Generic Image Library

Lubomir Bourdev

The Generic Image Library (GIL) is a C++ image library sponsored by Adobe Systems, Inc. and developed by Lubomir Bourdev and Hailin Jin. It is an open-source library, planned for inclusion in Boost 1.35.0. GIL is also a part of the Adobe Source Libraries. It is used in several Adobe projects, including some new features in Photoshop CS4. GIL's web page is http://opensource.adobe.com/gil

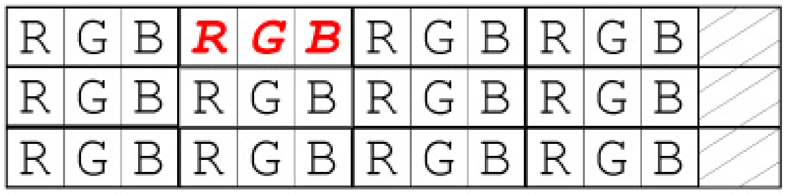
Lubomir Bourdev is a Sr. Computer Scientist at Adobe's Office of Technology. He is the original author of GIL and has developed features in Illustrator, Acrobat, Photoshop Elements and InDesign. He can be contacted at lbourdev@adobe.com

Introduction

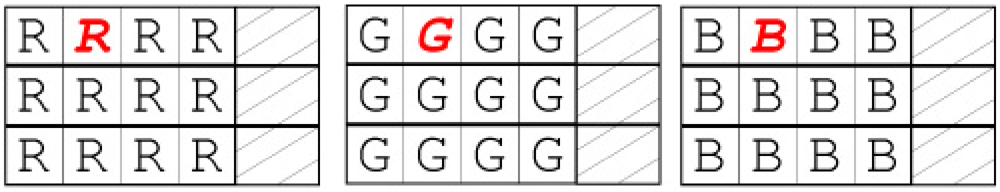
Nearly every visual output from a modern computer is in the form of an image. The user interface elements in your applications are at some point rasterized into an image before being displayed. When you watch a video, you are watching a sequence of images. When you print a document, it is rasterized into an image either inside your computer or in the printer itself.

An *image* is a 2D grid of pixels. A *pixel* defines the color at the corresponding point in the image. The color is specified with the values of *color components* in a given *color space*. For example, the color Red in an RGB color space is specified at 100% of the red channel, and 0% of the green and blue channels.
The contexts in which images are used impose various requirements on the memory footprint of images and the performance of image processing algorithms, which leads to a variability of image representations.

Certain algorithms traverse each pixel and access all of its channels, while other algorithms may prefer to access all the red components of an image, followed by the green, followed by the blue components. Each of these two types of algorithms suggests a different cache-friendly layout of the data: in an *interleaved image* the channels of a pixel are together in memory, whereas in a *planar image* all the channels of a given color are together in a *color plane*. Here is an example of how a 4x3 RGB interleaved image looks like in memory, with the second pixel hi-lighted:



And here is how the same image looks like in planar form:



Speed, precision, and memory footprint are conflicting goals, which give rise to a variety of channel representations. Channels may be represented with integral or floating point numbers with precision that typically varies from 1 bit to 64 or more bits. When memory is essential, the channels of an image may be represented with bit ranges that are often different for each color channel (such as a 16-bit 565 RGB pixel). Some video formats have very elaborate structure in which certain bit range channels are shared across different pixels.

Color spaces are another application-driven property of images. For example, the RGB color space is based on the way monitors model a color, whereas the CMYK color space represents the mixture of four inks used in color printers. Hardware architecture also influences the structure of an image. Certain platforms require alignment for efficient memory access. As a result, the rows of the image may be memory aligned, which results in a potential gap at the end of each row.

While images vary widely in their representation in memory, they share the same conceptual framework – an image is a grid of pixels. There is a common set of algorithms that are useful to apply to any image. Basic algorithms include setting the pixels of an image to a given value, copying the pixels from one image to another, or comparing two images for equality. In addition, there are domain-specific algorithms, such as performing a convolution on an image, scan-converting a vector object into an image and file I/O. Clearly the set of image representations is open, and so is the set of algorithms on images. In many image libraries the representation of an image is often hard-coded into the algorithm itself and, as a result, we often have the algorithm duplicated for each of a fixed set of image types. This results in code that is large, hard to extend, and a nightmare to maintain.

The goal of the Generic Image Library (GIL) is to abstract away image representations and allow for writing the algorithms once, while having them work efficiently on images in any representation. GIL leverages the promise of generic programming in C++ to allow for genericity without sacrificing performance.

Simple Example

Suppose we want to copy one image into another. For simplicity, lets assume that all pixels are consecutive in memory so we can do it in 1D. A non-generic code might look something like this:

```
struct rgb_pixel
{
    unsigned char red, green, blue;
};

void copy_pixels(const rgb_pixel* src, rgb_pixel* dst,
                 int width, int height)
{
    const rgb_pixel* src_end = src + width*height;
    while (src != src_end) {
        *dst++ = *src++;
    }
}
```

Suppose, however, that the destination image is planar, i.e. its red, green and blue channels are distributed in memory. We may have to do something like this:

```
void copy_pixels(const rgb_pixel* src,
    unsigned char* dst_red, unsigned char* dst_green,
    unsigned char* dst_blue, int width, int height)
{
      const rgb_pixel* src_end = src + width*height;
      while (src != src_end) {
           *dst_red++ = src->red;
           *dst_green++ = src->green;
           *dst_blue++ = src->blue;
            ++src;
      }
}
```

You can imagine two additional versions – copying from interleaved to planar and between two planar images. If you think four versions are manageable, imagine if we have to deal with multiple color spaces such as LAB, CMYK, Grayscale, which would require taking a different number of parameters. Imagine if we have to support different ordering of the channels (for example, copying from RGB into a BGR image) We will have to write dozens of versions of copy. Of course, copy is only one algorithm; the real pain comes when we have to create and maintain many versions of many different algorithms.

The GIL Version

While images vary widely in representation, conceptually they are simply a grid of pixels. If we think of an image as a grid of pixels, and the pixel as a container of channels, we can write our algorithms only once. Here is one way to write `copy_pixels` in GIL:

```
template <typename SrcView, typename DstView>
void copy_pixels(const SrcView& src, const DstView& dst)
{
   std::copy(src.begin(), src.end(), dst.begin());
}
```

If you are familiar with the STL, GIL will be easy to pick up. GIL makes a distinction between an image and a view. An image is a container of pixels. It owns the memory for the pixels and has a deep copy constructor. It is similar to `std::vector`. An image view is a shallow reference to a 2D grid of pixels. Views are lightweight and fast to construct and their copy constructors don't copy the pixels.[1] The STL equivalent of a GIL view is a range, which is often denoted with a pair of iterators. Most GIL algorithms operate on image views, just like most STL algorithms operate on ranges.

`copy_pixels` requires that the two views have the same dimensions, and that the destination be mutable. It also requires that the two views be compatible (which means they must have the same color space and channel type). Some of these requirements can be specified at compile time, and others must be done at run time:

```
template <typename SrcView, typename DstView>
void copy_pixels(const SrcView& src, const DstView& dst)
{
    boost::function_requires<ImageViewConcept<SrcView> >();
    boost::function_requires<MutableImageViewConcept<DstView> >();
    boost::function_requires<ViewsCompatibleConcept<SrcView,DstView> >();
    assert(src.dimensions() == dst.dimensions());

    std::copy(src.begin(), src.end(), dst.begin());
}
```

GIL has concept specifications for every construct, from the channel to the image. A *concept* is a set of syntactic requirements and complexity guarantees that a type (or a set of types) must have. For example, the first call to `function_requires` ensures that `SrcView` is a valid image view. If we attempt to instantiate `copy_pixels` with something else, we will get an error at that line, as opposed to somewhere deep in the algorithm.

A type satisfying the concept requirements is said to model the concept. For example, the class `gil::image_view` models `ImageViewConcept`. GIL algorithms take concepts rather than concrete models. This allows you to use the same algorithms with your own models of channels, pixels, pixel

[1] Views also don't propagate their constness to the underlying pixels, which explains why the destination is taken as a const reference. Mutability is part of the view type.

iterators, image views or images. Here is how we can use `copy_pixels` to copy from an interleaved to a planar 16-bit signed CMYK image.[2]

```
void copy_cmyk_planar2interleaved(const short* src_cmyk,
          ptrdiff_t src_row_bytes,
          short* dst_c, short* dst_m, short* dst_y, short* dst_k,
          ptrdiff_t dst_row_bytes,
          size_t width, size_t height)
{
    cmyk16sc_view_t src_view = interleaved_view(width,height,
                            (const cmyk16s_pixel_t*)src,src_row_bytes);
    cmyk16s_planar_view_t dst_view = planar_lab_view(width,height,
                            dst_c, dst_m, dst_y, dst_k, dst_row_bytes);
    copy_pixels(src_view, dst_view);
}
```

The first step is to take the raw data and construct a GIL view from it. While the two views may often have different types, they both model `ImageViewConcept`, and thus provide a common interface that the algorithms use. Since views are lightweight, constructing them from raw data has no measurable performance implications. A note on the conventions: "`cmyk`" stands for CMYK color space, "`16`" stands for 16-bit, "`s`" stands for signed integral channels, "`c`" stands for constant (immutable) type.

As you can imagine, the same algorithm can be used to copy between images of any color space, channel type, ordering of channels and planar/interleaved organization. GIL will not allow us, however, to copy an RGB source into an LAB destination or an 8-bit into a 16-bit channel. Most GIL algorithms require that the views be *compatible*, which means they must have the same color space and channel types. This makes copy_pixels a lossless operation.[3]

To see how GIL is used, let's replace `std::copy` with an explicit loop:

```
template <typename SrcView, typename DstView>
void copy_pixels(const SrcView& src, const DstView& dst)
{
   typename SrcView::iterator src_it = src.begin();
   typename DstView::iterator dst_it = dst.begin();
   while (src_it != src.end()) {
        *dst_it++ = *src_it++;
   }
}
```

Image views provide a random access iterator that iterates through all the pixels of a view in a left-to-right inside top-to-bottom order. Its `operator++` advances to the next pixel. It properly skips any potential gap at the end of a row. Its `operator*` returns a reference to a pixel. In the case of an interleaved view, the pixel reference is the built-in C++ reference (`pixel&`). Planar views, however, have channels that are disjoint in memory. Dereferencing a planar iterator returns a planar pixel *reference proxy* – a class that behaves like a built-in reference[4]. Regular pixels, as well as pixel reference proxies, model `PixelConcept`. Finally, models of `PixelConcept` have an `operator=` that takes another pixel concept (not necessarily of the same type, but compatible) and copies its channels. Many pixel-level algorithms, such as assignment, copy-construction and equality comparison, perform the same operation for each channel. GIL uses compile-time recursion to perform the operation for each channel efficiently, thus avoiding the overhead of a loop. Of course, all of this is hidden from the user.

Here is another way to write `copy_pixels` using `std::vector`-like interface:

[2] The rows of some images may be aligned, which may result in a gap at the end of each row. Thus we take the byte distance of the rows as a separate parameter.
[3] You can use `copy_and_convert_pixels` to copy between incompatible views.
[4] To the degree this is possible in C++.

```
template <typename SrcView, typename DstView>
void copy_pixels(const SrcView& src, const DstView& dst)
{
    for (size_t i=0; i<src.size(); ++i)
       dst[i] = src[i];
}
```

While one-dimensional algorithms are simpler to write, they are somewhat suboptimal, because advancing the iterator requires an extra check: Are we at the end of a row, so we can skip any padding and go to the beginning of the next row? It is more efficient to use two nested loops instead:

```
template <typename SrcView, typename DstView>
void copy_pixels(const SrcView& src, const DstView& dst)
{
    for (size_t y=0; y<src.height(); ++y) {
        typename SrcView::x_iterator src_xit = src.row_begin(y);
        typename DstView::x_iterator dst_xit = dst.row_begin(y);
        for (size_t x=0; x<src.width(); ++x)
            dst_xit[x] = src_xit[x];
   }
}
```

As the example demonstrates, views also provide `x_iterator`, an iterator navigating the pixels along a row of the image view. Since it doesn't need to worry about the end of a row, `x_iterator` is simpler and faster. For example, for an interleaved image view, `x_iterator` is just a pixel pointer. In a planar RGB image view `x_iterator` is a bundle of three pointers, so incrementing it amounts to three pointer increments.

GIL also has a `y_iterator`, an iterator along the column of an image. For an interleaved image, it is just a pointer with a step (the distance between two rows). Incrementing the iterator amounts to advancing the pointer by the given step. Here is how to write `copy_pixels` that copies by columns instead of by rows:

```
template <typename SrcView, typename DstView>
void copy_pixels(const SrcView& src, const DstView& dst)
{
    for (size_t x=0; x<src.width(); ++x) {
        typename SrcView::y_iterator src_yit = src.col_begin(x);
        typename DstView::y_iterator dst_yit = dst.col_begin(x);
        for (size_t y=0; y<src.height(); ++y)
            dst_yit[y] = src_yit[y];
   }
}
```

As you can see, all we have to do is switch `x` with `y`, `row` with `col` and `width` with `height`. Image views also allow for accessing their pixels given 2D coordinates. Here is another way to write `copy_pixels`:

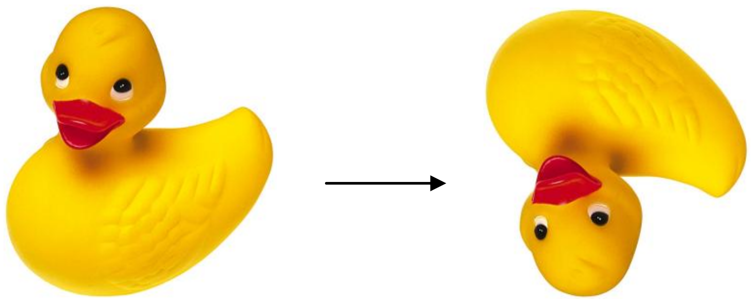
```
template <typename SrcView, typename DstView>
void copy_pixels(const SrcView& src, const DstView& dst)
{
    for (size_t y=0; y<src.height(); ++y)
        for (size_t x=0; x<src.width(); ++x)
             dst(x,y) = src(x,y);
}
```

While this is convenient, it is suboptimal, because the pixel location will be recomputed every time from its coordinates, which involve multiplication and addition. GIL also has a 2D equivalent of an iterator, called a locator. It can be advanced both horizontally and vertically and can return the pixel at a 2D offset from the current location.

Image Views

Suppose we want to create an upside down copy of the image. One way to do that is to write a new algorithm, something like `copy_pixels_upside_down`. Another way is to use GIL's image view transformation mechanism:

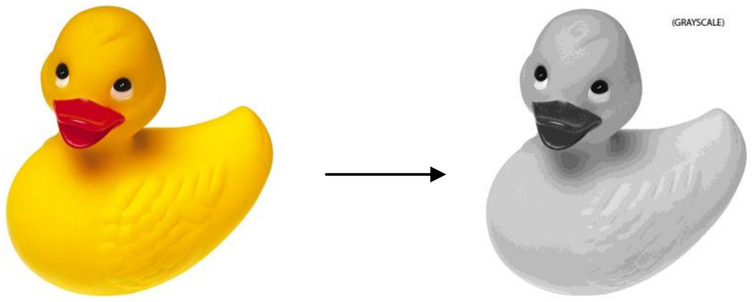
```
copy_pixels(rotated180_view(src), dst);
```



`rotated180_view` is an example of a GIL image view transformation. An image view transformation takes a view and returns another view (possibly of a different type) by changing some of the properties of the source. In this case, it returns a view whose first row and column are the last row and column of the source, and whose step in x and y is negated. Similarly, GIL provides `flipped_left_write_view`, `rotated90cw_view`, `rotated90ccw_view`, `rotated180_view`, and `transposed_view`. View transformations are shallow and lightweight. They don't copy the pixels.

View transformations can also perform color conversion or select color planes. For example, here is how to copy an image and at the same time convert it into 8-bit grayscale:

```
// dst must have 8-bit grayscale pixels
copy_pixels(color_converted_view<gray8_pixel_t>(src), dst);
```



`color_converted_view` takes any source image view and returns an immutable view that is just like the source, except has a different color space and channel type. Again, constructing `color_converted_view` is very fast; the actual color conversion is performed upon pixel access. Here is how to copy the (single channel) source into the first channel of the destination:

```
// src must be grayscale (i.e. single channel view)
copy_pixels(src, nth_channel_view(dst, 0));
```

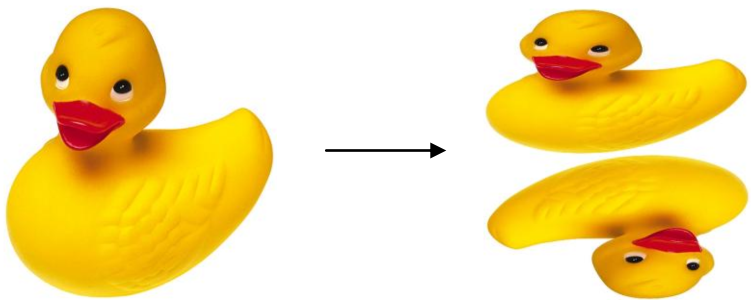
View transformations can be nested. Here is how to copy the source, scaled down, into the top half of the destination, and again, upside down, into the lower half of the destination[5]:

[5] For simplicity we assume the height of the source is even

```
copy_pixels(subsampled_view(src,1,2),
    subimage_view(dst, 0,0,              dst.width(), dst.height()/2));

copy_pixels(rotated180_view(subsampled_view(src,1,2)),
    subimage_view(dst, 0,dst.height()/2, dst.width(), dst.height()/2));
```

`subsampled_view` is a view that skips pixels horizontally and vertically. In this case it skips every other row of the source. `subimage_view` returns a view of a rectangular area within the source. In this case we use it to return a view of the top half and bottom half of the destination:



Virtual Image Views

GIL's `ImageViewConcept` does not require that the image view occupy physical memory. Here is how we can take an arbitrary function and construct a virtual image view from it.

First, we create a function object that will be given the pixel coordinates and will return the pixel value (in this case, 8-bit grayscale)[6]:

```
typedef point2<ptrdiff_t> point_t;

struct virtual_fn : public deref_base<virtual_fn, gray8_pixel_t,
       gray8_pixel_t, gray8_pixel_t, point_t, gray8_pixel_t, false>
{
    gray8_pixel_t operator()(const point_t& p) const {
         return gray8_pixel_t((sin(p.x/10.0f)+cos(p.y/10.0f)+2)*64);
    }
};
```

Then we use the function object to construct a virtual image view, whose size is the same as our source:

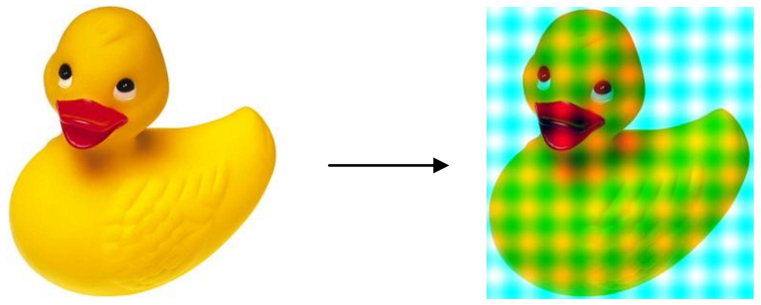
```
typedef virtual_2d_locator<virtual_fn,false> locator_t;
image_view<locator_t> virt_view(duck_view.dimensions(),locator_t());
```

We can now use the virtual view just like any other. For example, here is how to copy the function into the first (the red) channel of the sample image:

```
rgb8_view_t duck_view = …;
copy_pixels(virt_view, nth_channel_view(duck_view,0));
```

Here is the result of this operation:

[6] Please refer to the GIL documentation for details on the parameters



Dynamic Images

Some of the GIL image properties, such as its color space, layout and channel type, are part of the image type. We must know these properties at compile time if we want to achieve performance comparable to hand-writing the code for a specific image type. Unfortunately, sometimes these image properties are not available at compile time. For example, it would be difficult to create a function that reads an image from file and returns it in its native form. GIL's `dynamic_image` extension allows us to specify the image parameters at run time without compromising performance. Here is an example of using GIL's `dynamic_image` and I/O extensions to read an image from file in its native format:

```
#include <boost/gil/extension/dynamic_image/dynamic_image_all.hpp>
#include <boost/gil/extension/io/jpeg_dynamic_io.hpp>
#include <boost/mpl/vector.hpp>
using namespace boost;

typedef mpl::vector<rgb8_image_t,  rgb16_image_t,
                    gray8_image_t, gray16_image_t> img_types;
typedef gil::any_image<img_types> my_any_image_t;

my_any_image_t dyn_img;
gil::jpeg_read_image("test.jpg", dyn_img);
my_any_image_t::view_t dyn_view = gil::view(dyn_img);
```

While the image type is specified at run time, we still have to define the set of possible types the image could have. In this case, we are using Boost MPL to specify that the image could be a Grayscale or an RGB, and its depth could be 8 or 16 bit.

We then construct an instance of the image and use the GIL I/O extension to read it from a file. If the image on file is not compatible to any of the four possible types, the I/O code throws an exception.

GIL uses the Variant idiom to provide runtime polymorphism. The class `any_image` internally contains a buffer large enough to store any of the four images and an index specifying the current image. Invoking an operation on the image requires a switch statement followed by casting the image to the given type and invoking the operation. GIL tries to hide all of this machinery and provides an interface consistent with an image. For example, we can get an image view from the runtime image, which is also a runtime view, the same way we do it for compile time images. We can then use the view to call most of the GIL algorithms and image view functions. Here is how to write the image to file in its native format, but upside down. The code is identical for compile time or run time images:

```
jpeg_write_view(“test_out.jpg”, flipped_up_down_view(dyn_view));
```

Most GIL algorithms can also take runtime views. For example, `copy_pixels` allows both its source and destination to be compile time or run time views.[7]
Dynamic images and views result in efficient code, because they instantiate the algorithm for every possible combination of types and invoke the appropriate one at run time. Thus, the polymorphic check is performed only once per call to the algorithm. Since image algorithms typically run in time at least linear on the number of pixels, such a constant overhead results in no measurable effect on performance. However, dynamic images have the potential to increase code bloat and compile time. Note also that dynamic image views do not model the full requirements of `ImageViewConcept`. In particular, they don't allow access to the pixels. While polymorphic pixels are possible to implement, they would be too slow to use. You will need to provide overloads for your GIL algorithms if you want them to use dynamic views, as described in the documentation.

Performance

One of the most important goals of GIL's design is performance. GIL's algorithms often have performance overloads to handle specific cases. In our example we wrote `copy_pixels` to do an explicit loop. This loop is a fallback scenario in the real `copy_pixels`. When invoked for two contiguous interleaved views, `copy_pixels` resolves to `memmove`. If both views are planar, it resolves to `memmove` for each color plane. If there are gaps at the end of rows, `memmove` is called for each row. The generic framework allows clients to supply their own overloads for algorithms when using their own views or pixels. Performance is discussed more in the video tutorial on GIL's web page where the assembly of GIL and non-GIL code is compared.

Genericity

GIL maintains a clean separation between concepts, models and algorithms. There are concepts describing anything from a channel to N-dimensional image. This allows developer A to create a new model of, say, a 128-bit channel, developer B to create a new image view representing, for example, a space/time slice in a video clip, and developer C to provide support for, say, YUV color space. We can then use existing or new algorithms, like `copy_pixels`, to operate on 128-bit YUV space/time slices of video. Furthermore, developers A, B and C need not even be aware of each other's work; all they have to make sure is that their models satisfy the corresponding concepts.

Ease of use

GIL is very easy to integrate in your existing code. It has no external code dependencies, other than Boost, of course. It consists of header files only, so it doesn't require you to link against any library[8]. It doesn't require you to build Boost either. All you have to do is include the GIL headers and you are good to go. GIL is also portable - I have a face detector written with GIL running on my PDA phone.

GIL's ease of integration means that you don't have to sacrifice your favorite image library by switching to GIL, nor you have to throw away the image processing algorithms you have already invested your time developing. You can keep your existing code and use GIL snippets only where appropriate. In fact, one of the most common uses of GIL is as a glue between other often incompatible image libraries.

[7] Recall that `copy_pixels` is defined only between compatible image views. When dynamic views are involved, we can no longer determine at compile time whether the two views are compatible. When called on incompatible views, `copy_pixels` will throw an exception.
[8] If you require I/O support you will need to link against libgpeg, libtiff or libpng. However, there is a third-party BMP extension that imposes no external dependencies.s

Future Work

The core GIL library provides a framework for representing any image format, along with a set of models implementing many common image formats and a set of fundamental STL-like algorithms. Every context in which GIL is used requires domain-specific concepts, models and algorithms. For example, image processing and computer vision applications require algorithms like convolution and edge detection. Vector graphics applications require scan conversion and type rendering. Video processing requires its own concepts and algorithms. GIL has an extension mechanism to support such domain-specific code. We already provide two standard extensions to GIL – the i/o extension supports reading and writing images to file (currently JPEG, TIFF and PNG). The dynamic image extension allows for images whose type is specified at run time. We also have an experimental image-processing extension available as a separate download on GIL's web site. It provides some basic image processing algorithms such as resampling and convolution.

GIL is an open source library. We strongly encourage contributions. There are already about half a dozen extensions in progress contributed by the open source community, covering topics such as OpenCV and Freetype integration, BMP and PNM I/O, drawing antialiased lines, etc. If you would like to contribute to GIL please let us know!

Conclusion

This article provides a high level overview of the Generic Image Library, using one of its algorithms, `copy_pixels`, as a running example. We have demonstrated GIL's ability to abstract the image representation from algorithms operating on images, which allows for the same algorithm to be invoked on images of any color space, channel depth and layout, including synthetic images represented by a function or images whose type is specified at run time. Using generic programming, we are able to achieve such genericity without the need to sacrifice performance.

GIL's web page http://opensource.adobe.com/gil provides a wealth of information on GIL, including the source code, documentation, video tutorial, discussion forum and links to third party extensions. We are hoping you will find GIL useful in your work, and we are looking forward to having you join us in adding your contribution to the GIL initiative.