Table of Contents

- GcImaging Overview 2
- Key Features 3
- Getting Started 4-5
  - Quick Start 6-8
  - License Information 9-10
  - Technical Support 11
  - Redistribution 12
  - End-User License Agreement 13
- Product Architecture 14-16
- Features 17
  - Create Image 18-21
  - Load Image 22-24
  - Save Image 25
  - Work with GIF files 26-28
  - Work with TIFF Images 29-31
  - Process Image 32-41
  - Apply Effects 42-43
  - Work with Image Colors 44-46
  - Apply Transparency Mask 47-49
  - Work with Graphics 50
    - Draw and Fill Shapes 51-55
    - Clip Region 56-57
    - Align Image 58-59
    - Apply Matrix Transformation 60-61
  - Work with Text 62-75
  - Work with Exif Metadata 76-77
- Samples 78-80

GcImaging Overview

GrapeCity Documents is a cross-platform solution for document management which aims to provide a universal document, editor and viewer solution for all popular document formats

GrapeCity Documents for Imaging, referred to as GcImaging, is a part of GrapeCity Documents 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 in .NET Standard 2.0 applications with full support on Windows, macOS and Linux and deploy 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 and TIFF, 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

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

- **Fast and efficient library**
  GcImaging 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 GcImaging, 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**
  GcImaging lets you rotate, flip, crop, resize, composite, blend, apply dithering, thresholding and RGB effects on images.

- **Process GIF files**
  GcImaging allows you to read individual frames from a GIF file and save them as images in different formats supported by GcImaging. It also supports the creation of a GIF file by using multiple frames.

- **Create thumbnails**
  GcImaging allows you to downscale the images and apply various interpolation algorithms for creating image thumbnails.

- **Draw and fill graphics**
  Using GcImaging, you can draw and fill graphics like lines, polygons, rectangles, rounded rectangles, ellipses, paths on the graphics.

- **Advanced processing of image colors**
  GcImaging 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**
  GcImaging lets you draw text with advanced font and allows paragraph formatting on images. It also supports RTL text and Kashida on Arabic text.

- **Advanced TIFF processing**
  GcImaging 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**
  GcImaging 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.

Getting Started

System requirements

GcImaging system requirements, depending upon the framework you are using to create an application, are:

- Our packages include two targets, .NET Standard 2.0 and .NET Framework 4.6.1. In order to use them, your application needs to target either of the following:
  - .NET Core 2.0 or later
  - .NET Framework 4.6.1 or later
- Visual Studio 2015+/Visual Studio for MAC/Visual Studio Code for Linux

For OS versions supported in .NET Core 2.0+, see .NET Core 2.0+ - Supported OS versions.

Setting up an application

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

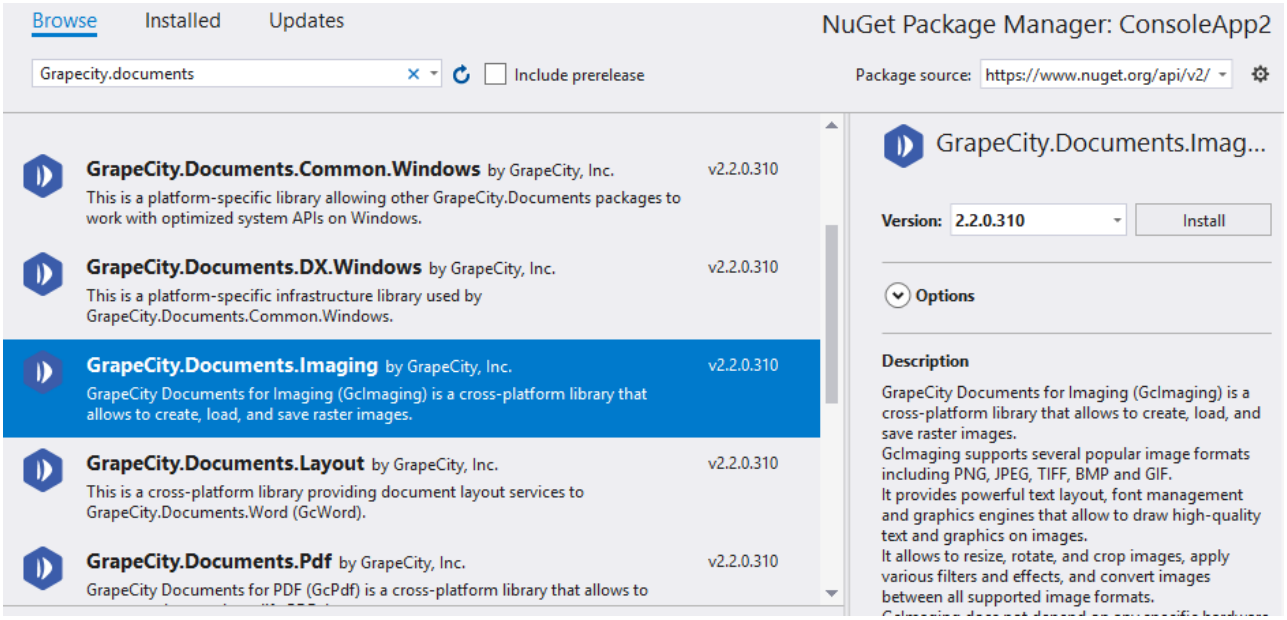
Reference	Purpose
GrapeCity.Documents.Imaging	To use GcImaging in an application, you need to reference (install) just the GrapeCity.Documents.Imaging package. It will pull in the required infrastructure packages.
GrapeCity.Documents.Common	GrapeCity.Documents.Common is an infrastructure package used by GcImaging. You do not need to reference it directly.
GrapeCity.Documents.Common.Windows	On a Windows system, you can optionally install GrapeCity.Documents.Common.Windows. It 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).
GrapeCity.Documents.DX.Windows	GrapeCity.Documents.DX.Windows is an infrastructure package used by GrapeCity.Documents.Common.Windows. You do not need to reference it directly.

Add reference to GcImaging in your application from NuGet.org

In order to use GcImaging 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 "Grapecity.Documents".
6. On the left panel, select **GrapeCity.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 "GrapeCity.Documents".
5. From the list of packages displayed in the left panel, select **GrapeCity.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 "**Grapecity**" and press Enter.
   Observe: GrapeCity packages get displayed in the dropdown.
10. Choose **GrapeCity.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 GcImaging library:

- **Create and save an image**
- **Load and modify an image**

Create and save an image

This quick start helps you in getting started with the GcImaging library. It 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
    - 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

Types of Licenses

GrapeCity Documents for Imaging supports the following types of license:

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

Unlicensed

After downloading the product, the product works in unlicensed mode. The following limitations are imposed when the product is used without license:

- Only 5 pages of the PDF file can be loaded for analyzing.
- On saving the PDF file, a watermark is displayed on all the pages in that file.
  'Unlicensed copy of GrapeCity PDF. Loading is limited to 5 pages. Contact us.sales@grapecity.com to get your 30-day evaluation key.'

> Note that if you run a sample that uses a signed PDF without a valid license key of GcPdf, the original signature in the generated PDF is invalidated. This happens because a license header is added to the PDF in such cases which changes the original signed document.

Evaluation License

GcImaging 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@grapecity.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, the product works in unlicensed mode with the above mentioned limitations.

In such case, following watermark is displayed in the PDF file:

'Created with expired evaluation copy of GrapeCity PDF. Loading is limited to 5 pages. Contact us.sales@grapecity.com to purchase license.'

> Note that if you run a sample that uses a signed PDF without a valid license key of GcPdf, the original signature in the generated PDF is invalidated. This happens because a license header is added to the PDF in such cases which changes the original signed document.

Licensed

GcPdf 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 GcPdf without any limitations.

Apply License

To apply evaluation/production license in GcPdf, 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 GcPdfDocument's ctor:
  `var doc = new GcImaging("key")`
  This licenses the instance being created.

- Call a static method on GcPdfDocument:
  `GcImaging.SetLicenseKey("key");`
  This licenses all the instances while the program is running.

Technical Support

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

- Product forum: https://www.grapecity.com/forums
- Email: us.sales@grapecity.com

Redistribution

In order to distribute the application, make sure you meet the installation criteria specified in the System Requirements page in this documentation. Further, you also need to have a valid Distribution License to successfully distribute the application.

GcImaging 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://www.grapecity.com/
E-mail	us.sales@grapecity.com
Phone	1.800.858.2739 or 412.681.4343
Fax	(412) 681-4384

End-User License Agreement

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

Product Architecture

Packaging

GcImaging is a collection of .NET Standard 2.0 class libraries written in C#, providing an API that allows to create image from scratch and to load, analyze and modify existing images.

GcImaging works on all platforms supported by .NET Standard, including .NET Core, ASP.NET Core, .NET Framework and so on.

GcImaging and supporting packages are available on nuget.org:

- **GrapeCity.Documents.Imaging**
- GrapeCity.Documents.Common
- GrapeCity.Documents.Common.Windows
- GrapeCity.Documents.DX.Windows

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

On a Windows system, you can optionally install **Grapecity.Documents.Common.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.

GcImaging API Overview

Namespaces

Namespaces	Description
GrapeCity.Documents.Common	Infrastructure and utility types (including fonts support).
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: • **GrapeCity.Documents.Drawing** • **GrapeCity.Documents.Imaging**
GrapeCity.Documents.Text	Text processing sub-system.

GcImaging 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 or multi-frame GIF

GcImaging 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 GcImaging package (GrapeCity.Document.Imaging) and in the related Windows

specific package (GrapeCity.Documents.Common.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 or decoded and loaded from the same set of image formats.
- **GrapeCity.Documents.Imaging.Windows.GcWicBitmap** which resides in the GrapeCity.Documents.Common.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, JPEGXR and loaded from BMP, PNG, JPEG, GIF, TIFF, 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 Readers/Writers

GcImaging 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 **GrapeCity.Documents.Common.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

GcImaging 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.

Features

This section comprises the features available in GcImaging.

Create Image
: Create images and thumbnails in GcImaging.

Load Image
: Load images from file, stream, and byte array in GcImaging.

Save Image
: Save images to different formats in GcImaging.

Work with TIFF Images
: Create a multiframed TIFF, save TIFF frames as separate images, and create tiled images in GcImaging.

Work with GIF Files
: Create a GIF file and read a GIF file to save the frames as separate images in GcImaging.

Process Image
: Resize, crop, rotate, flip, clear, and combine images, convert an image to indexed image and change its resolution in GcImaging.

Apply Effects
: Apply dithering, thresholding, gray scaling, and RGB effects on an image in GcImaging.

Work with Image colors
: Adjust color intensity and image histogram levels, work with color channels and color quantization in GcImaging.

Apply Transparency Mask
: Set transparency and set the background color for semi-transparent images in GcImaging.

Work with Graphics
: Draw and fill shapes, clip region, align image, and apply matrix transformation in GcImaging.

Work with Text
: Render and trim text, add watermark text on an image, draw text with antialiasing and different font types, draw text around images, use RTL, and format paragraphs in GcImaging.

Work with EXIF Metadata
: Extract and modify the EXIF metadata of an image using GcImaging.

Create Image

An image is a visual representation of information that can be created using a combination of graphics and text. GcImaging 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, and GIF.

GcImaging 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);
```

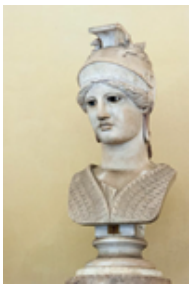
```
        g.DrawRoundRect(rec2, rx, ry, Color.Blue, 4);

        //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("GcImaging.jpeg");
}
```

Back to Top

Create Thumbnail

GcImaging 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.

Original Image	Thumbnail



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 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);
    }

}
```

Back to Top

For more information about processing images using GcImaging, see GcImaging sample browser.

Load Image

GcImaging 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 FlieStream 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");
}
```

Back to Top

Save Image

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

Format	Method
JPEG (with specified quality)	SaveAsJpeg
PNG	SaveAsPng
BMP	SaveAsBmp
TIFF	SaveAsTiff
GIF	SaveAsGif

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

C#

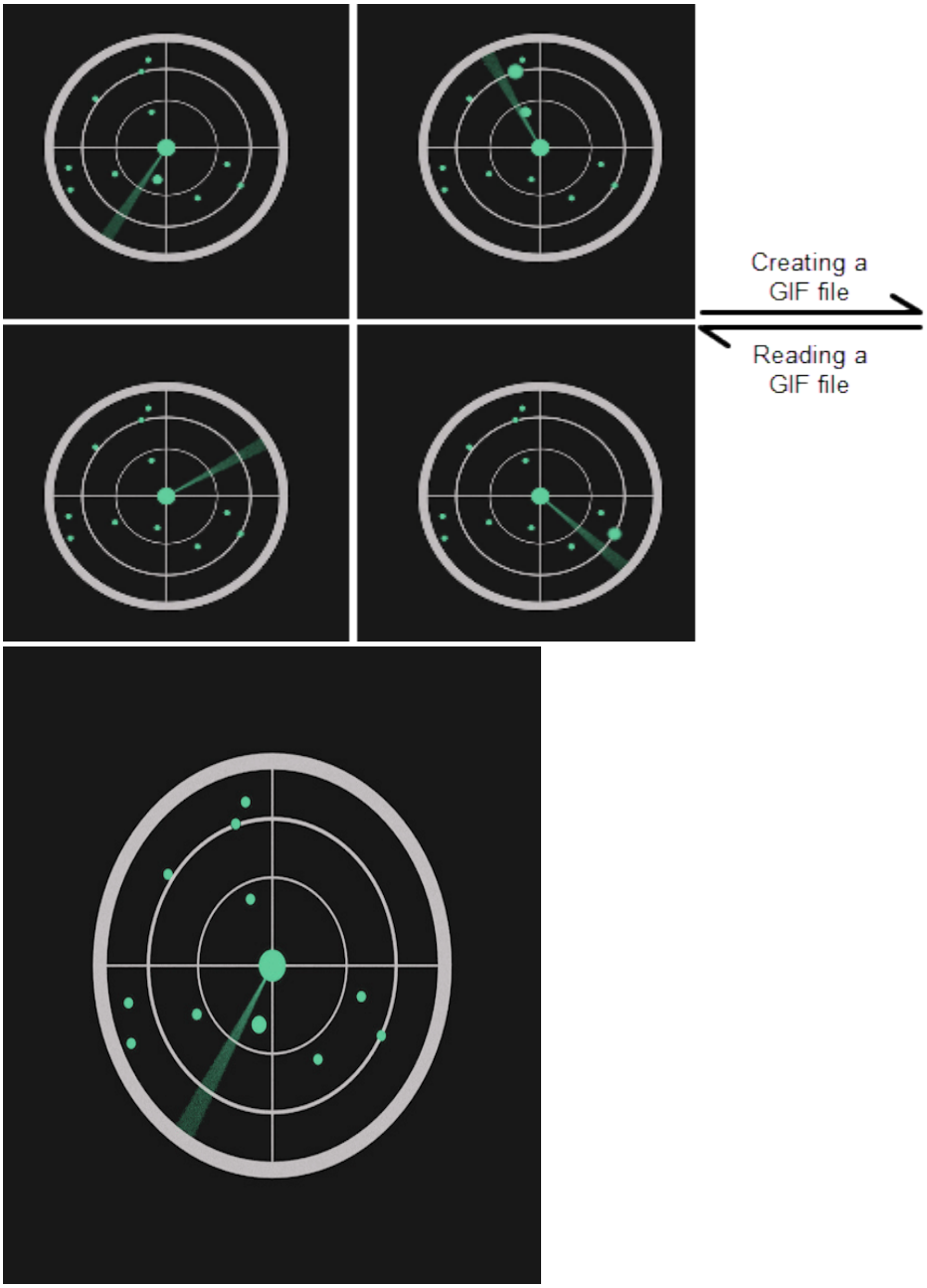
```
// Save image
bmp.SaveAsJpeg("color-postits.jpg");

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

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, GcImaging 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 GcImaging. 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

GcImaging 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 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 GcImaging 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 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 TIFF images using GcImaging, see GcImaging 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. GcImaging 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

GcImaging 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.Common.Windows package. 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 GcImaging, 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 JPEG format using SaveAsJpeg 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 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

GcImaging 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 TiffCompressioin.**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.

Back to Top

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 GcImaging, 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 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 GcImaging, see GcImaging sample browser.

Process Image

GcImaging 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

GcImaging 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.

Original Image	
	
Image with reduced size	**Enlarged image**
	

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 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. GcImaging 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.

Original Image	Cropped 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 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();
```

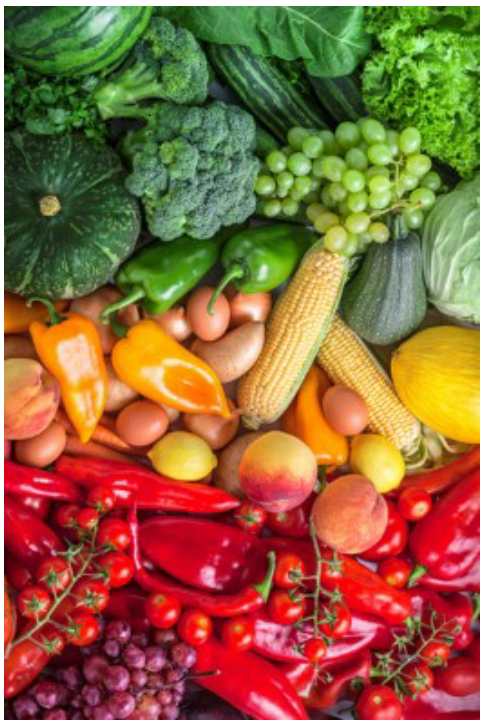
```
//Load image from file
origBmp.Load(origImagePath);

//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. GcImaging 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 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 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 GcImaging, 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. GcImaging 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 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

GcImaging 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. GcImaging 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.

Original Image	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 and pass the octree color palette as its parameter.
4. Save the indexed image using the SaveAsJpeg method of GcBitmap class.

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

GcImaging 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#

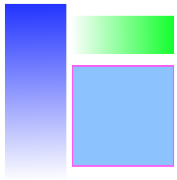
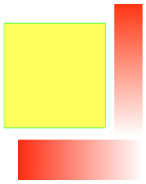
```
//Get the images paths
var jpgImagePath = Path.Combine("Resources", "Images",
                   "gray-puffins-small.jpg");
var pngImagePath = Path.Combine("Resources", "Images",
                   "gray-dog-small.png");
var bmpImagePath = Path.Combine("Resources", "Images",
                   "color-goldfish-small.bmp");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

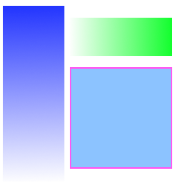
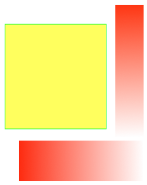
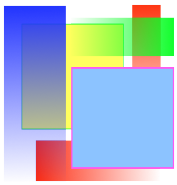
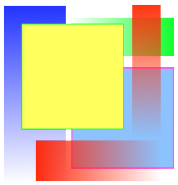
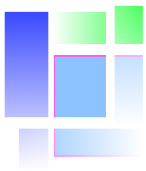
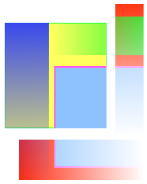
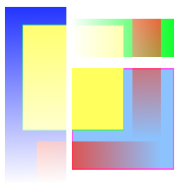
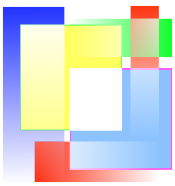
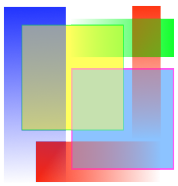
Back to Top

Compositing Images

Compositing defines various ways in which two bitmaps can be combined into a single image. GcImaging 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:

Source Image	Destination Image
	

Clear	Copy	Destination	Source Over	Destination Over
				
Source In	**Destination In**	**Source Out**	**Destination Out**	**Source Atop**
				
Destination Atop	**XOR**	**Lighter**		
				

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#

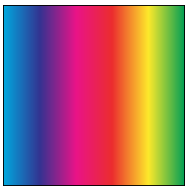
```
//Load the two images to be combined
using (var dst = new GcBitmap(@"in\dst.png"))
using (var src = new GcBitmap(@"in\src.png"))
//Combine the two images using various compositing and blending modes
{
    var tmp = dst.Clone();
    tmp.CompositeAndBlend(src, 0, 0, CompositeMode.DestinationOver);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

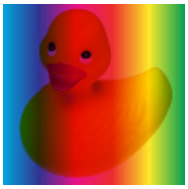
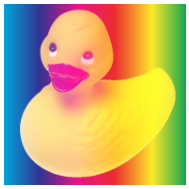
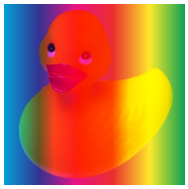
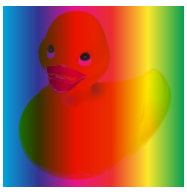
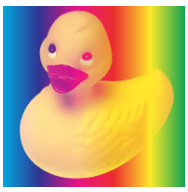
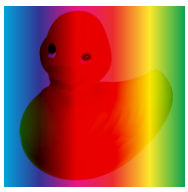
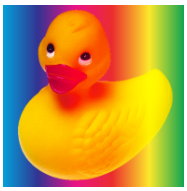
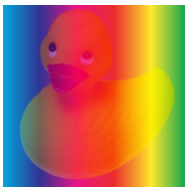
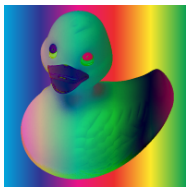
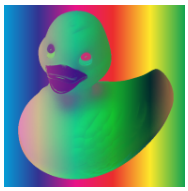
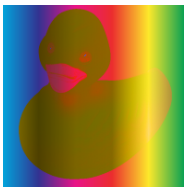
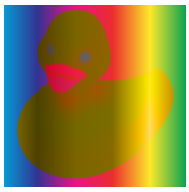
```
}
```

Back to Top

Blending Images

Blending defines various ways in which the colors of an image and the backdrop can interact with each other. GcImaging allows you to blend images by providing **CompositeAndBlend** method in the GcBitmap class. The method takes values from **BlendMode** enumeration as a parameter to generate the resultant image by blending the source and destination image. There are 16 blend modes which can be implemented through the BlendMode enumeration as displayed below:

Source Image	Destination Image
	

Normal	Multiply	Screen	Overlay
			
Darken	**Lighten**	**Color Dodge**	**ColorBurn**
			
Hard Light	**Soft Light**	**Difference**	**Exclusion**
			
Hue	**Saturation**	**Color**	**Luminosity**
			

To perform blending on two bitmaps using ColorDodge blend mode :

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

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");
```

Back to Top

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 GcImaging 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");
```

Back to Top

For more information about processing images using GcImaging, see GcImaging sample browser.

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.

GcImaging library offers great flexibility while working with these advanced effects which includes dithering, thresholding, gray scaling and various RGB effects. GcImaging 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.

Original Image	**BrightnessContrastEffect**	**TemperatureAndTintEffect**
		
Grayscale	**Thresholding**	**Dithering**
		

Effects	Classes	Descriptions
Dithering	DitheringEffect	Allows you to apply dithering effect through 9 different algorithms which are provided by the DitheringMethod enumeration. • Atkinson • Burks • FloydSteinberg • JarvisJudiceNinke • Sierra • SierraLite • Stucki • TwoRowSierra • NoDithering
Thresholding	• BradleyThresholdingEffect • OtsuThresholdingEffect	Allows you to apply two types of thresholding effects, Bradley's thresholding and Otsu's thresholding,

		through BradleyThresholdingEffect and OtsuThresholdingEffect class respectively.
Grayscaling	GrayscaleEffect	Allows you to apply grayscale effect as per the three grayscale standards provided by the GrayscaleStandard enumeration. • BT709 • BT601 • BT2100
RGB effects	• OpacityEffect • HueRotationEffect • SaturationEffect • SepiaEffect • TemperatureAndTintEffect • LuminanceToAlphaEffect • BrightnessContrastEffect • GammaCorrectionEffect	Allows you to apply various RGB effects using their corresponding classes mentioned in the column on left hand side.

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.

Back to Top

For more information about implementation of different effects using GcImaging, see GcImaging sample browser.

Work with Image Colors

GcImaging 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

GcImaging 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.

Original Image	Image after AutoLevel
	

To improve the colors of an image using AutoLevel method of GcBitmap class:

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, GcImaging library provides **AdjustLevels** method in both **GcBitmap** class and **GrayscaleBitmap** class.

Original Image	Image after AdjustLevels
	

To adjust levels of an image histogram using **AdjustLevels** method of GcBitmap class:

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("brighthouse.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. GcImaging 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.

Original Image	Blue Color Channel	Green Color Channel
		

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 of the GcBitmap class.

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();
}
```

Original Image	Red Color Channel

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 of the GcBitmap class.

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);
    }
}
```

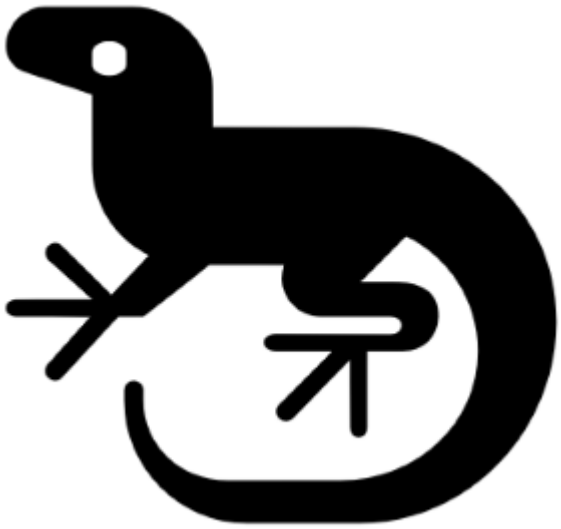
Back to Top

For more information about working with TIFF images using GcImaging, see GcImaging sample browser.

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 GcImaging, 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 AsBilevelBitmap or AsGrayscaleBitmap 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

Set Transparency

To set the transparency mask using a BilevelBitmap:

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 **AsGrayscaleBitmap** 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 result bitmap using the **ApplyTransparencyMask** method.
5. Convert the result 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 tranparency mask has to be applied
GcBitmap bmp = new GcBitmap("tudor.jpg");

//Define the transparency mask using mask image
GrayscaleBitmap grayscaleMask = mask.AsGrayscaleBitmap(true, 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

Set Background Color for Semi-transparent Images

To set the background color:

1. Initialize an instance of the GcBitmap class and load an image in the GcBitmap object.
2. Check whether the image contains transparent or semi-transparent pixels using **HasTransparentPixels** method of the GcBitmap class.
3. Convert the image with transparent or semi-transparent pixels to a fully opaque image with specified background color using the **ConvertToOpaque** method of the GcBitmap class.

C#

```
//Initialize GcBitmap by passing the image path
var imgPath = Path.Combine("Tranparent1.png");
GcBitmap origBmp = new GcBitmap(imgPath);

//Check for transarent pixels and convert to opaque
if (origBmp.HasTransparentPixels())
```

```
{
    origBmp.ConvertToOpaque(Color.LightBlue);
    origBmp.SaveAsJpeg("NoTransparent.jpg");
}
else
    Console.WriteLine("No transparent pixels");
```

Back to Top

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. GcImaging allows you to draw these shapes and graphics path using GcGraphics class methods.

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

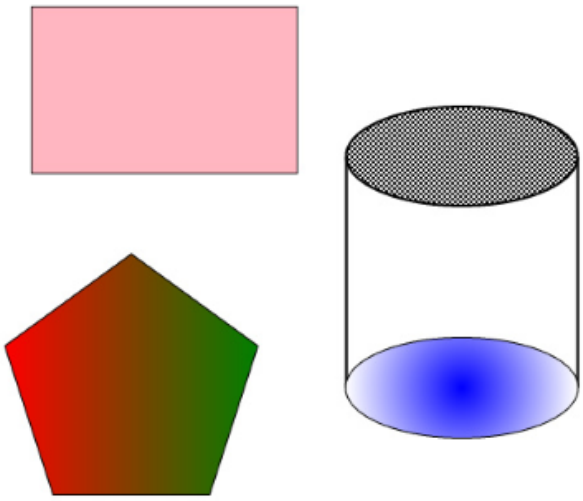
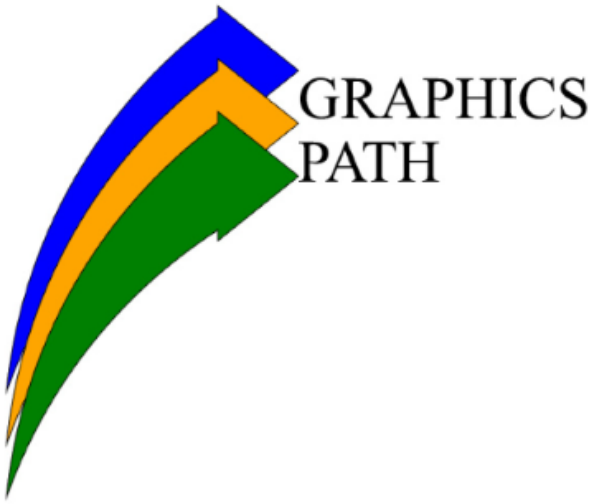
- Draw and Fill Shapes
- Clip Image
- Align Image
- Matrix Transformation

Draw and Fill Shapes

GcImaging 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, GcImaging 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

Shapes	Graphics Path
	

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 respectively.
5. Draw a cylinder with the help of line and ellipse using **DrawLine** and **DrawEllipse** methods.

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 drawng 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 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 which accepts a specified pen as its parameter.
7. Apply background color to the path using **FillPath** method 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

GcImaging, 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 GcImaging, see GcImaging sample browser.

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 GcImaging, 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");
```

For more information about implementation of clipping using GcImaging, see GcImaging sample browser.

Align Image

GcImaging 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 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");
```

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..

GcImaging 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.]
```

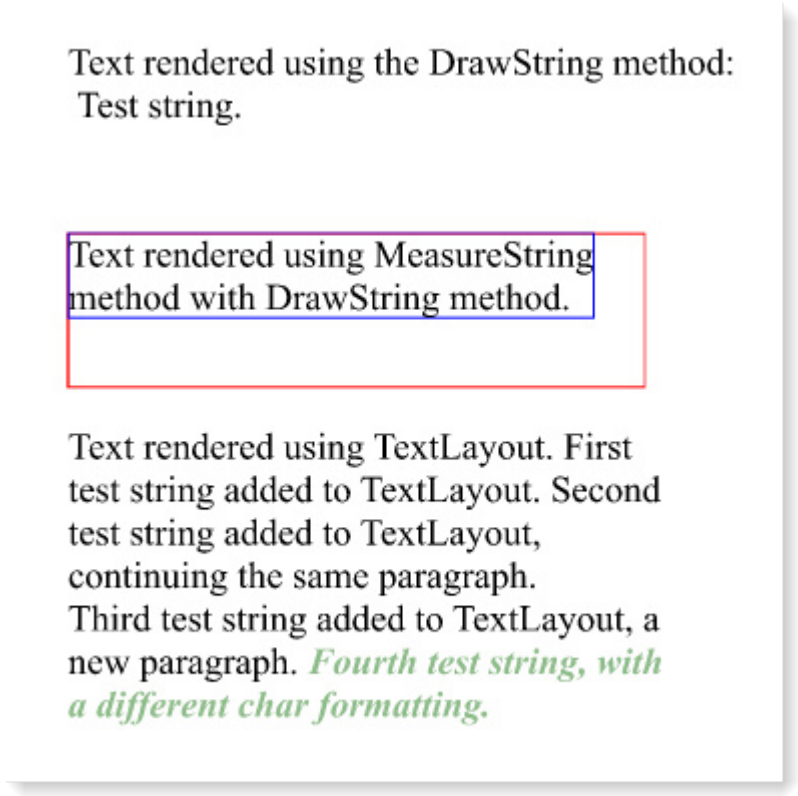
```
var rotate = Matrix3x2.CreateRotation((float)(-70 * Math.PI) / 180f);
var translate = Matrix3x2.CreateTranslation(Inch * 3, Inch * 5);
var scale = Matrix3x2.CreateScale(0.7f);

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 GcImaging, see GcImaging sample browser.

Work with Text

GcImaging 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.



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);
```

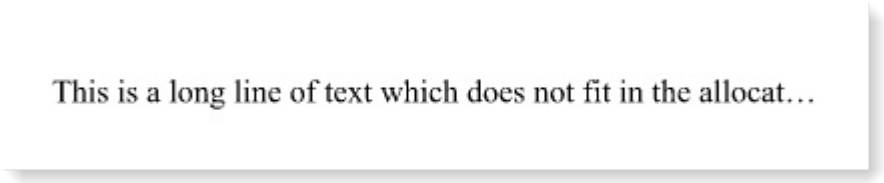
```
//TextFormat to specify font and other character formatting
var tf = new TextFormat()
{
    Font = Font.FromFile(Path.Combine("Resources", "Fonts",
    "times.ttf")),
    FontSize = fontSize
};

//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

GcImaging 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 overflowing 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.



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

GcImaging 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",
               "calibrib.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

GcImaging 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#

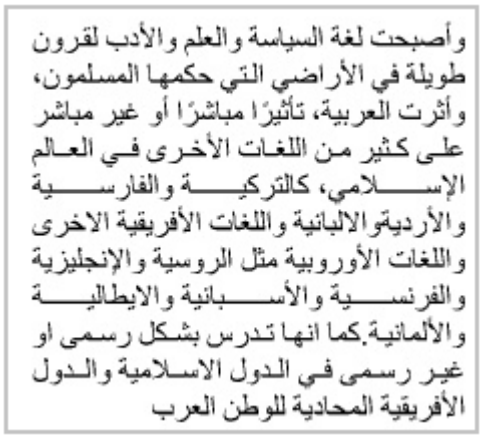
```
//Initialize Fonts
Font garlicFont = Font.FromFile(Path.Combine("Resources",
                  "Fonts", "Garlicembrace.woff"));
Font emojiFont = Font.FromFile(Path.Combine("Resources",
                 "Fonts", "seguiemj.ttf"));

//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 characrters 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

GcImaging 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.



To set the direction of text from right to left direction, you can set the value of the **RightToLeft** property to **true**.

C#

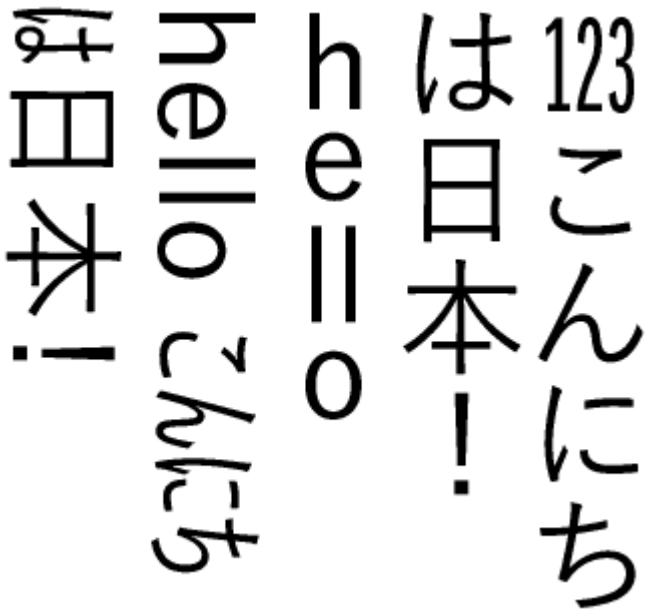
```
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(300, 300, true);
```

```
//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 الاخرى واللغات" +
    "الأوروبية مثل الروسية والإنجليزية والفرنسية والأسبانية والايطالية " +
    "والألمانية.كما انها تدرس بشكل رسمى " +
    "او غير رسمى في الدول الاسلامية 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

GcImaging 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#

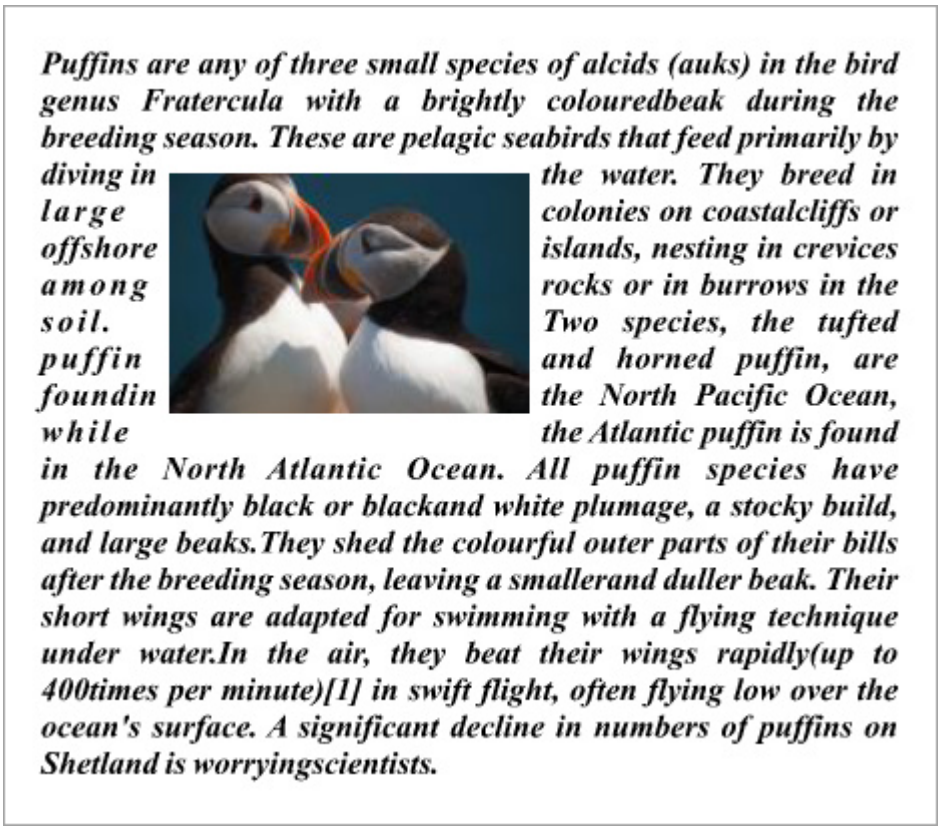
```
//Initialize GcBitmap
GcBitmap bmp = new GcBitmap(90 * 4, 80 * 4, true, 384f, 384f);

//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 GcImaging, 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.



C#

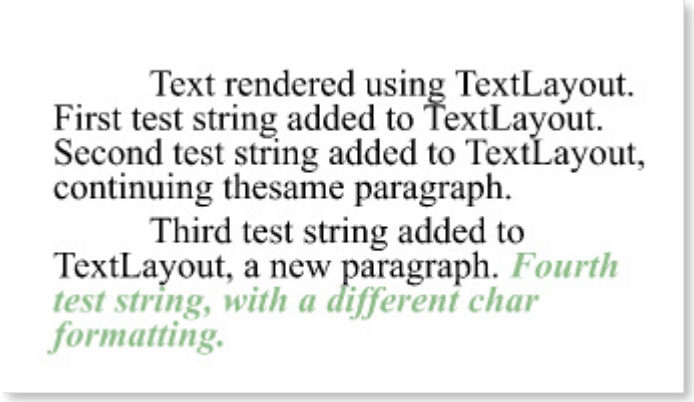
```
//Initialize GcBitmap
GcBitmap origBmp = new GcBitmap(500, 500, true);

//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

GcImaging 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.



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 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#

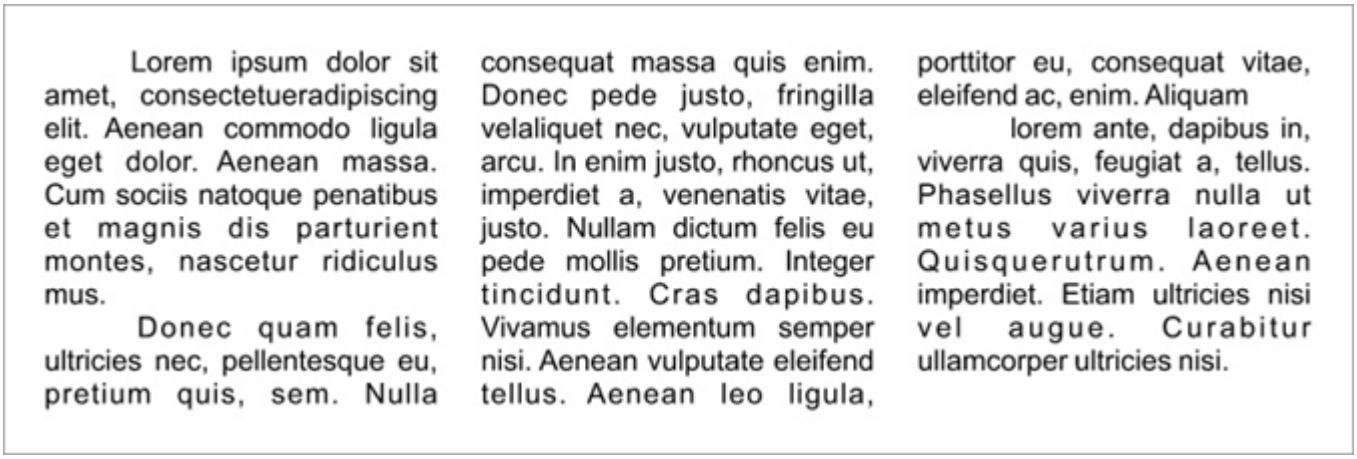
```
var Inch = 96;
const float fontSize = 14;

//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;

//Paragraph formatting can also be set, here we set first
//line offset, spacing between paragraphs and line spacing:
tl.FirstLineIndent = Inch * 0.5f;
tl.ParagraphSpacing = Inch * 0.05f;
tl.LineSpacingScaleFactor = 0.8f;

// Draw it on the page:
PointF pt = new PointF(Inch, Inch * 2);
g.DrawTextLayout(tl, pt);

//Save the image rendering different shapes
origBmp.SaveAsJpeg("ParagraphFormat.jpeg");
```

Back to Top

Text Splitting

GcImaging 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 splitted text which is rendered using the DrawTextLayout method.


Lorem ipsum dolor sit amet, consectetueradipiscing elit. Aenean commodo ligula eget dolor. Aenean massa. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus.

Donec quam felis, ultricies nec, pellentesque eu, pretium quis, sem. Nulla consequat massa quis enim. Donec pede justo, fringilla velaliquet 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

lorem ante, dapibus in, viverra quis, feugiat a, tellus. Phasellus viverra nulla ut metus varius laoreet. Quisquerutrum. Aenean imperdiet. Etiam ultricies nisi vel augue. Curabitur ullamcorper ultricies nisi.

The following example illustrates text splitting where 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, consectetuer" +
    "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");
```

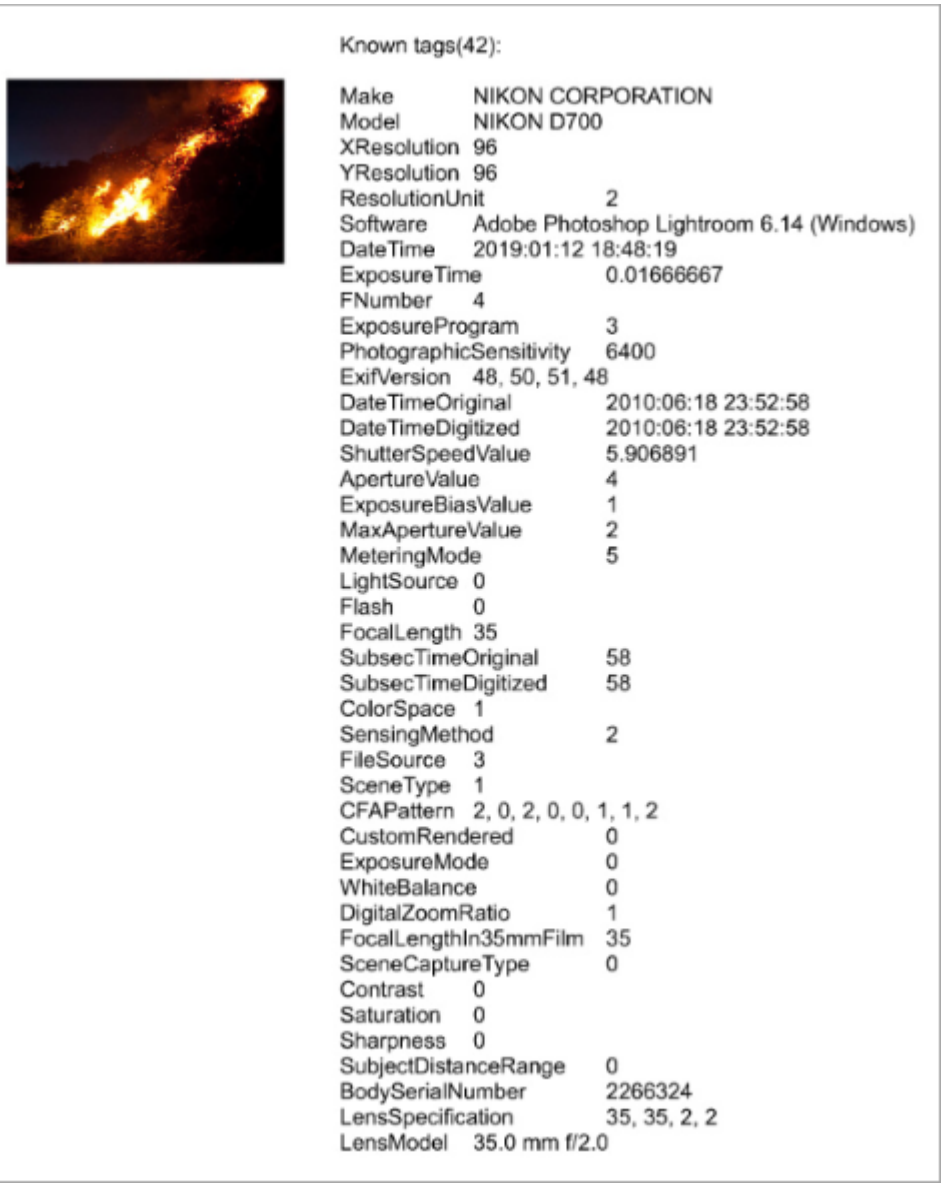
Back to Top

For more information about working with text using GcImaging, see GcImaging sample browser.

Work with Exif Metadata

The GcImaging library supports the extraction and modification of Exif metadata from various image formats, such as JPEG, PNG or TIFF files. With GcImaging, 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..

GcImaging 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.



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.
4. Access all the known tags of the profile using the **GetTags** method.
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 GcImaging, see GcImaging sample browser.

Samples

All the GcImaging 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.

List of samples

The complete list of available GcImaging sample projects is mentioned below:

Basic Feature Samples

Features	Sample	Description
	Hello, World!	Demonstrates how to create an image that includes some text.
Images	Bitmap Transforms	Demonstrates how to use bitmap transformations, such as, resizing, flipping and rotating.
	Create Thumbnails	Demonstrates how to create thumbnails using different interpolation modes.
	Enlarge Image	Demonstrates how to enlarge an image using different interpolation modes
	Indexing	Demonstrates how to convert full color images to indexed images.
	Round Clip	Demonstrates how to make a round clipping of an image.
Effects	Brightness and Contrast	Demonstrates how to adjust image brightness and contrast.
	Saturation	Demonstrates how to change the color saturation of an image.
	Gamma Correction	Demonstrates how to apply gamma correction.
	Dithering 1	Demonstrates how to use dithering methods NoDithering, FloydSteinberg, JarvisJudiceNinke.
	Dithering 2	Demonstrates how to use dithering methods Stucki, Atkinson, Burkes.
	Dithering 3	Demonstrates how to use dithering methods Sierra, TwoRowSierra, SierraLite.
	Thresholding Effects	Demonstrates how to apply thresholding effects.
	Grayscale Effects	Demonstrates how to apply grayscale effects.
	Matrix Effects 1	Demonstrates how to change the hue of an image based on the rotation angle.
	Matrix Effects 2	Demonstrates how to use LuminanceToAlphaEffect, OpacityEffect and SepiaEffect.
	Auto Levels	Demonstrates how to improve an image with autolevels.
	Auto Contrast	Demonstrates how to improve an image with auto contrast.

Text	Text Rendering	Demonstrates the basics of rendering text in GcImaging
	Text Trimming	Demonstrates how to display ellipsis at end of string that did not fit in the allocated space.
	Antialiasing	Demonstrates how to use different antialiasing text rendering modes.
	Text Columns	Demonstrates how to draw text paragraphs in columns.
	Characters and Fonts	Demonstrates how to draw text using different fonts.
	Japanese Text	Demonstrates how to draw Japanese text on a GcBitmap.
	Arabic Text	Demonstrates how to draw Arabic text on a GcBitmap.
	Tate Chu Yoko	Demonstrates how to render horizontal runs in a vertical text (tate chu yoko).
	Outlined Text	Demonstrates how to draw outlined text and filling text using brushes.
Drawing	Shapes	Demonstrates how to draw various shapes.
	Round Rectangles	Demonstrates how to draw round rectangles.
	Clip Region	Demonstrates how to clip drawing operations to an arbitrary region.
	Transformations	Demonstrates how to use transformation matrix to draw on graphics.
	Gradients	Demonstrates how to use linear and radial gradient brushes.
	Gradients 2	Demonstrates how to draw geometric shapes and using gradients.
	Watermark	Demonstrates how to draw watermark on an image.
	Watermark 2	Demonstrates how to render angled watermark text on an image.
	Hatch Styles	Demonstrates the different hatch brush styles.
Miscellaneous	Swap Colors	Demonstrates how to swap red and blue channels in a picture.
	Highlight Clipping	Demonstrates how to find and show blown out highlights.
	Black Clipping (TIFF)	Demonstrates how to find and show shadow clipping in a TIFF image.
	Black Clipping (JPEG)	Demonstrates how to find and show shadow clipping in a JPEG image.
	Limited Palette	Demonstrates how to reduce image color depth.
	Blending Modes	Demonstrates all 16 available blending modes in a 4x4 grid.
	All Blending Modes	Demonstrates how to find and show shadow clipping in a JPEG image.
Exif	Show Exif Info	Demonstrates how to extract Exif info.
	Clear Exif Info	Demonstrates how to remove Exif info from an image.
Tiff	Extract Frames	Demonstrates how to extract frames from a TIFF image.
	Make TIFF	Demonstrates how to create a multi-frame TIFF from a set of images.
	Different Image Types	Demonstrates how to create a multi-frame TIFF with different image type frames.
	Dithering Effects	Demonstrates how to create a multi-frame TIFF using different dithering methods.

Gif	Make Gif	Demonstrates how to create a GIF from a set of images.
	Indexed Gif	Demonstrates how to convert true color GIF frames to indexed 8bpp.
	Update Gif	Demonstrates how to change hue of an existing GIF.