The ImageMagick graphics library

A gentle introduction to Magick++

Rev 1.0.5

1 About this document

This document is an introductory tutorial to the free-software Magick++ C++ graphics library, and it thus covers only the basic Magick++ methods for image manipulation. For a complete reference guide to the Magick++ library the reader is advised to consult the original library documentation, as well as various application notes that address specific functionalities (these are freely available on the internet).

This document has been written such that it gradually introduces the various concepts that the Magick++ library is based upon: it starts with a brief overview of the library and how it is meant to be used as a component inside an application, continues with describing the meaning of a Canvas as the drawing area utilized by the Magick++ library, then it presents some essential characteristics of an Image object (which is itself based on the concept of Canvas), and ends with a detailed presentation of a collection of methods that the Image object provides for image generation, manipulation, and storage.

Only a limited set of Magick++ image manipulation methods is covered in this tutorial, but this set does however provide the necessary image manipulation tools for a large number of applications, including web-based server applications that need to generate dynamic images for embedding in web pages (e.g. pie charts, wire graphs, bar graphs, annotated pictures, etc).

> IMPORTANT:
> *The reader is assumed to be familiar with all the C++ terminology that is being used throughout this document.*

License

This document is Copyright (c) Information Technology Group www.itgroup.ro
(c) Alin Avasilcutei, Cornel Paslariu, Lucian Ungureanu, Virgil Mager.

Permission is granted to copy, distribute and/or modify this document
under the terms of the **GNU Free Documentation License, Version 1.2**
with no Invariant Sections, no Front-Cover Texts, no Back-Cover Texts.

2 Introducing the Magick++ library

The Magick++ library is *a set of C++ wrapper classes that provides access to the ImageMagick* package functionality from within a C++ application.

ImageMagick is a free software package used for image manipulation, and it is available for al the major operating systems: *Linux, Windows, MacOS X, and UN*X*. It includes a GUI for rendering images, command line utilities for image processing, and APIs for many programming languages: Magick++ (C++), MagickCore (C), MagickWand (C), ChMagick (Ch), JMagick (Java), L-Magick (Lisp), PerlMagick (Perl), MagickWand for PHP (PHP), PythonMagick (Python), TclMagick (Tcl/TK).

- Note: ImageMagick is provided with *automated installers* for Linux, Windows, and MacOS X

ImageMagick is released under a license that is compatible with, but less restrictive than, the GNU General Public License: significantly, *ImageMagick may be included in, and used by, a software package, fully or in part,* with proper attribution and with the ImageMagick license file attached.

The Magick++ library can perform the following classes of operations on images:

- *create* new images, or *read* existing images,
- *edit images* : flip, mirror, rotate, scale, transform images, adjust image colors, apply various special effects, or draw text, lines, polygons, arcs, ellipses and Bezier curves,
- *compose new images* based on other images,
- *save images* in a variety of formats, including GIF, JPEG, PNG, SVG, and PostScript.

Using the Magick++ library in an application

In order to use the Magick++ library in a C++ application one should first install the ImageMagick package (the automated installers may be used). One needs to include the relevant header files in the application's source(s), and link the application binary against the ImageMagick runtime libs:

- In the source files: after '`#include <Magick++.h>`' (in the application's files, as required), add the '`using namespace Magick;`' statement, or add the prefix '`Magick::`' to each Magick++ class/enumeration name.
- In a GNU environment the linker's LDFLAGS can be configured using the `Magick++-config` script: `` `Magick++-config --cppflags --cxxflags --ldflags --libs` ``

  NOTE: *the ` character is the backquote character, it cannot be replaced with ' or " !*
  For example, in Anjuta 1.2.x (on a GNU OS) paste the line above in the 'Settings->Compiler and Linker Settings->Options tab -> Additional Options: Linker Flags (LDFLAGS)' box
- Windows applications can be built using a MSVC6+ compiler: they can be derived from one of the demo projects, must ship with the ImageMagick DLLs and must “initialize the library” *prior to any Magick++ operation*: `InitializeMagick(path_to_ImageMagick_DLLs);`

Library versions

ImageMagick and the Magick++ libraries are advertised as being under development at the time of writing this document. As such, this document contains information about the functionality provided by the library **Version 6.2.5.4**, based on actually testing the library API for this version. It has been found during the tests that a number of advertised features work in an unpredictable manner, or are inconsistent with the library specifications. A very hilarious example is the DrawableRotation and DrawableTranslation objects that require '*angle/2*' (instead of 'angle') and '*displacement/2*' (instead of 'displacement') as arguments. If this could be regarded as some hacker joke made by one of the library implementers, more serious problems seem to exist: for example, the above two coordinate system transformations don't work together at all (they give nonsensical effects when drawing objects on a canvas).

IMPORTANT:
This document only presents features that were effectively tested on Magick++ version 6.2.5.4

3 Images in the Magick++ library

In object oriented graphics, an image (also known as a 'digital image') is described by an *object featured with a self-rendering method*. In the Magick++ implementation each such image object always has an *associated canvas* which actually holds the picture data (or it may also be an *empty* canvas, which is not the same with a "blank" canvas because in the latter case the canvas actually holds a background color).

What is a Canvas

A canvas can be described from several perspectives:

- From visual point of view, a canvas represents a *virtual surface where an application can draw*, paint or paste pieces of image.
- From the point of view of the internal representation, the canvas is stored as *an array of pixels* (also known as a 'bitmap'), with each pixel being stored in an certain format (usually based on the Red-Green-Blue-Alpha components, but other pixel formats may also be implemented).

The Geometry class

A number of Magick++ methods use a parameter of type 'Geometry' for specifying (mostly) the size of a rectangular object. The 'Geometry' class is relatively complex (actually reflecting the X11 geometry specification), but *for the purpose of this tutorial* it can be thought of as being a mechanism that *encapsulates two dimensions (x and y) into a single object* which is further used by Magick++ methods.

```
Geometry::Geometry(unsigned int x, unsigned int y);
Geometry::Geometry(const String& geometry);

// examples:
Geometry g1(100,200);    // numeric constructor
Geometry g2("300x400");  // string constructor
```

Pixels in Magick++ library

The PixelPacket structure

The data format of the pixel is of type '*PixelPacket*'. The 'PixelPacket' is a *C structure* used to represent pixels. The PixelPacket members are:

- 'red'
- 'green'
- 'blue'
- 'opacity' : this PixelPacket member defines the "opacity" of the pixel, thus allowing an image to include transparency information (an example of image file format that is able of preserving the image transparency information is the PNG format)

Quantizing levels: the Quantum type and the MaxRGB constant

Each member of the PixelPacket structure is an unsigned integer number, and depending on how the library was compiled it may be implemented on 8 bits or 16 bits (i.e. the total size of a PixelPacket is 4*8=32 bits or 4*16=64 bits); thus, the number of quantizing levels for each of the PixelPacket components can be 2^8=256 or 2^16=65536.

In any case, Magick++ provides the applications with the '*MaxRGB*' *constant which always specifies the number of quantizing levels - 1* as available on the particular version of the library that is being used (e.g. for a library version compiled with 8 bits per PixelPacket component the MaxRGB constant is 255).

In order to accommodate the implementation-specific values of quantizing levels, Magick++ provides the 'Quantum' data type. This type is an unsigned integer value that may be implemented as unsigned char, unsigned short int, etc, but it is always guaranteed to be able of holding all the quantizing levels as available on the particular version of the library that is being used (i.e. it can always hold the 'MaxRGB' value).

The PixelPacket structure members are of type Quantum, and are in range [0, MaxRGB]:

```
struct PixelPacket {
       Quantum red;
       Quantum green;
       Quantum blue;
       Quantum opacity;

};
```

The Color class

The Magick++ library provides a *class* 'Color' that can be assigned to, and from, PixelPacket data structures, thus facilitating an object-oriented approach for handling PixelPacket data structures.

The 'Color' class is also used as argument type by a number of Magick++ methods that set the color attributes of various graphical operations.

The following table lists the most important Color methods:

```
// constructors
Color::Color();
Color::Color(const string& color); // the string 'color' is one of a set of "inbuilt"
                                   // colors: "red", “blue”, "magenta", etc
Color::Color(Quantum red,          // parameters in range 0 to MaxRGB
             Quantum green,
             Quantum blue,
             Quantum alpha); // alpha is transparency: 0=opaque, MaxRGB=fully transparent
Color::Color(const PixelPacket& pixel);

// getter functions
Quantum Color::redQuantum();
Quantum Color::greenQuantum();
Quantum Color::blueQuantum();
Quantum Color::alphaQuantum();

// setter functions
void Color::redQuantum(Quantum red);
void Color::greenQuantum(Quantum green);
void Color::blueQuantum(Quantum blue);
void Color::alphaQuantum(Quantum alpha);

// conversion functions to and from PixelPacket
Color::operator PixelPacket() const;
const Color& Color::operator=(const PixelPacket& pixel);
```

The Magick++ Image object

The Image class

A Magick++ Image is a special object that implements a number of characteristic features:

- It has methods for *self-rendering* on an output interface (a computer screen, a file, etc)
- It has an associated *canvas*, where the canvas is the data storage area used to hold the picture data that the image object can render
- It has a set of specific methods that allow various *ways to access the canvas* of the Image such that the picture can be modified
- It can be used in conjunction with other Image objects in order to create *new composite images*

The canvas format

The Magick++ Image canvas is stored internally as a *contiguous array of pixels*, where each pixel is *a structure* of type 'PixelPacket'.

Note: *There is no special* '*Canvas*' *type* defined by the Magick++ library

Transparency

As it was already stated in the 'PixelPacket structure' section above, the format of an individual pixel (of type 'PixelPacket') encapsulates the three fundamental color components 'red', 'green', and 'blue', plus an extra 'opacity' information. *The* '*opacity*' *component is used when combining multiple image elements on the same canvas* (the various Magick++ methods of drawing and combining images are presented in subsequent sections throughout this document)

> Examples:
> When a green line is drawn over a red background, if the red background is totally opaque and the green line has 50% opacity then the resulting line will actually have a "grayish yellow" color (note that this "grayish yellow" looks *very different* from a "true yellow" on the screen).
> Similarly, when placing a semitransparent image over an opaque image, the resulting image will be a *blending* of the colors coming from both images (based on the opacity values involved in the pixel-by-pixel bending process).

Note: if pasting a fully transparent image over a 'support image', the resulting image's pixels will be exactly the pixels of the 'support image', i.e. *the opacity information of the pasted image is not copied over the value of the opacity of the support image, but rather a blending algorithm is used.*

Important limitations for transparency-related operations:

Not all image formats support transparency, and Magick++ itself has its own limitations when loading/saving images that contain transparency.

- Magick++ library functions can load images that *contain* transparency information if their format is GIF or PNG
- Magick++ library functions can *preserve* the transparency information if the save format is PNG

Creating an image

Images are created using the Image class contructors. A (possibly empty) canvas is automatically created when an Image object is created. The created Image object uses its associated canvas for storing the picture data, and for performing the various graphical operations.

```
Image::Image();
Image::Image(const Image& source_image);  // copy constructor
Image::Image(const Geometry& size, const Color& color);
Image::Image(const string& file_location_or_URL);
Image::Image(const Blob& source_blob);  // Blobs are discussed in a later section below


// create an empty-canvas image
//this is *not* a "blank" image, it's completely empty as its canvas has 0x0 dimnesions
Image empty_image();

// create a blank image canvas with 640x480 size and 'white' color as background:
Image blank_image( Geometry(640, 480), Color(MaxRGB, MaxRGB, MaxRGB, 0));
// or also, by using the automatic C++ type conversions for the arguments:
Image blank_image("640x480", "white");

// create an image from URL
Image url_image("http://www.serverName.com/image.gif");
Image local_file_image("my_image.gif");   // here the URL points to the local filesystem

// create an image from a Blob (details fololow in the Blob secttion below)
Image image_from_blob(my_blob);
```

Note: Magick++ documentation recommends automatic Image variables (i.e. allocated on the stack, via declaration of local variables) as the prefered way of creating Image objects, instead of explicit allocation (via 'new').

Accessing image attributes

```
// Canvas geometry
unsigned int Image::columns(); // returns an unsigned int representing the my_image width
unsigned int Image::rows();    // returns an unsigned int representing the my_image heigth


// the image Format
// this attribute is *not* related to the internal representation of the image on canvas,
// which is always a PixelPacket array;
// it is automaticlally set when reading an image based on the image encoding
// or it can be set within the program
void Image::magick(const string& image_format); // sets the my_image format;
                                                // the format string can be "GIF", etc
string Image::magick();                         // returns a string value representing the
                                                // image format (e.g. “GIF”, “JPEG”, etc)
// Notes:
// for a new image, this attribute is automatically set to a special 'Magick++ internal
// format' value of "XC"  (Constant image uniform color); the relevance of this attribute
// is presented in following sections
```

Reading and writing images

```
Image::read(const string& filename);
Image::write(const string& filename);
Image::write(Blob* blob);   // writing to Blobs is discussed in the Blobs section below


// Reading the contents of a disk file into an image object can be performed
// if the default Image costructor was used
// Example:
Image my_image();  // create an *empty* image using the default Image constructor
my_image.read("aGIFImageFile.gif");   // read a GIF image file from disk;
                                      // the image format is automatically set to GIF

// Writing an Image object to a disk file:
// the format of the image file is either specified by the file extension,
// or if the file extension is missing then the write format is taken from the image's
// "format" attribute (see the 'Image::magick()' method above)
// Example:
my_image.magick("png");                   // set the "format" attribute of my_image to PNG
my_image.write("file_name_no_extension"); // write to disk an image file (based on
                                          // my_image canvas); the image is saved to the
                                          // file file_name_no_extension in PNG format
my_image.write("file_name_explicit_extension.gif"); // write to disk an image file (based
                                                    // on my_image canvas); the format of
                                                    // file_name_explicit_extension is
                                                    // GIF, and *not* PNG
// NOTE : writing 'my_image' to disk via 'Magick::write()' in a GIF file format doesn't
// change the my_image format (i.e. the "format" attribute my_image format remains PNG)
```

- IMPORTANT: if an 'Image' object is saved in a format which does not support the full color set of the image to be saved, then the 'Image::write()' method will alter the image that is being saved. Such a situation can be avoided by first making a copy of the image to be saved, and then saving the copy in the desired format:

```
// example: save 'my_image' by means of a temporary copy
// in order to prevent the original from being altered
Image temp_image(my_image);           // make a copy of the image to be saved
temp_image.write("gif_version.gif");  // save the copy oin GIF format
```

Getting direct access to the canvas pixels

```
// set or get the color for the pixel at position (x,y) on the canvas
void Image::pixelColor(unsigned int x, unsigned int y, const Color& color);
Color Image::pixelColor(unsigned int x, unsigned int y);


// Example: setting pixels
Image my_image("640x480", "white"); // start with creating a white-background canvas
my_image.pixelColor(50,50,Color("red"));  // set the pixel at position (50,50) to red
my_image.pixelColor(5,5,Color(MaxRGB,0,0,MaxRGB/2)); // set semitransparent red at (5,5)

// Example (continued from above): reading pixels
Color pixel_sample;                       // create a color pixel object
pixel_sample = my_image.pixelColor(5,5);  // set pixel_sample with the value read form
                                          // pixel position (5,5)
```

The Pixel Cache: getting indirect access to an image canvas

The pixels (of type 'PixelPacket') constituting the canvas of an Image object can be also accessed using a special method that involves the creation of an 'image pixel cache' which, as its name suggests, is a temporary (cache) workspace where a number of graphical operations can be performed without having to update the image itself after each such operation.

The *data type* provided by the Magick++ library in order to assist the creation and manipulation of an image pixel cache is 'Pixels', and represents a rectangular window (a view) into the actual image pixels (the image may be in memory, memory-mapped from a disk file, or entirely on disk). The way the pixels are stored inside the image pixel cache is identical with the way they are stored for a canvas, i.e. a contiguous array of PixelPacket structures.
In order to use the image pixel cache one must:

- Create a pixel cache associated with an image by using the Pixels constructor (it takes an image reference as argument)
- Obtain a region from the existing image via the Pixels class 'get()' method; this method returns a PixelPacket* value which points to the first pixel in the pixel cache
- At this stage, the pixels from the cache can be read and/or written via direct access to the composing PiexlPacket structures
- Finally, any changes made to the pixel cache must be concluded with a 'sync()' function that updates the actual image with the changes that were made in the pixel cache.

Note: The main benefit of the 'Pixels' class is that it provides *efficient access to raw image pixels*. Depending on the capabilities of the operating system, and the relationship of the window to the image, the pixel cache may be a copy of the pixels in the selected window, or it may be the actual image pixels. In any case calling sync() insures that the base image is updated with the contents of the modified pixel cache.

```
Pixels::Pixels(const Image& image) // create a pixel cache and “connect” it to an image
PixelPacket* Pixels::get(unsigned int start_x, unsigned int start_y,
            unsigned int size_x, unsigned int size_y)  // cache a rectangular image area
// Notes:
// * before creating a Pixels cache from an image, the image must be "locked"
//   using the 'Image::modifyImage()' method
// * after finalizing the operations on the Pixels cache the image must be updated
//   with the 'Pixels::sync()' method


// Example of using an image pixel cache
Image my_image("640x480", "white"); // we'll use the 'my_image' object in this example
my_image.modifyImage();             // Ensure that there is only one reference to
                                    // underlying image; if this is not done, then the
                                    // image pixels *may* remain unmodified.   [???]

Pixels my_pixel_cache(my_image); // allocate an image pixel cache associated with my_image
PixelPacket* pixels;             // 'pixels' is a pointer to a PixelPacket array
// define the view area that will be accessed via the image pixel cache
int start_x = 10, start_y = 20, size_x = 200, size_y = 100;
// return a pointer to the pixels of the defined pixel cache
pixels = my_pixel_cache.get(start_x, start_y, size_x, size_y);
// set the color of the first pixel from the pixel cache to black (x=10, y=20 on my_image)
*pixels = Color("black");
// set to green the pixel 200 from the pixel cache:
// this pixel is located at x=0, y=1 in the pixel cache (x=10, y=21 on my_image)
*(pixels+200) = Color("green");
// now that the operations on my_pixel_cache have been finalized
// ensure that the pixel cache is transferred back to my_image
my_pixel_cache.sync();
```

Blobs: storing encoded images in memory

Encoded images (e.g. JPEG, GIF, etc) are most often written-to and read-from a disk file (by using the Image object's methods for saving/reading images in various formats to/from file), but they may also reside in memory. *Encoded images in memory are also known BLOBs* - Binary Large Objects. Magick++ provides the 'Blob' class for representing images that are stored in memory *in encoded format*.

```
Blob::Blob();
Blob::Blob(void* data, unsigned int size); // explicitly specifies a memory area to be
                                           // associated with the new Blob object
Blob::operator=(const Blob& blob);


// Examples of using Blobs in conjuction with Images
Blob my_blob;                      // create a blob
Image my_image("my_image.gif");    // create an image form a GIF image file
my_image.magick("JPEG");           // set JPEG output format
my_image.write(&my_blob);          // encode 'my_image' in JPEG format,
                                   // and store the encoded image in my_blob
Image image_from_blob(my_blob);  // create an image from the JPEG blob
                                 // (use the Blob-based Image constructor)
image_from_blob.magick("BMP");   // set the image format to bitmap
image_from_blob.write("image_from_blob.bmp");   // save the image on disk in BMP format
```

4 Drawable objects

The Magick++ drawing mechanism is based on *creating* '*drawable objects*', *which are afterwards placed on the canvas* using specific methods; these methods specify the position where the drawable objects are to be placed, the transparency of the drawing operations, etc. Thus, the Magick++ library takes an *object-oriented approach to drawing* (where the objects are the drawable objects), as opposed to other graphical libraries that only provide drawing *functions* that directly modify the pixels on a canvas.

This chapter incrementally introduces the concepts behind the Magick++ drawing mechanism, together with a selected set of drawable objects.

Foundation concepts of the Magick++ drawing mechanism

The generic Drawable class

The 'Drawable' class is a base class from which all drawable object classes are derived. All graphical object classes (such as 'DrawableLine', 'DrawableArc', 'DrawableBezier', etc) are derived from the base 'DrawableBase' class.

The Coordinate class

The 'Coordinate' structure represents a pair of (x,y) coodinates, and a 'Coordinate list' represents a list of objects of 'Coordinate' type (i.e. a list<Coordinate>). Some of the constructors of the Drawable classes accept and/or require as argument a Coordinate, or a Coordinate list.
The default values for the system of coordinates' (0,0) position is the top left corner of the image canvas where the drawings are performed, while the default rotation angle for drawings is 0.

> Note: Magick++ provides a method for modifying both the origin of the draw system of coordinates and the rotation angle from their default values. At the time of writing this document, this library feature has serious implementation bugs and thus it was chosen not to be included here. As a purely informative reference, this feature relies on a couple of "control objects" called 'DrawableRotation()' and 'DrawableTranslation()'

Types of drawable objects

There are two categories of drawable objects, based on what effect they produce on an image canvas when they are drawn: the "Renderable objects" and the "Control objects"

Renderable objects

This class of objects *effectively modify a specific set of pixels* on the canvas according to their geometry; an example of such an object is 'DrawableLine'

Control objects

Objects of this kind *do not produce an immediate effect* on the canvas when they are drawn, but rather they control the way in which Rendered objects are drawn. For example, such a control object is 'DrawableStrokeColor' which controls the color that will be used to draw the outline of the Rendered objects that are drawn (details on the control mechanism are presented in the paragraphs below)

GravityType

The GravityType is a enumeration type that is used for specifying the position of a graphical object within the bounds of an Image; it has the following values: 'NorthWestGravity', 'NorthGravity', 'NorthEastGravity', 'WestGravity', 'CenterGravity', 'EastGravity', 'SouthWestGravity', 'SouthGravity', 'SouthEastGravity'. The default value for gravity is NorthWestGravity (i.e. the upper-left corner).

Drawing on an Image canvas

All the drawable objects can be drawn on an Image's canvas using the 'Image::draw()' method, and this is the recommended way to perform drawing operations. The 'Image::draw()' method is specially provided for this purpose, and it can take as arguments a *list of Coordinates*, *individual Drawable* objects, or a *list of Drawable* objects.

The draw() method

- The drawable objects that the draw() method accepts as arguments can be both 'Rendering objects' and 'Control objects'.
- Drawable objects may be drawn '*one-by-one*' via successive invocations of the Image::draw(Drawable) method for each object to be drawn, or may be drawn '*all-at-once*' by passing a list<Drawable> objects to the Image::draw() method.

```
// draw a polygon shape by using its nodes' coordinates
   void Image::draw (const list<Coordinate>& polygon_node_list);
// draw one shape or text on a canvas using a single Drawable object
   void Image::draw (const Drawable& drawable_object);
// draw shapes or text on a canvas using a list of Drawable objects
   void Image::draw (const list<Drawable>& drawable_object_list);
```

Setting up the draw() parameters

A number of draw parameters-related settings can and/or must be made before starting to perform a sequence of draw operations, e.g. setting up a “pen” color (a.k.a. 'Stroke color') parameter before actually drawing a graphic shape, etc. The specifics of how these parameters are being set up depend on whether a 'one-by-one' or 'all-at-once' drawing method is used (these two drawing methods are exemplified in the paragraphs below).

> IMPORTANT: *It is highly advisable not to rely upon any implicit default values for the draw settings as they can produce very counterintuitive results. Instead, always explicitly set up all the draw characteristics through the use of the corresponding formatting objects and/or methods.*

Drawing objects one-by-one

In this drawing scenario the *settings are made to the Image object that will be drawn upon*, and then a series of draw operations are perfomed on the Image canvas. The Image class methods that support setting the draw options are listed below:

```
Image::strokeColor(const Color& color);  // set the outline color (i.e. the color of
                                         // the countour of the shape)
Image::fillColor(const Color& color);    // set the fill color (see also the note on
                                         // drawing open contours below)
Image::strokeWidth(double width);        // set the width to use when drawing lines
                                         // or object outlines
Image::strokeAntiAlias(bool mode);       // enable or disable draw anti-aliasing for
                                         // outlines; default is enabled
Image::font(const String& font_name);    // set the font face for text operations on the
                                         // canvas (details on the font_name format below)

// Example:
my_image.strokeColor("red");       // this will set the outline color to red for the
                                   // objects that will be subsequently drawn
my_image.strokeAntiAlias(false);   // this will disable the anti-aliasing for the outlines
                                   // that will be be subsequently drawn
```

Notes:

- the drawing settings listed above remain set until new drawing setting will be made.
- because it is *highly unadvisable to use implicit default values*, this method should only be used for draw operations that do not depend on any other settings except the above

Drawing objects all-at-once

In this drawing scenario a *list of drawable objects is generated* and then passed bulk to the draw method. In this case one has to build a set of *special* "*formatting*" *control objects* that determine the parameters of the "bulk draw" operation. For performing the actual draw on a canvas one has to *insert the formatting objects first, followed by the rendering objects* into a list<Drawable>, and then pass this list to the draw method.

> IMPORTANT: *The draw attributes (i.e. the "formatting" objects) at the beginning of the draw list determine the draw characteristics for all drawable objects from the list*. Changing the drawing attributes within the same draw list will generally result in erroneous output. For drawing several shapes, each with its own drawing attributes, *always use separate drawing lists, one for each set of draw attributes*.

Text formatting objects

These objects determine the font face and size when "drawing text" on the canvas

```
DrawableFont::DrawableFont(const string& font_name) // 'font_name' is a string containing
                                                    // a fully qualified X font name


// Following is a listing of several font names generally available on Linux systems
// (fully qualified X font names)
// "-*-helvetica-[medium/bold]-[r/o]-normal-*-*-[80/100/120/140/180]-75-75-*-*-iso8859-1"
//      » typical usage size is 120
// "-misc-fixed-[medium/bold]-[r/o]-[normal/semicondensed]-*-*-[100/120/140/200]-75-75-*-*-
iso8859-1";
//      » typical usage size is 120, and for this size all combinations are valid
//      » for all other sizes the combinations are restricted, but the …-medium-r-normal-…
//        font variants work for all sizes

// Example of setting up a DrawableFont object
   DrawableFont("-*-helvetica-medium-r-normal-*-*-120-*-*-*-*-iso8859-1");
```

this is a Magick++ default font: it varies depending on the system

font name: "-misc-fixed-medium-r-normal-*-*-120-75-75-*-*-iso8859-1"
font name: "-misc-fixed-medium-o-normal--*-120-75-75-*-*-iso8859-1"*
font name: "-misc-fixed-medium-r-normal-*-*-140-75-75-*-*-iso8859-1"

font name: "-*-helvetica-medium-r-normal-*-*-80-75-75-*-*-iso8859-1"
font name: "-*-helvetica-medium-r-normal-*-*-100-75-75-*-*-iso8859-1"
font name: "-*-helvetica-medium-r-normal-*-*-120-75-75-*-*-iso8859-1"
font name: "--helvetica-medium-o-normal-*-*-120-75-75-*-*-iso8859-1"*
font name: "-*-helvetica-bold-r-normal-*-*-180-75-75-*-*-iso8859-1"

Formatting objects for Outlines (i.e. contours)

These objects determine the draw characteristics for drawable objects' contours *(see also the note on drawing open contours below)*

```
// Set color to use when drawing lines or object outlines
// (see also the note on drawing open contours below)
   DrawableStrokeColor::DrawableStrokeColor(const Color& color);
// Set width to use when drawing lines or object outlines.
   DrawableStrokeWidth::DrawableStrokeWidth(double width);
// Enable/disable the anti-aliasing when drawing lines or object outline
// ( to disable, use 'false' as parameter)
   DrawableStrokeAntialias::DrawableStrokeAntialias(bool mode);
```

Formatting objects for Fill operations

These objects determine the fill color and opacity that will be used when a fill operation is required

```
// Specify drawing object fill color where filling is possible
   DrawableFillColor::DrawableFillColor(const Color& color);
```

Important note about drawing lines, text, and open contours

Prior to a draw operation one always has to *set the fill color to fully transparent* (i.e. 'Color(x,x,x,MaxRGB)') for any "open" drawable object (i.e. *straight lines, text, and contours that are not closed*). This is necessary *even for simple open countours such as a polyline* formed of two lines at a certain angle, because otherwise the triangle determined by the two lines plus the imaginary line that connects their non-common extremities would be filled by the fill color (unless set to fully transparent). Also, *even when drawing a simple straight line*, if the fill color is not set to fully transparent, then drawing the line over certain background colors can cause imprevisible results.

Example:
the left image below below presents the result of drawing a *two-segment polyline* when not setting a fully transparent fill color (the fill color has taken a Magick++ default value, which was in this case fully-opaque black), and the right image presents the result of drawing the same two-segment polyline when the fill color was explicitly set to fully transparent.

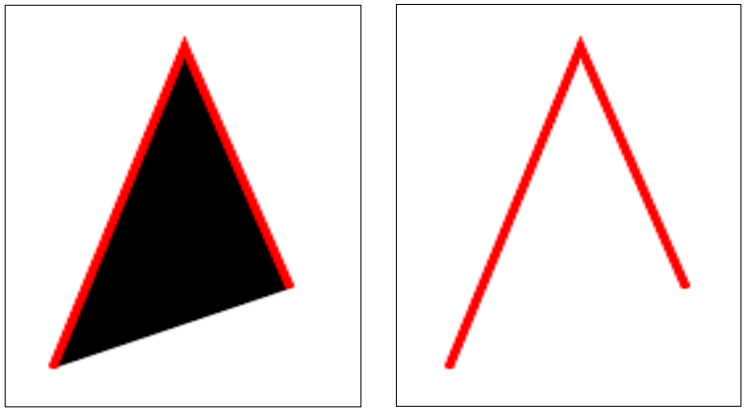


Illustration of the way Magick++ performs fill on open contours

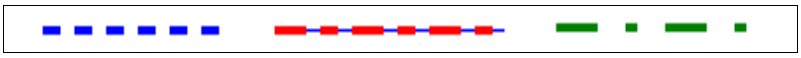
Formatting objects for setting the Drawing Pattern for Outlines

To set a non-continuous pattern for line drawing, Magick++ provides the 'DrawableDashArray' class. The DrawableDashArray constructor take as argument a *zero-terminated array of doubles* that defines the stroke line pattern, i.e. specifies the lengths of alternating dashes and gaps (blanks) in pixels.
Before setting the 'dash_pattern_array', one needs to set the StrokeColor - and optionally the StrokeWidth - for dashes, and the FillColor for gaps. *The width for FillColor is always 1 pixel*, and Magick++ doesn't provide any way to set other value for it (see 2nd line in the example below).

```
    DrawableDashArray::DrawableDashArray(double dash_pattern_array);

// Example for setting draw patterns
// The dash_pattern arrays which define the lines from the image below are:
    double dash_array_8stroke_8fill[] = {8,8,0}; // defines the 1st (blue) line pattern
    double dash_array_15stroke_8fill_8stroke_8fill[] = {15,8,8,8,0}; // 2nd line (red-blue)
    double dash_array_20stroke_15fill_5stroke_15fill[] = {20,15,5,15,0}; //3rd line (green)
// The drawn lines have the following "dashing patterns":
// the first line: 8dash_pixels - 8gap_pixels, and 0 indicating the end of the pattern.
// the second line: 15dash_pixels - 8gap_pixels - 8dash_pixels - 8gap_pixels, and 0 (end)
// the third line: 20dash_pixels - 15gap_pixels - 5dash_pixels - 15gap_pixels, and 0 (end)

// The portion of the source code that determines the draw pattern for the first two
// of the above lines is listed below:
list<Drawable> blue_dashed_line, red_blue_dashed_line;
// setting controls for 1st line
blue_dashed_line.push_back(DrawableStrokeColor("blue"));
blue_dashed_line.push_back(DrawableStrokeWidth(2));
blue_dashed_line.push_back(DrawableFillOpacity(0.0));
blue_dashed_line.push_back(DrawableDashArray(dash_array_8stroke_8fill));
// setting controls for 2nd line
red_blue_dashed_line.push_back(DrawableStrokeColor("red"));
red_blue_dashed_line.push_back(DrawableFillColor("blue"));
red_blue_dashed_line.push_back(DrawableStrokeWidth(2));
red_blue_dashed_line.push_back(DrawableDashArray(dash_array_15stroke_8fill_8stroke_8fill));
```



Renderable objects

This section presents a selection of basic rendering object types which can be used as the basic building blocks for a large set of graphical applications.

Line and Polyline

To draw a line(segment)/polyline on the canvas, use a 'DrawableLine'/ 'DrawablePolyline' object in conjunction with the 'draw()' method. A polyline is defined by at least three points.

```
// the 'DrawableLine' class constructor prototype
DrawableLine::DrawableLine(double start_x, double start_y, double end_x, double end_y);
// the 'DrawablePolyline' class constructor prototype
DrawablePolyline::DrawablePolyline(const list<Coordinate>& coordinates);

// Example:
list<Coordinate> polyline_coordinate_list;
polyline_coordinate_list.push_back(Coordinate(100,100));
polyline_coordinate_list.push_back(Coordinate(150,150));
polyline_coordinate_list.push_back(Coordinate(200,100));
my_image.draw ( DrawablePolyline(polyline_coordinate_list));
```

Polygons

To draw polygon shapes on the canvas, the Magick++ library provides a generic 'DrawablePolygon' class and a specialized classes to draw rectangles: 'DrawableRectangle'.

- The 'DrawablePolygon' constructor needs a *list of Coordinates* as parameter. Depending on the number of elements in the list<Coordinate>, the 'DrawablePolygon' object might describe a triangle (3 elements in list<Coordinate>), quadrilateral (4 elements), etc.
- The 'DrawableRectangle' constructor needs the coordinates of the top-left and bottom-right corner of the rectangle

```
DrawablePolygon::DrawablePolygon(const list<Coordinate>& polygon_coordinates)
DrawableRectangle::DrawableRectangle(double top_left_x, double top_left_y,
                                     double bottom_right_x, double bottom_right_y)

// Example:
list<Drawable> objects_to_draw;      // push in list the rendering objects and the
                                     // associated draw formatting objects
list<Coordinate> coords_of_triangle; // push in list the coordinates of a triangle
coords_of_triangle.push_back(Coordinate(0,0));
coords_of_triangle.push_back(Coordinate(20,100));
coords_of_triangle.push_back(Coordinate(40,0));
// create a triangle and "push" it in the drawable objects list
objects_to_draw.push_back(DrawablePolygon(coords_of_triangle));
// create a rectangle and "push" it in the drawable objects list
objects_to_draw.push_back(DrawableRectangle(100,100, 250,200));
// perform the actual draw on my_image (the list of objects_to_draw are drawn)
my_image.draw(objects_to_draw);
```

Bezier curves

To draw curves on the canvas, the Magick++ library provides the DrawableBezier object.

The DrawableBezier constructor has a *list of Coordinate* as argument type: DrawableBezier(const list<Coordinate>& coordinate_list). The points in the list determine the way in which the curve is drawn in the following way:

- the first point, p1, specifies the position from which the curve will start
- the second point, p2, is used in conjunction with the first point p1 to determine the angle at which the curve starts, and how how fast the curve will diverge from the start tangent
  - the oriented segment [p1,p2] determines the start angle (i.e. the curve will start from p1 by being tangent to the [p1,p2] segment)
  - the length of the segment [p1,p2] determines how fast the curve will diverge from the start tangent in order to reach its ending point
- the third point, p3, is used in conjunction with the fourth point p4 to determine the angle at which the curve ends, and how how fast the curve will approaches the end tangent
  - the oriented segment [p3,p4] determines the ending angle (i.e. the curve will end in p4 by being tangent to the [p3,p4] segment)
  - the length of the segment [p3,p4] determines how fast the curve will converge to the end tangent in order to reach its ending point
- the fourth point, p4, specifies the final position at which the curve will end

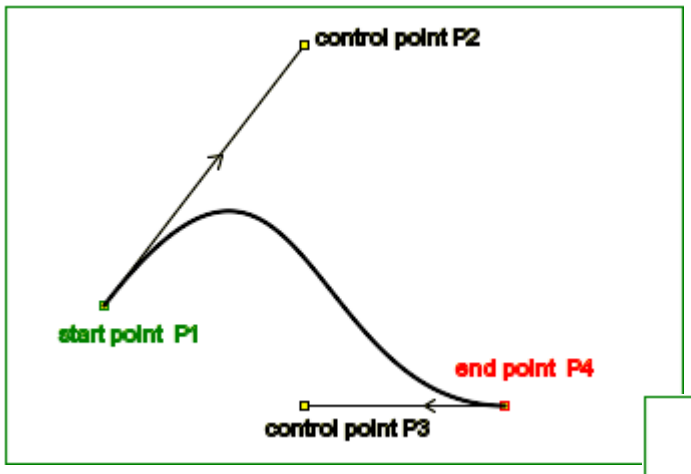
The Magick++ librarly only implements properly the cubic bezier curves (as described above); higher order Bezier curves must be obtained by joining multiple Cubic Bezier curves.

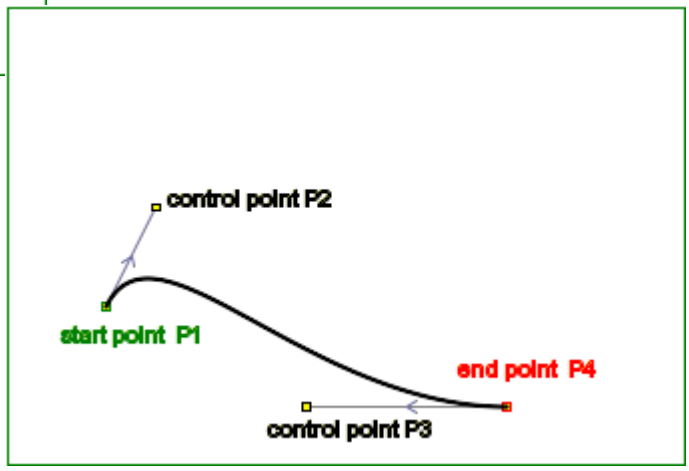
Following is a code example for drawing two Bezier curves, with the difference between them only residing in the positioning of one of the control points (namely P2):

```
DrawableBezier::DrawableBezier(cont list<Coordinate>& bezier_coords_and_control_points)

// Examples:
// The code for the LEFT-side curve below
list<Coordinate> left_cubic_bezier_coord;
left_cubic_bezier_coord.push_back(Coordinate(50,150));  // P1 from image
left_cubic_bezier_coord.push_back(Coordinate(150,20));  // P2 from image
left_cubic_bezier_coord.push_back(Coordinate(150,200)); // P3 from image
left_cubic_bezier_coord.push_back(Coordinate(250,200)); // P4 from image
my_image.draw( DrawableBezier(left_cubic_bezier_coord));// draw the left-side curve

// The code for the RIGHT-side curve below
list<Coordinate> right_cubic_bezier_coord;
right_cubic_bezier_coord.push_back(Coordinate(50,150)); // P1 from image (same as above)
right_cubic_bezier_coord.push_back(Coordinate(75,100)); // P2: DIFFERENT FROM CURVE ABOVE
right_cubic_bezier_coord.push_back(Coordinate(150,200));// P3 from image (same as above)
right_cubic_bezier_coord.push_back(Coordinate(250,200));// P4 from image (same as above)
my_image.draw( DrawableBezier(right_cubic_bezier_coord));// draw the right-side curve
```





Circles, Ellipses, Arcs

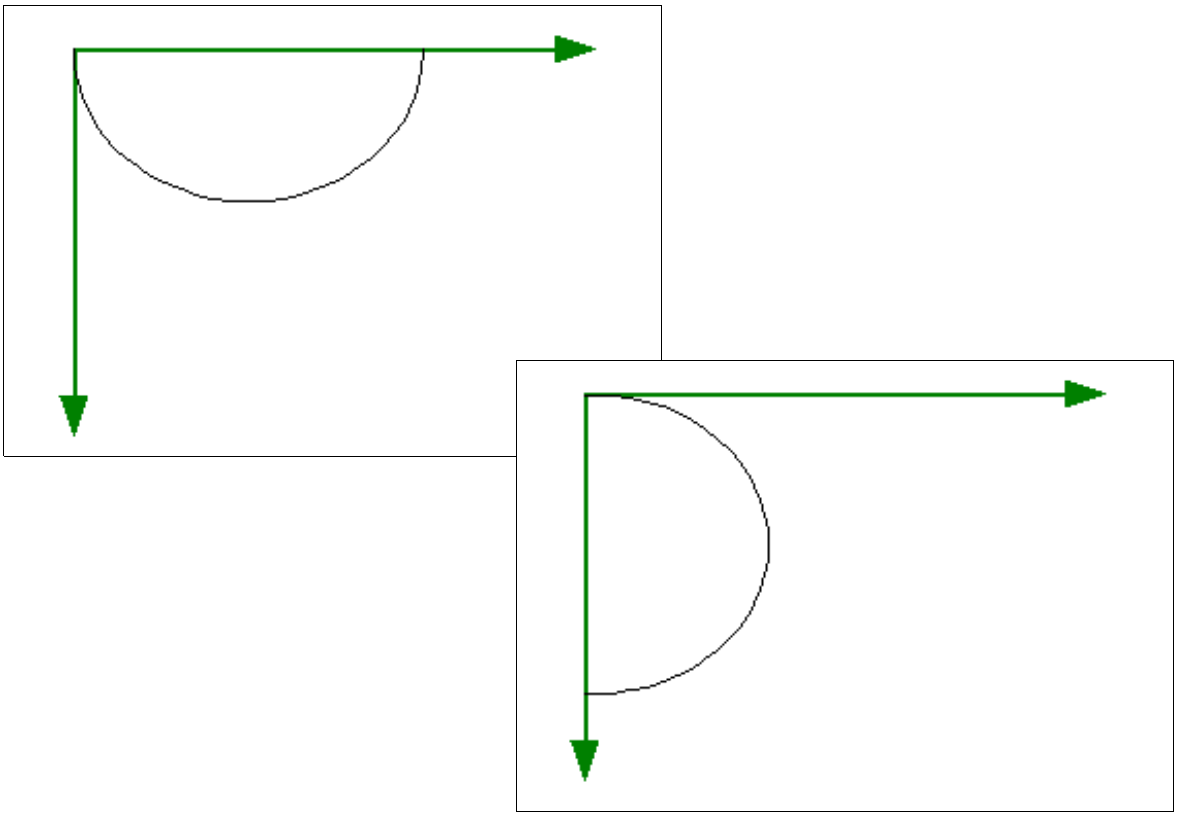
To draw arcs, full circles, ellipses, etc, the Magick++ library provides the DrawableArc object: DrawableArc(double x_top_left, y_top_left, x_bottom_right, y_bottom_right, start_angle, end_angle);

The way the arcs are drawn on the canvas can be understood via the following algorithm:

- first a virtual (i.e. invisible) rectangle is drawn according to the x_* and y_* parameters
- next a full virtual (i.e. invisible) ellipse is drawn inside the rectangle described above
- finally, the actual visible portion (i.e. the arc) is effectively drawn on the canvas over the virtual elipse, *in reverse-trigonometric direction*, using angle-based start and end positions; the start_angle and end_angle are both measured *from the center of the ellipse in reverse-trigonometric direction*.

```
DrawableArc::DrawableArc(double x_top_left, double y_top_left,
                         double x_bottom_right, double y_bottom_right,
                         start_angle, end_angle);

// Examples
// draw a bottom-half or a circle at the top left of the canvas (top-left image below):
my_image.draw(DrawableArc(0, -100, 200, 100, 0, 180));
// draw a right-half or a circle at the top left of the canvas (bottom-right image below):
my_image.draw(DrawableArc(-100, 0, 100, 200, 270, 90));
```



Text

Placing a text on a canvas ca be made using two basic methods: *drawing* the text inside a canvas, or *annotating* a full image

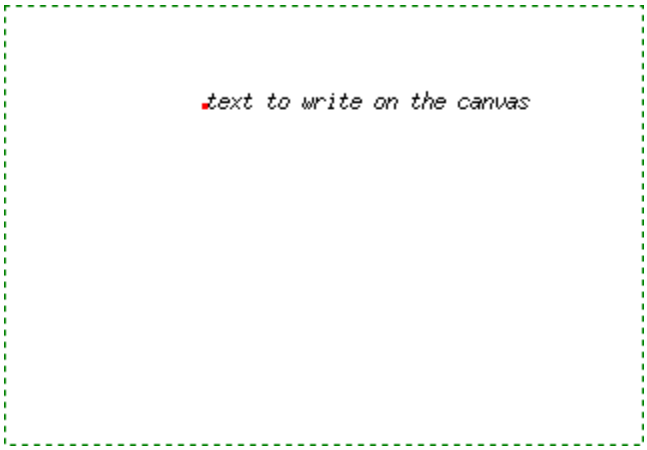
Using the Image 'draw()' method:

When using the 'Image::draw()' method for drawing text an an image canvas, a 'DrawableText' object must be created and used in conjunction with the draw() method. The 'DrawableText object contains both the information regarding the characters that are to be “printed” on the canvas, and the precise position on the canvas where the text must be placed.

> Note: The position specified for drawing represents the bottom-left corner of the imaginary box that surrounds the text

```
DrawableText::DrawableText(double x, double y, const string& text_to_write)

// Example:
   Image my_image( Geometry(320,220), Color("white"));
   list<Drawable> text_draw_list;
// set the text font: the font is specified via a string representing
// a fully qualified X font name (wildcards '*' are allowed)
   text_draw_list.push_back(
        DrawableFont("-misc-fixed-medium-o-semicondensed–13-*-*-*-c-60-iso8859-1"));
 // set the text to be drawn at specified position: x=101, y=50 this case
    text_draw_list.push_back( DrawableText(101, 50, "text to write on the canvas"));
 // set the text color (the fill color must be set to transparent)
    text_draw_list.push_back( DrawableStrokeColor(Color("black")));
    text_draw_list.push_back( DrawableFillColor(Color(0, 0, 0, MaxRGB)));
 // draw the "text to write on the canvas" string on the canvas with the above settings
    my_image.draw( text_draw_list);
 // Note: the red marking point in below image is located at position (100,50)
```

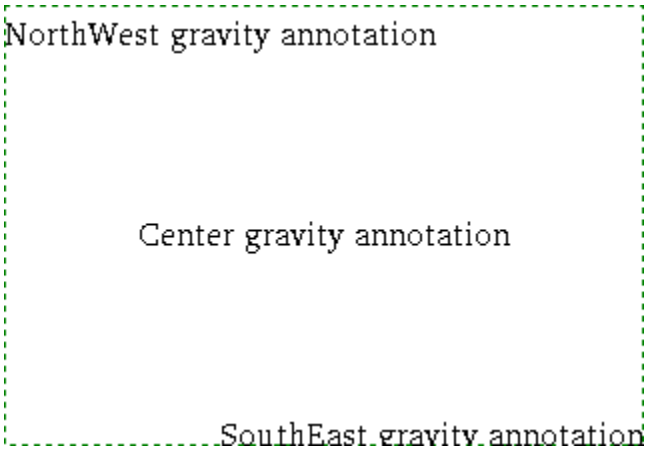


Annotating an image using the Image 'annotate()' method

The 'Image::annotate()' method allows placing text inside an image's canvas by only specifying the position relative to the image center at which the text will be placed, with said position being specified via "gravity" (which is a parameter of type GravityType)

```
 Image::annotate(const string& annotation, GravityType position)

// Example:
 Image my_image( Geometry(320,220), Color("white"));
 // set the text rendering font (the color is determined by the "current" image setting)
 my_image.font("-*-bitstream charter-medium-r-normal-*-*-*-*-*-*-*-iso8859-1");
 // draw text with different gravity position
 my_image.annotate("NorthWest gravity annotation", NorthWestGravity);
 my_image.annotate("SouthEast gravity annotation", SouthEastGravity);
 my_image.annotate("Center gravity annotation", CenterGravity);
```



Note: as it can be seen in the image above, the Magick++ library apparently has some bugs when annotating at the bottom if the image

5 Global image operations

Overlaying images

Magick++ supports image overlaying via the Image object's '*Image::composite()*' method. This method places an 'image_to_overlay' over a given 'support_image':

```
support_image.composite(image_to_overlay, parameters)
```

- The *composition method*
  This parameter is of type CompositeOperator (which is implemented as an enum).
  The following list presents a selection of essential composition methods:
  - The InCompositeOp is the *default value* for the CompositeOperator (i.e. when the CompositeOperator is omitted), and it instructs the following composition algorithm:
    - First, the 'support_image' is completely cleared and its canvas is made transparent
    - Next, the 'image_to_overlay' is placed over the 'support_image' canvas at a specified position. As a result, the 'support_image' canvas will only contain *exactly* the 'image_to_overlay' *surrounded by transparent area* on the 'support_image' canvas
  - The OverCompositeOp value specifies that the resulting composition image contains the original 'support_image' over which the 'image_to_overlay' is *blended* according the individual 'image_to_overlay' and 'support_image' pixels' transparency (i.e. Alpha values).
  - The OutCompositeOp value determines a composition algorithm that simply *cuts out an area* from the 'support_image' and leaves that area completely transparent; the size and shape of the cut-out area are determined by the 'image_to_overlay'
- The *positioning* of the image to be overlayed with respect to the support image.
  This positioning information can be passed to the 'composite()' method in two ways:
  - the Offset with respect to the support image system of coordinates
  - the Gravity with respect to the support image

```
// CompositeOperator below can be: InCompositeOp, OverCompositeOp, OutCompositeOp
Image::composite(const Image& image_to_overlay, int x, int y, CompositeOperator method)
Image::composite(const Image& image_to_overlay, GravityType pos, CompositeOperator method)

// Example
// Use the 'Over' method to place image_to_overlay over support_image at position (10,10)
support_image.composite(image_to_overlay, 10, 10, OverCompositeOp);
```

Extract a "sub-image" from another image

Magick++ supports the extraction of a "sub-image" from another image via two Image methods: 'chop()' and 'crop()'.

```
Image::chop(Geometry(double x, double y));   // see details below
Image::crop(Geometry(double x, double y));   // see details below
```

Both chop() and crop() methods *resize the image canvas that they operate on* by keeping only a certain region of the original, and simply discarding the rest. The region that will be retained is specific to each method.

Both 'chop()' and 'crop()' methods take only one argument which is of type 'Geometry'

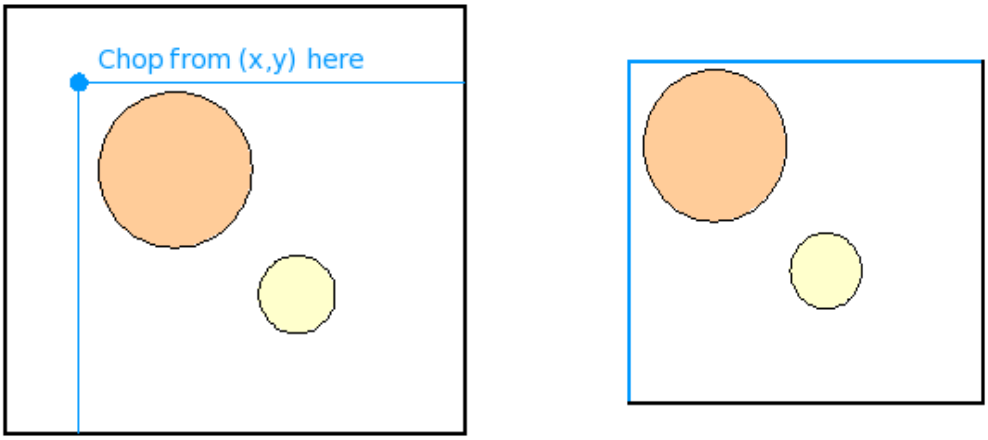
> Note: *because these methods modify the original image object for which they are invoked, they should be used on a copy of the original image in order to preserve the original image*

The chop() method

Let us consider an image 'my_image' with Geometry(original_size_x, original_size_y), and let us assume a call is made to my_image.chop(Geometry(x, y)). In this case, *after the chop() operation, my_image has changed* as follows:

- the *top-left corner* (i.e. the 0,0 point) is the point (x,y) of the original image
- the new size of my_image is (original_size_x-x, original_size_y-y)

Thus, the image size after a 'chop()' operation is reduced to (original_size_x-x, original_size_y-y), and the (0,0) point of the chopped image corresponds to the (x,y) point of the original image:

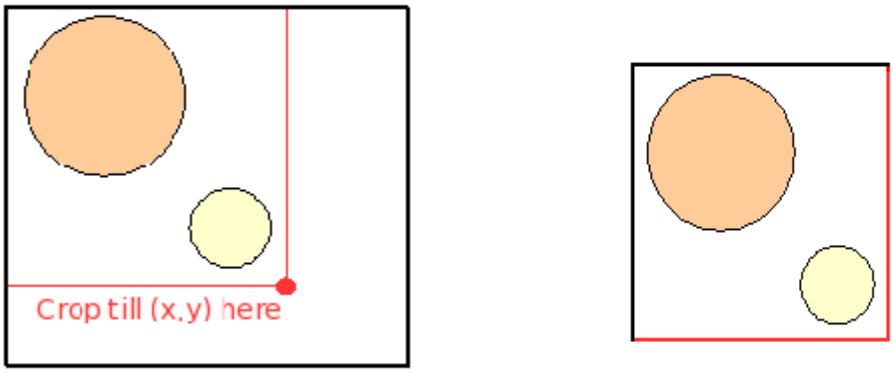


The crop() method

let us consider an image my_image with Geometry(original_size_x, original_size_y), and let us assume a call is made to my_image.crop(Geometry(x, y)). In this case, *after the crop() operation, my_image has changed* as follows:

- the *bottom-right corner* is be the point (x,y) of the original image
- the new size of my_image is (x,y)

Thus, the image size after a 'crop()' operation is reduced to (x,y), and the (0,0) point of the cropped image corresponds to the (0,0) point of the original image:



Extracting a region from within an image

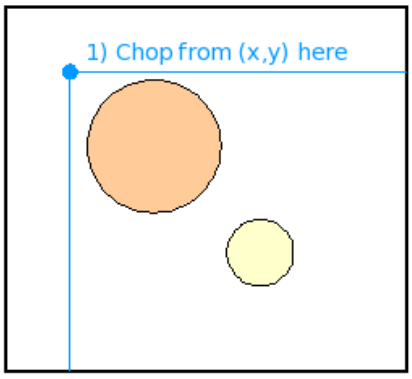
The following example illustrates a way of combining image *copying, chopping, and cropping*, for extracting a "sub-image" of size (200,200) from a source image, starting with the position (100,100) in the source image (i.e. *the resulting image will be a copy of the region ((100,100), (200,200))* from the original image):

```
// let us assume a source_image from 'source_image.gif' file that has 400x400 resolution
   Image source_image("source_image.gif");

// to avoid altering the source_image, make a copy of it and operate on the copy
  Image sub_image(source_image);

// chop the sub_image: after chop(), the sub_image will contain the
// (100, 100, 400, 400) area from source_image (chopped image)
   sub_image.chop(Geometry(100,100));

// crop the sub_image: after crop(), the sub_image will contain the
// (100, 100, 300, 300) area from source_image    (chopped and cropped image)
   sub_image.crop(Geometry(200,200));  // after this point in the code, 'sub_image'
                                       // contains the desired area from 'source_image'
```



copy of original image

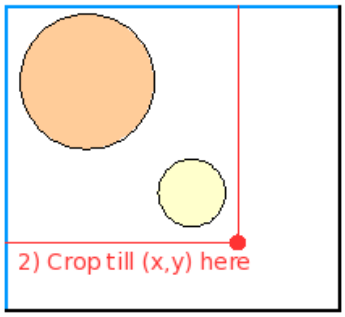


image after *chopping*

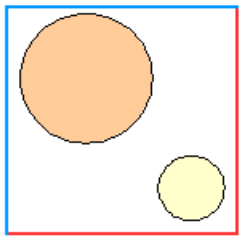


image after *chopping and cropping*

Global image modifications

Resizing an image

Magick++ provides the Image::zoom() method for image resizing. This method resizes the image with a given positive 'zoom_factor' value: if *0<zoom_factor*<1, the image will zoom out (i.e. it will be shrunk); if *zoom_factor*>1, the image will zoom in (i.e. it will be magnified).
Note: After a zoom operation, the image will have the same size_x:size_y ratio as before the operation.

```
   Image::zoom(const Geometry& new_geometry);

// Example:
// start with an image of 600x300 pixels (size_x:size_y ratio is 2:1)
// the image will first be magnified in by a factor of 2, and then shrunk by a factor
// of 2 (thus regaining its original dimensions)
   double size_x = 600, size_y = 300, zoom_in_factor = 2, zoom_out_factor = 0.5;
   Image my_image( Geometry(size_x,size_y), Color("white"));
// zoom in 'my_image' with the 'zoom_in_factor'
// after zoom in, the new 'my_image' size will be : 1200x600
   my_image.zoom( Geometry(size_x*zoom_in_factor, size_y*zoom_in_factor));
// zoom out 'my_image' with the 'zoom_out_factor'
// after zoom out, the new 'my_image' size will be brought back to 600x300
   my_image.zoom( Geometry(size_x*zoom_out_factor, size_y*zoom_out_factor));
```

Rotating an image

The 'Image::rotate()' method rotates an entire image with a given angle. The image canvas *is enlarged* if this is required for containing the rotated image; however, the image canvas *is never shrunk*, even if the rotated image could fit in a smaller canvas than the original.

If the image canvas after a rotation is larger than the original, the areas that do not correspond to the rotated original image are *filled with fully-opaque white in the current Magick++ version.*

```
Image::rotate(double angle);  // the angle is specified in counter-trigonometric degrees

// Example
Image my_image(Geometry(200,200), Color("Cyan"));  // Square image: Cyan background
{code to draw a yellow box}   // image 1
my_image.rotate(45);          // image 2
my_image.rotate(45);          // image 3
```

image1

image2

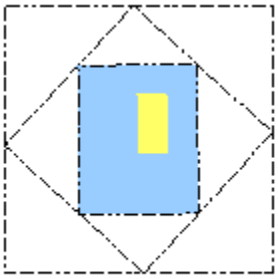


image3

Note:
the *dashed lines surrounding the above images are not part of the images* themselves;
they have been manually added only to show the image boundaries as the image is rotated

Color modifications on an image, or an image area

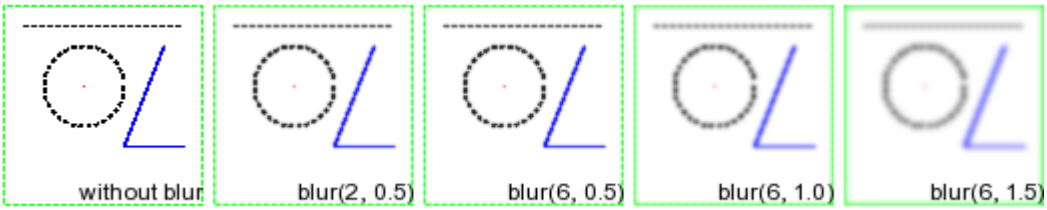
The method for changing the global coloring of an image, or of a portion of the image, is to create a pixel cache for the desired region and then sequentially alter the individual components for each of the pixels ('red', 'green', 'blue', and 'opacity'). At the end of the process the image will have to be update via the pixel buffer's 'Pixel::sync()' method (see the also 'pixel cache' section above).

- Note: these operations can also be performed directly on the image, but they will be less efficient than using a pixel cache if the number of pixels involved is relatively large (the actual efficiency improvement when using a pixel cache heavily depends on the Magick++ implementation)

Blurring an image

To apply a blur effect to an entire image canvas, Magick++ provides the Gaussian distribution-based 'Image::blur()' method. The blur filter acts on each pixel of the canvas, setting its value to a weighted average of its surrounding pixels within a given radius.

```
Image::blur(const double radius, const double sigma); // 'radius' defines the region
                                                      // around the pixel that is averaged
                                                      // 'sigma' represents the way the
                                                      // surrounding pixels are weighted
// the images below present the blur effect when using various values for radius and sigma
// the first image is the “base image” on which the blur filter will be applied
// the following code will generate the effect seen in the other images
my_image.blur(2, 0.5);
```



6 Briefly displaying an image

It is sometimes necessary (especially during an application's development phase) to briefly check the results of the image processing algorithms that are being implemented. For this purpose, the Magick++ library provides the 'Image::display()' method which pops up a window containing the image on the display, and halts program execution until the window is manually closed.

- IMPORTANT: *the image may be altered when it is displayed* via 'Image::display()'; in order to prevent such situations *always use a copy* of the image intended for visualization!

```
Image::display(); //display a pop-up window containing the image; THIS MAY ALTER THE IMAGE

//example: display an image via a temporary copy in order to prevent altering the original
Image temp_image(my_image); temp_image.display(); // display 'my_image' in a pop-up window
```

7 Coding style

The Magick++ documentation has two important recommendations with respect to the coding style to be used when developing an application, and they refer to how the various Magick++ objects should be created and what general coding rules should be followed.

Exceptions

Whenever a Magick++ operation fails, an exception is automatically thrown. There are two important base classes of exceptions in Magick++, and both are derived from the standard C++ exception class: 'Magick::Error' and 'Magick::Warning'.

Generally, an Error is thrown when a Magick++ operation failed in a way that would affect future operations in an unpredictable way, while a Warning generally allows the continuation of processing. Furthermore, each of Error and Warning are themselves base classes for a more detailed exception class tree that can accurately pinpoint the problem that has occurred.

```
// coding style example for using the Magick++ exceptions

try {
   my_image.read(my_image_file); // try to read an image from a file
}
catch (Error& my_error) {
   // because 'Error' is derived from the standard C++ exception, it has a 'what()' method
   std::cout << “a Magick++ error occurred: ” << my_error.what() << std::endl;
}
catch ( ... ) {
    std::cout << “an unhandled error has occurred; exiting application.” << std::endl;
    exit(1);
}
```

- Note: *two important types of errors* that might be correctable by user intervention during the execution of an application are 'ErrorFileOpen' which signals that an attempt to open a file has failed, and 'ErrorCorruptImage' which specifies that an image file is corrupt; also, an important *system error* is 'ErrorResourceLimit' which signals that a system resource has been exhausted (e.g. the system has run out of memory).
  For a detailed list of Magick++ Error and Warning types see the Magick++ documentation.

Variables

It is recommended to use automatic variables when instantiating Magick++ objects and lists of objects (as opposed to creating the objects on the heap via the 'new' operator). This recommendation is intended to increase the robustness of the application by making use of the C++ automatic “stack unrolling” mechanism which properly destroys the objects declared within a code block when execution leaves that code block (including when an exception occurs).

8 Further reading

The ImageMagick project home page is located at ImageMagick.org
The Magick++ library home page is located at ImageMagick.org/Magick++
The Magick++ reference is located at ImageMagick.org/Magick++/Documentation.html