The CImg Library

David Tschumperlé

To cite this version:

David Tschumperlé. The CImg Library. IPOL 2012 Meeting on Image Processing Libraries, Jun 2012, Cachan, France. 4 pp. hal-00927458

HAL Id: hal-00927458
https://hal.science/hal-00927458v1

Submitted on 13 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

The CImg Library

C++ Template Image Processing Toolkit

http://cimg.sourceforge.net

David Tschumperlé

GREYC Laboratory (CNRS UMR 6072), Image Team, 6 Bd Maréchal Juin, 14050 Caen/France

ABSTRACT

The CImg Library is a minimal, easy-to-use, and capable C++ image processing library aiming to help developers implementing new image processing algorithms, from scratch. The library is versatile enough to deal with a large number of image types (from 1D-signals to temporal sequences of 3D-hyperspectral volumes), with any type of pixel values. At the same time, it is extremely straightforward to learn and to use. It proposes indeed a minimal set of four classes, all defined in a single C++ header file `CImg.h` *that has very few (and adjustable) dependencies to third-party libraries. This makes CImg a small and handy image processing library, portable everywhere (supports multiple OS, CPU-architectures and compilers) and comfortable to maintain. Moreover, we introduce G'MIC, a simple script-language wrapping the CImg functionalities, so that all features of the library become available for a broader audience (non C++-programmers) who can use it either from the shell or from an interactive GUI.*

Index Terms— Image Processing, C++ Library, Template-based programming, Genericity, Straightforwardness, Script language.

1. CONTEXT AND MOTIVATIONS

The image processing world is full of very different people (computer scientists, mathematicians, physicians, biologists, ...), with diverse scientific backgrounds, working on a wide range of various image-related problems. This diversity, in terms of people, programming knowledges and data types is something one must take care of, when designing computer tools to assist those people in their goals : They don't all work on 2d gray-valued images only, and most of them do not have 15 years of experience in programming languages ! As a consequence, designing something as basic and important as an image processing library, should follow some minimal rules ensuring the *simplicity*, the *genericity*, the *usefulness*, the *extensibility*, the *portability* and the *freedom* of use of the library. This is what we have tried to achieve with CImg, a C++ library we started implementing in late 1999, designed primarily for researchers and students in image processing and computer vision. CImg is hosted on Sourceforge since 2003, at `http://cimg.sourceforge.net` and attracts each day a lot of people around the world (about 1200 visits and 100 downloads/day). In the followings, we detail the design choices we made to comply with the different rules mentioned above.

2. FEATURES

Simplicity + Genericity

The CImg Library is a *small library*, distributed as a .zip package ($\approx$ 12Mo) containing the library code ($\approx$ 40k loc), examples of use, the reference documentation and resource files. The library itself is composed of a *single C++ header file* `CImg.h` which must be included in the user's code to be functional. All CImg classes and methods are encompassed in the namespace `cimg_library` :

```
#include "CImg.h"              // Just do that...
using namespace cimg_library; // Ready to go !
```

It is obvious then that the CImg library code is compiled *at the same time as the user's code*. As we will see later, this has a lot of advantages in terms of flexibility.
The header file `CImg.h` defines only four different classes, only two of them having *one* template parameter :

1. **`CImg<T>` :** Represents an image, having 4 dimensions (width, height, depth and spectrum), each pixel value being of type `T`. The pixel data are stored in a simple linear buffer `T *data;` of size `width*height*depth*spectrum`.
2. **`CImgList<T>` :** Represents a list of images `CImg<T>`. It is useful to manage a sequence or a collection of images (that may have different sizes).
3. **`CImgDisplay` :** Represents a display window, where images or images lists can be displayed. Multiple windows can be opened, and user interactions are managed through the class methods.
4. **`CImgException` :** Used by the CImg library to throw exceptions, when errors are encountered in the library methods (I/O errors, bad arguments, ...).

This minimal set of classes already covers most of the image types we can encounter in real world application (from 1D scalar signals to collections of 3D hyperspectral images, with any type `T` of pixel values). The only template parameter `T` makes the understanding of the library affordable, *even for non C++ experts*.

Usefulness

CImg is a *general image processing library* that contains most of the *usual* image processing operators we would like to see in such a library. It is *not specialized* in a particular sub-field of image processing (e.g. mathematical morphology, variational calculus, spatial/spectral transforms, ...), but contains a lot of different functions to help *writing complex algorithms with very few C++ code* :

- **Data inputs/outputs :** CImg supports a large number of image file formats (e.g. JPEG, 16bits PNG, float-valued multipage TIFF, 3D Object File Format, PINK files, ...).
- **Usual IP operators :** Classical image processing operators are defined : filtering, mathematical morphology, histograms, color base conversions, interpolations, geometric transformations, non-linear blur/sharpening, displacement field estimation, FFT, and so on...

- **Image drawing :** Many methods allows to draw things in images : lines, polygons, ellipses, texts, vector fields, function graphs, 3d objects (*Fig.**1a***).
- **3d object viewer :** CImg owns a 3d object viewer (kind of mini-openGL), as well as many functions to generate 3d vector objects from dense image data (3d isolines, 3d isosurfaces, 3d elevations, 3d streamlines, ...). It does not rely on an external library for the 3d rendering. It supports light sources, texture mapping and transparent objects (*Fig.**1b***).
- **Arithmetic operators :** Most usual mathematical operations between images are defined (e.g. `operator+()`, `sqrt()`, `cos()`, `atan()`, `operator>>()`,...).
- **Expression evaluator :** CImg owns a numerical evaluator of mathematical expressions, allowing to quickly generate synthetic images from mathematical formula (whose formula can be chosen during the code execution) (*Fig.**1c*** shows an example of a synthetic image, generated from the evaluation of "`X=x-w/2;Y=y-h/2;D=sqrt(X^2+Y^2);if(D+u*20<80,abs(255*cos(D/(5+c))),10*(y%(20+c)))`").

Most of the image processing algorithms inside the library are defined as *methods* of the CImg classes (mostly for `CImg<T>` and `CImgList<T>`). In CImg, we decided *not to separate the classes from the algorithms*, as it is sometimes done in other template-based libraries (e.g. the STL). The separation between classes and algorithms is a nice concept when the data structures we want to work with are quite basic (e.g. 1-dimensional as vectors, lists, sets,...), as it makes sense that one generic algorithm can be applied to such many different structures (e.g. values sorting, re-ordering, searching for a value location). In the case of image processing, most of the algorithms we are interested in stay bounded to multi-dimensional image structures (who really wants to compute the FFT of a STL's `std::multimap` ?). Designing these algorithms in a too much generic way would be still possible, but would lead to a level of complexity that becomes discouraging for most average C++-programmers : it just means they will have to deal with *a lot of template parameters* as well as *lots of weird iterators* ! On the contrary, making the algorithms and the classes dependent as CImg does, allows to :

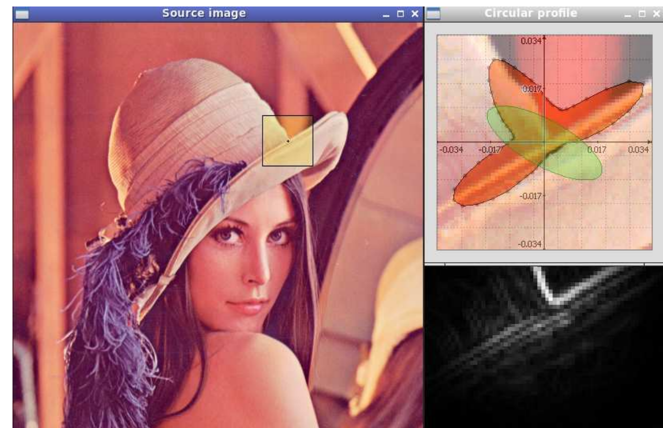
- Write algorithms that are optimized specifically for the CImg structures, e.g. by knowing in advance the pixel values ordering in the image data buffers, without needs for very specific iterators.
- Make the library more simple to use and to learn, by limiting the amount of genericity only to the classes (not to the algorithms).
- Make the library classes depending on a minimal set of required template types (only one !).

Beside this simplicity, this approach also allows CImg algorithms to be pipelined in a way that writing image processing algorithms can be done usually in very few lines. This pipeline capability is illustrated with the following (extreme) example :

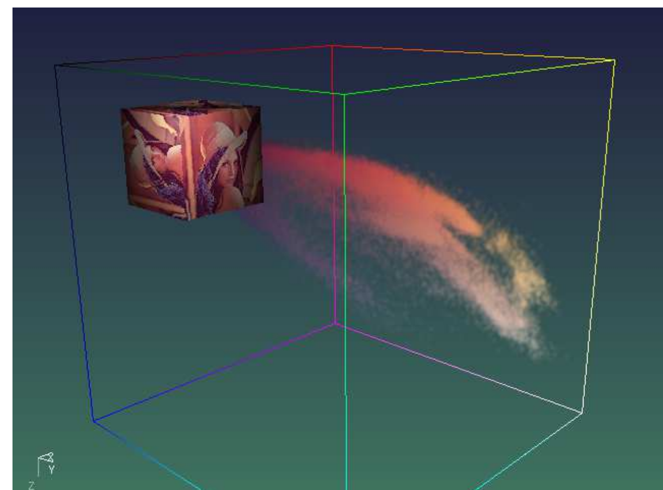
```
#include "CImg.h"
using namespace cimg_library;
int main() {
  CImg<> img("lena.bmp"); // Load color image.

  // Do some weird operations.
  img.RGBtoYCBCr().channel(0).quantize(10,false).
      map(CImg<>(3,1,1,3).rand(0,255).
          resize(10,1,1,3,3));
}
```

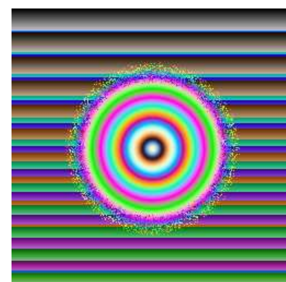
The C++ code above loads an image, computes its luminance channel, quantizes it in 10 levels, then maps a colormap on it, containing 3 random colors interpolated along 10 levels (see *Fig.**1d***).



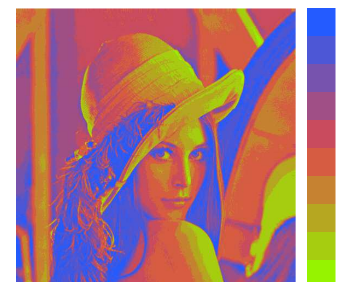
a) Example of drawing capabilities of CImg.



b) Example of 3d rendering with CImg.



c) Using the expression evaluator to generate a synthetic 2d color image (1 loc).



d) Result of the pipeline example.

Fig. 1. **Usefulness :** Illustrating some CImg functionnalities.

Extensibility

As the entire code of the CImg library is contained inside a single header file, the library code is not pre-compiled but compiled *on-the-fly*, i.e. when the library user compiles its own code. This has a lot of advantages in terms of configuration flexibility :

• The user decides what should be the configuration of the library environment, e.g. the dependencies of CImg to third-party libraries, at the compile time, *independently for each of its project*. CImg defines indeed a lot of configuration macros that tells about which third-party libraries have to be linked in the final code : `cimg_use_png, cimg_use_jpeg, cimg_use_tiff, cimg_use_lapack, cimg_use_fftw3, ....`
Depending on these configuration flags, extra functionalities are enabled in the library, as for instance the native support of some image file formats (PNG, JPEG, TIFF, ...) in CImg load/save methods.

• CImg defines a *plug-in mechanism* allowing users to *add their own methods* to the CImg classes, *without having to modify the library source file*. This is how it works in practice :

```
#define cimg_plugin "foo.h"
#include "CImg.h"
using namespace cimg_library;
int main() {
  CImg<> img("lena.bmp");
  // Call to a new custom method
  // of CImg<T> defined in 'foo.h' :
  img.my_method();
}
```

If `my_method` is intended to compute the gradient norm of a `CImg<T>`, we can write the plug-in file `foo.h` simply as :

```
CImg<T>& my_method() {
  const CImgList<T> g = get_gradient("xyz");
  (g[0].sqr() + g[1].sqr() + g[2].sqr()).
    sqrt().move_to(*this);
  return *this;
}
```

Several plug-ins are already available in the CImg package (NLmeans, Skeleton, VRML reader, CImg to Matlab conversion, ...). Note that this kind of plug-in mechanism is not feasible when using a classical library whose code is pre-compiled.

Portability + **Freedom**

The CImg library code is small, does not depend on many external libraries (can be compiled only with dependencies to `libc` and `libm`), and is easy to maintain. As a consequence, it has been possible to make it very portable and robust to different compilers, OS, and CPU architectures. It is known to run flawlessly on all Unix flavors (incl. Android), Windows and Mac OS. The library package is distributed under the permissive open-source **CeCILL-C** license (LGPL-like). The structures defined in CImg are insanely simple to manage, and the integration and communication of CImg with other libraries is facilitated.

3. A SCRIPT LANGUAGE FOR IMAGE PROCESSING

G'MIC stands for *GREYC's Magic Image Converter*. This project is hosted on Sourceforge since 2008, at `http://gmic.sourceforge.net`. It aims to :

• Define a lightweight but powerful script language (the G'MIC language) dedicated to the design of image processing pipelines.

• Provide an interpreter of this language, distributed as a C++ library embeddable in third-party applications (*libgmic*).

• Propose 3 usable binary tools embedding this interpreter :

1. The command-line executable `gmic` to use the G'MIC framework from a shell. In this setting, G'MIC may be seen as a direct (and friendly) competitor of the *ImageMagick* or *GraphicsMagick* software suites.
2. The interactive and extensible plug-in `gmic_gimp` to bring G'MIC capabilities to the image retouching software GIMP.
3. `ZArt`, a real-time Qt-based interface for webcam images manipulation with G'MIC.

Due to its openness, this project attracts a lot of people everyday ($\approx$ 450 downloads/day, more than 600.000 since Aug. 2008). The need for G'MIC has come from several observations :

1. CImg requires (basic) C++ knowledges to be used properly. Still, many people in the image processing field do not know C++ enough, but could be interested by some of the CImg functionalities anyway.
2. When we get new image data, we often want to perform the same basic operations on them, for instance visualization, gradient computation, noise reduction, frequency analysis, ... Sometimes, the operations we need to apply are very specific to the images we just got (e.g. masking the Fourier transform to eliminate some frequency noise).
3. It is certainly not optimal to be forced to create new C++ code specifically for these minor tasks. This requires code edition, compilation time, and most often the code we have just wrote will be used only once in our lifetime !

From these observations, we created G'MIC as a script language interfacing all the CImg methods, to be usable *from the shell*. No compilation steps are required anymore for manipulating generic images. From a technical point of view, G'MIC is based on these simple properties :

- G'MIC manages one list of images in memory (one instance of a `CImgList<T>`). The pixel type `T` can be chosen to be {`bool|char|uchar|short|ushort|int|uint|float|double` }.
- Each G'MIC instruction runs an image processing algorithm (calling then one CImg method) either on one or several images of the list, or control the program execution : `-blur, -mirror, -rgb2hsv, -isosurface3d, -if, -endif, -repeat, -done, -return, ....`
- User-defined scripts can be saved as G'MIC script files, and be recognized by the G'MIC interpreter.
- One G'MIC script can be called from the command line, or from any external project (through the *libgmic* library, embedding the G'MIC interpreter).

G'MIC is a very pleasant tool to manipulate generic images from the command line. Here, we show some examples of use.

• **Add a synthetic degradation to an image** (*Fig.2*) :

```
gmic lena.bmp -blur 3 -sharpen 1000 -noise 30
  -+ "'cos(x/3)*30'"
```

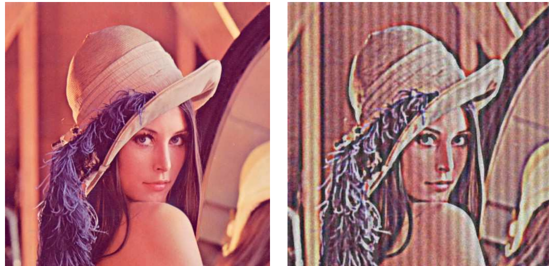


Fig. 2. Create a synthetic image degradation with G'MIC.

• **Extract 3d structures from a dense volumetric image** (*Fig.3*) :

```
gmic reference.inr --flood 23,53,30,50,1,1000
 -flood[-2] 0,0,0,30,1,1000 -blur 1
 -isosurface3d 900 -opacity3d[-2] 0.2
 -color3d[-1] 255,128,0 -+3d
```

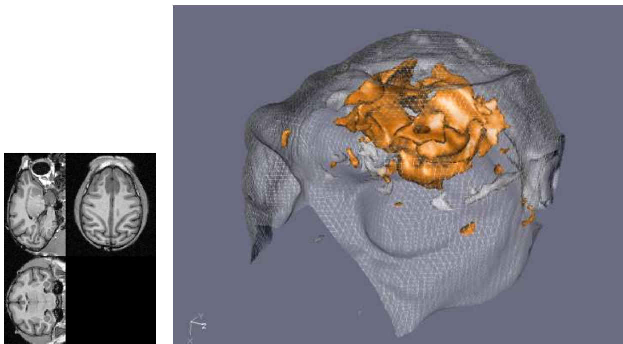


Fig. 3. Extract 3d structures from a volumetric image with G'MIC.

• **Display histograms of an image and its gamma correction** (*Fig.4*) :

```
gmic milla.bmp --f '255*(i/255)^1.7'
  -histogram 128,0,255 -append c -plot
```

is the G'MIC equivalent code to :

```
#include "CImg.h"
using namespace cimg_library;
int main(int argc,char **argv) {
const CImg<>
  img("milla.bmp"),
  hist = img.get_histogram(128,0,255),
  img2 = img.get_fill("255*((i/255)^1.7)",true),
  hist2 = img2.get_histogram(128,0,255);
  (hist,hist2).get_append('c').
    display_graph("Histograms");
}
```

• **Using the G'MIC capabilities from GIMP** (*Fig.5*).

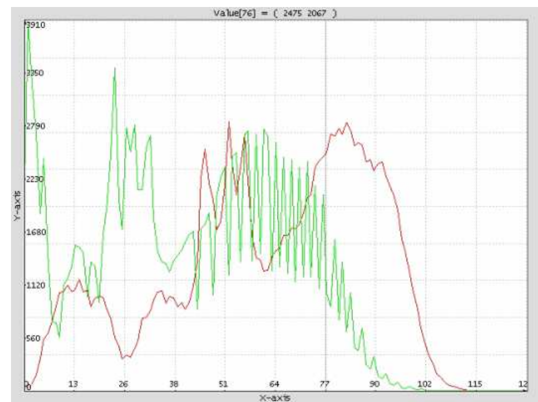


Fig. 4. Display histogram of an image and its gamma correction.

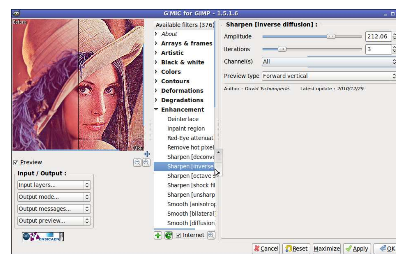


Fig. 5. Example of G'MIC running inside the plug-in for GIMP.

4. CONCLUSION

We have defined complete and generic open-source frameworks for image processing, covering different scales of use : A low-level library for C++ programmers allowing to do algorithm prototyping (`CImg`), a script-language for command line users allowing to easily create image processing pipelines (`gmic`), and graphical interfaces (through the plug-in `gmic_gimp` and `ZArt`) to provide image processing tools for a more general audience. This has been our way to try to reach as many people as possible interested by image processing techniques. Keep the things simple, affordable and useful !