl.Y +
    height / 2 - radY), shapePen);
g.DrawLine(new PointF(center_cyl.X + radX,
    center_cyl.Y - height / 2 + radY), new PointF(center_cyl.X +
    radX, center_cyl.Y + height / 2 - radY), shapePen);

//Save the image rendering different shapes
origBmp.SaveAsJpeg("DrawShape.jpeg");
}
```


[Back to Top](#)

Fill Shapes

To fill different shapes with different types of brushes:

1. Initialize an instance of **SolidBrush** class to fill rectangle with a solid color.
2. Apply the background color to the rectangle using **FillRectangle** method of GcGraphics class which accepts the instance of SolidBrush as its parameter.
3. Similarly, fill the remaining shapes as well by passing the instance of the required brush as a parameter to the corresponding method.

```
C#
//Initialize an instance of SolidBrush class to fill
//rectangle with solid color
SolidBrush solidBrush = new SolidBrush(Color.LightPink);
g.FillRectangle(rectangleBounds, solidBrush);

//Initialize an instance of LinearGradientBrush class to
//fill pentagon with linear gradient
LinearGradientBrush linearBrush = new
    LinearGradientBrush(Color.Red, Color.Green);
g.FillPolygon(pts, linearBrush);

//Initialize an instance of HatchBrush class to fill
//cylinder top ellipse with hatch style
HatchBrush hatchBrush = new HatchBrush(HatchStyle.Diagonal);
g.FillEllipse(topEllipse, hatchBrush);

//Initialize an instance of RadialGradientBrush class
//to fill bottom ellipse with radial gradient
RadialGradientBrush radialBrush = new
    RadialGradientBrush(Color.Blue, Color.White);
g.FillEllipse(bottomEllipse, radialBrush);

//Save the image rendering different shapes
origBmp.SaveAsJpeg("FillShape.jpeg");
```

[Back to Top](#)

Draw and Fill Path

To draw a graphics path:

1. Create a graphics path using **CreatePath** method of the **GcGraphics** class which returns an instance of **IPath** interface.
2. Create a new figure for the path starting at a specified point using **BeginFigure** method of the **IPath** interface.
3. Add arcs and lines to the figure using **AddArc** and **AddLine** methods of the **IPath** interface for completing a graphics path.
4. Close the figure using **EndFigure** method of the **IPath** interface to complete the graphics path.
5. Return the graphics path.
6. Draw the graphics path using the **DrawPath** method of GcGraphics class which accepts a specified pen as its parameter.
7. Apply background color to the path using **FillPath** method of GcGraphics class which accepts specified color as its parameter.

```
C#
//Define and return the graphic path
public IPath CreatePath(RectangleF rec, GcGraphics g, SizeF sz)
{
    var path = g.CreatePath();
    path.BeginFigure(new PointF(rec.X + 50, rec.Y + rec.Height));
    path.AddArc(new ArcSegment() { Point = new
        PointF(rec.X + 250, rec.Y + 50), RotationAngle = 30,
        SweepDirection = SweepDirection.Clockwise, Size = sz });
    path.AddLine(new PointF(rec.X + 250, rec.Y + 40));
    path.AddLine(new PointF(rec.X + 325, rec.Y + 100));
    path.AddLine(new PointF(rec.X + 250, rec.Y + 160));
    path.AddLine(new PointF(rec.X + 250, rec.Y + 150));
}
```

```
path.AddArc(new ArcSegment() { Point = new
    PointF(rec.X + 50, rec.Y + rec.Height),
    RotationAngle = 30, SweepDirection =
    SweepDirection.CounterClockwise, Size = sz });
path.EndFigure(FigureEnd.Closed);

return path;
}

//Create an image using the Graphic Path
public void DrawPath()
{
    //Initialize GcBitmap
    GcBitmap origBmp = new GcBitmap(640, 530, true);

    //Create the graphics for the Bitmap
    GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

    //Define the start point and pen for drawing shapes
    Pen shapePen = new Pen(Color.Black, 2);

    //Size of the graphics path image
    SizeF sz = new SizeF(500, 500);

    RectangleF rect1 = new RectangleF(0, 0, 500, 400);
    var path1 = CreatePath(rect1, g, sz);

    g.DrawPath(path1, shapePen); // Draw graphic path
    g.FillPath(path1, Color.Blue); // Fill graphic path

    RectangleF rect2 = new RectangleF(0, 50, 500, 400);
    var path2 = CreatePath(rect2, g, sz);
    g.DrawPath(path2, shapePen);
    g.FillPath(path2, Color.Orange);

    RectangleF rect3 = new RectangleF(0, 100, 500, 400);
    var path3 = CreatePath(rect3, g, sz);
    g.DrawPath(path3, shapePen);
    g.FillPath(path3, Color.Green);

    //Define TextFormat to render text in the image
    TextFormat tf = new TextFormat
    {
        Font = Font.FromFile(Path.Combine("Resources",
            "Fonts", "times.ttf")),
        FontSize = 42
    };

    g.DrawString("GRAPHICS", tf, new PointF(325, 95));
    g.DrawString("PATH", tf, new PointF(325, 155));

    //Save the image rendering different shapes
    origBmp.SaveAsJpeg("GraphicPath.jpeg");
}
```

[Back to Top](#)

Antialiasing

Dslmaging, by default, renders the graphics in fast antialiasing mode that gives the good quality result with fast rendering. However, if you want to render the graphics in slow antialiasing mode to get the highest quality while compromising on the speed, you can set the **SlowAntialiasing** property of **BitmapRenderer** class to **true**. Similarly, you can also render the graphics without antialiasing which gives you poor quality but very fast rendering, by setting the **Aliased** property to **true**.

In addition, `BitmapRenderer` class also provides **ForceAntialiasingForText** property which when **true**, forces the text layout to draw with antialiasing even if the **Aliased** property is set to true. This is generally required in scenarios where graphics are required to be aliased but text needs to be antialiased. For instance, in the case of rendering barcodes with digits, barcodes should be aliased to make them crisp and readable by devices while digits under the barcodes needs to be drawn with better quality.

To render a text with slow antialiasing on an image:

1. Initialize the `GcBitmap` class.
2. Create a drawing surface using **CreateGraphics** method of the `GcBitmap` class which returns an instance of the `GcBitmapGraphics` class.
3. Create an instance of **TextLayout** class using the **CreateTextLayout** method.
4. Set the **SlowAntialiasing** property to true to render a good quality text with fast speed.

```
C#
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(1000, 500, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

//Render using multithreaded mode
g.Renderer.Multithreaded = true;

var text = @"Different(anti)aliasing modes of rendering
            text are no antialiasing, fast antialiasing
            and slow antialiasing.";

var tfcap = new TextFormat()
{
    Font = Font.FromFile(Path.Combine("Resources", "Fonts",
        "timesbd.ttf")),
    FontSize = 16,
};


var tl = g.CreateTextLayout();
tl.TextAlignment = TextAlignment.Justified;
//Render text without antialiasing
//origBmp.Renderer.Aliased = true;

//Render text with slow antialiasing
origBmp.Renderer.SlowAntialiasing = true;
tl.AppendLine("Fast antialiasing (default quality)", tfcap);
tl.Append(text, tfcap);
g.DrawTextLayout(tl, new PointF(50, 200));

//Save the image depicting different antialiasing modes
origBmp.SaveAsJpeg("Antialiasing.jpeg");
```

Back to Top

For more information about drawing and filling geometric shapes using `DslImaging`, see [DslImaging sample browser](#).

 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Clip Region

A clip region refers to a specific part of an image and is defined to limit the drawing operations for an image to a specific part of the image.

In DsImaging, the clip region can be created using the **CreateClipRegion** method of the **GcBitmapGraphics** class which takes a **Rectangle** or a **GraphicsPath** as a parameter. The defined clip region is applied to the image using the **PushClip** method which limits the drawing operations to the clip region. After the required drawing operations are done, you can use the **PopClip** method to remove the clip region and make the complete image surface available for any further drawing operations.



To clip an image:

1. Create an instance of GcBitmap.
2. Load an image into the GcBitmap instance.
3. Define a clip rectangle for adding text.
4. Add text to the rectangle.
5. Pass the rectangle as a parameter in **PushClip** method of the GcBitmapGraphics class.

C#

```
var backColor = Color.FromArgb(unchecked((int)0xff0066cc));  
var foreColor = Color.FromArgb(unchecked((int)0xffffcc00));
```

```
float cw = 450, ch = 300, pad = 10, bord = 4;
int pixelWidth = 1024, pixelHeight = 1024;

GcBitmap origBmp = new GcBitmap(pixelWidth, pixelHeight, true);
var path = Path.Combine("Resources", "Images", "tudor.jpg");
origBmp.Load(path);

GcBitmapGraphics g = origBmp.CreateGraphics();


RectangleF clipRc = new RectangleF(pixelWidth - cw - pad,
    pad, cw, ch);

using (g.PushClip(clipRc))
{
    g.FillRectangle(clipRc, Color.Blue);
    g.DrawString("This is a beautiful home",
        new TextFormat()
        {
            Font = Font.FromFile(Path.Combine("Resources",
                "Fonts", "times.ttf")),
            FontSize = 16,
            ForeColor = foreColor
        },
        clipRc
    );
}

//Save the image with clipped region
origBmp.SaveAsJpeg("ClipImageTest.jpeg");
```

Back to Top

For more information about implementation of clipping using DslImaging, see [DslImaging sample browser](#).

 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Align Image

DsImaging provides you an option to set the alignment of an image within its container using the **ImageAlign** class. This class provides you an option to center, scale, or stretch an image with respect to the bitmap.



To align an image at the center of its container:

1. Initialize the GcBitmap class.
2. Create a drawing surface to draw shapes using **CreateGraphics** method of the **GcBitmap** class which returns an instance of the **GcBitmapGraphics** class.
3. Invoke the **DrawImage** method of GcGraphics class to draw an image and set the image alignment to center using the **ImageAlign** class to pass it as a parameter to the method.

C#

```
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(1000, 1000, true);


//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.AliceBlue);

//Get the image and define the image rectangle
var image = Image.FromFile(Path.Combine("Resources", "Images", "tudor.jpg"));
var imgRec = new Rectangle(50, 50, image.Width + 100, image.Height + 100);
```

```
g.FillRectangle(imgRec, Color.Gray);

//Draw the image with "CenterImage" alignment mode
g.DrawImage(image, imgRec, null, ImageAlign.CenterImage);

//Save the image
origBmp.SaveAsJpeg("AlignImageCenter.jpeg");
```

 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Apply Matrix Transformation

Transformation plays a vital role when it comes to graphics. The purpose of using transformation in graphics is to reposition the graphics and alter their orientation and size. It may involve a sequence of operations such as, translation, scaling, rotation, etc..

DsImaging supports graphics transformation through **Transform** property of the **GcBitmapGraphics** class which is of type **Matrix3x2**. The **Matrix3x2** struct represents a 3x2 matrix and is a member of **System.Numerics** namespace. The transformations are applied in the order reverse to which they are added to the matrix.



To apply matrix transformation:

1. Initialize the **GcBitmap** class.
2. Create a drawing surface using **CreateGraphics** method of the **GcBitmap** class which returns an instance of the **GcBitmapGraphics** class.
3. Draw a rectangle using **DrawRectangle** method and apply the background color using **FillRectangle** method of the **GcBitmapGraphics** class.
4. Define the text to be rendered in a rectangle.
5. Add text to the rectangles using **DrawString** method of the **GcBitmapGraphics** class
6. Create a transformation matrix with different transformation types. For example, create rotation, translation, and scaling matrix using **CreateRotation**, **CreateTranslation** and **CreateScale** method of the **Matrix3x2** class respectively.
7. Apply the transformation matrix using the **Transform** property.

Note that the sequence of transformations applied to the text is done in reverse order, which means first is scaling followed by translation and rotation.

C#

```
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(1024, 1024, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

//Define text to be rendered in box/rectangle
const string baseTxt = "Text drawn at (10,36) in a 4\"x2\" box";
var Inch = origBmp.DpiX;

// Render the image with tranformed text
// Transforms are applied in order from last to first.]
```




```
var rotate = Matrix3x2.CreateRotation((float)(-70 * Math.PI) / 180f);
var translate = Matrix3x2.CreateTranslation(Inch * 3, Inch * 5);
var scale = Matrix3x2.CreateScale(0.7f);

g.Transform =
    rotate *
    translate *
    scale;

var box = new RectangleF(10, 36, origBmp.DpiX * 4, origBmp.DpiY * 2);
g.FillRectangle(box, Color.FromArgb(80, 0, 184, 204));
g.DrawRectangle(box, Color.FromArgb(0, 193, 213), 1);
box.Inflate(-6, -6);
g.DrawString(baseTxt, new TextFormat()
{
    Font = Font.FromFile(Path.Combine("Resources", "Fonts",
        "times.ttf")),
    FontSize = 14,
},
box);

//Save the image rendering different shapes
origBmp.SaveAsJpeg("MatrixTransform.jpeg");
```

For more information about using transformation matrix in DslImaging, see [DslImaging sample browser](#).

 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Add Transparency Layer

DsImaging allows you to add a transparency layer to your drawings and hence lets you manipulate a group of drawing operations.

The GcGraphics class provides **PushTransparencyLayer** method that allows you to add new layers to the GcGraphics object which receives all subsequent drawing operations until the **PopTransparencyLayer** method is called. On calling the **PopTransparencyLayers** method, the contents of the layers are merged into the drawing surface.



When pushing a transparency layer, you can specify the bound of new layer so that the subsequent operations are performed inside the specified bounds only. When bounds are not specified, bounds of the original drawing surface are considered as bounds of the layer and all the layer operations take effect on the whole surface.

You can also pass opacity of the transparency layer as a parameter of PushTransparencyLayer method. The opacity of layer gets composited to the drawing surface when the layer is popped out.

The example below uses GcGraphics to create a drawing and then merges additional operations using a transparency layer. Similarly, transparency layers can be used with the following classes derived from GcGraphics class:

GcPdfGraphics, GcBitmapGraphics, GcSkiaGraphics, GcSvgGraphics, GcD2DBitmapGraphics, GcWicBitmapGraphics.

C#

```
// Draw a figure with layer using GcGraphics
static void DrawFigure(GcGraphics g)
{
    const float DegToRad = (float)(Math.PI / 180);
    var m = Matrix3x2.CreateScale(g.Resolution / 48f);
    m = Matrix3x2.CreateSkew(DegToRad * 30, 0) * m;
    m = Matrix3x2.CreateRotation(DegToRad * 30) * m;
    g.Transform = m;

    var rect = new RectangleF(50, 50, 150, 100);
```

```
g.FillRectangle(rect, new HatchBrush(HatchStyle.ZigZag)
{
    BackColor = Color.Yellow,
    ForeColor = Color.Purple
});
g.DrawRectangle(rect, new GrapeCity.Documents.Drawing.Pen(Color.LightGreen, 6f));

//Define bounds and push an opaque transparency layer
var clipRect = new RectangleF(70, 50, 110, 100);
g.PushTransparencyLayer(clipRect, 0.5f);

g.FillRectangle(rect, Color.Green);
rect.Height -= 60;
rect.Y += 30;
g.DrawRectangle(rect, new GrapeCity.Documents.Drawing.Pen(Color.Red, 6f));

var tl = g.CreateTextLayout();
tl.DefaultFormat.ForeColor = Color.White;
tl.DefaultFormat.FontSize = 38;
tl.DefaultFormat.FontSizeInGraphicUnits = true;
tl.DefaultFormat.FontName = "Tahoma";
tl.MaxWidth = 150;
tl.TextAlignment = TextAlignment.Center;
tl.Append("Hello World!");
g.DrawTextLayout(tl, new PointF(50, 50));

// merge the transparency layer
g.PopTransparencyLayer();
}
```

Limitations: Browsers such as Google Chrome and Mozilla Firefox may not notice the changes in the SVG image generated from the contentBounds parameter passed to the GcSvgGraphics.PushTransparencyLayer method.

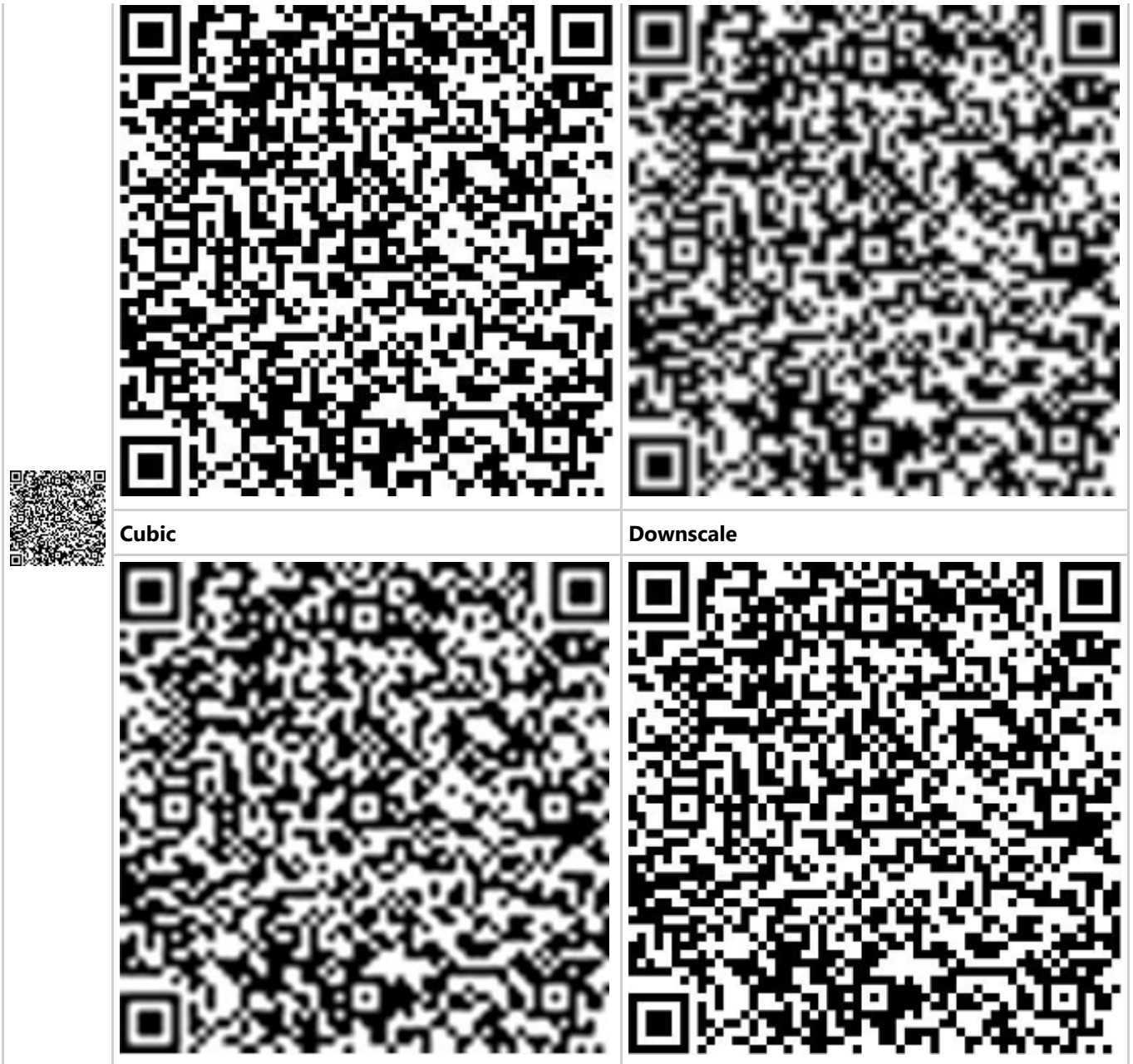
Interpolation Mode

Sometimes, you might want to draw an image using nearest-neighbor interpolation mode without resizing the image. The interpolation mode influences the way images stretch and shrink. The interpolation modes try to attain the best estimation of the intensity of a pixel based on neighboring pixel values on a proximity basis. To stretch an image, each pixel in the original image must be mapped to a group of pixels in the larger image. However, to shrink an image, groups of pixels in the original image must be mapped to single pixels in the smaller image. Dslmaging supports interpolation modes to allow you to control interpolation mode in a common way for all implementations of **GcGraphics** class.

The GcGraphics class provides **InterpolationMode** property and **IsInterpolationModeSupported** method, which are used to control interpolation mode in a common way for all graphics. The **InterpolationMode** enumeration defines following interpolation modes:

- NearestNeighbor
- Linear
- Cubic
- Downscale

Original Image	NearestNeighbor	Linear
----------------	-----------------	--------



The following table lists the interpolation modes supported by different GcGraphics implementations.

GcGraphics Implementations	NearestNeighbor	Linear	Cubic	Downscale
GcPdfGraphics	Yes	No	No	No
GcBitmapGraphics	Yes	Yes	Yes	Yes
GcSvgGraphics	Yes	Yes	Yes	No
GcSkiaGraphics	Yes	Yes	Yes	No
GcWicBitmapGraphics	Yes	Yes	No	No
GcD2DBitmapGraphics	Yes	Yes	No	No

The following are the usage examples of the InterpolationMode enumeration in different GcGraphics implementations:

- The current value of the `InterpolationMode` property in `GcGraphics` class affects the way bitmap images are drawn by the **DrawImage** method in `GcGraphics` class.
- The current value of the `InterpolationMode` property in **BitmapRenderer** class affects the way bitmap images are drawn by the **DrawBitmap** method in `BitmapRenderer` class.
- The **Resize** method of **GcBitmap** class accepts `InterpolationMode` as a parameter, which affects the resulting `GcBitmap` image.

Refer to the following example code to enlarge a small image using different interpolation modes:

```
C#
public class EnlargeQRCode
{
    public GcBitmap GenerateImage(Size pixelSize, float dpi, bool opaque, string[]
sampleParams = null)
    {
        // Create and clear the target bitmap.
        var targetBmp = new GcBitmap(pixelSize.Width, pixelSize.Height, opaque, dpi,
dpi);
        targetBmp.Clear(Color.Transparent);

        const int fontSize = 16;
        var xpad = (int)(dpi * .5f);
        var ypad = (int)(dpi * .7f);

        // Initialize Font.
        TextFormat tf = new TextFormat
        {
            Font = GCTEXT.Font.FromFile(Path.Combine("Resources", "Fonts",
"times.ttf")),
            FontSize = fontSize,
        };

        // Load the image.
        using var origBmp = new GcBitmap();
        using (var stm = File.OpenRead(Path.Combine("Resources", "ImagesBis",
"QRCode-57x57.png")))
            origBmp.Load(stm);

        // Match the opaqueness of the original bitmap and the target.
        origBmp.Opaque = targetBmp.Opaque;

        var ip = new Point(xpad, ypad);

        // Draw the original image at its original size.
        targetBmp.BitBlt(origBmp, ip.X, ip.Y);
        using (var g = targetBmp.CreateGraphics(null))
            g.DrawString($"← Original image ({origBmp.PixelWidth} by
{origBmp.PixelHeight} pixels)", tf, new PointF(xpad * 2 + origBmp.Width, ip.Y));
        ip.Y += origBmp.PixelHeight + ypad;

        // Enlarge the original small image by a factor of 6.
        var f = 6;
    }
}
```

```
int twidth = origBmp.PixelWidth * f;
int theight = origBmp.PixelHeight * f;

// Enlarge and draw four copies of the image using the four different
// available interpolation modes.
using (var bmp = origBmp.Resize(twidth, theight,
InterpolationMode.NearestNeighbor))
    targetBmp.BitBlt(bmp, ip.X, ip.Y);
drawCaption("InterpolationMode.NearestNeighbor", ip.X, ip.Y + theight);


using (var bmp = origBmp.Resize(twidth, theight, InterpolationMode.Cubic))
    targetBmp.BitBlt(bmp, ip.X + twidth + xpad, ip.Y);
drawCaption("InterpolationMode.Cubic", ip.X + twidth + xpad, ip.Y + theight);

ip.Y += theight + ypad;

using (var bmp = origBmp.Resize(twidth, theight, InterpolationMode.Linear))
    targetBmp.BitBlt(bmp, ip.X, ip.Y);
drawCaption("InterpolationMode.Linear", ip.X, ip.Y + theight);

using (var bmp = origBmp.Resize(twidth, theight,
InterpolationMode.Downscaled))
    targetBmp.BitBlt(bmp, ip.X + twidth + xpad, ip.Y);
drawCaption("InterpolationMode.Downscaled", ip.X + twidth + xpad, ip.Y +
theight);

void drawCaption(string caption, float x, float y)
{
    using var g = targetBmp.CreateGraphics(null);
    g.DrawString(caption, tf, new PointF(x, y));
}
return targetBmp;
}
}
```

 **Note:** The interpolation mode only affects the way raster images are drawn on a graphic, i.e., the result of DrawImage method and raster image resizing. Interpolation mode does not affect any other graphics operations. In particular, if a PDF is saved to an image format, the only items affected by interpolation mode would be raster images embedded in the original PDF, if they exist.

When a raster image is drawn on an SVG graphic (a vector graphic), the original raster image is not modified; instead, the specified interpolation mode is saved in the SVG markup as a hint to viewers on how to show the image, so it is not directly affected by the interpolation mode. The hint may be ignored, depending on the viewer. Graphics on a PDF are also vector graphics, but these graphics only support NearestNeighbor mode of InterpolationMode, meaning that raster images embedded in a PDF are always shown by PDF viewers using that mode.

Add Shadow

DsImaging provides **ApplyGaussianBlur** and **ToShadowBitmap** methods in **GrayscaleBitmap** class. These methods make the process of drawing an image with a shadow easier and more straightforward. The **ApplyGaussianBlur** method accepts the **borderColor**, **radius**, and **borderMode** arguments. The **ToShadowBitmap** method simplifies moving a transparency mask from **GrayscaleBitmap** to a **GcBitmap**, and has the ability to pass the **shadowColor** and **opacity factor**. Also, the **ToShadowBitmap** always treats the **GrayscaleBitmap** as a transparency mask, even if it was created from any color channel (not necessarily the alpha channel).

To add a shadow to an image:

1. Draw an image with some shapes and text.



C#

```
// Initialize GcBitmap.
using var bmp = new GcBitmap(800, 600, false);

// Draw an image.
using (var g = bmp.CreateGraphics(Color.AliceBlue))
{
    Draw(g, 0, 0);
}

// Save the image without a shadow.
bmp.SaveAsPng("WithoutShadow.png");

static void Draw(GcGraphics g, float offsetX, float offsetY)
{
    // Define the transformation matrix.
    var baseT = Matrix3x2.CreateTranslation(offsetX, offsetY);
    g.Transform = baseT;

    // Draw an ellipse.
```



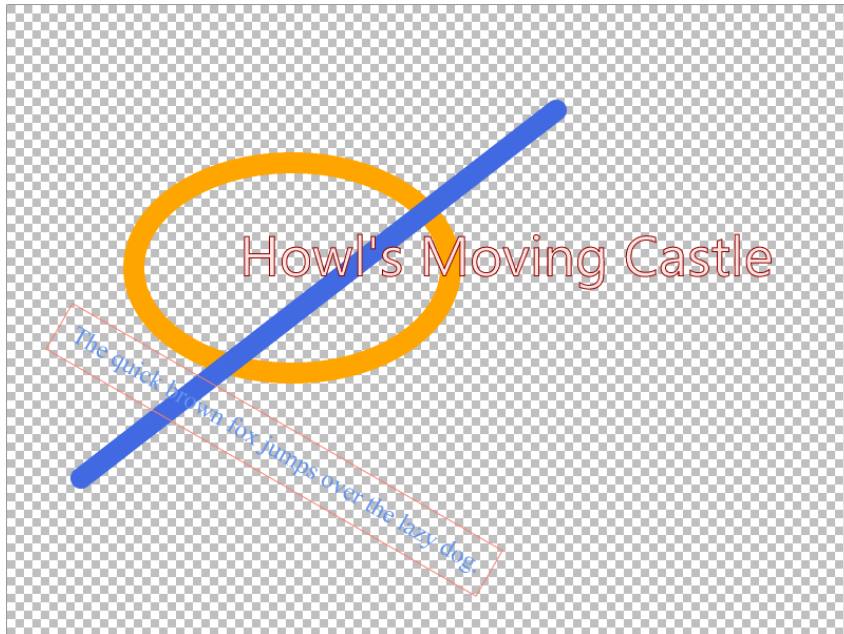
```
g.DrawEllipse(new RectangleF(100, 100, 300, 200),
    new Pen(Color.Orange, 20f));

// Draw a line.
g.DrawLine(new PointF(50, 400), new PointF(500, 50),
    new Pen(Color.RoyalBlue, 20f)
    {
        LineCap = PenLineCap.Round
    });

// Draw strings.
g.DrawString("Shadow",
    new TextFormat
    {
        FontName = "Segoe UI",
        FontSize = 40,
        ForeColor = Color.MistyRose,
        StrokePen = new Pen(Color.DarkRed, 1f)
    },
    new PointF(200, 150));
g.Transform = Matrix3x2.CreateRotation((float)(Math.PI / 6)) *
    (Matrix3x2.CreateTranslation(50, 250) * baseT);
g.DrawString("The shadow is added to both text and shapes.",
    new TextFormat
    {
        FontName = "Times New Roman",
        FontSize = 18,
        ForeColor = Color.CornflowerBlue
    },
    new PointF(0, 0));

// Draw a rectangle.
g.DrawRectangle(new RectangleF(-15, -10, 470, 50),
    new Pen(Color.Salmon, 1f));
}
```

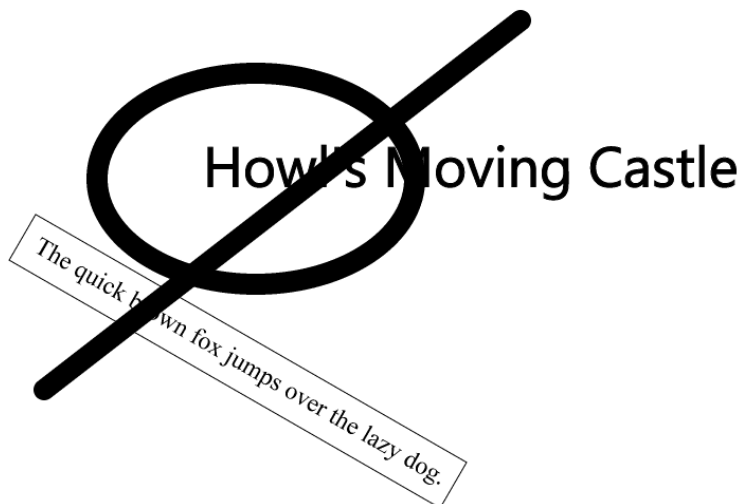
2. Draw the image on a transparent background with an offset for the shadow.



C#

```
// Draw the image to the transparent background with an offset for shadow.  
using (var g = bmp.CreateGraphics(Color.Transparent))  
{  
    Draw(g, 20, 50);  
}
```

3. Extract the alpha channel from GcBitmap to a GrayscaleBitmap.



C#

```
// Extract the alpha channel from GcBitmap to a GrayscaleBitmap.  
using var gs = bmp.ToGrayscaleBitmap(ColorChannel.Alpha);
```

4. Apply some blur to the GrayscaleBitmap using the ApplyGaussianBlur method.



C#

```
// Apply some blur to GrayscaleBitmap.  
gs.ApplyGaussianBlur(9);
```

5. Convert the transparency mask from GrayscaleBitmap to GcBitmap, filling the opaque pixels with the shadow color. Draw the transparency mask into the same bitmap using the ToShadowBitmap; there is no need to create another GcBitmap instance.



C#

```
// Convert the transparency mask from GrayscaleBitmap to GcBitmap. Apply an  
additional opacity factor.  
gs.ToShadowBitmap bmp, Color.CadetBlue, 0.4f);
```

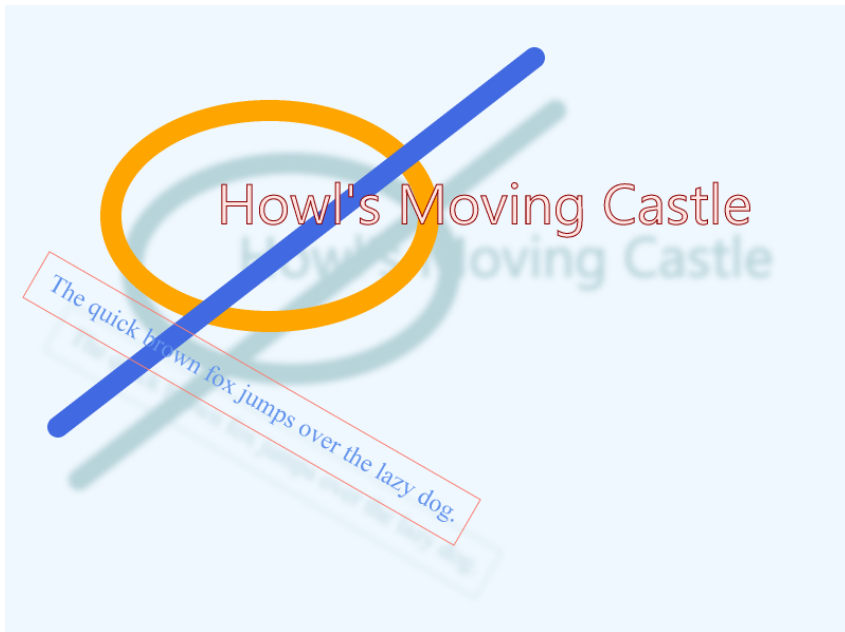
6. Substitute the transparent background with an opaque background color.



C#

```
// Substitute the transparent background with an opaque background color.  
bmp.ConvertToOpaque(Color.AliceBlue);
```

7. Draw the main image onto the shadow image and save the image.

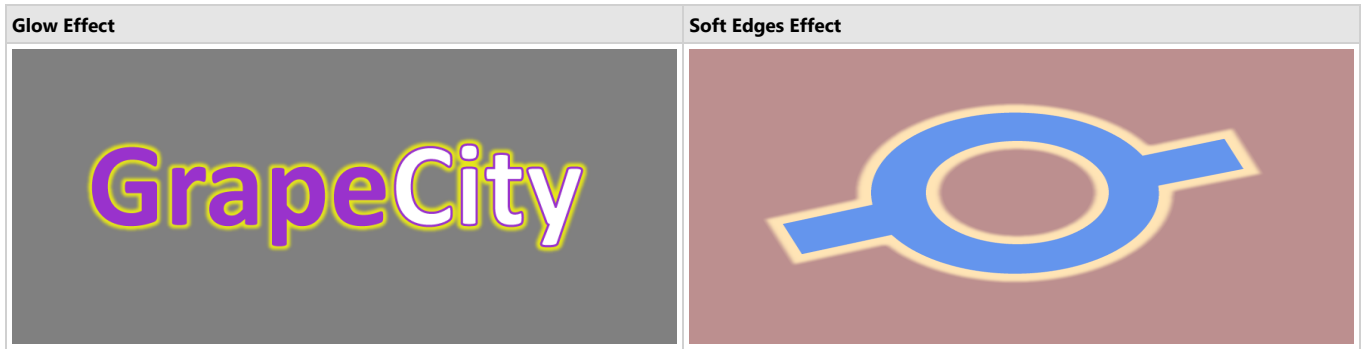


C#

```
// Draw the main image.  
using (var g = bmp.CreateGraphics())  
{  
    Draw(g, 0f, 0f);  
}  
  
// Save the image with a shadow.  
bmp.SaveAsPng("WithShadow.png");
```

Add Glow and Soft Edges

DslImaging provides **ApplyGlow** method in **GrayscaleBitmap** class that will be used to add glow effect as well as soft edges to graphics. The glow effect inflates all non-transparent areas of the image by the specified amount and then applies the Gaussian blur to make the border smooth. The soft edges effect deflates non-transparent areas, then applies the Gaussian blur. The glow and soft edges effects are usually applied to a full-color image within a **GcBitmap**. The first parameter of the method is set to a positive value (for glow effect) or a negative value (for soft edges effect) as per the requirement.



Refer to the following example to apply the glow effect:

```

C#
// Initialize TextLayout.
var tl = new TextLayout(96f);

// Configure text format.
var f1 = new TextFormat
{
    FontName = "Calibri",
    FontSize = 120,
    ForeColor = Color.DarkOrchid,
    FontBold = true
};
var f2 = new TextFormat(f1)
{
    ForeColor = Color.White,
    StrokePen = new Pen(Color.DarkOrchid, 3)
};

// Append the text.
tl.Append("Grape", f1);
tl.Append("City", f2);

// Initialize GcBitmap.
using var bmp = new GcBitmap(880, 390, false);

// Create the graphic using text layout defined.
using (var g = bmp.CreateGraphics(Color.DarkGray))
{
    g.DrawTextLayout(tl, new PointF(100, 80));
}

// Save the image without glow.
bmp.SaveAsPng("WithoutGlow.png");

// Draw the text on a transparent bitmap at first.
using (var g = bmp.CreateGraphics(Color.Transparent))
{
    g.Renderer.SlowAntialiasing = true;
    g.DrawTextLayout(tl, new PointF(100, 80));
}

// Convert the image to a transparency mask.
using var gs = bmp.ToGrayscaleBitmap(ColorChannel.Alpha);

// Apply the glow effect to the transparency mask.
gs.ApplyGlow(4, 6);
    
```

```
/* Map a shadow from the transparency mask to the source GcBitmap drawing opaque pixels with glow color (yellow).
Apply some additional transparency.*/
gs.ToShadowBitmap bmp, Color.Yellow, 0.8f);

// Fill the background.
bmp.ConvertToOpaque(Color.Gray);

// Draw the text over the prepared background.
using (var g = bmp.CreateGraphics())
{
    g.Renderer.SlowAntialiasing = true;
    g.DrawTextLayout(tl, new PointF(100, 80));
}

// Save the image with glow.
bmp.SaveAsPng("WithGlow.png");
```

Refer to the following example to apply the soft edges effect:

```
C#

// Initialize PathBuilder.
var pb = new PathBuilder();
pb.BeginFigure(100, 350);
pb.AddLine(210, 310);

// Define an arc.
var arc = new ArcSegment
{
    Size = new SizeF(183, 173),
    SweepDirection = SweepDirection.Clockwise,
    Point = new PointF(550, 205),
};

// Add arcs and lines.
pb.AddArc(arc);
pb.AddLine(650, 170);
pb.AddLine(680, 250);
pb.AddLine(575, 285);
arc.Point = new PointF(240, 390);
pb.AddArc(arc);
pb.AddLine(130, 430);
pb.EndFigure(true);
pb.Figures.Add(new EllipticFigure(new RectangleF(295, 197, 200, 190)));
var gpFill = pb.ToPath();
var gpStroke = gpFill.Widen(new Pen(Color.Black, 20));

// Draw the image.
using var bmp = new GcBitmap(800, 600, false);
var renderer = bmp.EnsureRendererCreated();

bmp.Clear(Color.RosyBrown);
renderer.FillPath(gpFill, Color.CornflowerBlue);
renderer.FillPath(gpStroke, Color.Moccasin);

// Save the image without soft edges.
bmp.SaveAsPng("WithoutSoftEdges.png");

// Draw the figure on a transparent background.
bmp.Clear(Color.Transparent);
renderer.FillPath(gpFill, Color.CornflowerBlue);
renderer.FillPath(gpStroke, Color.Moccasin);

// Convert the image to the transparency mask.
using var gs = bmp.ToGrayscaleBitmap(ColorChannel.Alpha);

// Apply the soft edges effect to the transparency mask.
gs.ApplyGlow(-4, 8);

// Draw the original image.
```

```
bmp.Clear(Color.RosyBrown);
renderer.TransparencyMaskBitmap = gs;
renderer.FillPath(gpFill, Color.CornflowerBlue);
renderer.FillPath(gpStroke, Color.Moccasin);

// Save the image with soft edges.
bmp.SaveAsPng("WithSoftEdges.png");
```

Work with Text

Dslmaging allows you to draw text on an image through **GcBitmapGraphics** class of **Grapecity.Documents.Imaging** namespace. There are two ways in which we can render the text:

- **Using DrawString method:** The **DrawString** method is used when you simply need to draw a string at a specified location on an image. However, when there is a possibility that the string might not fit in the available space, you can use the **MeasureString** method in conjunction with the **DrawString** method. **MeasureString** method measures the string along with the width allocated to draw it and makes it possible to draw a string in the allocated space using the **DrawString** method.
- **Using TextLayout class:** This approach gives you more control over the text to be rendered and provides various advanced options such as formatting. In this approach, you create an instance of the **TextLayout** class and invoke the **Append** or **AppendLine** methods to add the text runs to the **TextLayout**. Finally, you can invoke the **DrawTextLayout** method, which uses the instance of **TextLayout** class to draw the text layout at a specified location.

Text rendered using the **DrawString** method:
Test string.



Text rendered using **MeasureString** method with **DrawString** method.

Text rendered using **TextLayout**. First test string added to **TextLayout**. Second test string added to **TextLayout**, continuing the same paragraph. Third test string added to **TextLayout**, a new paragraph. *Fourth test string, with a different char formatting.*

```
C#
var Inch = 96;
const float fontSize = 14;

//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(500, 500, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

//TextFormat to specify font and other character formatting
var tf = new TextFormat()
{
    Font = Font.FromFile(Path.Combine("Resources", "Fonts",
        "times.ttf")),
    FontSize = fontSize
};

//First Way:
//DrawString: Render text using DrawString method at
//a specific location
g.DrawString("Text rendered using the DrawString method:" +
    "\r\n Test string.", tf, new PointF(Inch, Inch));

//Using MeasureString method along with DrawString
//to have more control over text layout
```



```

const string tstr = "Text rendered using MeasureString method" +
    " with DrawString method.";

SizeF layoutSize = new SizeF(Inch * 3, Inch * 0.8f);
SizeF s = g.MeasureString(tstr, tf, layoutSize, out int fitCharCount);

// Show the passed in size in red, the measured size in blue,
// and draw the string within the returned size as bounds:
PointF pt = new PointF(Inch, Inch * 2);
g.DrawRectangle(new RectangleF(pt, layoutSize), Color.Red);
g.DrawRectangle(new RectangleF(pt, s), Color.Blue);
g.DrawString(tstr, tf, new RectangleF(pt, s));

// Second Way:
// TextLayout: A much more powerful and with better performance,
// way to render text
var tl = g.CreateTextLayout();
// To add text, use Append() or AppendLine() methods:
tl.Append("Text rendered using TextLayout. ", tf);
tl.Append("First test string added to TextLayout. ", tf);
tl.Append("Second test string added to TextLayout, continuing the" +
    " same paragraph. ", tf);
tl.AppendLine(); // Add a line break, effectively starting a new paragraph
tl.Append("Third test string added to TextLayout, a new paragraph. ", tf);
tl.Append("Fourth test string, with a different char formatting. ",
    new TextFormat(tf)
    {
        Font = Font.FromFile(Path.Combine("Resources", "Fonts",
            "timesbi.ttf")),
        FontSize = fontSize,
        FontBold = true,
        FontItalic = true,
        ForeColor = Color.DarkSeaGreen,
    });

//Setting layout options
tl.MaxWidth = g.Width - Inch * 2;

// Draw it on the page:
pt = new PointF(Inch, Inch * 3);
g.DrawTextLayout(tl, pt);

//Save the image rendering different shapes
origBmp.SaveAsJpeg("RenderText.jpeg");

```

[Back to Top](#)

Text Trimming and Wrapping

DsImaging supports text trimming and wrapping to handle the text that does not fit in the allocated space. The **TextLayout** class provides the **TrimmingGranularity** property which sets the text granularity as character or word and trims the over flowing text to display an ellipsis at the end. This property accepts value from the **TrimmingGranularity** enumeration and works in conjunction with the **WrapMode** property which provides the text wrapping options. To enable trimming, text wrapping should be disabled by setting the WrapMode property to **NoWrap**. The **WrapMode** property also provides options to wrap a text at the grapheme cluster boundaries or as per the Unicode line breaking algorithm.

This is a long line of text which does not fit in the allocat...

C#

```

var Inch = 96;
const float fontSize = 12;
var str = "This is a long line of text which does not fit in" +

```

```
        "the allocated space.");
var wid = Inch * 4;
var dy = 0.3f;
var ip = new PointF(50, 200);

//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(500, 500, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

//Rendering text with Character trimming:
TextLayout tl = g.CreateTextLayout();
tl.DefaultFormat.Font = Font.FromFile(Path.Combine("Resources",
    "Fonts", "times.ttf"));
tl.DefaultFormat.FontSize = fontSize;
tl.MaxWidth = wid;
tl.WrapMode = WrapMode.NoWrap;
tl.Append(str);
tl.TrimmingGranularity = TrimmingGranularity.Character;
g.DrawTextLayout(tl, ip);

//Render rectangle indicating the area which defines text trimming
g.DrawRectangle(new RectangleF(50, 200, wid, ip.Y - 200),
    Color.OrangeRed);

//Save the image rendering different shapes
origBmp.SaveAsJpeg("TrimText.jpeg");
```

[Back to Top](#)

Add Watermark

Dslmaging provides a mechanism to add watermarks on top of an image by rendering the watermark text using semi-transparent color. In order to render a watermark text, you can use the **DrawString** method which takes the text format as a parameter. This text format is represented by the **TextFormat** class and should have **ForeColor** property set to a semi-transparent color to render the string as a semi-transparent text, i.e, watermark.



C#

```
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(800, 800, true);
```

```
//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

Image image = Image.FromFile(Path.Combine("Resources",
    "Images", "reds.jpg"));
RectangleF rc = new RectangleF(0, 0, image.Width, image.Height);

//Render the image
g.DrawImage(image, rc, null, ImageAlign.Default);

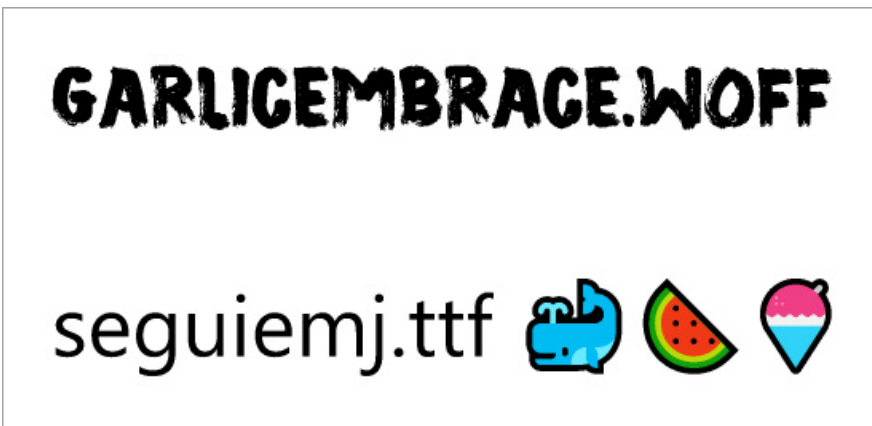
//Add text watermark to the image using a semitransparent color
g.DrawString(
    "Watermark",
    new TextFormat()
    {
        Font = Font.FromFile(Path.Combine("Resources", "Fonts",
            "calibri.ttf")),
        FontSize = 96,
        ForeColor = Color.FromArgb(128, Color.Yellow),
    },
    rc, TextAlignment.Center, ParagraphAlignment.Center, false);

//Save the image with watermark
origBmp.SaveAsJpeg("Watermark.jpeg");
```

[Back to Top](#)

Characters and Fonts

DsImaging provides support for drawing text with different font types, such as OpenType, TrueType and WOFF, and characters with codes greater than 0xFFFF. In addition, you can also draw colored fonts such as Segoe UI Emoji using **Palette** property of the **TextFormat** class. In this example, we have drawn *Garlicembrace.woff* and *seguiemj.ttf* fonts on the drawing surface.



```
C#
//Initialize Fonts
Font garlicFont = Font.FromFile(Path.Combine("Resources",
    "Fonts", "Garlicembrace.woff"));
Font emojiFont = Font.FromFile(Path.Combine("Resources",
    "Fonts", "seguiemj.ttf"));

//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(500, 500, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

//Define TexFormat and render text with specific font
TextFormat tf = new TextFormat
{
    Font = garlicFont,
```

```

    FontSize = 40
};
g.DrawString("Garlicembrace.woff", tf, new RectangleF(4, 4, 500, 50));

//Define TexFormat and render characters with codes greater than 0xFFFF
var pals = emojiFont.CreateFontTables(TableTag.CpalDraw).GetPalettes();
tf = new TextFormat
{
    Font = emojiFont,
    FontSize = 40,
    Palette = pals[0]
};
g.DrawString("seguiemj.ttf \U0001F433\U0001F349\U0001F367", tf,
    new RectangleF(4, 140, 550, 50));

//Save the image
origBmp.SaveAsJpeg("CharacterFonts.jpeg");

```

[Back to Top](#)

Right to Left

Dslmaging provides support for rendering text in right to left direction using **RightToLeft** property of the **TextLayout** class. This property can be used in a scenario where you use a language which is written in right to left direction, such as Arabic, Hebrew, etc.

وأصبحت لغة السياسة والعلم والأدب لقرون طويلة في الأراضي التي حكمها المسلمون، وأثرت العربية، تأثيرًا مباشرًا أو غير مباشر على كثير من اللغات الأخرى في العالم الإسلامي، كالتركية والفارسية والأردية واللبانية واللغات الأفريقية الأخرى واللغات الأوروبية مثل الروسية والإنجليزية والفرنسية والأسبانية والإيطالية والألمانية. كما أنها تدرس بشكل رسمي أو غير رسمي في الدول الإسلامية والدول الأفريقية المحادية للوطن العرب

To set the direction of text from right to left direction, you can set the value of the **RightToLeft** property to **true**.

```

C#
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(300, 300, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

string text = " وأصبحت لغة السياسة والعلم والأدب لقرون طويلة في الأراضي " +
    " التي حكمها المسلمون، وأثرت العربية، تأثيرًا مباشرًا " +
    " أو غير مباشر على كثير من اللغات الأخرى في العالم الإسلامي، كالترك " +
    " ية والفارسية والأردية واللبانية واللغات الأفريقية الأخرى واللغات " +
    " الأوروبية مثل الروسية والإنجليزية والفرنسية والأسبانية والإيطالية " +
    " والألمانية. كما أنها تدرس بشكل رسمي " +
    " أو غير رسمي في الدول الإسلامية والدول الأفريقية المحادية للوطن العرب";

TextLayout tl = g.CreateTextLayout();
tl.MaxWidth = 72 * 3;
tl.RightToLeft = true;
tl.TextAlignment = TextAlignment.Justified;
tl.Append(text);

g.DrawTextLayout(tl, new PointF(40, 50));

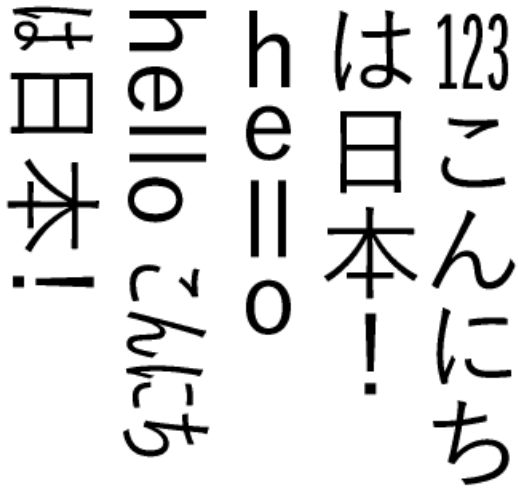
//Save the image
origBmp.SaveAsJpeg("RightToLeft.jpeg");

```

[Back to Top](#)

Vertical Text

DslImaging supports vertical text through **FlowDirection** property of the **TextLayout** class which accepts value from the **FlowDirection** enumeration. To set the vertical text alignment, this property needs to be set to **VerticalLeftToRight** or **VerticalRightToLeft**. Additionally, the **TextLayout** class provides an option to rotate the sideways text in counter clockwise direction using the **RotateSidewaysCounterclockwise** property. Further, **SidewaysInVerticalText** and **UprightInVerticalText** property of the **TextFormat** class also provides options to display the text sideways or upright respectively. These properties are especially useful for rendering Latin text within the East-Asian language text.



```
C#
//Initialize GcBitmap
GcBitmap bmp = new GcBitmap(90 * 4, 80 * 4, true, 384f, 384f);

//Create the graphics for the Bitmap
GcBitmapGraphics g = bmp.CreateGraphics(Color.White);

//Intialize TextLayout
var tl = g.CreateTextLayout();

//Define TexFormat and render text with specific font
var fmt1 = new TextFormat()
{
    Font = Font.FromFile(@"c:\Windows\Fonts\YuGothM.ttc"),
    FontSize = 12,
    UprightInVerticalText = true,
    GlyphWidths = GlyphWidths.QuarterWidths,
    TextRunAsCluster = true
};
tl.Append("123", fmt1);

//Define TexFormat and render text with specific font
var fmt2 = new TextFormat(fmt1)
{
    UprightInVerticalText = false,
    GlyphWidths = GlyphWidths.Default,
    TextRunAsCluster = false
};
tl.Append("こんにちは日本!", fmt2);

fmt2.TransformToFullWidth = true;
tl.Append("he", fmt2);
tl.Append("ll", fmt1);
tl.Append("o ", fmt2);

fmt2.TransformToFullWidth = false;
fmt2.UseVerticalLineGapForSideways = true;
tl.Append("hello ", fmt2);
```

```

fmt2.SidewaysInVerticalText = true;
fmt2.GlyphWidths = GlyphWidths.HalfWidths;
tl.Append("こんにちは日本!", fmt2);

tl.MaxHeight = 80;
tl.MaxWidth = 90;

//Specify text lines should be placed vertically from right to left
tl.FlowDirection = FlowDirection.VerticalRightToLeft;

//Render TextLayout
g.DrawTextLayout(tl, new PointF(0f, 0f));

//Save the image
bmp.SaveAsPng("VerticalText.png");


```

[Back to Top](#)

Text Around Images

In DslImaging, you can show text around images by identifying the area occupied by the embedded object, for instance, an image. The embedded object can be represented by an object rectangle which can be defined using an instance of the **ObjectRect** class. This object rectangle is assigned to the text layout using **ObjectRects** property of the **TextLayout** class in order to draw the text around the specified object rectangle.

*Puffins are any of three small species of alcids (auks) in the bird genus *Fratercula* with a brightly coloured beak during the breeding season. These are pelagic seabirds that feed primarily by diving in large offshore among soil. puffin foundin while*



the water. They breed in colonies on coastal cliffs or islands, nesting in crevices rocks or in burrows in the North Pacific Ocean, the Atlantic puffin is found in the North Atlantic Ocean. All puffin species have predominantly black or black and white plumage, a stocky build, and large beaks. They shed the colourful outer parts of their bills after the breeding season, leaving a smaller and duller beak. Their short wings are adapted for swimming with a flying technique under water. In the air, they beat their wings rapidly (up to 400 times per minute)[1] in swift flight, often flying low over the ocean's surface. A significant decline in numbers of puffins on Shetland is worrying scientists.

C#

```

//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(500, 500, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

//Get the demo image
Image imgPuffins = Image.FromFile("Resources/Images/puffins-small.jpg");
var rectPuffins = new RectangleF(100, 70, 180, 180);
// Set up ImageAlign that would fit and center an image within a
//specified area, preserving the image's original aspect ratio
ImageAlign ia = new ImageAlign(ImageAlignHorz.Center,
    ImageAlignVert.Center, true, true, true, false, false);

// Draw image, providing an array of rectangles as an output
//parameter to get the actual image rectangle

```

```
g.DrawImage(imgPuffins, rectPuffins, null, ia,
            out RectangleF[] rectsPuffins);

//Sample Text
string sampleText = "Puffins are any of three small species of" +
    " alcids (auks) in the bird genus Fratercula with a brightly" +
    " coloured beak during the breeding season. These are pelagic" +
    " seabirds that feed primarily by diving in the water. They" +
    " breed in large colonies on coastal cliffs or offshore" +
    " islands, nesting in crevices among rocks or in burrows in" +
    " the soil. Two species, the tufted puffin and horned puffin," +
    " are found in the North Pacific Ocean, while the Atlantic" +
    " puffin is found in the North Atlantic Ocean. All puffin" +
    " species have predominantly black or black and white plumage," +
    " a stocky build, and large beaks.They shed the colourful outer" +
    " parts of their bills after the breeding season, leaving a" +
    " smaller and duller beak. Their short wings are adapted for" +
    " swimming with a flying technique under water.In the air, they" +
    " beat their wings rapidly(up to 400 times per minute)[1] in" +
    " swift flight, often flying low over the ocean's surface. A" +
    " significant decline in numbers of puffins on Shetland is" +
    " worrying scientists.";

//Create and set up a TextLayout object to print the text:
var tl = g.CreateTextLayout();
tl.DefaultFormat.Font = Font.FromFile(Path.Combine("Resources", "Fonts",
            "timesbi.ttf"));
tl.DefaultFormat.FontSize = 12;
tl.TextAlignment = TextAlignment.Justified;
tl.MaxWidth = origBmp.Width;
tl.MaxHeight = origBmp.Height;
tl.MarginAll = 72 / 2;
// ObjectRect is the type used to specify the areas to flow around
//to TextLayout.
// Create an ObjecRect based on an image rectangle, adding some
//padding so that the result looks nicer
tl.ObjectRects = tl.ObjectRects = new List<ObjectRect>()
{ new ObjectRect(rectsPuffins[0].X - 6, rectsPuffins[0].Y - 2,
    rectsPuffins[0].Width + 12, rectsPuffins[0].Height + 4) };

// Add text:
tl.Append(sampleText);

// draw layout for the text:
g.DrawTextLayout(tl, PointF.Empty);

//Save the image
origBmp.SaveAsJpeg("TextAroundImage.jpeg");
```

[Back to Top](#)

Paragraph Formatting

DsImaging uses **GrapeCity.Documents.Text** namespace which provides **TextLayout** class that represents one or more paragraphs of text with same formatting. This class also provides various properties to align and format paragraphs. For example, this class provides **ParagraphAlignment** property which takes the values from **ParagraphAlignment** enumeration to set the alignment of paragraphs along the flow direction axis. The **FirstLineIndent** and **LineSpacingScaleFactor** properties of the **TextLayout** class can be used to apply the basic paragraph formatting options such as line indentation and spacing.

Text rendered using TextLayout.
 First test string added to TextLayout.
 Second test string added to TextLayout,
 continuing the same paragraph.
 Third test string added to
 TextLayout, a new paragraph. *Fourth
 test string, with a different char
 formatting.*

To format a paragraph:

1. Initialize the GcBitmap class.
2. Create a drawing surface using **CreateGraphics** method of the GcBitmap class which returns an instance of the GcBitmapGraphics class.
3. Add text using the **Append** and **AppendLine** methods of **TextLayout** class to create a paragraph.
4. Set the text formatting attributes such as font, font size, color, etc. using the **TextFormat** class properties.
5. Set first line offset, spacing between paragraphs and line spacing to format a paragraph using **FirstLineIndent**, **ParagraphSpacing** and **LineSpacingScaleFactor** properties of the **TextLayout** class respectively.

C#

```
var Inch = 96;
const float fontSize = 14;

//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(500, 500, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

// TextFormat to specify font and other character formatting:
var tf = new TextFormat()
{
    Font = Font.FromFile(Path.Combine("Resources", "Fonts", "times.ttf")),
    FontSize = fontSize
};

// TextLayout: A much more powerful way to render text with
//better performance
var tl = g.CreateTextLayout();
// To add text, use Append() or AppendLine() methods:
tl.Append("Text rendered using TextLayout. ", tf);
tl.Append("First test string added to TextLayout. ", tf);
tl.Append("Second test string added to TextLayout, continuing the " +
    "same paragraph. ", tf);

//Add a line break, effectively starting a new paragraph
tl.AppendLine();
tl.Append("Third test string added to TextLayout, a new paragraph. ",
    tf);
tl.Append("Fourth test string, with a different char formatting. ",
    new TextFormat(tf)
    {
        Font = Font.FromFile(Path.Combine("Resources", "Fonts",
            "timesbi.ttf")),
        FontSize = fontSize,
        FontBold = true,
        FontItalic = true,
        ForeColor = Color.DarkSeaGreen,
    });

//Setting layout options
tl.MaxWidth = g.Width - Inch * 2;
```



```
C#
// Initialize Fonts.
var arialbd = GCEXT.Font.FromFile(Path.Combine("Resources", "Fonts", "arialbd.ttf"));
var arialuni = GCEXT.Font.FromFile(Path.Combine("Resources", "Fonts", "arialuni.ttf"));

// Initialize GcBitmap.
var bmp = new GcBitmap(pixelSize.Width, pixelSize.Height, opaque, dpi, dpi);

// Create graphic for the Bitmap.
using var g = bmp.CreateGraphics(Color.White);

// Initialize TextLayout and set its properties.
var tl = g.CreateTextLayout();
tl.TextAlignment = TextAlignment.Distributed;
tl.JustifiedSpaceExtension = 0f;
tl.JustifiedTextExtension = 20f;


// Initialize TextFormat and set its properties.
var tf = new TextFormat { FontSize = 26f, Font = arialuni };
var tfInfo = new TextFormat { FontSize = 11f, Font = arialbd };
float marginx = 260, marginy = 36;
tl.MaxWidth = pixelSize.Width - marginx * 2;
var text = "abcdefg!1010101010abc;999999本列島で使され99 555";

// Render TextLayout and set the LineBreakingRules and WordBoundaryRules properties.
float DrawText(TextLayout tl, float y)
{
    var pt = new PointF(marginx, y + 20);
    tl.Append(text, tf);

    // Perform layout for the whole text.
    tl.PerformLayout(true);
    var rc = new RectangleF(pt, new SizeF(tl.ContentWidth, tl.ContentHeight));
    g.FillRectangle(rc, Color.PaleGoldenrod);

    // Render text using DrawString method at a specific location.
    g.DrawString($"LineBreakingRules.{tl.LineBreakingRules}, TextExtensionStrategy.
{tl.TextExtensionStrategy}:",
        tfInfo, new PointF(marginx / 2f, y));
    g.DrawTextLayout(tl, pt);
    tl.Clear();
    return rc.Bottom + 16;
}

float y = marginy, dy = marginy * 3.5f;
y = DrawText(tl, y);
tl.TextExtensionStrategy = TextExtensionStrategy.EastAsianExcel;
y = DrawText(tl, y);
tl.TextExtensionStrategy = TextExtensionStrategy.Excel;
y = DrawText(tl, y);
tl.LineBreakingRules = LineBreakingRules.Simplified;
tl.WordBoundaryRules = WordBoundaryRules.Simplified;
tl.TextExtensionStrategy = TextExtensionStrategy.Default;
y = DrawText(tl, y);
tl.TextExtensionStrategy = TextExtensionStrategy.EastAsianExcel;
y = DrawText(tl, y);
tl.TextExtensionStrategy = TextExtensionStrategy.Excel;
y = DrawText(tl, y);
```

 **Note:** The following properties are internally mapped to new properties and, hence, are marked obsolete:

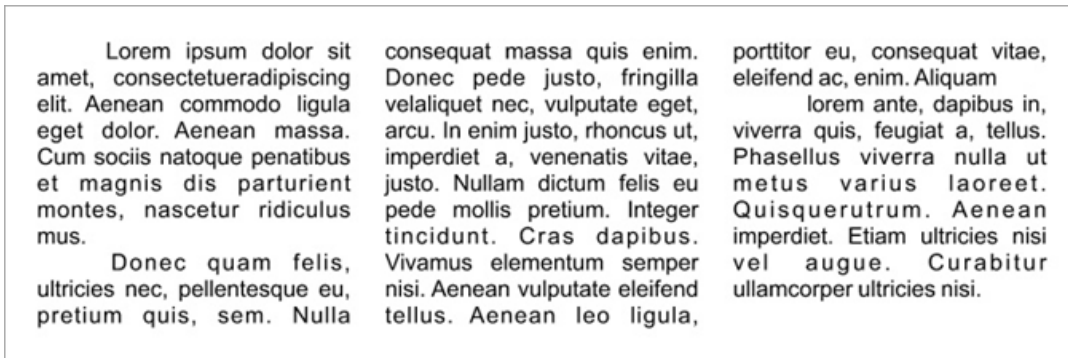
- TextLayout.SimplifiedWordBreak
- TextLayout.SimplifiedAlignment
- TextLayout.NoExcelAlignment

Limitation

In a few uncommon instances, behaviour of the old properties might change.

Text Splitting

DslImaging supports splitting of text layout through **Split** method of the TextLayout class. The **Split** method splits the text based on the bounds defined by the TextLayout and returns the individual text which is rendered using the DrawTextLayout method.



The following example illustrates text splitting where the text is split into multiple columns by invoking the **Split** method which creates a magazine style multi-column layout.

```
C#
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(800, 300, true);

//Create the graphics for the Bitmap
GcBitmapGraphics g = origBmp.CreateGraphics(Color.White);

g.Renderer.Multithreaded = true;
g.Renderer.SlowAntialiasing = true;

var tl = g.CreateTextLayout();
tl.TextAlignment = TextAlignment.Justified;
tl.FirstLineIndent = 96 / 2;

// Add some text (note that TextLayout interprets "\r\n",
//"\r" and "\n" as paragraph delimiters)
tl.Append("Lorem ipsum dolor sit amet, consectetur" +
    "adipiscing elit. Aenean commodo ligula eget dolor. " +
    "Aenean massa. " +
    "Cum sociis natoque penatibus et magnis dis parturient " +
    "montes, nascetur ridiculus mus. \r\n Donec quam felis, " +
    "ultricies" + " " + " nec, pellentesque eu, pretium quis, sem." +
    " Nulla consequat massa quis enim. Donec pede justo, fringilla vel" +
    "aliquet nec, vulputate eget, arcu. In enim justo, rhoncus ut, " +
    "imperdiet a, venenatis vitae, justo. Nullam dictum felis eu" +
    "pede mollis pretium. Integer tincidunt. Cras dapibus." +
    " Vivamus elementum semper nisi. Aenean vulputate eleifend" +
    "tellus. Aenean leo ligula, porttitor eu, consequat vitae," +
    "eleifend ac, enim. Aliquam" +
    "\r\n lorem ante, dapibus in, viverra quis, feugiat a, tellus." +
    "Phasellus viverra nulla ut metus varius laoreet. Quisque" +
    "rutrum. Aenean imperdiet. Etiam ultricies nisi vel augue." +
    "Curabitur ullamcorper ultricies nisi.");

// Set up columns
const int colCount = 3;
const float margin = 96 / 2; //1/2" margins all around
const float colGap = margin / 2; //1/4" gap between columns
float colWidth = (origBmp.Width - margin * 2 -
    colGap * (colCount - 1)) / colCount;
tl.MaxWidth = colWidth;
```

```

tl.MaxHeight = origBmp.Height - margin * 2;
// Calculate glyphs and perform layout for the whole text
tl.PerformLayout(true);

// In a loop, split and render the text in the current column
int col = 0;
while (true)
{
    // The TextLayout that will hold the rest of the text
    //which did not fit in the current layout
    var tso = new TextSplitOptions(tl)
    {
        MinLinesInLastParagraph = 2,
        MinLinesInFirstParagraph = 2
    };
    var splitResult = tl.Split(tso, out TextLayout rest);
    g.DrawTextLayout(tl, new PointF(margin + col * (colWidth +
        colGap), margin));
    if (splitResult != SplitResult.Split)
        break;
    tl = rest;
    if (++col == colCount)
        break;
}

//Save the image
origBmp.SaveAsJpeg("Columns.jpeg");

```

Back to Top

Support for Bitmap Glyphs

DsImaging library supports embedded bitmap glyphs or scaler bits (Sbits) specified by EBDT (Embedded bitmap data) table. DsImaging provides **AllowFontSbits** and **UseBitmapCache** properties in the **TextFormat** class. These properties can be used in cross-platform OpenType CJK fonts for representing complex glyphs at very small sizes.

When UseBitmapCache is set to True	When UseBitmapCache is set to False
	

To add Bitmap Glyph support for OpenType CJK font:

1. Load a Japanese Font file.
2. Configure Text format.
3. Set the **UseBitmapCache** property to True to use Bitmap Glyph Cache.
4. Set the **AllowFontSbits** property to True to get embedded bitmaps from EBDT font table.
5. Draw the Japanese string with Bitmap Glyph Cache.
6. Save the image.

C#

```

static void Main(string[] args)
{
    //Load the japanese font file
    var font = Font.FromFile("msgothic.ttc");

    //Configure the text format
    TextFormat tf = new TextFormat
    {
        Font = font,
        FontSizeInGraphicUnits = true,
        FontSize = 12,
    }

```

```

//Allows to use Bitmap Glyph Cache
UseBitmapCache = true,
// Allows to use embedded bitmaps from the EBDT font table
AllowFontSbits = true
};
// Japanese string
var s = "這是演示示例";
using (var bmp = new GcBitmap(90, 40, true))
{
    using (var g = bmp.CreateGraphics(Color.White))
    {
        // Draws the japanese string with Bitmap Glyph Cache
        g.DrawString(s, tf, new RectangleF(4, 4, 130, 20));

        //The code lines below are used to showcase how the Japanese text is drawn
        //when UseBitmapCache is set to false.
        tf.UseBitmapCache = false;
        g.DrawString(s, tf, new RectangleF(4, 24, 130, 20));
    }
    //Save the image
    bmp.SaveAsPng("BitmapGlyphSupport.png");

    Console.WriteLine("\n----Image Saved----");
    Console.ReadLine();
}
}

```

Support TrueType Hinting Instructions

Hinting instructions are included in some TrueType fonts which improve their look by reusing some glyph parts in different glyphs regardless of their font size. The TrueType hinting instructions are also supported in Dslmaging which supports drawing CJK characters as combinations of other smaller glyph pieces which enhances their final look.

Dslmaging library supports TrueType hinting instructions when rendering text on **GcGraphics**.

For fonts which include TrueType glyph hinting instructions, the **EnableHinting** property of the **Font** class is set to true, for the others it is set to False. Further, to apply the hinting instructions of the font, **EnableFontHinting** property of the **TextFormat** class must be set to true (the default value).

However, if the **EnableHinting** property is explicitly set to false, then the hinting instructions cannot be enabled.

As the default value of both the properties is true, hence the hinting instructions are supported for any TrueType font which includes them. Also, both properties affect text drawing on GcBitmapGraphics only.

Disabled Hinting Intructions

The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog.

入秋空污警報！這幾招遠離PM2.5學起來

Enabled Hinting Intructions

The quick brown fox jumps over the lazy dog.

The quick brown fox jumps over the lazy dog.


入秋空污警報！這幾招遠離PM2.5學起來

To enable TrueType hinting instructions for Chinese string:

1. Load a Chinese Font file.
2. Initialize the GcBitmap class.
3. Create a drawing surface using CreateGraphics method of the GcBitmap class which returns an instance of the GcBitmapGraphics class.
4. Define a Chinese string and configure **TextFormat** properties.
5. Set the **EnableFontHinting** property to true to enable hinting instructions.
6. Draw the Chinese string.
7. Save the image.

```
C#  
  
//Load the Chinese font file  
var font = Font.FromFile("kaiu.ttf");  
var bmp = new GcBitmap(750 * 2, 180 * 2, true, 192f, 192f);  
{  
    var g = bmp.CreateGraphics(Color.White);  
    {  
        //Draw the string with hinting instructions set to true  
        string s = @"一年之计在于春，一日之计在于晨";  
  
        //Define text formatting attributes  
        var tf = new TextFormat()  
        {  
            Font = font,  
            FontSize = 20,  
            EnableFontHinting = true  
        };  
  
        g.DrawString(s, tf, new PointF(10, 110));  
    }  
    bmp.SaveAsPng("ChineseFontwithHintingInstructions.png");  
}
```


For more information about working with text using DslImaging, see [DslImaging sample browser](#).


 **Note:** For rendering large or complex text and graphics, you can use **Skia** library. For more information about the library and its usage, see [Render using Skia Library](#).

Draw Rotated Text

DsImaging allows you to draw rotated text in unrotated rectangular bounds using **DrawRotatedText** and **MeasureRotatedText** methods of **GcGraphics** class. **DrawRotatedText** draws text at an angle in a specified rectangle, whereas **MeasureRotatedText** calculates the bounds where to draw the text.

Both methods accept the same parameters despite their different functioning. The following table lists the parameters these methods accept:

Parameters	Description
textLayout	<p>TextLayout to draw. This includes one or multiple text lines with various text formats. TextAlignment property specifies the alignment of text in each text line. It is important to draw a rotated text.</p> <p> Note: A few other properties of TextLayout have no effect when drawing a rotated text: MaxWidth, MaxHeight, FlowDirection, CanSkipFirstLineWithIndentation, ObjectRects, ParagraphAlignment, MarginLeft, MargingRight, MarginTop, MarginBottom, ColumnWidth, and RowHeight.</p>
angle	Text rotation angle in degrees. The expected range is -90 and +90, specifying an angle in degrees. Positive angles refer to clockwise rotation. Angles less than -90 are treated as -90 degrees, and angles greater than +90 are treated as +90 degrees.
verticalStacking	Stacks text lines either horizontally (along the top and bottom sides of the rectangle) or vertically (along the left and right sides of the rectangle).
rect	Target rectangle for the text.
alignment	Alignment of the whole text rectangle within the target rectangle using RotatedTextAlignment enumeration to: Top Left, Top Right, Top Center, Bottom Left, Bottom Right, Bottom Center, Middle Left, Middle Right, and Middle Center.

 **Note:** The methods may change or split the original TextLayout into multiple parts. Hence, if necessary, create a clone of the TextLayout in advance.

Refer to the example code to draw multiple rotated texts in different settings:

```
C#
// Initialize GcWicBitmap.
using var bmp = new GcWicBitmap(1050, 310, true);

// Draw rotated text with specified angle and alignment.
using (var g = bmp.CreateGraphics(Color.White))
{
    Draw(g, 10, angle: -90, false, RotatedTextAlignment.BottomLeft,
    TextAlignment.Leading);
    Draw(g, 240, angle: -60, false, RotatedTextAlignment.BottomLeft,
    TextAlignment.Leading);
    Draw(g, 480, angle: -45, false, RotatedTextAlignment.BottomLeft,
    TextAlignment.Leading);
    Draw(g, 720, angle: -30, false, RotatedTextAlignment.BottomLeft,
    TextAlignment.Leading);
}
```

```
}

// Save the image.
bmp.SaveAsPng("RotatedText.png");

// Define Draw method.
static void Draw(GcGraphics g, int x, int angle, bool verticalStacking,
    RotatedTextAlignment rotatedAlign, TextAlignment textAlign)
{
    // Initialize RectangleF.
    var rect = new RectangleF(x, 100, 200, 200);

    // Draw rectangle.
    g.DrawRectangle(rect, new Pen(Color.Green, 1));

    // Initialize TextLayout.
    var tl = g.CreateTextLayout();

    // Set text format.
    var fmt = new TextFormat
    {
        FontName = "Calibri",
        FontSize = 18,
    };

    // Add the text.
    tl.Append("This is long text, very long text, very long long.", fmt);

    // Set text alignment.
    tl.TextAlignment = textAlign;

    // Clone TextLayout.
    var tlCopy = tl.Clone(true);

    // Calculate bounds of rotated text inside a rectangle.
    var tlRect = g.MeasureRotatedText(tlCopy, angle, verticalStacking, rect,
rotatedAlign);

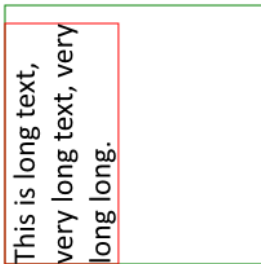
    // Draw rectangle with text.
    g.DrawRectangle(tlRect, new Pen(Color.Red, 1));

    // Draw rotated text.
    g.DrawRotatedText(tl, angle, verticalStacking, rect, rotatedAlign);

    // Draw strings with all text format details.
    fmt.FontSize = 12;
    g.DrawString($"angle = {angle}°", fmt, new PointF(x, 10));
    g.DrawString($"TextAlignment = {tl.TextAlignment}", fmt, new PointF(x, 30));
    g.DrawString($"alignment = {rotatedAlign}", fmt, new PointF(x, 50));
    g.DrawString($"verticalStacking = {verticalStacking}", fmt, new PointF(x, 70));
}
```

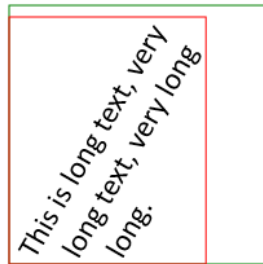

angle = -90°

TextAlignment = Leading
alignment = BottomLeft
verticalStacking = False



angle = -60°

TextAlignment = Leading
alignment = BottomLeft
verticalStacking = False



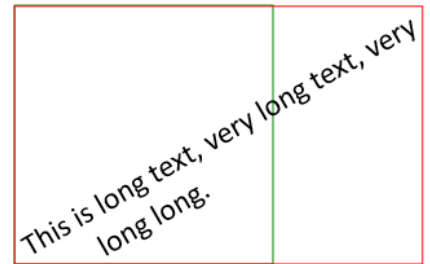
angle = -45°

TextAlignment = Leading
alignment = BottomLeft
verticalStacking = False



angle = -30°

TextAlignment = Leading
alignment = BottomLeft
verticalStacking = False



Draw Excel Like Rotated Text

With **DrawRotatedText** method, DsImaging allows you to define DrawExcelText method, which simulates the Excel renderer used for drawing rotated text. This method will differ from the DrawRotatedText method, as it considers the positive angles to render the text in a counterclockwise direction.

Refer to the example code to draw rotated text in specified unrotated rectangular bounds similar to Excel:

C#

```
namespace ExcelTextSimulator
{
    // Define enums.
    enum HorizontalAlignment
    {
        Left,
        Right,
        Center
    }

    enum VerticalAlignment
    {
        Top,
        Bottom,
        Center
    }

    internal class Program
    {
        static void Main(string[] _)
        {
            // Initialize GcBitmap.
            using (var bmp = new GcBitmap(700, 450, true))
            {
                // Draw rotated text.
                using (var g = bmp.CreateGraphics(Color.White))
                {
                    Draw(g, new RectangleF(100, 100, 500, 250));
                }
            }
            bmp.SaveAsPng("ExcelLikeRotatedText.png");
        }
    }
}
```

```
    }
}

// Define Draw method.
static void Draw(GcGraphics g, RectangleF rect)
{
    // Draw rectangle.
    g.DrawRectangle(rect, new Pen(Color.Green, 1));

    // Initialize TextLayout.
    var tl = g.CreateTextLayout();

    // Set text format.
    var fmt = new TextFormat
    {
        FontName = "Calibri",
        FontSize = 30,
        FontSizeInGraphicUnits = true
    };

    // Add the text.
    tl.Append("Quick brown", fmt);
    fmt.FontSize = 60;
    tl.Append(" fox", fmt);
    fmt.FontSize = 30;
    tl.Append(" jumps over the lazy dog.", fmt);
    fmt.FontSize = 50;
    tl.Append(" Quick brown fox jumps", fmt);
    fmt.FontSize = 20;
    tl.Append(" over the lazy dog.", fmt);

    int angle = 45;

    // Draw text.
    DrawExcelText(g, tl, angle, rect, HorizontalAlignment.Right,
VerticalAlignment.Top);
}

// Define DrawExcelText method.
static void DrawExcelText(GcGraphics g, TextLayout tl, int degrees,
RectangleF rect, HorizontalAlignment hAlign, VerticalAlignment vAlign)
{
    if (degrees == 90)
    {
        if (vAlign == VerticalAlignment.Bottom)
            tl.TextAlignment = TextAlignment.Leading;
        else if (vAlign == VerticalAlignment.Top)
            tl.TextAlignment = TextAlignment.Trailing;
        else
            tl.TextAlignment = TextAlignment.Center;
    }
}
```

```
else if (degrees == -90)
{
    if (vAlign == VerticalAlignment.Top)
        tl.TextAlignment = TextAlignment.Leading;
    else if (vAlign == VerticalAlignment.Bottom)
        tl.TextAlignment = TextAlignment.Trailing;
    else
        tl.TextAlignment = TextAlignment.Center;
}
else
{
    if (hAlign == HorizontalAlignment.Left)
        tl.TextAlignment = TextAlignment.Leading;
    else if (hAlign == HorizontalAlignment.Right)
        tl.TextAlignment = TextAlignment.Trailing;
    else
        tl.TextAlignment = TextAlignment.Center;
}

RotatedTextAlignment align;
if (vAlign == VerticalAlignment.Top)
{
    if (hAlign == HorizontalAlignment.Left)
        align = RotatedTextAlignment.TopLeft;
    else if (hAlign == HorizontalAlignment.Right)
        align = RotatedTextAlignment.TopRight;
    else
        align = RotatedTextAlignment.TopCenter;
}
else if (vAlign == VerticalAlignment.Bottom)
{
    if (hAlign == HorizontalAlignment.Left)
        align = RotatedTextAlignment.BottomLeft;
    else if (hAlign == HorizontalAlignment.Right)
        align = RotatedTextAlignment.BottomRight;
    else
        align = RotatedTextAlignment.BottomCenter;
}
else
{
    if (hAlign == HorizontalAlignment.Left)
        align = RotatedTextAlignment.MiddleLeft;
    else if (hAlign == HorizontalAlignment.Right)
        align = RotatedTextAlignment.MiddleRight;
    else
        align = RotatedTextAlignment.MiddleCenter;
}

// Draw rotated text inside a specific rectangle.
g.DrawRotatedText(tl, -degrees, false, rect, align);
}
```

```
}  
}
```



Draw Text in Slanted Rectangles

With **DrawSlantedText** method of **GcGraphics** class, DsImaging also allows you to draw rotated text in specified slanted rectangular bounds, similar to Excel. This method is similar to **DrawRotatedText** method except for the parameter of **SlantedTextAlignment** type.

SlantedTextAlignment enumeration provides the following six different modes for the slanted rectangles:

Modes	Description
BelowRotatedInside	The text appears below the rectangle side, rotated to the same angle as text. The side above the text is rotated inside the rectangle.
BelowRotatedOutside	The text appears below the rectangle side, rotated to the same angle as text. The side above the text is rotated outside the rectangle.
AboveRotatedInside	The text appears above the rectangle side, rotated to the same angle as text. The side below the text is rotated inside the rectangle.
AboveRotatedOutside	The text appears above the rectangle side, rotated to the same angle as text. The side below the text is rotated outside the rectangle.
CenterInsideOutside	The text appears at the center between the rectangle sides, rotated to the same angle as text. The side above the text is rotated inside the rectangle.
CenterOutsideInside	The text appears at the center between the rectangle sides, rotated to the same angle as text. The side above the text is rotated outside the rectangle.

Refer to the example code to draw rotated text in specified slanted rectangular bounds in different modes similar to Excel:

```
C#  
  
// Initialize GcWicBitmap.  
using var bmp = new GcWicBitmap(940, 640, true);
```

```
// Draw rotated text in slanted rectangles.
using (var g = bmp.CreateGraphics(Color.White))
{
    int angle = -70;
    var slantedAlign1 = SlantedTextAlignment.BelowRotatedInside;
    var slantedAlign2 = SlantedTextAlignment.BelowRotatedOutside;
    var slantedAlign3 = SlantedTextAlignment.AboveRotatedInside;
    var slantedAlign4 = SlantedTextAlignment.AboveRotatedOutside;
    var slantedAlign5 = SlantedTextAlignment.CenterInsideOutside;
    var slantedAlign6 = SlantedTextAlignment.CenterOutsideInside;
    bool verticalStacking = false;

    int x1 = 100;
    int y1Head = 10;
    int y1 = 100;
    int y2Head = 320;
    int y2 = 410;

    // Draw text and rectangle with specified angle and alignment.
    Draw(g, x1, y1Head, y1, angle, verticalStacking, slantedAlign1,
    TextAlignment.Leading);
    Draw(g, x1 + 270, y1Head, y1, angle, verticalStacking, slantedAlign2,
    TextAlignment.Trailing);
    Draw(g, x1 + 540, y1Head, y1, angle, verticalStacking, slantedAlign3,
    TextAlignment.Center);

    Draw(g, x1, y2Head, y2, angle, verticalStacking, slantedAlign4,
    TextAlignment.Leading);
    Draw(g, x1 + 270, y2Head, y2, angle, verticalStacking, slantedAlign5,
    TextAlignment.Trailing);
    Draw(g, x1 + 540, y2Head, y2, angle, verticalStacking, slantedAlign6,
    TextAlignment.Center);
}

// Save the image.
bmp.SaveAsPng("TextinSlantedRectangle.png");

// Define Draw method.
static void Draw(GcGraphics g, int x, int yHead, int y, int angle, bool
verticalStacking, SlantedTextAlignment slantedAlign, TextAlignment textAlign)
{
    RectangleF rect;
    if (!verticalStacking)
    {
        // Initialize RectangleF.
        rect = new RectangleF(x, y, 160, 200);
        float dx = (float)(200.0 / Math.Tan(Math.PI * angle / -180.0));

        // Set switch cases for different slanted text alignments.
        switch (slantedAlign)
```

```
{
    case SlantedTextAlignment.BelowRotatedInside:
    case SlantedTextAlignment.AboveRotatedOutside:
    case SlantedTextAlignment.CenterInsideOutside:

        // Draw the polygon.
        g.DrawPolygon([
            new PointF(x + dx, y),
            new PointF(x + dx + 160, y),
            new PointF(x + 160, y + 200),
            new PointF(x, y + 200)],
            new Pen(Color.Red, 1));
        break;
    case SlantedTextAlignment.BelowRotatedOutside:
    case SlantedTextAlignment.AboveRotatedInside:
    case SlantedTextAlignment.CenterOutsideInside:

        // Draw the polygon.
        g.DrawPolygon([
            new PointF(x, y),
            new PointF(x + 160, y),
            new PointF(x - dx + 160, y + 200),
            new PointF(x - dx, y + 200)],
            new Pen(Color.Red, 1));
        break;
    }
}
else
{
    // Initialize RectangleF.
    rect = new RectangleF(x, y, 200, 160);
    float dy = (float)(200.0 * Math.Tan(Math.PI * angle / 180.0));

    // Set switch cases for different slanted text alignments.
    switch (slantedAlign)
    {
        case SlantedTextAlignment.BelowRotatedInside:
        case SlantedTextAlignment.AboveRotatedOutside:
        case SlantedTextAlignment.CenterInsideOutside:
            if (angle >= 0)

                // Draw the polygon.
                g.DrawPolygon([
                    new PointF(x, y),
                    new PointF(x + 200, y + dy),
                    new PointF(x + 200, y + dy + 160),
                    new PointF(x, y + 160)],
                    new Pen(Color.Red, 1));

            else

                // Draw the polygon.
```

```
        g.DrawPolygon([
            new PointF(x, y - dy),
            new PointF(x + 200, y),
            new PointF(x + 200, y + 160),
            new PointF(x, y - dy + 160)],
            new Pen(Color.Red, 1));
        break;
    case SlantedTextAlignment.BelowRotatedOutside:
    case SlantedTextAlignment.AboveRotatedInside:
    case SlantedTextAlignment.CenterOutsideInside:
        if (angle >= 0)

            // Draw the polygon.
            g.DrawPolygon([
                new PointF(x, y - dy),
                new PointF(x + 200, y),
                new PointF(x + 200, y + 160),
                new PointF(x, y - dy + 160)],
                new Pen(Color.Red, 1));
        else

            // Draw the polygon.
            g.DrawPolygon([
                new PointF(x, y),
                new PointF(x + 200, y + dy),
                new PointF(x + 200, y + dy + 160),
                new PointF(x, y + 160)],
                new Pen(Color.Red, 1));
        break;
    }
}

// Draw the rectangle.
g.DrawRectangle(rect, new Pen(Color.Blue, 0.5f) { DashStyle = DashStyle.Dash });

// Initialize TextLayout.
var tl = g.CreateTextLayout();

// Set text format.
var fmt = new TextFormat
{
    FontName = "Calibri",
    FontSize = 18,
};

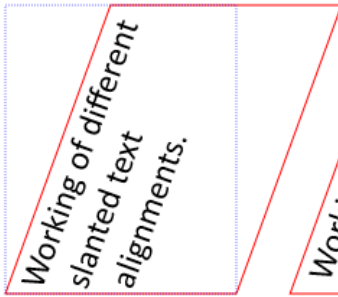
// Add the text.
tl.Append("Working of different slanted text alignments.", fmt);

// Set text alignment.
tl.TextAlignment = textAlign;
```

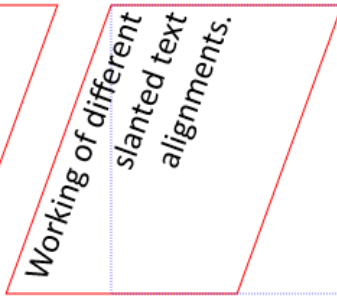
```
// Draw text in slanted rectangle.
g.DrawSlantedText(tl, angle, verticalStacking, rect, slantedAlign);

// Draw strings with all details.
fmt.FontSize = 12;
g.DrawString($"angle = {angle}°", fmt, new PointF(x, yHead));
g.DrawString($"TextAlignment = {tl.TextAlignment}", fmt, new PointF(x, yHead + 20));
g.DrawString($"alignment = {slantedAlign}", fmt, new PointF(x, yHead + 40));
g.DrawString($"verticalStacking = {verticalStacking}", fmt, new PointF(x, yHead + 60));
}
```

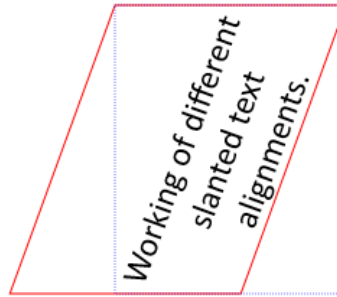
angle = -70°
TextAlignment = Leading
alignment = BelowRotatedInside
verticalStacking = False



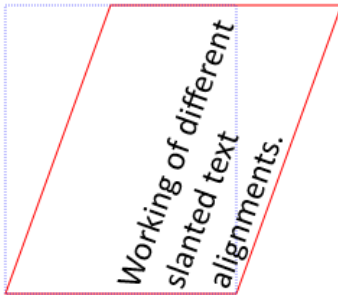
angle = -70°
TextAlignment = Trailing
alignment = BelowRotatedOutside
verticalStacking = False



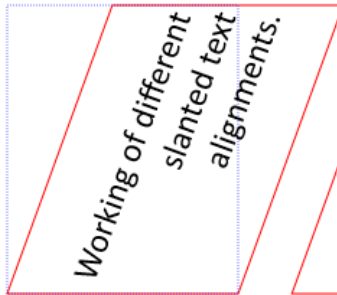
angle = -70°
TextAlignment = Center
alignment = AboveRotatedInside
verticalStacking = False



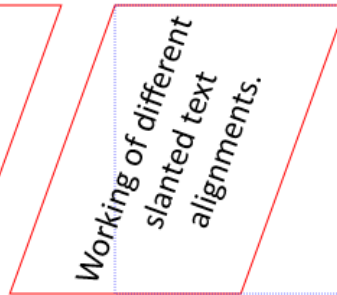
angle = -70°
TextAlignment = Leading
alignment = AboveRotatedOutside
verticalStacking = False



angle = -70°
TextAlignment = Trailing
alignment = CenterInsideOutside
verticalStacking = False




angle = -70°
TextAlignment = Center
alignment = CenterOutsideInside
verticalStacking = False



Work with Exif Metadata

The DslImaging library supports the extraction and modification of Exif metadata from various image formats, such as JPEG, PNG or TIFF files. With DslImaging, a developer can extract all Exif metadata from the images as mentioned in the Exif specifications sheet, such as the shutter speed, time it was taken, focal length, light value, use of flash, title, date, creator, copyright, description location (GPS data) etc..

DslImaging provides all the Exif metadata of an image in the **ExifProfile** class which is available in **GrapeCity.Documents.Imaging.Exif** namespace. The library also provides **ExifProfile** property of the **GcBitmap** class through which Exif metadata of the images can be accessed. The ExifProfile class mainly includes two methods, namely **GetTags** and **GetValues**. The **GetTags** method returns an array of all known tags in the profile. On the other hand, the **GetValues** method returns a list of all known tags in the profile along with their corresponding values. These tags are represented by the **ExifTag** enumeration and values are represented by the **ExifValue** class. These values can also be accessed through special properties such as **Orientation**, **ResolutionUnit**, **LensModel**, etc. provided by the ExifProfile class. The class also caters the unknown tags using the **UnknownTags** property which gets a list of values for the unknown tags. Moreover, if required, you can save the Exif metadata to a stream or a byte array using **SaveToStream** and **ToByteArray** methods of the **ExifProfile** class respectively and also load the Exif metadata from a stream or a byte array using **Load** method of the ExifProfile class.



Known tags(42):

Make	NIKON CORPORATION
Model	NIKON D700
XResolution	96
YResolution	96
ResolutionUnit	2
Software	Adobe Photoshop Lightroom 6.14 (Windows)
DateTime	2019:01:12 18:48:19
ExposureTime	0.01666667
FNumber	4
ExposureProgram	3
PhotographicSensitivity	6400
ExifVersion	48, 50, 51, 48
DateTimeOriginal	2010:06:18 23:52:58
DateTimeDigitized	2010:06:18 23:52:58
ShutterSpeedValue	5.906891
ApertureValue	4
ExposureBiasValue	1
MaxApertureValue	2
MeteringMode	5
LightSource	0
Flash	0
FocalLength	35
SubsecTimeOriginal	58
SubsecTimeDigitized	58
ColorSpace	1
SensingMethod	2
FileSource	3
SceneType	1
CFAPattern	2, 0, 2, 0, 0, 1, 1, 2
CustomRendered	0
ExposureMode	0
WhiteBalance	0
DigitalZoomRatio	1
FocalLengthIn35mmFilm	35
SceneCaptureType	0
Contrast	0
Saturation	0
Sharpness	0
SubjectDistanceRange	0
BodySerialNumber	2266324
LensSpecification	35, 35, 2, 2
LensModel	35.0 mm f/2.0

To extract and modify the EXIF metadata of an image:

1. Initialize the GcBitmap class.
2. Create an instance of **ExifProfile** class and get the instance with the Exif metadata of the image using the **ExifProfile** property.
3. Get all the known tags values using the **GetValues** method of ExifProfile class.
4. Access all the known tags of the profile using the **GetTags** method of ExifProfile class.
5. Initialize an instance of the **TextLayout** class and add all the known tags and values to the **TextLayout** object.
6. Render the EXIF metadata of the image along with the image using the **DrawTextLayout** and **DrawImage** methods respectively.

C#

```
//Image path
var imgPath = Path.Combine("Resources", "Images", "fire.jpg");

//Initialize GcBitmap and create bitmap graphics
GcBitmap origbmp = new GcBitmap(imgPath);

//Get all the known tags values
ExifProfile ep = origbmp.ExifProfile;
List<KeyValuePair<ExifTag, ExifValue>> knownTagsValues = ep.GetValues();

//Create TextLayout used to show EXIF metadata of the image
TextLayout tl = new TextLayout();
if (knownTagsValues.Count > 0)
{
    tl.Append("Known tags(" + knownTagsValues.Count.ToString() + "): \r\n");
    tl.AppendLine();

    //Add known tags values to the textlayout
    foreach (KeyValuePair<ExifTag, ExifValue> tag in knownTagsValues)
        tl.AppendLine(tag.Key + "\t" + tag.Value);
}
else
    tl.Append("No known tags");

//Render the created TextLayout and the original image on the output image
GcBitmap targetBmp = new GcBitmap(700, 850, true);
GcBitmapGraphics g = targetBmp.CreateGraphics(Color.White);
using (var img = Image.FromFile(imgPath))
    g.DrawImage(img, new RectangleF(20, 30, 200, 200), null,
        ImageAlign.ScaleImage);
g.DrawTextLayout(tl, new PointF(260, 30));

//Save the image
targetBmp.SaveAsJpeg("ExifMetadata.jpg");
}
```

Back to Top

For more information about working with EXIF metadata using DslImaging, see [DslImaging sample browser](#).

Render HTML to Image

DsImaging library along with **DsHtml** library lets you easily render HTML content to Images. When you browse through the content on a website, you may want to capture images to incorporate them into a professional presentation. Sometimes, one may also want to take the snapshot of online pricing details. With a utility library like DsHtml, the user can conveniently render HTML content to high resolution images. With DsHtml, you can convert webpages, HTML strings or even URIs to different image formats (JPEG, PNG, BMP, TIFF, GIF and WebP).


DsHtml is based on the industry standard Chrome web browser engine working in headless mode, offering advantage of rendering HTML to image on any platform - Windows, Linux and macOS. It doesn't matter whether your .NET application is built for x64, x86 or AnyCPU platform target. The browser is always working in a separate process.

The DS.Documents.Html package contains the following namespaces:

- GrapeCity.Documents.Html namespace provides **GcHtmlRenderer**(Obsolete), **GcHtmlBrowser**, **PdfOptions**, **ImageOptions**, **PageOptions**, **HtmlPage** classes etc.
- GrapeCity.Documents.Pdf namespace provides the **GcPdfGraphicsExt** and **HtmlToPdfFormat** classes.
- GrapeCity.Documents.Drawing namespace provides **GcBitmapGraphicsExt** and **HtmlToImageFormat** class.

Install DsHtml Package

1. Open Visual Studio and create a .Net Core Console application.
2. Right-click **Dependencies** and select **Manage NuGet Packages**.
3. With the **Package source** set to Nuget website, search for DS.Documents.Imaging under the **Browse** tab and click **Install**.
4. Similarly, install DS.Documents.Html package.

 **Note:** During installation, you'll receive two confirmation dialogs. Click **OK** in the **Preview Changes** dialog box and click **I Agree** in the **License Acceptance** dialog box to proceed installation.

5. Once, the DsHtml package has been installed successfully, add the namespace in Program.cs file.

```
C#  
  
using GrapeCity.Documents.Html;  
using GrapeCity.Documents.Pdf;  
using GrapeCity.Documents.Drawing;
```

6. Apply DsImaging license to **GcHtmlBrowser** class of DsHtml library to convert HTML to image. Without proper license, the count is limited to only 5 image conversions. The license can be applied in one of the following ways as shown below:
 - To license the instance being created

```
var html = "<html><body><h1>My First Heading</h1><p>My first paragraph.</p></body></html>";  
var re = new GcHtmlBrowser(html);  
re.ApplyGcImagingLicenseKey("key");
```
 - To license all the instances

```
GcHtmlBrowser.SetGcImagingLicenseKey("key");
```
7. Write the sample code.

Render HTML Webpage as Image

1. Get the URI of the HTML webpage you wish to render.
2. Configure image settings using the **PageOptions** class.
3. Convert the HTML page to JPEG, PNG, or WebP image using the **SaveAsPng**, **SaveAsJpeg** or **SaveAsWebP** methods of the **HtmlPage** class.

```
C#  
  
using GrapeCity.Documents.Html;  
using System.Drawing;  
  
var browserPath = BrowserFetcher.GetSystemChromePath();  
using var browser = new GcHtmlBrowser(browserPath);  
  
var uri = new Uri("Sample2.html", UriKind.Relative);  
using var pg = browser.NewPage(uri, new PageOptions
```

```
{
    WindowSize = new Size(700, 300),
    DefaultBackgroundColor = Color.AliceBlue
});
pg.SaveAsPng("Sample2.png");
```

The resulting image is shown below:



Note:

- In order to render an HTML page to image, the fonts used on that page should be already installed on your system.
- It is important to dispose every instance of the GcHtmlBrowser class after use.

Render HTML Markup as a Bitmap and Save as Image

1. Get the HTML string or mark up that you wish to render.
2. Store the HTML markup on a new page of the browser instance.
3. Configure image settings using the **PageOptions** class.
4. Create a new instance of **GcBitmap** and apply any transformations you need.
5. Use the **SaveAsTiff** method to save the image in TIFF format.

Similarly, you can use other SaveAs methods of GcBitmap class to save HTML markup in other image formats.

```
C#
using GrapeCity.Documents.Imaging;
using GrapeCity.Documents.Html;
using System.Drawing;

var browserPath = BrowserFetcher.GetSystemEdgePath();
using var browser = new GcHtmlBrowser(browserPath);

string html = "<p style=\"color: green; text-shadow: 3px 3px 3px gray;\">JavaScript can
change HTML content.</p>";
using var pg = browser.NewPage(html, new PageOptions { DefaultBackgroundColor =
Color.LemonChiffon });

using var bitmap = new GcBitmap();
pg.RenderAndCrop(bitmap, new PngOptions { Scale = 3 }, Color.LemonChiffon, 100, 50, 20, 100);
bitmap.SaveAsTiff("Sample3.tiff");
```

The resulting image is shown below:

JavaScript can change HTML content.

Draw HTML String or Page on GcGraphics at Specified Position

The user can also render HTML content to image using the **DrawHtml** method of **GcBitmapGraphicsExt** class. The **DrawHtml** method allows to convert an HTML text or page into an image. It also allows to insert HTML fragments in images along with other content. Moreover, the **DrawHtml** method has two overloads. The `GcBitmapGraphics.DrawHtml (string html, float x, float y, HtmlToImageFormat format, out SizeF size)` can be used to draw an HTML text on `GcBitmapGraphics` at a specified position, while the `GcBitmapGraphics.DrawHtml (Uri htmlUri, float x, float y, HtmlToImageFormat format, out SizeF size)` can be used to draw an HTML page specified by an URI on `GcBitmapGraphics` at a specified position.

To render HTML string or page on **GcBitmapGraphics** at specified position, follow the steps below:

1. Create an instance of **GcBitmap** class.
2. Configure the image settings using the **HtmlToImageFormat** class.
3. Save the HTML page or string on `GcBitmapGraphics` using the `DrawHtml` extension method of **GcBitmapGraphicsExt** class.
4. Call the **SaveAsJpeg**, **SaveAsPng**, **SaveAsGif**, **SaveAsBmp** and **SaveAsTiff** methods of **GcBitmap** class.

```
C#
//Create an instance of GcBitmap class
var bmp = new GcBitmap(1000, 1000, true, 96, 96);

//Configure image settings
HtmlToImageFormat htmlToImage = new HtmlToImageFormat(true)
{ WindowSize = new Size(500, 500) };

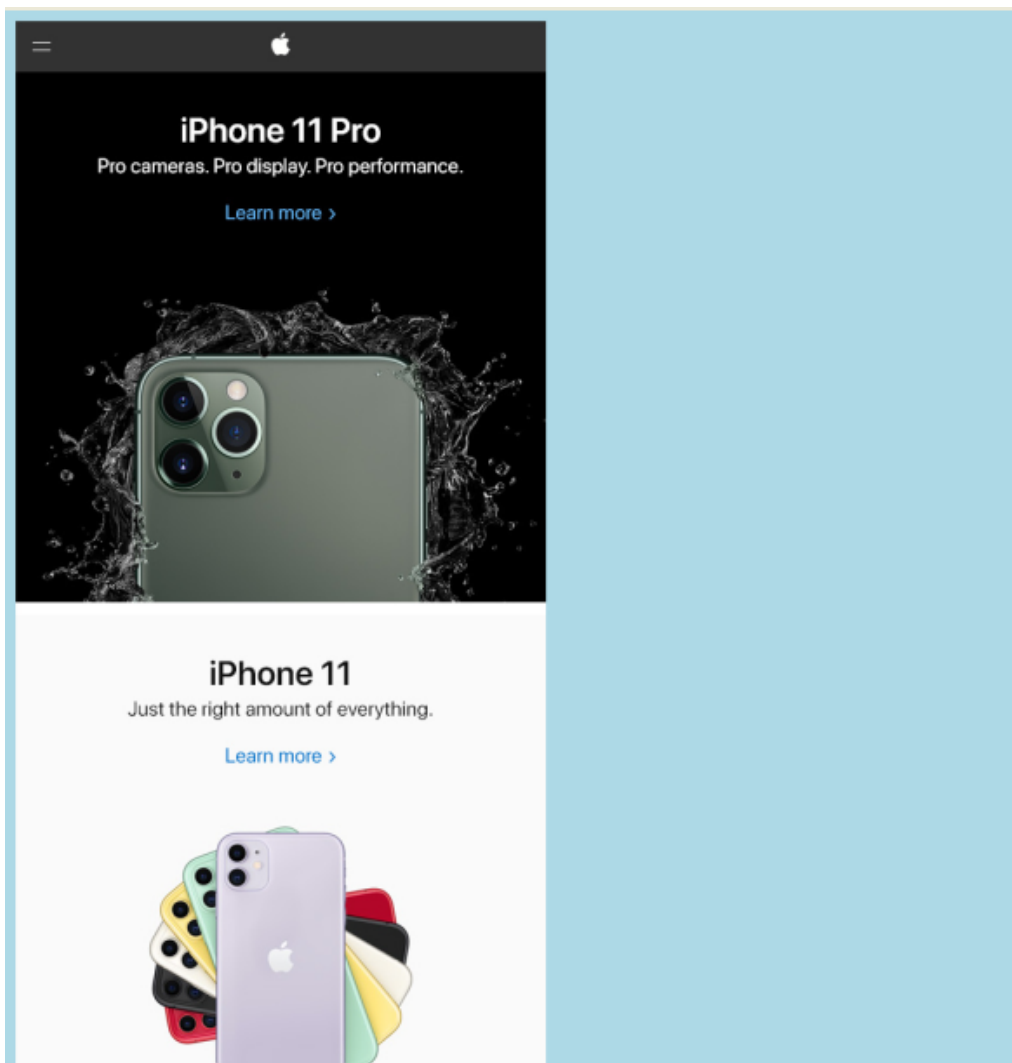
htmlToImage.DefaultBackgroundColor = Color.White;

//Draw HTML Page on GcBitmapGraphics
using (var g = bmp.CreateGraphics(Color.LightBlue))
{
    SizeF addSize = new SizeF();

    // Create an instance of GcHtmlBrowser that is used to render HTML:
    var browserPath = BrowserFetcher.GetSystemChromePath();
    var browser = new GcHtmlBrowser(browserPath);

    g.DrawHtml(browser, new Uri("https://www.apple.com/in/"), 10, 10, htmlToImage, out
addSize);
    bmp.SaveAsJpeg("DrawHtml.jpeg");
}
```

The resulting image is shown below:



For more information about rendering HTML to Image using DslImaging, see [DslImaging demo](#).

Tips to Migrate from Obsolete GcHtmlRenderer Class

If your application uses obsolete **GcHtmlRenderer** class to convert the HTML pages or content to an image format, you can use following steps to quickly update to the new **GcHtmlBrowser** class which does not depend on a custom build of Chromium and does not require GPL or LGPL licenses.

1. In Solution Explorer, go to Project > **Dependencies** > **Packages** and remove any references to
 - o GrapeCity.Documents.Html.Windows.X64
 - o GrapeCity.Documents.Html.Linux.X64
 - o GrapeCity.Documents.Html.Mac.X64
2. Check for licensing calls and if they exist, change them as follows:

C#

```
GcHtmlRenderer.SetGcImagingLicenseKey(key); -> GcHtmlBrowser.SetGcImagingLicenseKey(key);  
GcHtmlRenderer.SetGcPdfLicenseKey(key); -> GcHtmlBrowser.SetGcPdfLicenseKey(key);
```

3. In order to create and use an instance of GcHtmlBrowser, you require the path to a Chromium based browser on the current system. For instance, in case of Chrome browser:

- o You can get the path to an existing instance of Chrome installed on the current system.

C#

```
var path = BrowserFetcher.GetSystemChromePath();
```

- o Or, you can download and install Chrome in a location of your choice.

C#

```
var tp = Path.GetTempPath();
var bf = new BrowserFetcher() { DestinationFolder = Path.Combine(tp, ".gc-chromium") };
var path = bf.GetDownloadedPath();
```

 **Note:** We recommend using **Chrome** browser with GcHTMLBrowser class as Edge has some differences in the implementation of some DevTools features.

4. Create an instance of GcHtmlBrowser. Note that **RunWithNoSandbox** option may be needed on some Linux systems.

C#

```
if (RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
    return new GcHtmlBrowser(path, new LaunchOptions { RunWithNoSandbox = true });
else
    return new GcHtmlBrowser(path)
```

5. In all calls to GcGraphics.DrawHtml() method, insert browser instance as the first parameter.

C#

```
g.DrawHtml(html, ...); -> g.DrawHtml(browser, html, ...);
```

6. Look for instances of GcHtmlRenderer class which are rendering Uri such as


C#

```
using var re = new GcHtmlRenderer(uri);
...
re.RenderToJpeg(file, new JpegSettings() {...});
re.RenderToPng(file, new PngSettings() {...});
```

and replace them with

C#

```
// Create an HtmlPage from the URI
// (DefaultBackgroundColor and WindowSize options from Pdf/Jpeg/PngSettings
// have moved to PageOptions, while some other options are now in LaunchOptions):
using var htmlPage = browser.NewPage(uri, new PageOptions() { WindowSize = pixelSize;... });
...
htmlPage.SaveAsJpeg(file, new JpegOptions() {...});
htmlPage.SaveAsPng(file, new PngOptions() {...});
```

 **Note:** Few methods and properties of JpegSettings and PngSettings classes have been moved to **LaunchOptions** and **PageOptions** classes.

Render Using Skia Library

Skia is an open source 2D graphics library that provides common APIs that work as the graphics engine for Google Chrome and Chrome OS, Android, Flutter, Mozilla Firefox, Firefox OS, and many other products. In addition to this, SkiaSharp is a comprehensive cross-platform 2D graphics API for .NET platform used across mobile, server and desktop models to render images.

The **Skia** library, just like DslImaging, offers a rendering engine for drawing text and graphics. However, in case of **Skia**, the rendering engine is based on SkiaSharp and has dependency on **SkiaSharp** and **SkiaSharp.NativeAssets.Linux** nuget packages. The Skia library uses the exactly same implementation to render text and graphics as that of **GcGraphics** library. The difference is that the Skia library uses Skia engine at backend. In other words, you can use same approach to draw text and graphics while using the two libraries. However, both of them have their own merits and any of them could be used depending on the requirement of your application. Below are few recommended scenarios for each of them.

Skia versus DslImaging

You should use **Skia** when your application:

- Requires rendering large or complex images.
- Requires rendering text with fonts hinting and subpixel rendering
- Does not require access to individual pixels, DPI other than 96, EXIF/ICC profiles support, or effects such as dithering.
- Can afford to have two heavy nuget packages of SkiaSharp and SkiaSharp.NativeAssets.Linux.

You should use DslImaging when your application:

- Requires advanced drawing features such as transparency marks or logical operations on clip regions that are available via BitmapRenderer.
- Needs large fonts for creating images such as CJK.
- Requires pixel level access.
- Processes large images but has limited physical memory size.
- Requires small footprint.

For detailed information regarding the structure of these two libraries, see [DslImaging](#) and [Skia](#) in product architecture.

The code below shows how to render text and graphics using Skia library:

C#

```
// GcSkiaGraphics calls CreateTextLayout method to render text
using var g = new GcSkiaGraphics(800, 600, false, Color.White);
g.DrawRoundRect(new RectangleF(5, 5, 790, 590), 20, Color.Blue, 2,
DashStyle.DashDot);
float DegToRad = (float)Math.PI / 180;
g.Transform = Matrix3x2.CreateRotation(30 * DegToRad) *
Matrix3x2.CreateTranslation(100, 50);
var tl = g.CreateTextLayout();
tl.Append("Hello World!", new TextFormat()
{
    FontName = "Segoe UI",
    ForeColor = Color.SandyBrown,
    FontSize = 50f
});
```



```
});  
g.DrawTextLayout(tl, PointF.Empty);  
using var skiaImage = g.ToSkiaImage();  
skiaImage.SaveAsPng("result_text.png");  
  
// GcSkiaBitmap calls CreateGraphics method to render graphics  
using var bmp = new GcSkiaBitmap(800, 600, false);  
using (var h = bmp.CreateGraphics(Color.White))  
{  
    h.Transform = Matrix3x2.CreateRotation(-30 * DegToRad) *  
Matrix3x2.CreateTranslation(400, 400);  
    var rect = new RectangleF(0, 0, 300, 200);  
    h.FillEllipse(rect, new HatchBrush(HatchStyle.Backslashes) { ForeColor =  
Color.MediumPurple });  
    h.DrawEllipse(rect, new Pen(Color.Red, 3));  
}  
bmp.SaveAsPng("result_graphics.png");
```

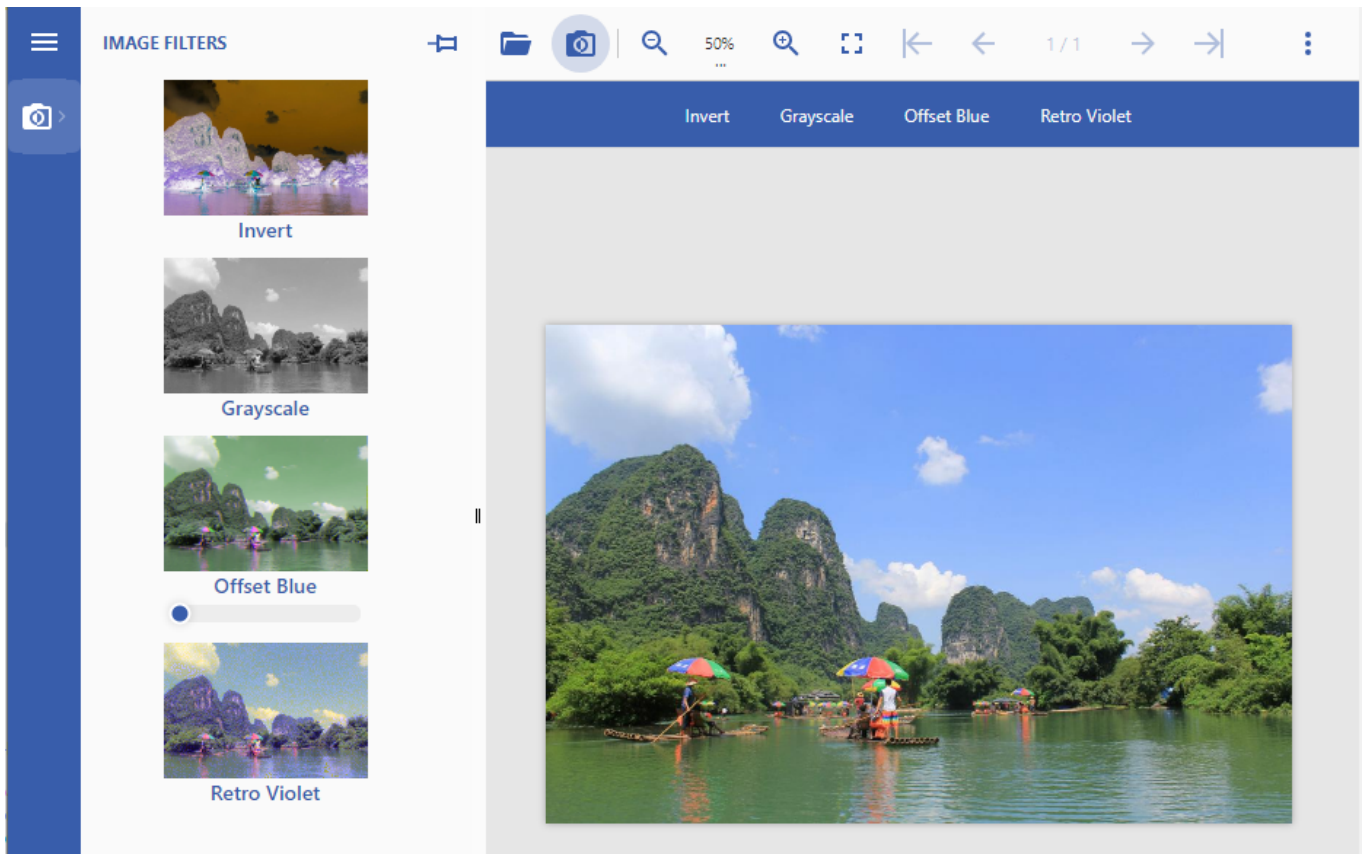
Limitation

- The Skia library has dependency on SkiaSharp and SkiaSharp.NativeAssets.Linux packages.
- Skia does not support changing image resolution, hence all the images are loaded and saved at 96 dpi only. However, you can implement a partial workaround by using scaling transformation.
- GcSkiaGraphics does not support hardware acceleration and transparency group feature.
- GcSkiaGraphics does not support the "Transparency Groups" feature required for drawing some PDF files to images. However, you can use **GcBitmapGraphics** or **GcD2DBitmapGraphics** classes to work around the same.

Document Solutions Image Viewer

Document Solutions Image Viewer (DslImageViewer, previously GclImageViewer) is a JavaScript based client-side image viewer and editor which allows you to view, edit, process, and save images on client side. The control supports all popular image formats, such as JPEG, PNG, TIFF, GIF, BMP, WebP etc. You can easily integrate it with Dslmaging to meet your client-side needs.

For more information about the viewer, see the [Document Solutions Image Viewer documentation](#).



Samples

DsImaging Samples

All the DsImaging samples are available through the online [sample browser](#). You can browse the source code of samples, run them on the server, view and download the images in different formats, or download individual samples to build and run on your own system (Windows, Mac or Linux). For more information, see [Quick Start](#), introductory page for the samples.

If you choose to download the samples, you can run them using following simple steps:

1. Click the **Download** action on the top right of the sample page.
2. Unzip the downloaded .zip file of sample.
3. Run the sample.

API Reference

This section contains documentation for all the assemblies required to create applications using Dslmaging.

Assembly	Description
DS.Documents.Html	Cross-platform library that provides HTML processing and rendering features.
DS.Documents.Imaging	Cross-platform library for working with raster images.
DS.Documents.Imaging.Windows	Platform-specific library that allows Dslmaging to use Windows system APIs when running on Windows operating systems.
DS.Documents.Imaging.Skia	Cross-platform library based on SkiaSharp that offers a rendering engine for drawing text and graphics.

Release Notes

Refer to the following release notes for the major releases of the product.

- [Release Notes for Version 7.1.0](#)
- [Release Notes for Version 7.0.0](#)
- [Release Notes for Version 6.2.0](#)
- [Release Notes for Version 6.1.0](#)
- [Release Notes for Version 6.0.0](#)
- [Release Notes for Version 5.2.0.800](#)
- [Release Notes for Version 5.1.0.790](#)
- [Release Notes for Version 5.0.0.762](#)
- [Release Notes for Version 4.2.0.719](#) - No Changes
- [Release Notes for Version 4.2.0.715](#)
- [Release Notes for Version 4.1.0.658](#)
- [Release Notes for Version 4.0.0.616](#)
- [Release Notes for Version 3.1.0.548](#) - No Changes
- [Release Notes for Version 3.1.0.508](#)
- [Release Notes for Version 3.0.0.414](#)
- [Release Notes for Version 2.2.0.310](#)

For details about latest hotfixes, see the [nuget page](#).

Breaking Changes

Refer to the following release notes for breaking changes:

- [Version 6.1.0](#)
- [Version 5.1.0.790](#)
- [Version 5.0.0.762](#)
- [Version 2.2.0.310](#)

Version 7.1.0

New Features and Improvements

- Added GcGraphics.DrawRotatedText() method: Draws rotated text inside an unrotated rectangle (similar to how MS Excel draws rotated text in borderless cells).
- Added GcGraphics.DrawSlantedText() method: Draws rotated text inside a slanted rectangle (similar to how MS Excel draws rotated text in cells with borders).
- Added GcGraphics.MeasureRotatedText() method: Calculates the bounds of rotated text inside an unrotated rectangle.

Version 7.0.0

Important Note

This is the initial release of the DS.Documents.Imaging package. This package replaces GrapeCity.Documents.Imaging, and provides the same functionality, ensures future enhancements, and is backwards compatible with GrapeCity.Documents.Imaging. Existing subscriptions will continue to apply to this new package.

New Features and Improvements

- Added rotation property to the GrapeCity.Documents.Drawing.IImage interface, which provides flip and rotate transformations that can be applied to the image.

Version 6.2.0

New Features and Improvements

- Added following helper classes in GrapeCity.Documents.Layout.Composition namespace to simplify drawing layouts on a GcGraphics:
 - Surface Class: Represents a surface that can draw its views on a GcGraphics.
 - Layer Class: Represents a drawing layer with visuals, optional clipping, and nested layers.
 - View Class (derived from Layer Class): Represents a Layer with an associated LayoutView object and transformation.
 - Space Class: Represents a space on a Layer with an associated LayoutRect.
 - Visual Class (derived from Space Class): Represents a figure, text, or image on a Layer.
- Synced version with other GrapeCity.Documents packages.

Version 6.1.0

Breaking Changes

- `GrapeCity.Documents.Svg.SvgMatrix` class renamed to `Matrix` and moved to `GrapeCity.Documents.Common` namespace.
- `GrapeCity.Documents.Imaging.InterpolationMode` enum has been moved to `GrapeCity.Documents.Drawing` namespace.

New Features and Improvements

- Added `GcGraphics.InterpolationMode` property that gets or sets the sampling mode to use when drawing images with resizing.
- Added `GcGraphics.IsInterpolationModeSupported()` method, which indicates whether the current graphics implementation supports a specified interpolation mode.
- Added the following classes and enums to `GrapeCity.Documents.Layout` namespace that implements a flat layout model based on constraints. Instead of setting the exact position of a visual element, constraints define rules for how that position depends on the positions of other elements:
 - `GrapeCity.Documents.Layout.LayoutHost` class: Represents the host and origin of a coordinate system for `LayoutView` objects.
 - `GrapeCity.Documents.Layout.LayoutView` class: Represents a transformed surface with a set of `LayoutRect` objects.
 - `GrapeCity.Documents.Layout.LayoutRect` class: Represents a rectangle with constraints.
 - `GrapeCity.Documents.Layout.AnchorPoint` class: Represents a point to be used as an anchor.
 - `GrapeCity.Documents.Layout.Contour` class: Represents a closed figure on a `LayoutView`.
 - `GrapeCity.Documents.Layout.LayoutException` class: Represents an error that occurred when resolving constraints in a `LayoutRect`.
 - `GrapeCity.Documents.Layout.Constraint` class: The base class for `LayoutRect` constraints.
 - `GrapeCity.Documents.Layout.AngleConstraint` class: Determines the rotation angle of the target `LayoutRect`.
 - `GrapeCity.Documents.Layout.AspectRatioConstraint` class: Determines the aspect (width to height) ratio of the target `LayoutRect`.
 - `GrapeCity.Documents.Layout.StarSizeConstraint` class: Determines the proportional width or height (weight) of the target `LayoutRect`.
 - `GrapeCity.Documents.Layout.SizeConstraint` class: Restricts the width or height of the target `LayoutRect`.
 - `GrapeCity.Documents.Layout.PositionConstraint` class: Determines the position of the sides or centers of the target `LayoutRect`.
 - `GrapeCity.Documents.Layout.ContourConstraint` class: Determines the min/max position of sides relative to the contour.
 - `GrapeCity.Documents.Layout.AnchorParam` enum: Specifies the source parameter of the anchor `LayoutRect`.
 - `GrapeCity.Documents.Layout.TargetParam` enum: Specifies the target parameter of the constraint's target `LayoutRect`.
 - `GrapeCity.Documents.Layout.ContourPosition` enum: Specifies the position of the anchor for a contour constraint.
- Added the following classes that use `LayoutHost` and related classes and enums to draw simple or complex tables with merged, rotated, auto-sized, multilayer cells with customizable styles:
 - `GrapeCity.Documents.Drawing.TableRenderer` class: A helper class for drawing tables on a `GcGraphics` (e.g., `GcPdfGraphics` or `GcBitmapGraphics`).
 - `GrapeCity.Documents.Drawing.TableCell` class: Represents the layout, style, and data of a table cell. Cells can contain simple text, multi-formatted `TextLayout`, or owner-drawn content.
 - `GrapeCity.Documents.Drawing.FrameStyle` class: Describes the inner border and filling of a table cell or table frame.

- GrapeCity.Documents.Drawing.CellStyle class: Describes the relative position, inner border, filling, and layout of a table cell.
- GrapeCity.Documents.Drawing.FrameBorders enum: Specifies which border lines are drawn in a table cell or table frame.
- GrapeCity.Documents.Drawing.FixedTableSides enum: Specifies which sides of a table are fixed. The position of those sides does not depend on the table's content.
- GrapeCity.Documents.Drawing.CellPosition enum: Specifies whether a table cell appears behind or on top of other cells.
- The default encoding used by GcBitmap.SaveAsIco() changed from Argb32 to Png.

Resolved Issues

- Miscellaneous minor big fixes.

Version 6.0.0

New Features and Improvements

- Added new properties such as `TextExtensionStrategy`, `LineBreakingRules`, and `WordBoundaryRules` in `TextLayout` class that specify the rules for text extension, line breaking algorithms, and word breaking algorithms respectively.
- Added the ability to generate text elements in place of paths elements while rendering text by setting `GcGraphicsDrawTextAsPath` property to `false`.
- Added Gaussian blur effect that applies Gaussian blur to the images.
- Added `PermissionToEmbedGranted` property to the `Font` class to indicate explicit permission from the legal owner of the font to embed it in documents produced by the software.
- Updated `SvgCommentElement` class with XML comment in the SVG file.
- Added support for new class `GcHTMLBrowser` for converting HTML markup to images (PNG, JPEG, WEBP).
- Added other supporting classes such as `BrowserFetcher`, `HtmlPage`, `LaunchOptions`, `PageOptions`, `TimeoutOptions`.
- Added `ImageOptions` and its other derived classes `PngOptions`, `JpegOptions`, and `WebpOptions` which provide output settings for rendering HTML to images.
- Added another class, `GcBitmapGraphicsExt` to provide extension methods for rendering HTML to `GcBitmapGraphics`.
- Marked `GcHtmlRenderer` and some other classes and methods in `GrapeCity.Documents.Html` namespace as obsolete.
- Replace custom-built Chromium with Chrome or Edge which is installed in OS, or you can now download chromium from a public website as well.

Version 5.2.0.800

New Features and Improvements

- Added support for WebP image format.
- Added WebP member to the ImageEncoding enumeration.
- Changed GcBitmap.Load() method overloads to accept images in WebP format.
- Added overloads GcBitmap.SaveAsWebp() method to save the images in WebP format.
- Changed Image.FromFile(), Image.FromStream(), Image.FromBytes() methods to accept images in WebP format.
- Added Skia (new package GrapeCity.Documents.Imaging.Skia) class representing a rendering engine for drawing text and graphics based on SkiaSharp.
- Added GcSkiaBitmap class representing a SkiaSharp.SKBitmap with an object model similar to GcBitmap.
- Added GcSkiaImage class representing an immutable image based on SkiaSharp.SKImage.
- Added GcSkiaGraphics class which implements a drawing surface based on SkiaSharp.SKSurface and SkiaSharp.SKCanvas.

Resolved Issues

- Fixed incorrect rendering of Thai text.

Version 5.1.0.790

Breaking Changes

- Setting the Resolution property of GcBitmapGraphics, GcWicBitmapGraphics and GcD2DBitmapGraphics to a value other than 96 now throws an exception.

New Features and Improvements

- Added the GcSvgGraphics class, which represents a graphics object that can be used to draw on a GcSvgDocument. The GcSvgGraphics.ToSvgDocument() method can be used to create a GcSvgDocument from graphics.
- Added SvgElement.ShapeRendering property to get or set a hint to the implementation about what tradeoffs to make as it renders vector graphics elements.
- Added SvgElement.ImageRendering property to get or set a hint to the implementation about how to make speed versus quality tradeoffs as it performs image processing.

Version 5.0.0.762

New Features and Improvements

- Support for rendering SVG (Scalable Vector Graphics) to PDF and raster images.
- Class `GcSvgDocument`: represents an SVG document.
- Methods `DrawSvg()` and `MeasureSvg()` added to `GcGraphics` class.
- `GrapeCity.Documents.Svg` namespace: contains types that provide SVG support, some of the more important classes are listed below.
- Abstract base class `SvgElement` and derived classes: represent various SVG elements.
- `SvgGraphicsElement` class (derived from `SvgElement`) and derived classes: represent various graphics SVG elements.
- `SvgGeometryElement` class (derived from `SvgGraphicsElement`) and derived classes: represent graphics elements that are defined by paths.
- `SvgPathBuilder` class: helper class for creating instances of `SvgPathData`.
- Other utility types (such as `SvgLength`, `SvgPaint` etc.) added in the `GrapeCity.Documents.Svg` namespace.

Breaking Changes

Breaking changes affecting all `GrapeCity.Documents` packages:

- `GrapeCity.Documents.Common` package has been removed, types defined in it have been moved to `GrapeCity.Documents.Imaging`.
- `GrapeCity.Documents.Common.Windows` package has been replaced by `GrapeCity.Documents.Imaging.Windows`.
- `GrapeCity.Documents.Pdf.Resources` has been removed, types defined in it have been moved to `GrapeCity.Documents.Pdf`.

Version 4.2.0.715

New Features and Improvements

- Added GcBitmapGraphics.BlendMode implementation. The specified blend mode affects all drawing (graphics, text and images).
- Added BitmapRenderer.BackgroundBitmap property which specifies a bitmap providing background for all drawing operations.
- Added BitmapRenderer.TransparencyMaskBitmap property which specifies a grayscale bitmap providing transparency mask for all drawing operations.
- Added GcBitmap.StoreInTempFile property which indicates whether pixel data should be dynamically mapped to a temporary file rather than kept fully in memory.
- Added Image.ToGcBitmap(GcBitmap) method which retrieves the underlying GcBitmap, or creates a new GcBitmap that contains the image data.

Bug Fix

- Miscellaneous bug fixes.

Version 4.1.0.658

Changes From the Previous Release

This version of the product has following changes.

- Updated encoders/decoders for JPEG, PNG and BMP image formats.

Version 4.0.0.616

New Features and Improvements

- Added support for ICO image format.
- Added Gclco class which represents a set of images stored in ICO format.
- Added IcoFrame class which represents a single frame in an ICO file.
- Added IcoFrameEncoding enumeration which specifies the encoding of an ICO frame image.

Version 3.1.0.508

New Features and Improvements

- Added support for TrueType hinting instructions in BitmapRenderer.

Bug fixes

- Fixed the exception that occurred when rendering a certain PDF.

Version 3.0.0.414

New Features and Improvements

- Added feature to decode and read images stored in JPEG 2000 format.
- Added new constructor to GcBitmap, GrayscaleBitmap and BilevelBitmap that accepts the existing pixel data as IntPtr.
- Added new GetContentRect method to the GcBitmap class.

Bug Fix

- Fixed a bug in GcBitmap.GetContentRect method that caused an AccessViolationException.

Version 2.2.0.310

Breaking Changes

This version of the product has the following breaking changes.

- Use the `GcBitmap.EnsureRendererCreated()` method instead of `GcBitmap.Renderer` property to make sure a non-null instance of `BitmapRenderer` is returned.
- Renamed `GcBitmap.AsBilevelBitmap()` method to `ToBilevelBitmap` (the `transparencyMask` parameter replaced with the `colorChannel` parameter, `blackIsZero` replaced with `whiteIsZero` with opposite meaning and default value).
- Renamed `GcBitmap.AsGrayscaleBitmap()` method to `ToGrayscaleBitmap()` method (the `transparencyMask` parameter has been replaced with the `colorChannel` parameter, `blackIsZero` parameter has been replaced with `whiteIsZero` with opposite meaning and default value).
- Replaced `BlackIsZero` property in the `BilevelBitmap` and `GrayscaleBitmap` classes with `WhiteIsZero` property having opposite meaning.
- Replaced `blackIsZero` parameter of `BilevelBitmap` and `GrayscaleBitmap` constructors with `whiteIsZero` parameter having opposite meaning and default value.
- Removed `Image.ConvertToGrayscale()` method (instead, you can use the `Image.ToGcBitmap()` and apply `GrayscaleEffect` to `GcBitmap`).
- Moved the `Disposed` property from the `Image` class to the `Image` interface.
- Replaced `Image.AsGcBitmap()` method with the `Image.ToGcBitmap()` method (`Image` interface is supported in `Image` and various bitmap classes).
- Removed `ToPngStream()`, `ToJpegStream()`, `GifStream()`, `FromGcBitmap()`, `FromFileDeferred()`, `FromStreamDeferred()` and `FromBytesDeferred()` methods from the `Image` class. Instead of the removed methods, you can call the `Image.ToGcBitmap()` and then call any of the `GcBitmap.SaveAs()` methods to accomplish similar tasks.
- Removed `withICC` argument from `FromFile()`, `FromStream()` and `FromBytes()` methods of the `Image` class.
- Added `frameIndex` as second parameter to the constructors of `GcBitmap` class which accepts path, stream, or byte array as the first argument.
- Replaced `ImageRect` type with `System.Drawing.Rectangle` in method arguments of the `GcWicBitmap` class.
- Renamed `TiffFrame.ReadAsGcBitmap()` method to `TiffFrame.ToGcBitmap()`.
- Renamed `WicTiffFrame.ReadAsGcWicBitmap()` method to `WicTiffFrame.ToGcWicBitmap()`.
- Removed `WicImage` class (instead, use `GcWicBitmap` class).
- Removed the `Clone()` method from the `Image` class.
- Added the `lowerBitsFirst` parameter to the `Indexed4bppBitmap` class constructor.

Changes From the Previous Release

This version of the product has the following changes.

- `Image` class is now lightweight and contains just the image metadata and a binding to the actual image data (e.g. to a disk file or to a stream).
- Now, users can convert `TiffFrame` and `WicTiffFrame` to an `Image` object.
- `GcBitmap` and `GcWicBitmap` can be created from an `Image` object.
- The `Image` interface has been implemented in the following classes: `GcBitmap`, `GcWicBitmap`, `BilevelBitmap`, `GrayscaleBitmap`, `Indexed4bppBitmap`, `Indexed8bppBitmap`.
- Optimized the `GcTiffReader` and `GcWicTiffReader` for a scenario wherein users want to load a single frame from a large TIFF file.

New Features and Improvements

The following features have been added with this version of the product.

- Added `GcGraphics.DrawRoundRect(RectangleF bounds, Pen left, Pen top, Pen right, Pen bottom, CornerRadius cornerRadius)` method, which allows rendering of multi-style rounded border.
- Added `GcBitmap.CompositeAndBlend()` method supporting all Porter Duff compositing operators and the advanced blending modes for combining two bitmaps into a single image.
- Added `AutoLevel()`, `AdjustLevels()`, `ExportColorChannel()` and `ImportColorChannel()` methods to the `GcBitmap` class.
- Added `AutoContrast()` and `AdjustLevels()` methods to the `GrayscaleBitmap` class.
- Implemented the `IImage` interface in the `GcBitmap`, `GcWicBitmap`, `BilevelBitmap`, `GrayscaleBitmap`, `Indexed4bppBitmap` and `Indexed8bppBitmap` classes.
- `TiffFrame` and `WicTiffFrame` can be converted to an `Image` object.
- Optimized `GcTiffReader` and `GcWicTiffReader` for a situation where only frame is loaded from large TIFF file.
- It is possible to load second, third, and other frames from a TIFF file or stream with `GcBitmap`, `GcWicBitmap`, and `Image` classes.
- `GcBitmap` and `GcWicBitmap` can be created from an `Image` object.
- Added constructors to `GcWicBitmap` class which accepts path, stream, or byte array.
- Added the `ToGcBitmap()` method overload that accepts an existing instance of `GcBitmap` to the following classes: `TiffFrame`, `BilevelBitmap`, `GrayscaleBitmap`, `Indexed4bppBitmap` and `Indexed8bppBitmap`.
- Added `GcBitmap.ToIndexed4bppBitmap()` and `GcBitmap.ToIndexed4bppBitmap()` method overloads that accept a custom palette and a dithering method.
- Added `GcBitmap.ToIndexed4bppBitmap()` and `GcBitmap.ToIndexed4bppBitmap()` method overloads based on the Octree quantizer algorithm.
- Added `GcBitmap.GenerateOctreePalette()` method which creates an Octree quantizer based palette for the current image.
- Added `LowerBitsFirst` property to the `Indexed4bppBitmap` class.
- Added `Clip()` method to the `Indexed4bppBitmap`, `Indexed8bppBitmap`, `BilevelBitmap`, and `GrayscaleBitmap` classes.
- Added the `GcGifReader` and `GcGifWriter` classes that allow users to read and write multi-frame GIF files.
- Added `IccProfileData` property to the `GcBitmap` class and other bitmap classes.
- ICC profile can now be loaded and saved to the following formats: JPEG, PNG, TIFF, and GIF.
- Added new constructors to the `GcBitmap` and `GrayscaleBitmap` classes that accept existing pixel data to be modified in-place.
- Added `ToPngStream()` method to the `IImage` interface and all its related classes.
- `Image` class is now lightweight. It contains the image metadata and binding to actual image data.
- Added support for all MS Excel pattern fills in the `HatchStyle` enumeration.
- It is now possible to load any frame (not only the first one) from a TIFF file or stream into `GcBitmap`, `GcWicBitmap` and `Image` objects.

Bug Fixes

The following issue has been resolved since the last release.

- While storing the global palette with less than 129 colors, `GcGifWriter` doesn't throw any errors now.