

Introduction to The CImg Library

C++ Template Image Processing Toolbox (version 1.5)



```
// Bouncing bubble
//-----
CImg<unsigned char> back(320,256,1,3,0),img;
cimg_forXY(back,x,y) back(x,y,2) = (unsigned char)((y<2*ba
CImgDisplay disp(back,"Bouncing bubble",0,1);
const unsigned char col1[3]={40,100,103}, col2[3]={20,70,0}
double u = std::sqrt(2.0), cx = back.dimx()/2, t = 0, vt=
while (!disp.is_closed && disp.key!=cimg::keyQ && disp.key
img = back;
int xm=(int)cx, ym = (int)(img.dimy()/2-70 + (img.dimy(
float r1 = 50, r2 = 50;
vt=0.05;
if (xm+r1>img.dimx()) { const float delta = (xm+r1)-
if (xm-r1<0) { const float delta = -(xm-r1)
if (ym+r2>img.dimy()-40) { const float delta = (ym+r2)-
if (ym-r2<0) { const float delta = -(ym-r2)
img.draw_ellipse(xm,ym,r1,r2,1,0,col1);
img.draw_ellipse((int)(xm+0.03*r1*u),(int)(ym-0.03*r2*u)
img.draw_ellipse((int)(xm+0.1*r1*u),(int)(ym-0.1*r2*u),0
img.draw_ellipse((int)(xm+0.2*r1*u),(int)(ym-0.2*r2*u),r
img.draw_ellipse((int)(xm+0.3*r1*u),(int)(ym-0.3*r2*u),r
```

David Tschumperlé

CNRS UMR 6072 (GREYC) - Image Team

- Document available at : http://cimg.sourceforge.net/CImg_slides.pdf



Attribution-NonCommercial-ShareAlike 2.5

You are free:

- to **Share** -- to copy, distribute, display, and perform the work
- to **Remix** -- to make derivative works

Under the following conditions:



Attribution. You must attribute the work in the manner specified by the author or licensor.



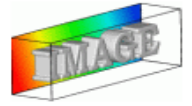
Noncommercial. You may not use this work for commercial purposes.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Outline - PART I of II : CImg Library Overview



- **Context** : Image Processing with C++.
 - Aim and targeted audience.
 - Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.
 - Drawing things on images.
- **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

Outline - PART II of II : More insights

- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - Morphomaths - Median Filter.
 - Anisotropic smoothing.
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.
- The plug-in mechanism.
- Dealing with 3D objects.
- Shared images.

PART I of II

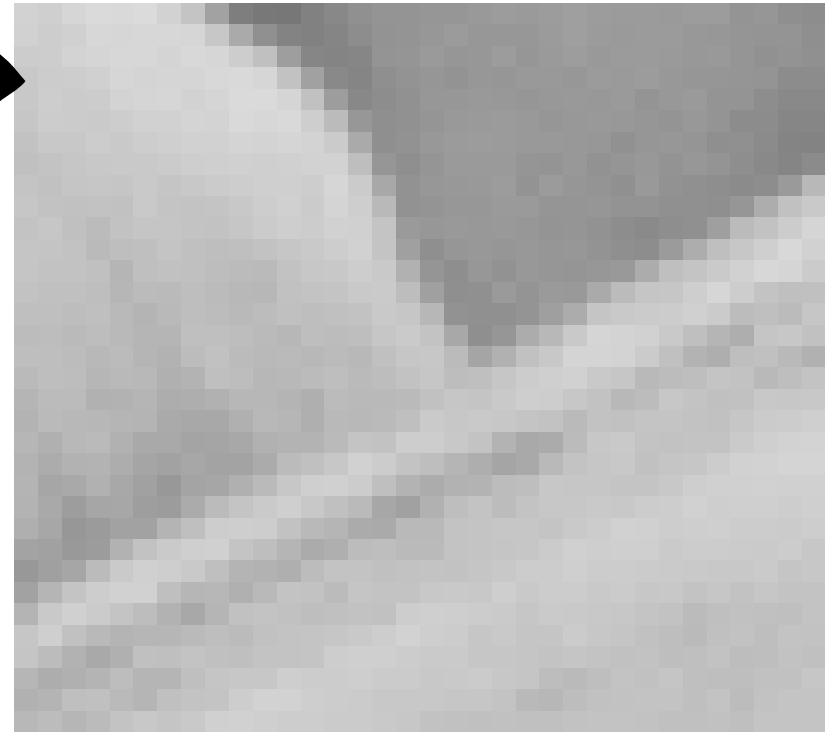
⇒ Context : Image Processing with C++.

- Aim and targeted audience.
- Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.
 - Drawing things on images.
- **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

Outline - PART I of II : CImg Library Overview

- **Context** : Image Processing with C++.
 - ⇒ **Aim and targeted audience.**
 - Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.
 - Drawing things on images.
- **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

- Digital Images.



- On a computer, image data stored as a **discrete array of values** (pixels or voxels).

- Acquired digital images have a lot of different types :
 - Domain dimensions : $2D$ (static image), $2D + t$ (image sequence), $3D$ (volumetric image), $3D + t$ (sequence of volumetric images), ...

- Acquired digital images have a lot of different types :
 - **Domain dimensions** : $2D$ (static image), $2D + t$ (image sequence), $3D$ (volumetric image), $3D + t$ (sequence of volumetric images), ...
 - **Pixel dimensions** : Pixels can be scalars, colors, $N - D$ vectors, matrices, ...

- Acquired digital images have a lot of different types :
 - **Domain dimensions** : $2D$ (static image), $2D + t$ (image sequence), $3D$ (volumetric image), $3D + t$ (sequence of volumetric images), ...
 - **Pixel dimensions** : Pixels can be scalars, colors, $N - D$ vectors, matrices, ...
 - **Pixel data range** : depends on the sensors used for acquisition, can be N -bits (usually 8,16,24,32...), sometimes float-valued.

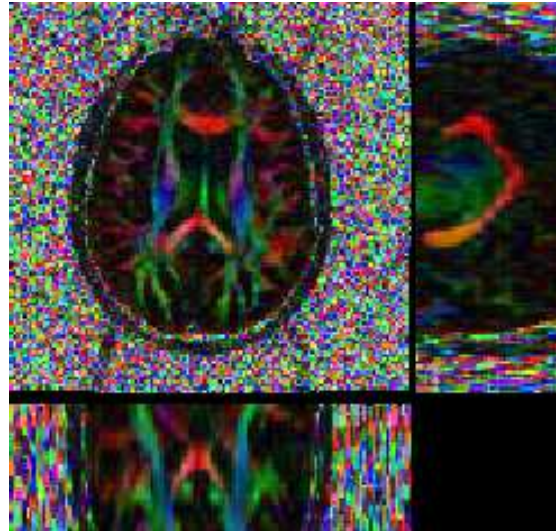
- Acquired digital images have a lot of different types :
 - **Domain dimensions** : $2D$ (static image), $2D + t$ (image sequence), $3D$ (volumetric image), $3D + t$ (sequence of volumetric images), ...
 - **Pixel dimensions** : Pixels can be scalars, colors, $N - D$ vectors, matrices, ...
 - **Pixel data range** : depends on the sensors used for acquisition, can be N -bits (usually 8,16,24,32...), sometimes float-valued.
 - **Type of sensor grid** : Rectangular, Octagonal, ...

- Acquired digital images have a lot of different types :
 - **Domain dimensions** : $2D$ (static image), $2D + t$ (image sequence), $3D$ (volumetric image), $3D + t$ (sequence of volumetric images), ...
 - **Pixel dimensions** : Pixels can be scalars, colors, $N - D$ vectors, matrices, ...
 - **Pixel data range** : depends on the sensors used for acquisition, can be N -bits (usually 8,16,24,32...), sometimes float-valued.
 - **Type of sensor grid** : Rectangular, Octagonal, ...
- All these different image types are digitally stored using different file formats :
 - PNG, JPEG, BMP, TIFF, TGA, DICOM, ANALYZE, ...

Context



(a) $I_1 : W \times H \rightarrow [0, 255]^3$



(b) $I_2 : W \times H \times D \rightarrow [0, 65535]^{32}$



(c) $I_3 : W \times H \times T \rightarrow [0, 4095]$

- I_1 : classical *RGB* color image (digital photograph, scanner, ...) (8 bits)
- I_2 : DT-MRI volumetric image with 32 magnetic field directions (16 bits)
- I_3 : Sequence of echography images (12 or 16 bits).

- Image Processing and Computer Vision aim at the **elaboration of numerical algorithms** able to automatically **extract features** from images, **interpret** them and then **take decisions**.

⇒ **Conversion of a pixel array to a semantic description of the image.**

- Is there any **white pixel** in this image ?
- Is there any **contour** in this image ?
- Is there any **object** ?
- **Where's the car** ?
- Is there **anybody** driving the car ?



Some observations about Image Processing and Computer Vision :

- There are huge and active research fields.
- The final goal is almost impossible to achieve !
- There are been thousands (millions?) of algorithms proposed in this field, most of them relying on strong mathematical modeling.
- The community is varied and not only composed of very talented programmers.

⇒ How to design a reasonable and useable programming library for such people ?

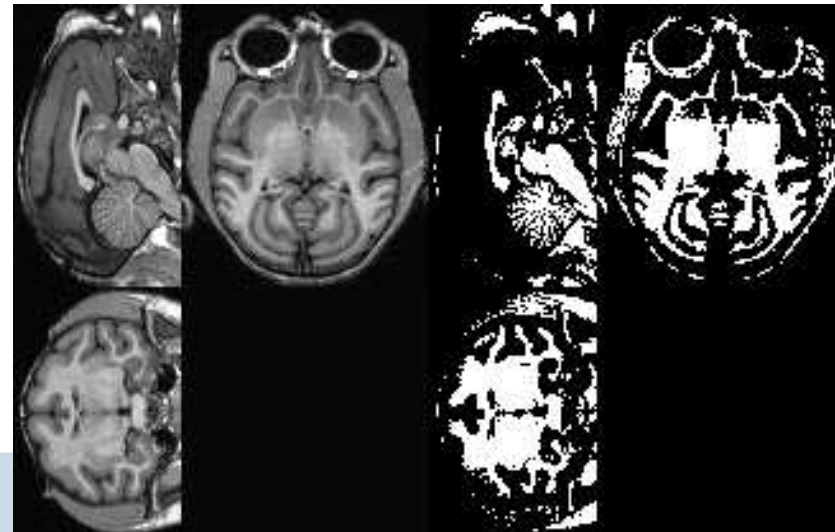
Observation

- Most of advanced image processing techniques are “type independent”.
- Ex : **Binarization** of an image $I : \Omega \rightarrow \Gamma$ by a threshold $\epsilon \in \mathbb{R}$.

$$\tilde{I} : \Omega \rightarrow \{0, 1\} \quad \text{such that} \quad \forall p \in \Omega, \quad \tilde{I}(p) = \begin{cases} 0 & \text{if } \|I(p)\| < \epsilon \\ 1 & \text{if } \|I(p)\| \geq \epsilon \end{cases}$$



$$I_1 : \Omega \in \mathbb{R}^2 \longrightarrow [0, 255]$$



$$I_2 : \Omega \in \mathbb{R}^3 \longrightarrow \mathbb{R}$$

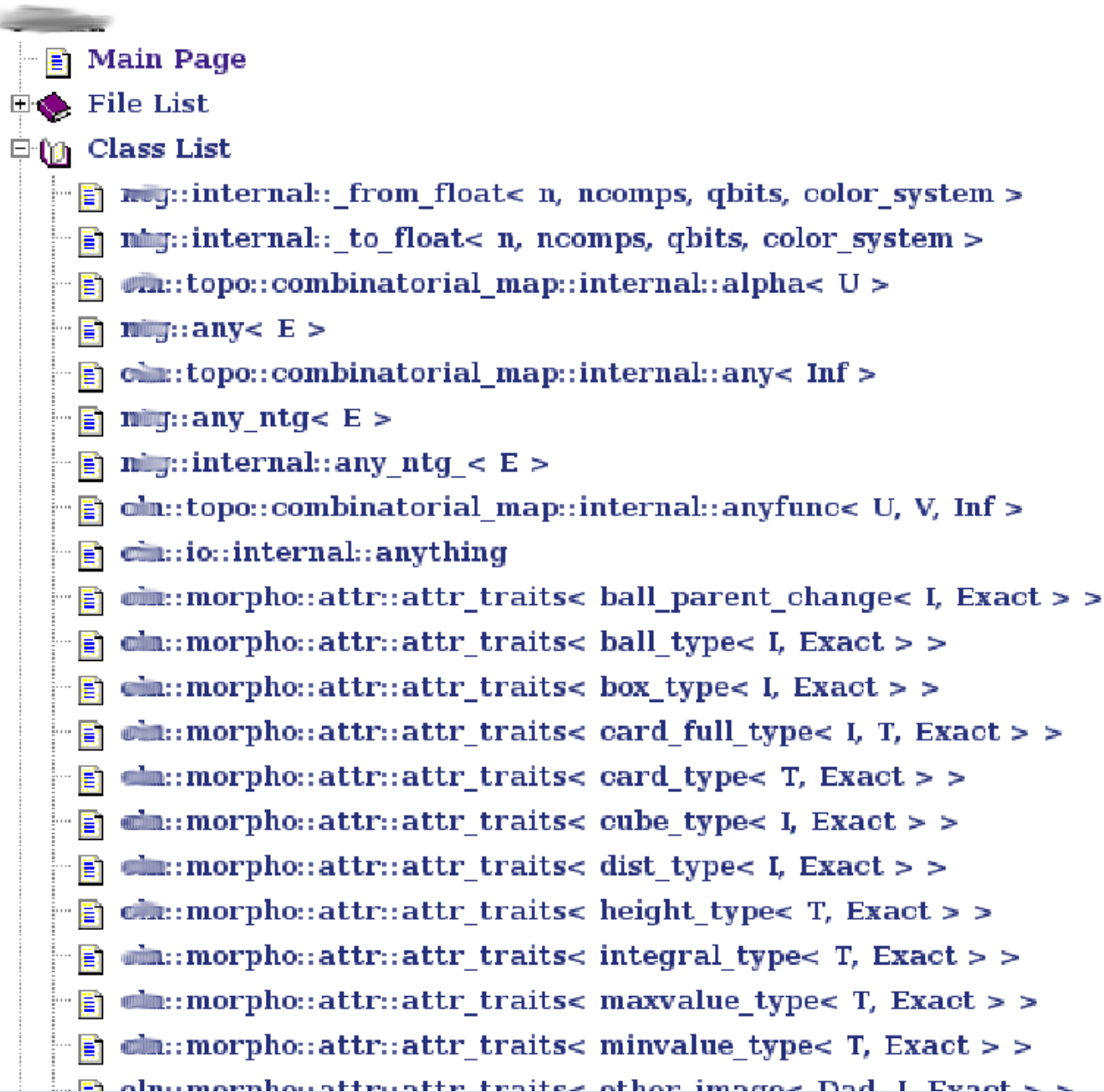
- Implementing an image processing algorithm should be as independent as possible on the image format and coding.

⇒ Generic Image Processing Libraries :

(...), FreeImage, Devil, (...), OpenCV, Pandore, CImg, Vigna, GIL, Olena, (...)

- C++ is a “good” programming language for solving such a problem :
 - Genericity is possible, quite elegant and flexible ([template mechanism](#)).
 - Compiled code. Fast executables (good for time-consuming algorithms).
 - Portable , huge base of existing code.
- ***Danger : Too much genericity may lead to unreadable code.***

Too much genericity... (Example 1).



Too much genericity... (Example 2).

```
typedef cross_vector_image_view_types
< mpl::vector<bits8, bits16>,
  mpl::vector<rgb_t, cmyk_t>,
  kInterleavedAndPlanar,
  kNonStepAndStep,
  false // false == mutable; true == read-only
>::type my_views_t;
typedef any_image_view<my_views_t> my_any_image_view_t;
```

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

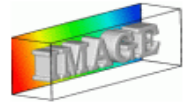
typedef mpl::vector<gray8_image_t, gray16_image_t, rgb8_image_t, rgb16_image_t> my_img_types;
any_image<my_img_types> runtime_image;
jpeg_read_image("input.jpg", runtime_image);

gray8s_image_t gradient(get_dimensions(runtime_image));
x_luminosity_gradient(const_view(runtime_image), view(gradient));
jpeg_write_view("x_gradient.jpg", color_converted_view<gray8_pixel_t>(const_view(gradient)));
```

- Strictly speaking, this is more C++ stuffs (problems?) than image processing.

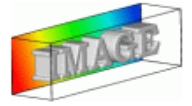
⇒ **Definitely not suitable for non computer geeks !!**

The CImg Library



- An open-source C++ library aiming to **simplify** the development of image processing algorithms for generic (enough) datasets (CeCILL License).

The CImg Library



- An open-source C++ library aiming to **simplify** the development of image processing algorithms for generic (enough) datasets (CeCILL License).
- **Primary audience** : Students and researchers working in Computer Vision and Image Processing labs, and having **standard notions** of C++.

The CImg Library

- An open-source C++ library aiming to **simplify** the development of image processing algorithms for generic (enough) datasets (CeCILL License).
- **Primary audience** : Students and researchers working in Computer Vision and Image Processing labs, and having **standard notions** of C++.
- It defines a set of C++ classes able to **manipulate and process** image objects.

The CImg Library

- An open-source C++ library aiming to **simplify** the development of image processing algorithms for generic (enough) datasets (CeCILL License).
- **Primary audience** : Students and researchers working in Computer Vision and Image Processing labs, and having **standard notions** of C++.
- It defines a set of C++ classes able to **manipulate and process image objects**.
- Started in **late 1999**, the project is now hosted on Sourceforge since December 2003 :

<http://cimg.sourceforge.net/>



THE CIMG LIBRARY

C++ Template Image Processing Library.

Outline - PART I of II : CImg Library Overview

- **Context** : Image Processing with C++.
 - Aim and targeted audience.
- ⇒ **Why considering The CImg Library ?**
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.
 - Drawing things on images.
- **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

Main characteristics

Clmg is **lightweight** :

- Total size of the full Clmg (.zip) package : approx. **12.5 Mb**.

Main characteristics

CImg is **lightweight** :

- Total size of the full CImg (.zip) package : approx. **12.5 Mb**.
- All the library is contained in a **single header file CImg.h**, that must be included in your C++ source :

```
#include "CImg.h"           // Just do that...  
using namespace cimg_library; // ...and you can play with the library
```

Main characteristics

CImg is **lightweight** :

- Total size of the full CImg (.zip) package : approx. **12.5 Mb**.
- All the library is contained in a **single header file CImg.h**, that must be included in your C++ source :

```
#include "CImg.h"           // Just do that...  
using namespace cimg_library; // ...and you can play with the library
```

- The library itself only takes **2.2Mb of sources** (approximately **45000** lines).
- The library package contains the file **CImg.h** as well as documentation, examples of use, and additional plug-ins.

Main characteristics

CImg is **lightweight** :

- What ? a library defined in a single header file ?
 - Simplicity “a la STL”.

Main characteristics

CImg is **lightweight** :

- What ? a library defined in a single header file ?
 - Simplicity “a la STL”.
 - Used template functions and structures know their type only during the compilation phase :
 - ⇒ No relevance in having pre-compiled objects (.cpp→.o).

Main characteristics

CImg is **lightweight** :

- What ? a library defined in a single header file ?
 - Simplicity “a la STL”.
 - Used template functions and structures know their type only during the compilation phase :
 - ⇒ No relevance in having pre-compiled objects (.cpp→.o).
 - Why not several headers (one for each class) ?
 - ⇒ Interdependence of the classes : all headers would be always necessary.

Main characteristics

CImg is **lightweight** :

- What ? a library defined in a single header file ?
 - Simplicity “a la STL”.
 - Used template functions and structures know their type only during the compilation phase :
 - ⇒ No relevance in having pre-compiled objects (.cpp→.o).
 - Why not several headers (one for each class) ?
 - ⇒ Interdependence of the classes : all headers would be always necessary.
 - Only used functions are actually compiled :
 - ⇒ Small generated executables.

Main characteristics

CImg is **lightweight** :

- What ? a library defined in a single header file ?
 - Simplicity “a la STL”.
 - Used template functions and structures know their type only during the compilation phase :
 - ⇒ No relevance in having pre-compiled objects (.cpp→.o).
 - Why not several headers (one for each class) ?
 - ⇒ Interdependence of the classes.
 - Only used functions are actually compiled :
 - ⇒ Small generated executables.
- **Drawback** : Compilation time and needed memory important when optimization flags are set.

Main characteristics

CImg is (sufficiently) **generic** :

- CImg implements static genericity by using the **C++ template mechanism**.
- **One template parameter only** : the type of the image pixel.

Main characteristics

CImg is (sufficiently) **generic** :

- CImg implements static genericity by using the C++ template mechanism.
- One template parameter only : the type of the image pixel.
- CImg defines an image class that can handle hyperspectral volumetric (i.e 4D) images of generic pixel types.

Main characteristics

CImg is (sufficiently) **generic** :

- CImg implements static genericity by using the C++ template mechanism.
- One template parameter only : the type of the image pixel.
- CImg defines an image class that can handle hyperspectral volumetric (i.e 4D) images of generic pixel types.
- CImg defines an image list class that can handle temporal image sequences.

Main characteristics

Clmg is (sufficiently) **generic** :

- Clmg implements static genericity by using the C++ template mechanism.
- One template parameter only : the type of the image pixel.
- Clmg defines an image class that can handle hyperspectral volumetric (i.e 4D) images of generic pixel types.
- Clmg defines an image list class that can handle temporal image sequences.
- ... But, Clmg is limited to images having a rectangular grid, and cannot handle images having more than 4 dimensions.

Main characteristics

CImg is (sufficiently) **generic** :

- CImg implements static genericity by using the **C++ template mechanism**.
- **One template parameter only** : the type of the image pixel.
- CImg defines **an image class** that can handle **hyperspectral volumetric** (i.e 4D) images of **generic pixel types**.
- CImg defines **an image list class** that can handle **temporal image sequences**.
- ... But, CImg is limited to images having a **rectangular grid**, and cannot handle images having more than 4 dimensions.

⇒ CImg covers actually 99% of the image types found in real world applications.

Main characteristics

CImg is **multi-platform** :

- It does not depend on many libraries.

It can be compiled only with existing system libraries.

Main characteristics

CImg is **multi-platform** :

- It does not depend on many libraries.
It can be compiled only with existing system libraries.
- Advanced tools or libraries may be used by CImg (ImageMagick, XMedcon, libpng, libjpeg, libtiff, libfftw3...), these tools being freely available for any platform.

Main characteristics

CImg is **multi-platform** :

- It does not depend on many libraries.
It can be compiled only with existing system libraries.
- Advanced tools or libraries may be used by CImg (ImageMagick, XMedcon, libpng, libjpeg, libtiff, libfftw3...), these tools being freely available for any platform.
- Successfully tested platforms : Win32, Linux, Solaris, *BSD, Mac OS X.
- It is also “multi-compiler” : g++, Visual Studio .NET, Intel ICL, Clang++.

Main characteristics

And most of all, CImg is **very simple to use** :

- Only 1 single file to include.

Main characteristics

And most of all, CImg is **very simple to use** :

- Only 1 single file to include.

- Only 4 C++ classes to know :

`CImg<T>`, `CImgList<T>`, `CImgDisplay`, `CImgException`.

Main characteristics

And most of all, CImg is **very simple to use** :

- Only 1 single file to include.
- Only 4 C++ classes to know :
`CImg<T>`, `CImgList<T>`, `CImgDisplay`, `CImgException`.
- Very basic low-level architecture, simple to apprehend (and to hack if necessary!).

Main characteristics

And most of all, CImg is **very simple to use** :

- Only 1 single file to include.
- Only 4 C++ classes to know :
`CImg<T>`, `CImgList<T>`, `CImgDisplay`, `CImgException`.
- Very basic low-level architecture, simple to apprehend (and to hack if necessary!).
- Enough genericity and library functions, allowing complex image processing tasks.

Main characteristics

And most of all, CImg is **very simple to use** :

- Only 1 single file to include.
- Only 4 C++ classes to know :
`CImg<T>`, `CImgList<T>`, `CImgDisplay`, `CImgException`.
- Very basic low-level architecture, simple to apprehend (and to hack if necessary!).
- Enough genericity and library functions, allowing complex image processing tasks.

.... and **extensible** :

- Simple plug-in mechanism to easily add your own functions to the library core (without modifying the file `CImg.h` of course).

Hello World step by step

```
#include "CImg.h"

using namespace cimg_library;

int main(int argc, char **argv) {

    return 0;
}
```

Hello World step by step

```
#include "CImg.h"
using namespace cimg_library;

int main(int argc, char **argv) {

    CImg<unsigned char> img(300,200,1,3);

    return 0;
}
```


Hello World step by step

```
#include "CImg.h"
using namespace cimg_library;

int main(int argc, char **argv) {

    CImg<unsigned char> img(300,200,1,3);
    img.fill(32);

    return 0;
}
```

Hello World step by step

```
#include "CImg.h"
using namespace cimg_library;

int main(int argc, char **argv) {

    CImg<unsigned char> img(300,200,1,3);
    img.fill(32);
    img.noise(128);

    return 0;
}
```

Hello World step by step

```
#include "CImg.h"
using namespace cimg_library;

int main(int argc, char **argv) {

    CImg<unsigned char> img(300,200,1,3);
    img.fill(32);
    img.noise(128);
    img.blur(2,0,0);

    return 0;
}
```

Hello World step by step

```
#include "CImg.h"

using namespace cimg_library;

int main(int argc, char **argv) {

    CImg<unsigned char> img(300,200,1,3);
    img.fill(32);
    img.noise(128);
    img.blur(2,0,0);
    const unsigned char white[] = { 255,255,255 };
    img.draw_text(80,80,"Hello World",white,0,32);

    return 0;
}
```

Hello World step by step

```
#include "CImg.h"

using namespace cimg_library;

int main(int argc, char **argv) {

    CImg<unsigned char> img(300,200,1,3);
    img.fill(32);
    img.noise(128);
    img.blur(2,0,0);
    const unsigned char white[] = { 255,255,255 };
    img.draw_text(80,80,"Hello World",white,0,32);
    img.display();

    return 0;
}
```

Hello World step by step



Hello World step by step : animated

```
#include "CImg.h"
using namespace cimg_library;

int main(int argc, char **argv) {

    const CImg<unsigned char> img =
        CImg<unsigned char>(300,200,1,3).fill(32).noise(128).blur(2,0,0).
        draw_text(80,80,"Hello World",CImg<unsigned char>::vector(255,255,255).ptr(),0,32);

    CImgDisplay disp(img,"Moving Hello World",0);
    for (float t=0; !disp.is_closed(); t+=0.04) {
        CImg<unsigned char> res(img);
        cimg_forYC(res,y,v)
            res.get_shared_row(y,0,v).shift((int)(40*std::sin(t+y/50.0)),0,0,0,2);
        disp.display(res).wait(20);
        if (disp.is_resized()) disp.resize();
    }
    return 0;
}
```

Another example : Computing gradient norm of a 3D volumetric image

- Let $I : \Omega \in \mathbb{R}^3 \rightarrow \mathbb{R}$, compute

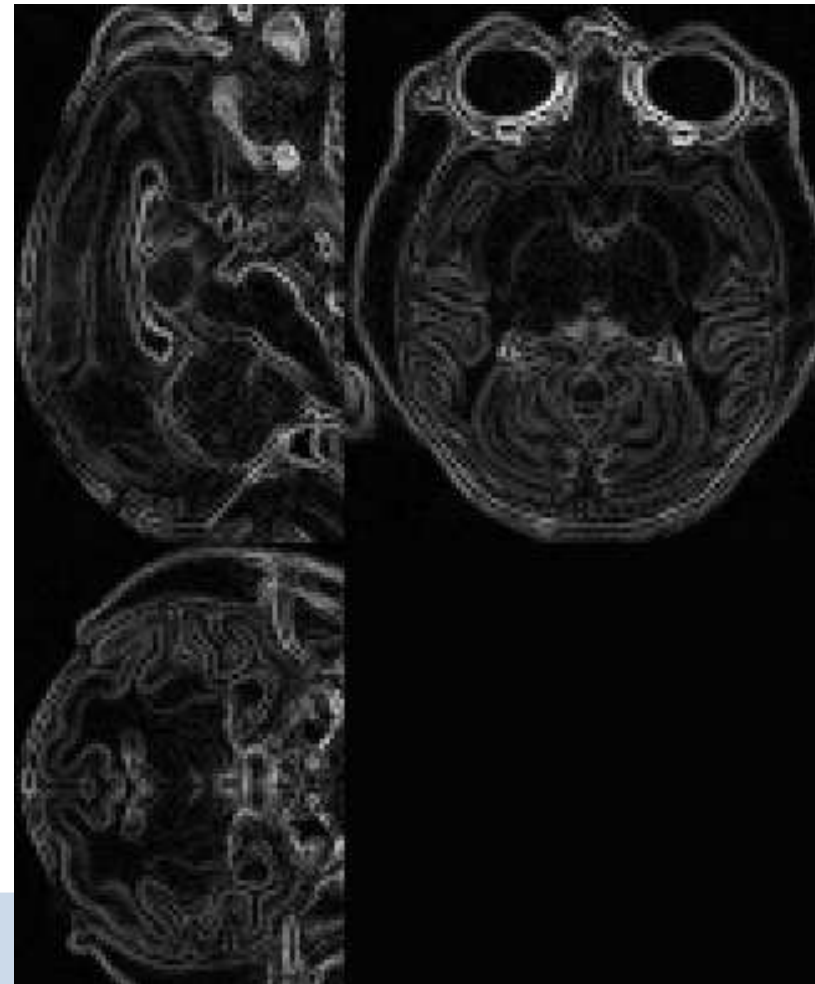
$$\forall p \in \Omega, \quad \|\nabla I\|_{(p)} = \sqrt{\left(\frac{\partial I}{\partial x}\right)^2 + \left(\frac{\partial I}{\partial y}\right)^2 + \left(\frac{\partial I}{\partial z}\right)^2}$$

- Code :

```
#include 'CImg.h'
using namespace cimg_library;

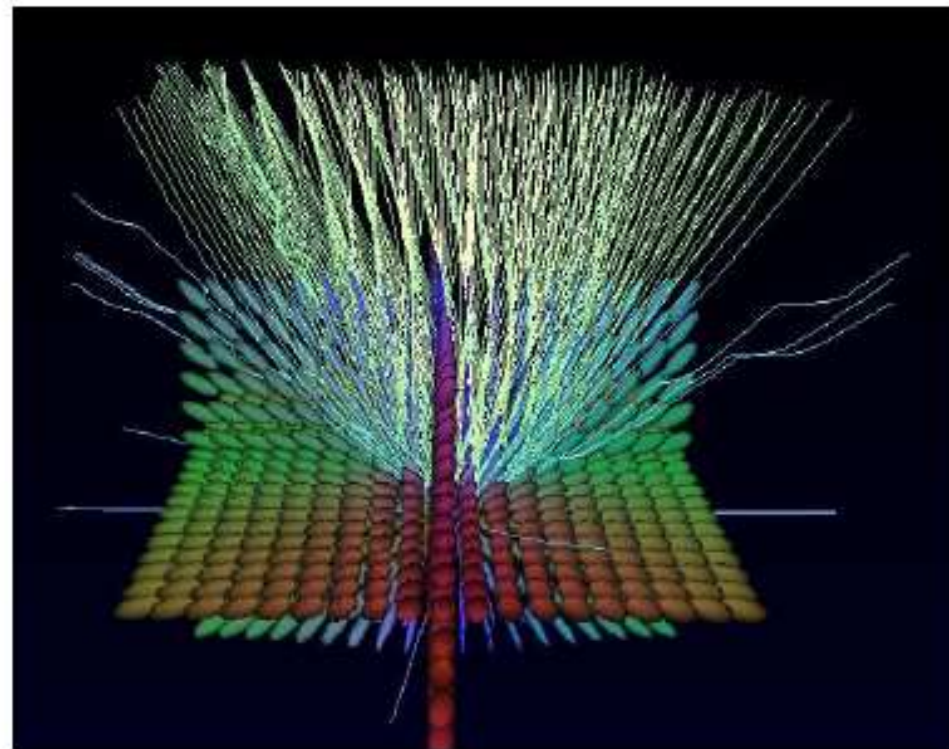
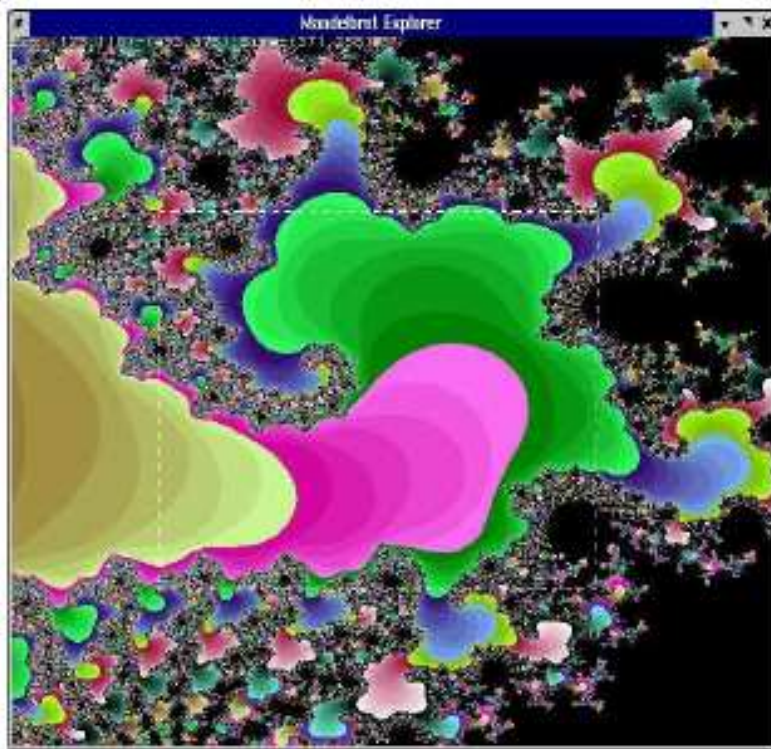
int main(int argc, char **argv) {
    const CImg<float> img('brain_irm3d.hdr');
    const CImgList<float> grad = img.get_gradient('xyz');
    CImg<float> norm = (grad[0].pow(2) + grad[1].pow(2) + grad[2].pow(2));
    norm.sqrt().get_normalize(0,255).save('brain_gradient3d.hdr');
    return 0;
}
```


Another example : Computing gradient norm of a 3D volumetric image

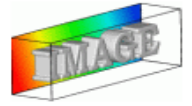


Live Demo !

- Let see what we can do with this library.



Overall Library Structure



- The whole library classes and functions are defined in the `cimg_library::` namespace.

Overall Library Structure

- The whole library classes and functions are defined in the `cimg_library::` namespace.
- The library is composed of only **four C++ classes** :
 - **CImg<T>**, represents an image with pixels of type T.

Overall Library Structure

- The whole library classes and functions are defined in the `cimg_library::` namespace.
- The library is composed of only **four C++ classes** :
 - **CImg<T>**, represents an image with pixels of type T.
 - **CImgList<T>**, represents a list of images `CImg<T>`.

Overall Library Structure

- The whole library classes and functions are defined in the `cimg_library::` namespace.
- The library is composed of only **four C++ classes** :
 - **CImg<T>**, represents an image with pixels of type T.
 - **CImgList<T>**, represents a list of images `CImg<T>`.
 - **CImgDisplay**, represents a display window.

Overall Library Structure

- The whole library classes and functions are defined in the `cimg_library::` namespace.
- The library is composed of only **four C++ classes** :
 - **CImg<T>**, represents an image with pixels of type T.
 - **CImgList<T>**, represents a list of images `CImg<T>`.
 - **CImgDisplay**, represents a display window.
 - **CImgException**, used to throw library exceptions.

Overall Library Structure

- The whole library classes and functions are defined in the `cimg_library::` namespace.
- The library is composed of only **four C++ classes** :
 - **CImg<T>**, represents an image with pixels of type T.
 - **CImgList<T>**, represents a list of images `CImg<T>`.
 - **CImgDisplay**, represents a display window.
 - **CImgException**, used to throw library exceptions.
- A sub-namespace `cimg_library::cimg::` defines some low-level library functions (including some useful ones as `rand()`, `grand()`, `min<T>()`, `max<T>()`, `abs<T>()`, `sleep()`, etc...).

Overall Library Structure

cimg_library::

cimg::

Low-level functions

CImg<T>

Image

CImgList<T>

Image List

CImgException

Error handling

CImgDisplay

Display Window

- All CImg classes incorporate two different kinds of methods :
 - Methods which **act directly on the instance object** and modify it. These methods **returns a reference to the current instance**, so that writting **function pipelines** is possible :

```
CImg<>('toto.jpg').blur(2).mirror('y').rotate(45).save('tutu.jpg');
```

- All CImg classes incorporate two different kinds of methods :
 - Methods which **act directly on the instance object** and modify it. These methods **returns a reference to the current instance**, so that writting **function pipelines** is possible :

```
CImg<>('toto.jpg').blur(2).mirror('y').rotate(45).save('tutu.jpg');
```

- Other methods **return a modified copy of the instance**. These methods start with `get_*` :

```
CImg<> img('toto.jpg');
```

```
CImg<> img2 = img.get_blur(2); // 'img' is not modified
```

```
CImg<> img3 = img.get_rotate(20).blur(3); // 'img' is not modified
```

- All CImg classes incorporate two different kinds of methods :
 - Methods which **act directly on the instance object** and modify it. These methods **returns a reference to the current instance**, so that writting **function pipelines** is possible :

```
CImg<>('toto.jpg').blur(2).mirror('y').rotate(45).save('tutu.jpg');
```

- Other methods **return a modified copy of the instance**. These methods start with `get_*` :

```
CImg<> img('toto.jpg');  
CImg<> img2 = img.get_blur(2);           // 'img' is not modified  
CImg<> img3 = img.get_rotate(20).blur(3); // 'img' is not modified
```

⇒ **Almost all CImg methods are declined into these two versions.**

Outline - PART I of II : CImg Library Overview

- **Context** : Image Processing with C++.

- Aim and targeted audience.
- Why considering The CImg Library ?

⇒ **CImg<T> : A class for image manipulation.**

- Image construction, data access, math operators.
- Basic image transformations.
- Drawing things on images.

- **CImgList<T>** : Image collection manipulation.

- Basic manipulation functions.

- **CImgDisplay** : Image display and user interaction.

- Displaying images in windows.

CImg<T> : Overview

- This is the **main class** of the CImg Library. It has a **single template parameter** `T`.
- A `CImg<T>` represents an image with pixels **of type** `T` (default template parameter is `T=float`). Supported types are the C/C++ basic types : `bool`, `unsigned char`, `char`, `unsigned short`, `short`, `unsigned int`, `int`, `float`, `double`, ...

CImg<T> : Overview

- This is the **main class** of the CImg Library. It has a **single template parameter** T.
- A **CImg<T>** represents an image with pixels **of type T** (default template parameter is **T=float**). Supported types are the C/C++ basic types : bool, unsigned char, char, unsigned short, short, unsigned int, int, float, double,...
- An image has always **3 spatial dimensions** (width, height, depth) + **1 hyperspectral dimension** (dim) : It can represent any data from a **scalar 1D signal** to a **3D volume of vector-valued pixels**.

CImg<T> : Overview

- This is the **main class** of the CImg Library. It has a **single template parameter** T.
- A **CImg<T>** represents an image with pixels **of type T** (default template parameter is **T=float**). Supported types are the C/C++ basic types : bool, unsigned char, char, unsigned short, short, unsigned int, int, float, double,...
- An image has always **3 spatial dimensions** (width, height, depth) + **1 hyperspectral dimension** (dim) : It can represent any data from a **scalar 1D signal** to a **3D volume of vector-valued pixels**.
- Image processing algorithms are **methods of CImg<T>** (**≠ STL**) :
blur(), resize(), convolve(), erode(), load(), save()....
- Method implementation aims to handle **the most general case** (3D volumetric hyperspectral images).

CImg<T> : Low-level Architecture (for hackers!)

- The structure CImg<T> is defined as :

```
template<typename T> struct CImg {  
    unsigned int _width;  
    unsigned int _height;  
    unsigned int _depth;  
    unsigned int _dim;  
    bool _is_shared;  
    T* _data;  
};
```

CImg<T> : Low-level Architecture (for hackers!)

- The structure CImg<T> is defined as :

```
template<typename T> struct CImg {  
    unsigned int _width;  
    unsigned int _height;  
    unsigned int _depth;  
    unsigned int _dim;  
    bool _is_shared;  
    T* _data;  
};
```

- A CImg<T> image is **always entirely stored in memory**.
- A CImg<T> is **independent** : it has its own pixel buffer.

CImg<T> : Low-level Architecture (for hackers!)

- The structure CImg<T> is defined as :

```
template<typename T> struct CImg {  
    unsigned int width;  
    unsigned int height;  
    unsigned int depth;  
    unsigned int dim;  
    T* data;  
};
```

- A CImg<T> image is **always entirely stored in memory**.
- A CImg<T> is **independent** : it has its own pixel buffer most of the time.
- CImg member functions (destructor, constructors, operators,...) **handle memory allocation/desallocation efficiently**.

CImg<T> : Memory layout (for hackers!)

```
template<typename T> struct CImg {  
    unsigned int _width;  
    unsigned int _height;  
    unsigned int _depth;  
    unsigned int _dim;  
    bool _is_shared;  
    T* _data;  
};
```

- Pixel values are not stored in a typical “RGBRGBRGBRGBRGB” order.
- Pixel values are stored first along the X-axis, then the Y-axis, then the Z-axis, then the C-axis :

$R(0,0) R(1,0) \dots R(W-1,0) \dots R(0,1) R(1,1) \dots R(W-1,1) \dots R(0,H-1) R(1,H-1)$
 $\dots R(W-1,H-1) \dots G(0,0) \dots G(W-1,H-1) \dots B(0,0) \dots B(W-1,H-1).$

Outline - PART I of II : CImg Library Overview

- **Context** : Image Processing with C++.
 - Aim and targeted audience.
 - Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - ⇒ **Image construction, data access, math operators.**
 - Basic image transformations.
 - Drawing things on images.
- **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

CImg<T> : Constructors (1)

- Default constructor, constructs an empty image.
`CImg<T>();`
- No memory allocated in this case, images dimensions are zero.
- Useful to declare an image without allocating its pixel values.

```
#include 'CImg.h'
using namespace cimg_library;

int main() {
    CImg<unsigned char> img_8bits;
    CImg<unsigned short> img_16bits;
    CImg<float> img_float;
    return 0;
}
```

CImg<T> : Constructors (2)

- Constructs a 4D image **with specified dimensions**. Omitted dimensions are set to 1 (default parameter).

`CImg<T>(unsigned int, unsigned int, unsigned int, unsigned int);`

```
#include 'CImg.h'
using namespace cimg_library;

int main() {
    CImg<float> img(100,100); // 2D scalar image.
    CImg<unsigned char> img2(256,256,1,3); // 2D color image.
    CImg<bool> img3(128,128,128); // 3D scalar image.
    CImg<short> img4(64,64,32,16); // 3D hyperspectral image (16 bands).
    return 0;
}
```

- No initialization of pixel values is performed. Can be done with :

`CImg<T>(unsigned int, unsigned int, unsigned int, unsigned int, const T&);`

CImg<T> : Constructors (3)

- Create an image by reading an image from the disk (format deduced by the filename extension).

```
CImg<T>(const char *filename);
```

```
#include 'CImg.h'
```

```
using namespace cimg_library;
```

```
int main() {
```

```
    CImg<unsigned char> img('nouns.jpg');
```

```
    CImg<unsigned short> img2('toto.png');
```

```
    CImg<float> img3('toto.png');
```

```
    return 0;
```

```
}
```

- Pixel data of the file format are converted (static cast) to the specified template parameter.

CImg<T> : In-place constructors

- CImg<T>& assign(...)

Each constructor has an **in-place** version with same parameters.

```
CImg<float> img;  
img.assign('toto.jpg');  
img.assign(256,256,1,3,0);  
img.assign();
```

- This principle is extended to the other CImg classes.

```
CImgList<float> list;  
list.assign(img1,img2,img3);  
CImgDisplay disp;  
disp.assign(list,'List display');
```

CImg<T> : Access to image data informations

- Get the dimension along the X,Y,Z or C-axis (width, height, depth or channels).

```
int width() const;
```

```
int W = img.width(), H = img.height(), D = img.depth(), S = img.spectrum();
```

CImg<T> : Access to image data informations

- Get the dimension along the X,Y,Z or C-axis (width, height, depth or channels).

```
int width() const;
```

```
int W = img.width(), H = img.height(), D = img.depth(), S = img.spectrum();
```

- Get the pixel value at specified coordinates. Omitted coordinates are set to 0.

```
T& operator()(unsigned int, unsigned int, unsigned int, unsigned int);
```

```
unsigned char R = img(x,y), G = img(x,y,0,1), B = img(x,y,2);
```

```
float val = volume(x,y,z,v);
```

```
img(x,y,z) = x*y;
```

(Out-of-bounds coordinates are not checked !)

CImg<T> : Access to image data informations

- Get the dimension along the X,Y,Z or C-axis (Width, Height, Depth or Channels).

```
int width() const;
```

```
int W = img.width(), H = img.height(), D = img.depth(), S = img.spectrum();
```

- Get the pixel value at specified coordinates. Omitted coordinates are set to 0.

```
T& operator()(unsigned int, unsigned int, unsigned int, unsigned int);
```

```
unsigned char R = img(x,y), G = img(x,y,0,1), B = img(x,y,2);
```

```
float val = volume(x,y,z,v);
```

```
img(x,y,z) = x*y;
```

(Out-of-bounds coordinates are not checked !)

- Get the pixel value at specified sub-pixel position, using bicubic interpolation. Out-of-bounds coordinates are checked.

```
float cubic_pix2d(float, float, unsigned int, unsigned int);
```

```
float val = img.get_cubic_pix2d(x-0.5f,y-0.5f);
```

CImg<T> : Copies and assignments

- Construct an image **by copy**. Perform static pixel type cast if needed.

```
template<typename t> CImg<T>(const CImg<t>& img);
```

```
CImg<float> img_float(img_double);
```

CImg<T> : Copies and assignments

- Construct an image **by copy**. Perform static pixel type cast if needed.

```
template<typename t> CImg<T>(const CImg<t>& img);
```

```
CImg<float> img_float(img_double);
```

- Assignment operator. **Replace the instance image by a copy** of img.

```
template<typename t> CImg<T>& operator=(const CImg<t>& img);
```

```
CImg<float> img;
```

```
CImg<unsigned char> img2('toto.jpg'), img3(256,256);
```

```
img = img2;
```

```
img = img3;
```

- Modifying a copy **does not modify the original image** (own pixel buffer).

CImg<T> : Math operators and functions

- Most of the usual math operators are defined : $+$, $-$, $*$, $/$, $+=$, $-=$, \dots

```
CImg<float> img('toto.jpg'), dest;  
dest =(2*img+5);  
dest+=img;
```

CImg<T> : Math operators and functions

- Most of the usual math operators are defined : + , - , * , / , += , -= , ...

```
CImg<float> img('toto.jpg'), dest;  
dest =(2*img+5);  
dest+=img;
```

- Operators always try to return images with the best datatype.

```
CImg<unsigned char> img('toto.jpg');  
CImg<float> dest;  
dest = img*0.1f;  
img*=0.1f;
```


CImg<T> : Math operators and functions

- Most of the usual math operators are defined : $+$, $-$, $*$, $/$, $+=$, $-=$, ...

```
CImg<float> img('toto.jpg'), dest;  
dest =(2*img+5);  
dest+=img;
```

- Operators always try to return images with the best datatype.

```
CImg<unsigned char> img('toto.jpg');  
CImg<float> dest;  
dest = img*0.1f;  
img*=0.1f;
```

- Usual math functions are also defined : `sqrt()`, `cos()`, `pow()`...

```
img.pow(2.5);  
res = img.get_pow(2.5);  
res = img.get_cos().pow(2.5);
```

CImg<T> : Matrices operations

- The * and / operators corresponds to a **matrix product/division** !

```
CImg<float> A(3,3), v(1,3);
```

```
CImg<float> res = A*v;
```

- Use `CImg<T>::mul()` and `CImg<T>::div()` for **pointwise operators**.

CImg<T> : Matrices operations

- The * and / operators corresponds to a **matrix product/division** !

```
CImg<float> A(3,3), v(1,3);  
CImg<float> res = A*v;
```

- Use `CImg<T>::mul()` and `CImg<T>::div()` for **pointwise operators**.
- **Usual matrix functions and transformations** are available in CImg : determinant, SVD, eigenvalue decomposition, inverse, ...

```
CImg<float> A(10,10), v(1,10);  
const float determinant = A.det();  
CImg<float> pseudo_inv =  
((A*A.get_transpose()).inverse())*A.get_transpose();  
CImg<float> pseudo_inv2 = A.get_pseudoinverse();
```

CImg<T> : Matrices operations

- The * and / operators corresponds to a **matrix product/division** !

```
CImg<float> A(3,3), v(1,3);  
CImg<float> res = A*v;
```

- Use `CImg<T>::mul()` and `CImg<T>::div()` for **pointwise operators**.
- **Usual matrix functions and transformations** are available in CImg : determinant, SVD, eigenvalue decomposition, inverse, ...

```
CImg<float> A(10,10), v(1,10);  
const float determinant = A.det();  
CImg<float> pseudo_inv =  
((A*A.get_transpose()).inverse())*A.get_transpose();  
CImg<float> pseudo_inv2 = A.get_pseudoinverse();
```

- **Warning : Matrices are viewed as images, so first indice is the column number, second is the line number : $A_{ij} = A(j, i)$**

CImg<T> : Image destruction

- Image destruction is done in the `~CImg()` method.
- Used pixel buffer memory (if any) is automatically freed by the destructor.
- Destructor is automatically called at the end of a block.
- Memory deallocation **can be forced** by the `assign()` function.

```
CImg<float> img(10000,10000); // Need 4*10000^2 bytes = 380 Mo  
float det = img.det();
```

```
// We won't use img anymore...  
img.assign();
```

```
// Equivalent to :  
img = CImg<float>();
```

Outline - PART I of II : CImg Library Overview

- **Context** : Image Processing with C++.
 - Aim and targeted audience.
 - Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.

⇒ **Basic image transformations.**

 - Drawing things on images.
- **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

CImg<T> : Image manipulation

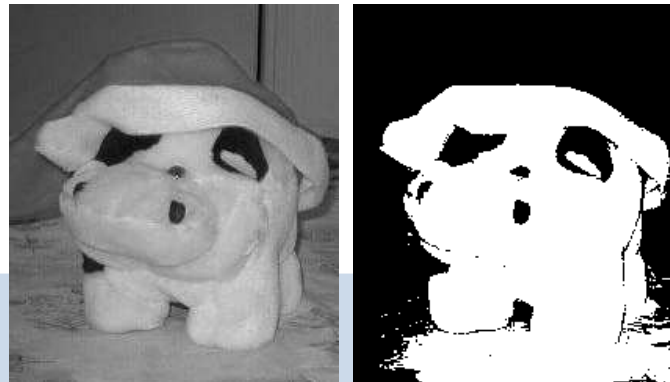
- `fill()` : Fill an image with one or several values.

```
CImg<> img(256,256), vector(1,6);  
img.fill(0);  
vector.fill(1,2,3,4,5,6);
```

- Apply basic global transformations on pixel values.

`normalize()`, `cut()`, `quantize()`, `threshold()`.

```
CImg<float>  
img("toto.jpg");  
img.quantize(16);  
img.normalize(0,1);  
img.cut(0.2f,0.8f);  
img.threshold(0.5f);  
img.normalize(0,255);
```



CImg<T> : Image manipulation

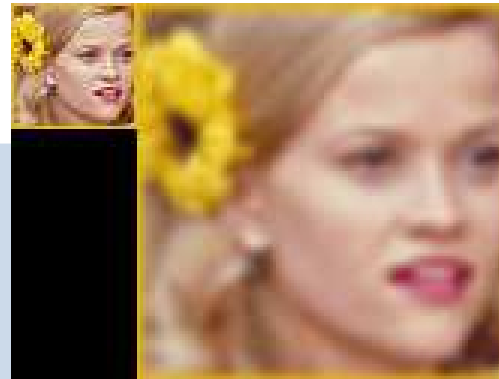
- `rotate()` : **Rotate** an image with a given angle.

```
CImg<> img('milla.png');  
img.rotate(30);
```

- `resize()` : **Resize** an image with a given size.

```
CImg<> img('mini.jpg');  
img.resize(-300,-300); // -300 = 300%
```

⇒ Border conditions and interpolation types can be chosen by the user.



CImg<T> : Image manipulation



- `get_crop()` : Get a **sub-image** of the instance image.

```
CImg<> img(256,256);  
img.get_crop(0,0,128,128); // Get the upper-left half image
```

- **Color space-conversions** : `RGBtoYUV()`, `RGBtoLUT()`, `RGBtoHSV()`, ... and inverse transformations.
- **Filtering** : `blur()`, `convolve()`, `erode()`, `dilate()`, `FFT()`, `deriche()`,
- In the reference documentation, functions are grouped by themes....

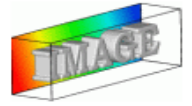
<http://cimg.sourceforge.net/reference/>

CImg<T> : Image manipulation

```
#include 'CImg.h'
using namespace cimg_library;
int main() {
    CImg<unsigned char> img('milla.jpg');
    img.blur(1).crop(15,52,150,188).dilate(10).mirror('x');
    img.save('result.png');
    return 0;
}
```



Outline - PART I of II : CImg Library Overview



- **Context** : Image Processing with C++.
 - Aim and targeted audience.
 - Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.

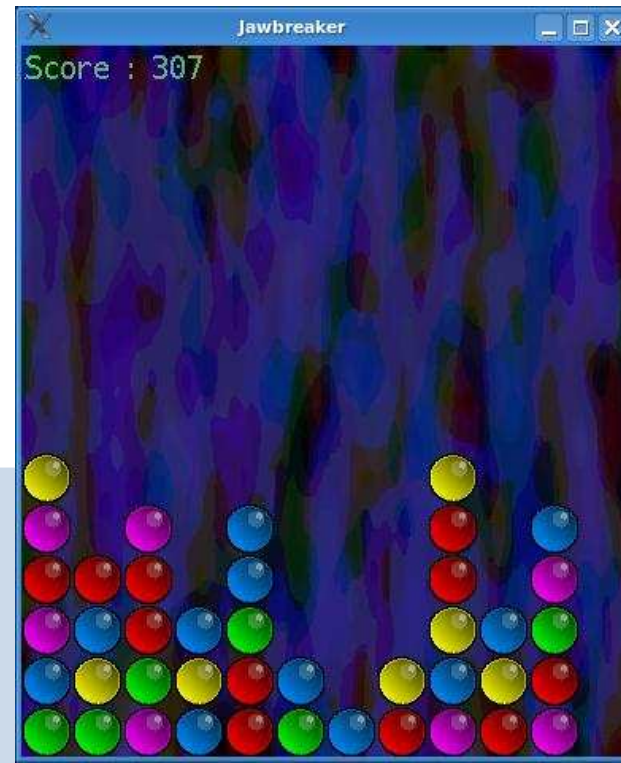
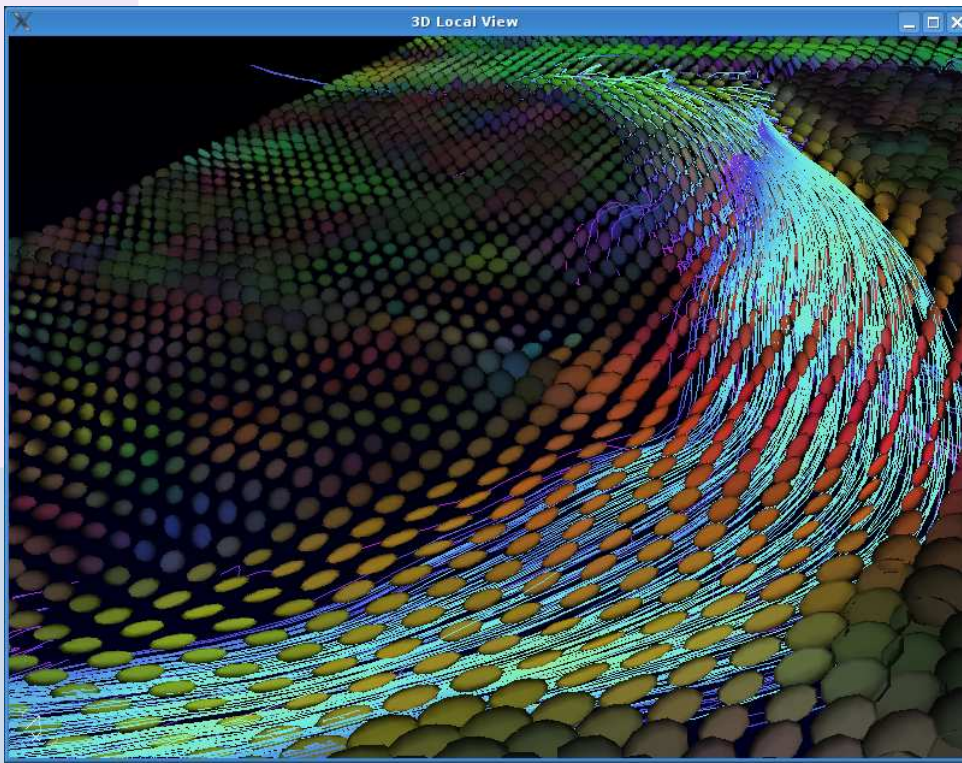
⇒ **Drawing things on images.**
- **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

CImg<T> : Drawing functions

- CImg proposes a lot of functions to draw features in images.

⇒ Points, lines, circles, rectangles, triangles, text, vector fields, 3D objects, ...

- All drawing function names begin with `draw_*`.
- Features are drawn directly on the instance image (so there are not `const`).



CImg<T> : Drawing functions

- All drawing functions **work the same way** : They need **the instance image**, **feature coordinates**, and a **color** (eventual other optional parameters can be set).

CImg<T> : Drawing functions

- All drawing functions **work the same way** : They need **the instance image**, **feature coordinates**, and a **color** (eventual other optional parameters can be set).
- They **return a reference** to the instance image, so they can be **pipelined**.

CImg<T> : Drawing functions

- All drawing functions **work the same way** : They need **the instance image**, **feature coordinates**, and a **color** (eventual other optional parameters can be set).
- They **return a reference** to the instance image, so they can be **pipelined**.
- They **clip** objects that are out of image bounds.

CImg<T> : Drawing functions

- All drawing functions **work the same way** : They need **the instance image**, **feature coordinates**, and a **color** (eventual other optional parameters can be set).
- They **return a reference** to the instance image, so they can be **pipelined**.
- They **clip** objects that are out of image bounds.
- **Ex** : `CImg& draw_line(int,int,int,int,T*)` ;

```
CImg<unsigned short> img(256,256,1,5); // hyperspectral image of ushort
unsigned short color[5] = { 0,8,16,24,32 }; // color used for the drawing
img.draw_line(x-2,y-2,x+2,y+2,color).
    draw_line(x-2,y+2,x+2,y-2,color).
    draw_circle(x+10,y+10,5,color);
```


CImg<T> : Drawing functions

- All drawing functions **work the same way** : They need **the instance image**, **feature coordinates**, and a **color** (eventual other optional parameters can be set).
- They **return a reference** to the instance image, so they can be **pipelined**.
- They **clip** objects that are out of image bounds.

- **Ex** : `CImg& draw_line(int,int,int,int,T*)` ;

```
CImg<unsigned short> img(256,256,1,5); // hyperspectral image of ushort
unsigned short color[5] = { 0,8,16,24,32 }; // color used for the drawing
img.draw_line(x-2,y-2,x+2,y+2,color).
    draw_line(x-2,y+2,x+2,y-2,color).
    draw_circle(x+10,y+10,5,color);
```

- `CImg<T>::draw_object3d()` can draw 3D objects (mini Open-GL!)

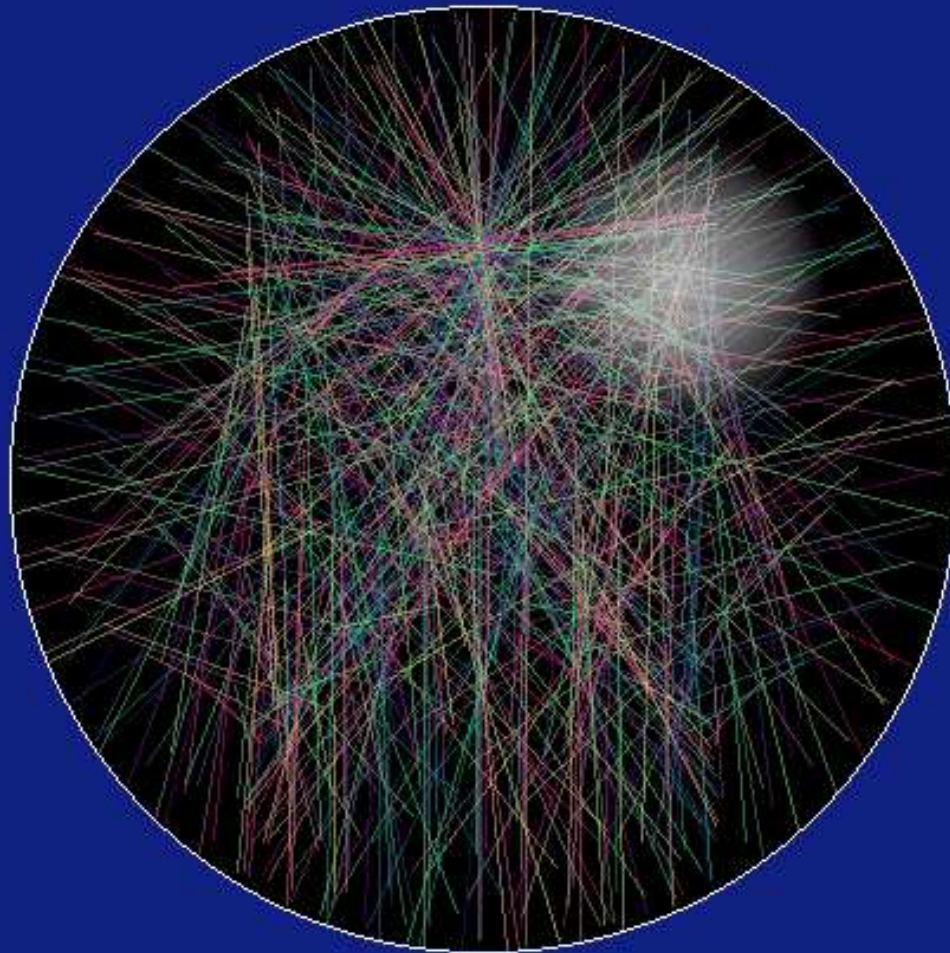
CImg<T> : Plasma ball (source code)

- The following code draws a “plasma ball” from scratch :

```
CImg<unsigned char> img(512,512,1,3,0);
for (float alpha=0, beta=0; beta<100; alpha+=0.21f, beta+=0.18f) {
    const float
        ca = std::cos(alpha), cb = std::cos(beta),
        sa = std::sin(alpha), sb = std::sin(beta);
    img.draw_line(256+200*ca*sa,256+200*cb*sa,
                  256+200*sa*sb,256+200*sb*ca,
                  CImg<unsigned char>::vector(alpha*256,beta*256,128).
                  ptr(),0.5f);
}

const unsigned char white[3] = { 255,255,255 }, blue[3] = { 16,32,128 };
img.draw_circle(256,256,200,white,1.0f,~0U).draw_fill(0,0,blue);
for (int radius = 60; radius>0; --radius)
    img.draw_circle(340,172,radius,white,0.02f);
```

CImg<T> : Plasma ball (result)

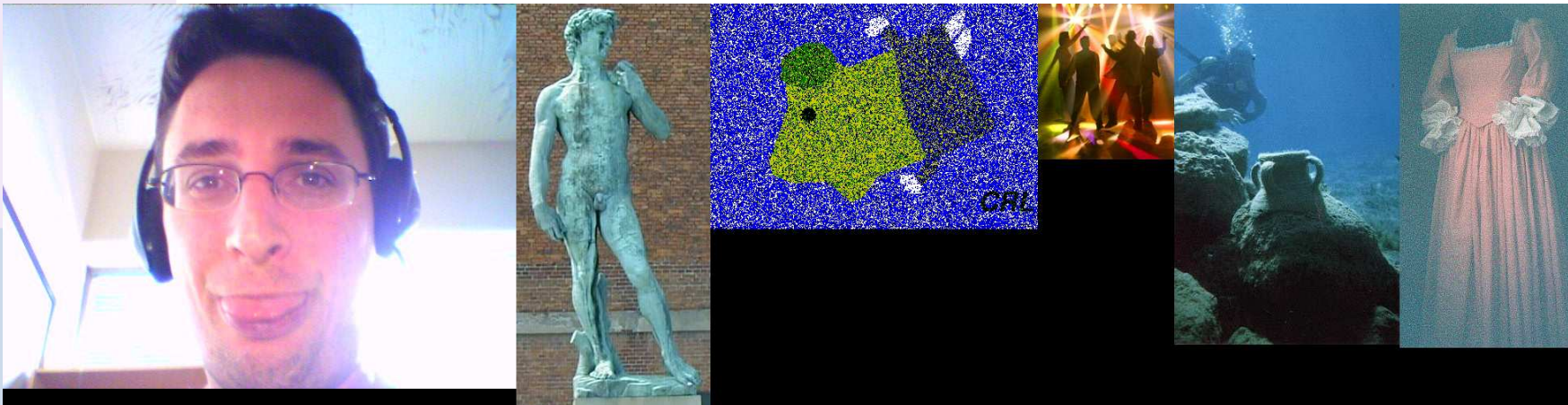


Outline - PART I of II : CImg Library Overview

- **Context** : Image Processing with C++.
 - Aim and targeted audience.
 - Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.
 - Drawing things on images.
- ⇒ **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

CImgList<T> : Overview

- A `CImgList<T>` represents an **array of `CImg<T>`**.
- Useful to handle **a sequence or a collection of images**.
- Here also, the memory is **not shared** by other `CImgList<T>` or `CImg<T>` objects.
- Looks like a `std::vector<CImg<T> >`, specialized for image processing.
- Can be used as **a flexible and ordered set of images**.



Outline - PART I of II : CImg Library Overview

- **Context** : Image Processing with C++.
 - Aim and targeted audience.
 - Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.
 - Drawing things on images.
- **CImgList<T>** : Image collection manipulation.
 - ⇒ **Basic manipulation functions.**
- **CImgDisplay** : Image display and user interaction.
 - Displaying images in windows.

CImgList<T> : Main functions

```
// Create a list of 20 color images 100x100.  
CImgList<float> list(20,100,100,1,3);  
  
// Insert two images at the end of the list.  
list.insert(CImg<float>(50,50));  
list.insert(CImg<unsigned char>('milla.ppm'));  
  
// Remove the second image from the list.  
list.remove(1);  
  
// Resize the 5th image of the list.  
CImg<float> &ref = list[4];  
ref.resize(50,50);
```

- Lists can be saved (and loaded) as **.cimg** files (simple binary format with ascii header).

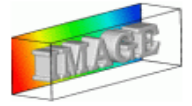
CImgList<T> : .cimg files

- Functions `CImgList<T>::load_cimg()` and `CImgList<T>::save_cimg()` allow to load/save portions of .cimg image files.
- Single images (`CImg<T>` class) can be also loaded/saved into .cimg files.
- Useful to work with big image files, video sequences or image collections.

Outline - PART I of II : CImg Library Overview

- **Context** : Image Processing with C++.
 - Aim and targeted audience.
 - Why considering The CImg Library ?
 - **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.
 - Drawing things on images.
 - **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- ⇒ **CImgDisplay** : Image display and user interaction.
- Displaying images in windows.

Outline - PART I of II : CImg Library Overview



- **Context** : Image Processing with C++.
 - Aim and targeted audience.
 - Why considering The CImg Library ?
- **CImg<T>** : A class for image manipulation.
 - Image construction, data access, math operators.
 - Basic image transformations.
 - Drawing things on images.
- **CImgList<T>** : Image collection manipulation.
 - Basic manipulation functions.
- **CImgDisplay** : Image display and user interaction.
 - ⇒ **Displaying images in windows.**

CImgDisplay : Overview

- A `CImgDisplay` allows to **display** `CImg<T>` or `CImg1<T>` instances in a window, and **can handle user events** that may happen in this window (mouse, keyboard, ...)

CImgDisplay : Overview

- A `CImgDisplay` allows to **display** `CImg<T>` or `CImgL<T>` instances in a window, and **can handle user events** that may happen in this window (mouse, keyboard, ...)
- The construction of a `CImgDisplay` **opens a window**.

CImgDisplay : Overview

- A `CImgDisplay` allows to **display** `CImg<T>` or `CImgL<T>` instances in a window, and **can handle user events** that may happen in this window (mouse, keyboard, ...)
- The construction of a `CImgDisplay` **opens a window**.
- The destruction of a `CImgDisplay` **closes the corresponding window**.

CImgDisplay : Overview

- A `CImgDisplay` allows to **display** `CImg<T>` or `CImgL<T>` instances in a window, and **can handle user events** that may happen in this window (mouse, keyboard, ...)
- The construction of a `CImgDisplay` **opens a window**.
- The destruction of a `CImgDisplay` **closes the corresponding window**.
- The display of an image in a `CImgDisplay` is done by a call to the `CImgDisplay::display()` function.

CImgDisplay : Overview

- A `CImgDisplay` allows to **display** `CImg<T>` or `CImgList<T>` instances in a window, and **can handle user events** that may happen in this window (mouse, keyboard, ...)
- The construction of a `CImgDisplay` **opens a window**.
- The destruction of a `CImgDisplay` **closes the corresponding window**.
- The display of an image in a `CImgDisplay` is done by a call to the `CImgDisplay::display()` function.
- A `CImgDisplay` has its **own pixel buffer**. It does not store any references to the `CImg<T>` or `CImgList<T>` passed at the last call to `CImgDisplay::display()`.

CImgDisplay : Handling events

- When opening the window, an **event-handling thread** is created.
- This thread automatically updates volatile fields of the **CImgDisplay** instance, when events occur in the corresponding window :
 - **Mouse events** : `mouse_x()`, `mouse_y()` and `button()`.
 - **Keyboard event** : `key()`.
 - **Window events** : `is_resized()`, `is_closed()` and `is_moved()`.
- Only **one thread** is used to handle display events of all opened **CImgDisplay**.
- This thread is killed **when the last display window is destroyed**.
- The **CImgDisplay** class is fully coded **both for GDI32 and X11** graphics libraries.
- Display automatically handles **image normalization** to display float-valued images correctly.

CImgDisplay : Useful functions

- Construction :

```
CImgDisplay disp1(img, 'My first display');  
CImgDisplay disp2(640,400, 'My second display');
```

- Display/Refresh image:

```
img.display(disp);  
disp.display(img);
```

- Handle events :

```
if (disp.key()==cimg::keyQ) { ... }  
if (disp.is_resized()) disp.resize();  
if (disp.mouse_x()>20 && disp.mouse_y()<40) { ... }  
disp.wait();
```

- Temporize (for animations) : `disp.wait(20);`

CImgDisplay : Example of using CImgDisplay

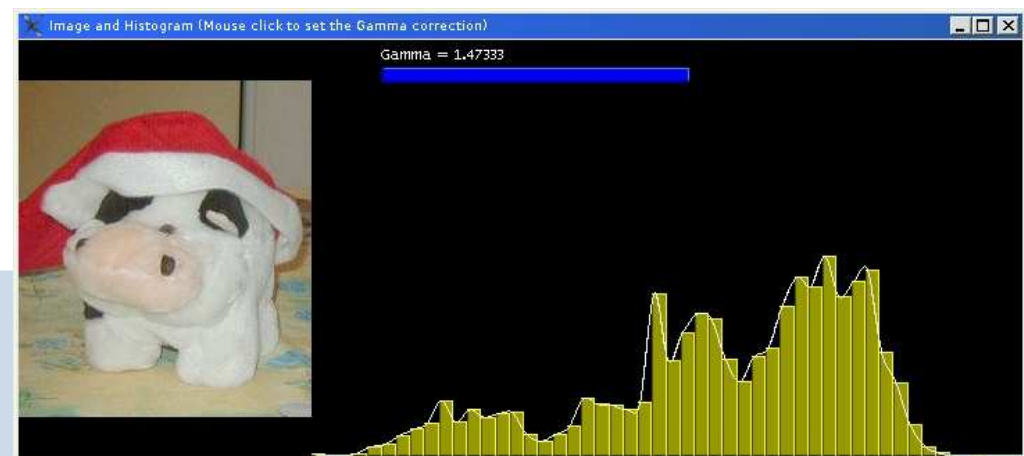
```
#include "CImg.h"
using namespace cimg_library;
int main() {
    CImgDisplay disp(256,256,"My Display");
    while (!disp.is_closed()) {
        if (disp.button&1) {
            const int x = disp.mouse_x(), y = disp.mouse_y();
            CImg<unsigned char> img(disp.width(),disp.height());
            unsigned char col[1] = {255};
            img.fill(0).draw_circle(x,y,40,col).display(disp);
        }
        if (disp.button()&2) disp.resize(-90,-90);
        if (disp.is_resized()) disp.resize();
        disp.wait();
    }
    return 0;
}
```

A more complete example of using CImg<T> (14 C++ lines)

```
CImg<> img = CImg<>("img/milla.ppm").normalize(0,1);
CImg<unsigned char> visu(img*255, CImg<unsigned char>(512,300,1,3,0));
const unsigned char yellow[3] = {255,255,0}, blue[3]={0,155,255}, blue2[3]={0,0,255}, blue3[3]={0,0,155},
    white[3]={255,255,255};
CImgDisplay disp(visu,"Image and Histogram (Mouse click to set the Gamma correction)",0);
for (double gamma=1;!disp.closed() && disp.key()!=cimg::keyQ && disp.key()!=cimg::keyESC; ) {
    cimg_forXYZC(visu[0],x,y,z,k) visu[0](x,y,z,k) = (unsigned char)(pow((double)img(x,y,z,k),1.0/gamma)*256);
    const CImg<> hist = visu[0].get_histogram(50,0,255);
    visu[1].fill(0).draw_text(50,5,"Gamma = %g",white,NULL,1,gamma).
    draw_graph(hist,yellow,1,20000,0).draw_graph(hist,white,2,20000,0);
    const int xb = (int)(50+gamma*150);
    visu[1].draw_rectangle(51,21,xb-1,29,blue2).draw_rectangle(50,20,xb,20,blue).draw_rectangle(xb,20,xb,30,blue);
    visu[1].draw_rectangle(xb,30,50,29,blue3).draw_rectangle(50,20,51,30,blue3);
    if (disp.button() && disp.mouse_x()>=img.width()+50 && disp.mouse_x()<=img.width()+450) gamma = (disp.mouse_x()-img.width()-50)/10;
    disp.resize(disp).display(visu).wait();
}
```

Result :

Histogram manipulation and gamma correction (example from example file `CImg_demo.cpp`)



PART II of II

⇒ **Image Filtering : Goal and principle.**

- Convolution - Correlation.
- Morphomaths - Median Filter.
- Anisotropic smoothing.
- Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.
- The plug-in mechanism.
- Dealing with 3D objects.
- Shared images.

Context : Image Filtering

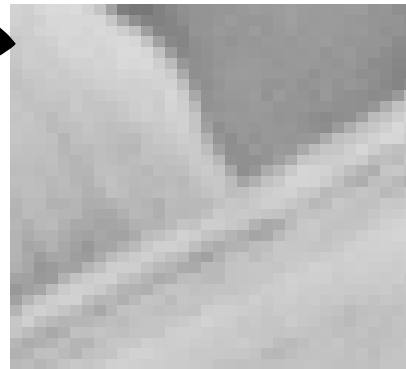
- **Image filtering** is one of the most common operations done on images in order to retrieve informations.

- **Image filtering** is one of the most common operations done on images in order to retrieve informations.
- Filtering is needed in the following cases :
 - Compute **image derivatives** (gradient) $\nabla I = \left(\frac{\partial I}{\partial x} \quad \frac{\partial I}{\partial y} \right)^T$.
 - **Noise removal** : Gaussian or Median filtering.
 - **Edge enhancement & Deconvolution** : Sharpen masks, Fourier Transform.
 - **Shape analysis** : Morphomath filters (erosion, dilatation,..)
 - ...

- **Image filtering** is one of the most common operations done on images in order to retrieve informations.
- Filtering is needed in the following cases :
 - Compute **image derivatives** (gradient) $\nabla I = \left(\frac{\partial I}{\partial x} \quad \frac{\partial I}{\partial y} \right)^T$.
 - **Noise removal** : Gaussian or Median filtering.
 - **Edge enhancement & Deconvolution** : Sharpen masks, Fourier Transform.
 - **Shape analysis** : Morphomath filters (erosion, dilatation,..)
 - ...
- A filtering process generally needs **the image** and a **mask** (a.k.a **kernel** or **structuring element**).

How filtering works ?

- For each point $p \in \Omega$ of the image I , consider its neighborhood $\mathcal{N}_I(p)$ and combine it with a user-defined mask M .



$$\bullet \begin{bmatrix} -2 & 3 & \dots & 7 & 1 \\ 1 & \ddots & \vdots & \ddots & -3 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ -4 & \ddots & \vdots & \ddots & 6 \\ 1 & -2 & \dots & 8 & -5 \end{bmatrix}$$

- Neighborhood $\mathcal{N}_I(p)$ and mask M have the same size.
- The operator \bullet may be **linear**, but not necessarily.
- The result of the filtering operation is the new value at p :

$$\forall p \in \Omega, \quad J(p) = \mathcal{N}_I(p) \bullet M$$

Filtering examples



(a) Original image



(b) Derivative along x



(c) Erosion

- Derivative obtained with $\bullet = *$ and $M = \begin{bmatrix} 0.5 & 0 & -0.5 \end{bmatrix}$
- Erosion obtained with $\bullet = \min()$.

- **Image Filtering** : Goal and principle.
 - ⇒ **Convolution - Correlation.**
 - Morphomaths - Median Filter.
 - Anisotropic smoothing.
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.
- The plug-in mechanism.
- Dealing with 3D objects.
- Shared images.

- Convolution and Correlation implements **linear filtering** ($\bullet = *$)

Convolution :
$$J(x, y) = \sum_i \sum_j I(x - i, y - j) M(i, j)$$

Correlation :
$$J(x, y) = \sum_i \sum_j I(x + i, y + j) M(i, j)$$

- `CImg<T>::get_convolve()`, `CImg<T>::convolve()` and `CImg<T>::get_correlate()`, `CImg<T>::correlate()`.
- Compute image derivative along the X-axis :

```
CImg<> img('toto.jpg');  
CImg<> mask = CImg<>(3,1).fill(0.5,0,-0.5);  
img.convolve(mask);
```

Linear filtering (2)

- You can set the **border condition** in `convolve()` and `correlate()`
- Common linear filters are already implemented :
 - Gaussian kernel for **image smoothing** :
`CImg<T>::get_blur()` and `CImg<T>::blur()`.
 - **Image derivatives** :
`CImg<T>::get_gradient("xy")` and `CImg<T>::get_gradient("xyz")`.

⇒ **More faster versions** than using the `CImg<T>::convolve()` function !



Blur an image with a Gaussian kernel with $\sigma = 10$.

Using `CImg<T>::convolve()` : 1129 ms.

Using `CImg<T>::blur()` : 7 ms.

Linear filtering (3)

- When mask size is big, you can efficiently convolve the image by a multiplication in the Fourier domain.
- `CImg<T>::get_FFT()` returns a `CImgList<T>` with the real and imaginary part of the FT.
- `CImg<T>::get_FFT(true)` returns a `CImgList<T>` with the real and imaginary part of the inverse FT.



Outline - PART II of II : More insights

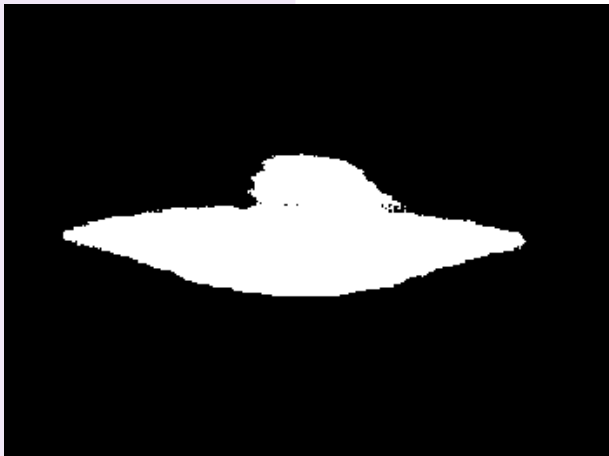
- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - ⇒ **Morphomaths - Median Filter.**
 - Anisotropic smoothing.
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.
- The plug-in mechanism.
- Dealing with 3D objects.
- Shared images.

- Nonlinear filters.
- **Erosion** : Keep the minimum value in the image neighborhood having the same shape than the structuring element mask.

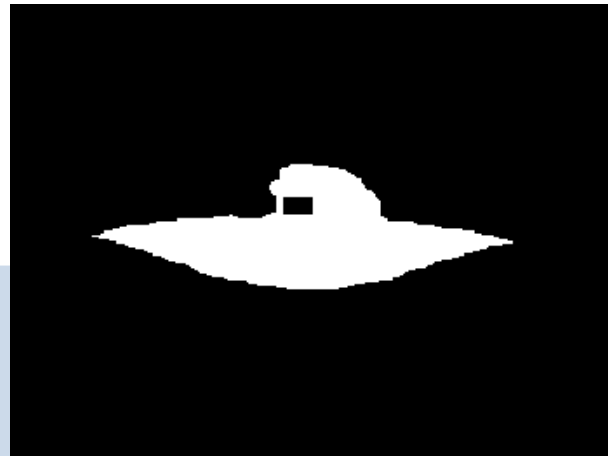
`CImg<T>::erode()` and `CImg<T>::get_erode()`.

- **Dilatation** : Keep the maximum value in the image neighborhood having the same shape than the structuring element mask.

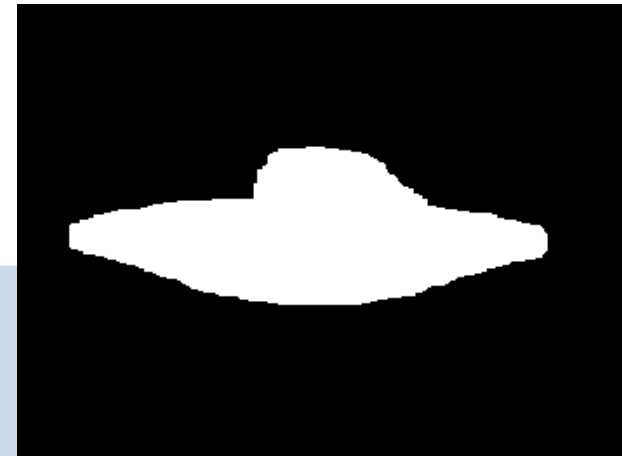
`CImg<T>::dilate()` and `CImg<T>::get_dilate()`.



(a) Original image



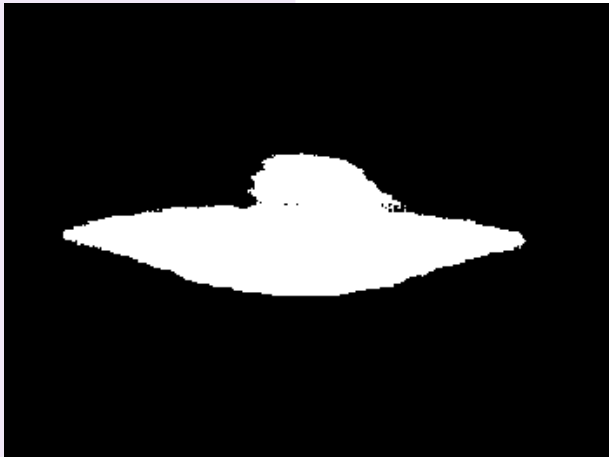
(b) Erosion by a 10×10 kernel



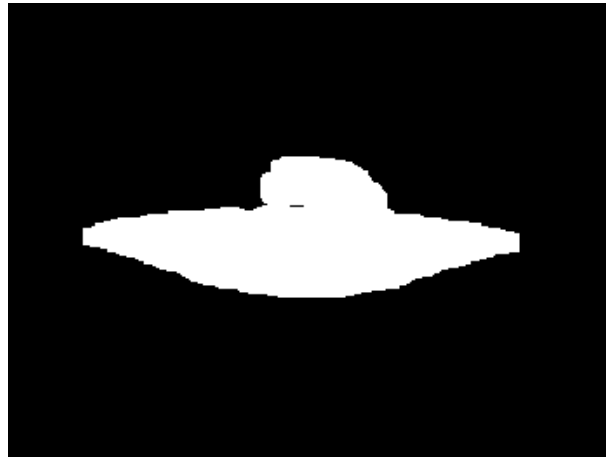
(b) Dilatation by a 10×10 kernel

Morphomaths (2)

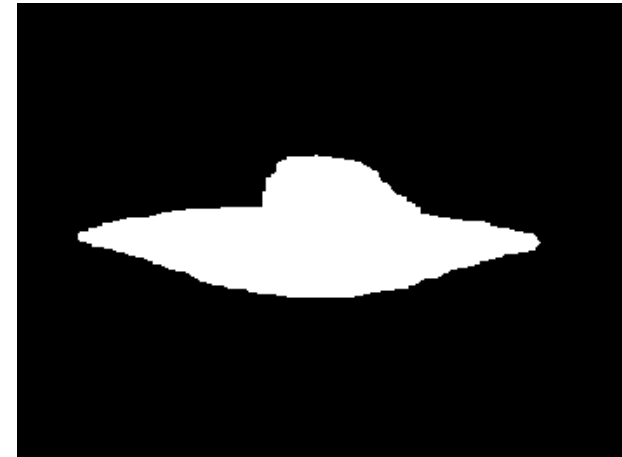
- **Opening** : Erode, then dilate :
`img.erode(10).dilate(10);`
- **Closing** : Dilate, then erode :
`img.dilate(10).erode(10);`



(a) Original image



(b) Opening by a 10×10 kernel



(b) Closing by a 10×10 kernel

Median filtering

- **Nonlinear filter** : Keep the median value in the image neighborhood having the same shape than the mask.
- Functions `CImg<T>::get_blur_median()` and `CImg<T>::blur_median()`.
- Near optimal to remove Salt&Pepper noise.



- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - Morphomaths - Median Filter.
 - ⇒ **Anisotropic smoothing.**
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.
- The plug-in mechanism.
- Dealing with 3D objects.
- Shared images.

Anisotropic smoothing

- Non-linear edge-directed diffusion, very optimized PDE-based algorithm.
- Very efficient in removing Gaussian noise, or other additive noise.
- Able to work on $2D$ and $3D$ images.
- Function `CImg<T>::blur_anisotropic()`.
- A lot of applications : Image denoising, reconstruction, resizing.

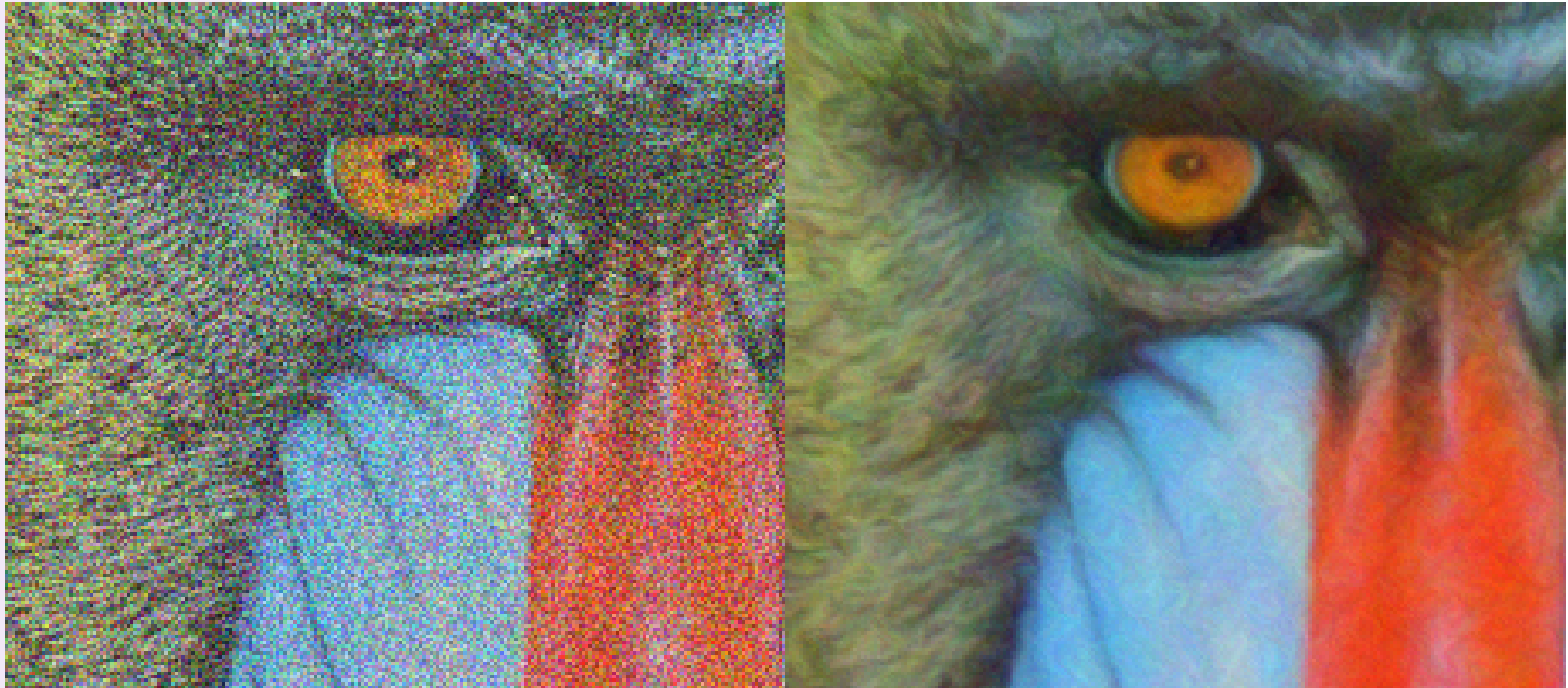
- `CImg<T>::blur_anisotropic()` implements the following diffusion PDE :

$$\forall i = 1, \dots, n, \quad \frac{\partial I_i}{\partial t} = \text{trace}(\mathbf{T}\mathbf{H}_i) + \frac{2}{\pi} \nabla I_i^T \int_{\alpha=0}^{\pi} \mathbf{J}_{\sqrt{\mathbf{T}}a_\alpha} \sqrt{\mathbf{T}}a_\alpha d\alpha$$

$$\text{where } \mathbf{J}_w = \begin{pmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{pmatrix} \quad \text{and} \quad \mathbf{H}_i = \begin{pmatrix} \frac{\partial^2 I_i}{\partial x^2} & \frac{\partial^2 I_i}{\partial x \partial y} \\ \frac{\partial^2 I_i}{\partial x \partial y} & \frac{\partial^2 I_i}{\partial y^2} \end{pmatrix}.$$

- Image smoothing while preserving discontinuities (edges).
- One of the advanced filtering tool in the CImg Library.

Application of `CImg<T>::blur_anisotropic()`



“Babouin” (détail) - 512x512 - (1 iter., 19s)

Application of `CImg<T>::blur_anisotropic()`



“Tunisie” - 555x367

Application of `CImg<T>::blur_anisotropic()`



“Tunisie” - 555x367 - (1 iter., 11s)

Application of `CImg<T>::blur_anisotropic()`



“Tunisie” - 555x367 - (1 iter., 11s)

Application of `CImg<T>::blur_anisotropic()`



“Bébé” - 400x375

Application of `CImg<T>::blur_anisotropic()`



“Bébé” - 400x375 - (2 iter, 5.8s)

Application of `CImg<T>::blur_anisotropic()`



“Bébé” - 400x375 - (2 iter, 5.8s)

Application of `CImg<T>::blur_anisotropic()`



“Van Gogh”

Application of `CImg<T>::blur_anisotropic()`



“Van Gogh” - (1 iter, 5.122s).

Application of `CImg<T>::blur_anisotropic()`



“Fleurs” (JPEG, 10% quality).

Application of `CImg<T>::blur_anisotropic()`



“Corail” (1 iter.)

Application : Image Inpainting



“Bird”, original color image.

Application : Image Inpainting



“Bird”, inpainting mask definition.

Application : Image Inpainting



“Bird”, inpainted with our PDE.

Application : Image Inpainting



“Chloé au zoo”, original color image.

Application : Image Inpainting



“Chloé au zoo”, inpainting mask definition.

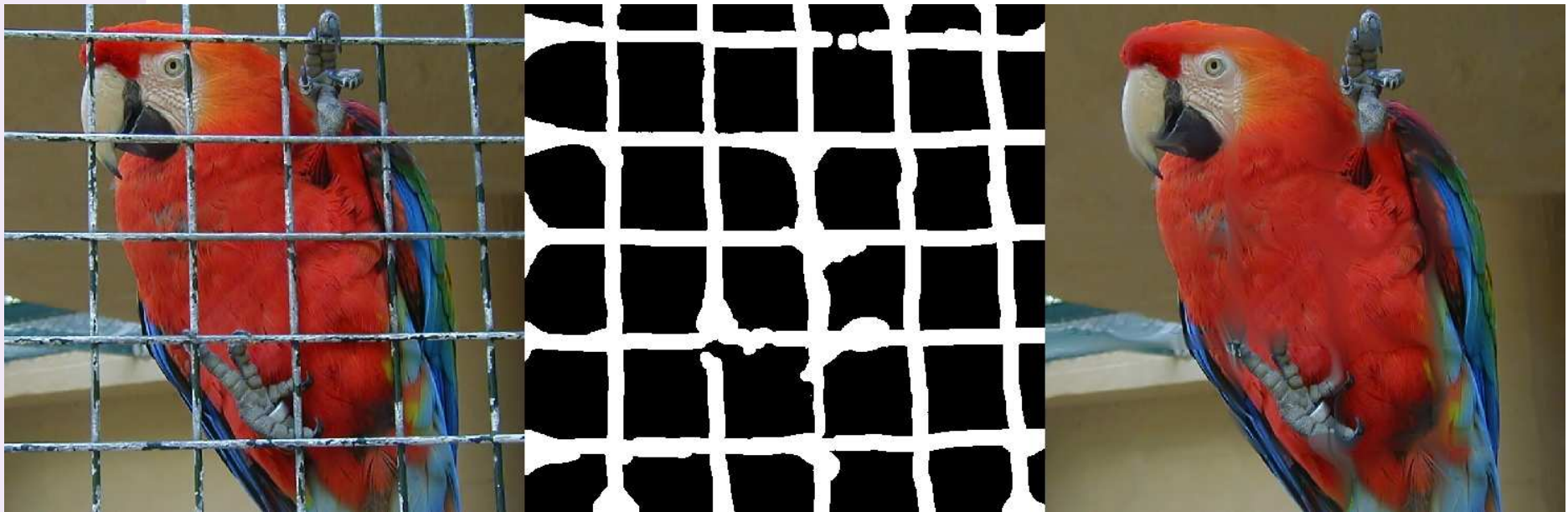
Application : Image Inpainting



“Chloé au zoo”, inpainted with our PDE.

Application : Image Inpainting and Reconstruction

"Parrot"
500x500
(200 iter.,
4m11s)



"Owl"
320x246
(10 iter., 1m01s)



Application : Image Resizing



(c) Details from the image resized by bicubic interpolation.



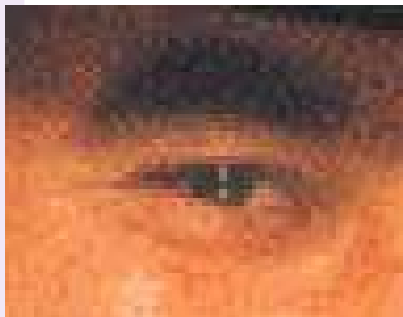
(d) Details from the image resized by a non-linear regularization PDE.

Application : Image Resizing

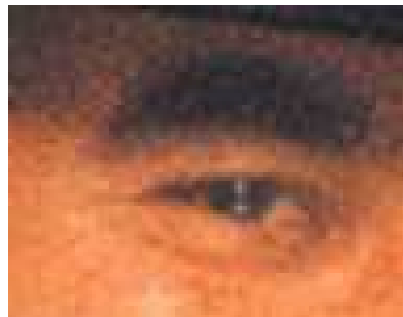


(a) Original

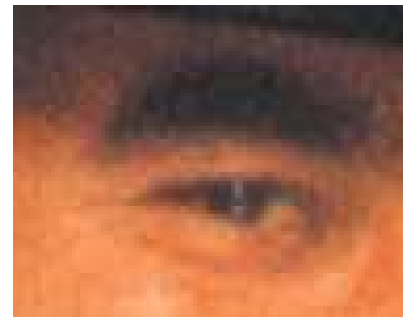
color image



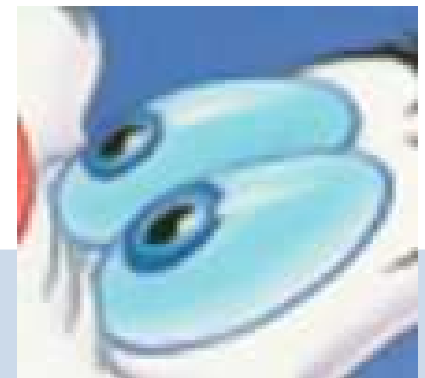
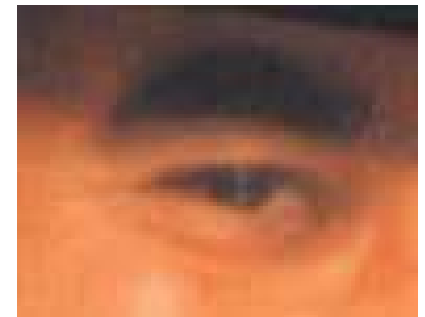
(b) Bloc Interpolation



(c) Linear Interpolation



(d) Bicubic Interpolation



(e) PDE/LIC Interpolation

- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - Morphomaths - Median Filter.
 - Anisotropic smoothing.

⇒ **Other related functions.**
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.
- The plug-in mechanism.
- Dealing with 3D objects.
- Shared images.

Adding noise to images

- `CImg<T>::noise()` and `CImg<T>::get_noise()`.
- Can add different kind of noise to the image with specified distribution : Uniform, Gaussian, Poisson, Salt&Pepper.
- One parameter that set the amount of noise added.



Retrieving image similarity

- Two indices defined to measure “distance” between two images $I1$ and $I2$: **MSE** and **PSNR**.
- MSE, Mean Squared Error : `CImg<T>::MSE(img1, img2)`.

$$\text{MSE}(I1, I2) = \frac{\sum_{p \in \Omega} (I1_{(p)} - I2_{(p)})^2}{\text{card}(\Omega)}$$

The lowest the MSE is, the closest the images $I1$ and $I2$ are.

- PSNR, Peak Signal to Noise Ratio : `CImg<T>::PSNR(img1, img2)`.

$$\text{PSNR}(I1, I2) = 20 \log_{10} \left(\frac{M}{\sqrt{\text{MSE}(I1, I2)}} \right)$$

where M is the maximum value of $I1$ and $I2$.

Filtering in Clmg : Conclusions

- A lot of useful functions that does the common image filtering tasks.
- Linear and Nonlinear filters.
- But what if we want to define to following filter ???

$$\forall p \in \Omega, \quad J(x, y) = \sum_{i,j} \text{mod}(I(x - i, y - j), M(i, j))$$

⇒ There are smart ways to define your own nonlinear filters, using neighborhood loops.

- **Image Filtering** : Goal and principle.

- Convolution - Correlation.
- Morphomaths - Median Filter.
- Anisotropic smoothing.
- Other related functions.

⇒ **Image Loops** : Using predefined macros.

- Simple loops.
- Neighborhood loops.
- The plug-in mechanism.
- Dealing with 3D objects.
- Shared images.

- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - Morphomaths - Median Filter.
 - Anisotropic smoothing.
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - ⇒ **Simple loops.**
 - Neighborhood loops.
 - The plug-in mechanism.
 - Dealing with 3D objects.
 - Shared images.

- Image loops are very useful in image processing, to scan pixel values iteratively.
- Cimg define **macros** that replace the corresponding `for(...;...;...)` instructions.

```
cimg_forX(img,x)  ⇔  for (int x=0; x<img.width(); x++)  
cimg_forY(img,y)  ⇔  for (int y=0; y<img.height(); y++)  
cimg_forZ(img,z)  ⇔  for (int z=0; z<img.depth(); z++)  
cimg_forC(img,c)  ⇔  for (int c=0; c<img.spectrum(); c++)
```


- Image loops are very useful in image processing, to scan pixel values iteratively.
- Cimg define **macros** that replace the corresponding **for(...;...;...)** instructions.

```
cimg_forX(img,x)  ⇔  for (int x=0; x<img.width(); x++)  
cimg_forY(img,y)  ⇔  for (int y=0; y<img.height(); y++)  
cimg_forZ(img,z)  ⇔  for (int z=0; z<img.depth(); z++)  
cimg_forC(img,c)  ⇔  for (int c=0; c<img.spectrum(); c++)
```

- Cimg also defines :

```
cimg_forXY(img,x,y) ⇔ cimg_forY(img,y) cimg_forX(img,x)  
cimg_forXYZ(img,x,y,z) ⇔ cimg_forZ(img,z) cimg_forXY(img,x,y)  
cimg_forXYZC(img,x,y,z,c) ⇔ cimg_forC(img,c) cimg_forXYZ(img,x,y,z)
```

Simple loops (2)

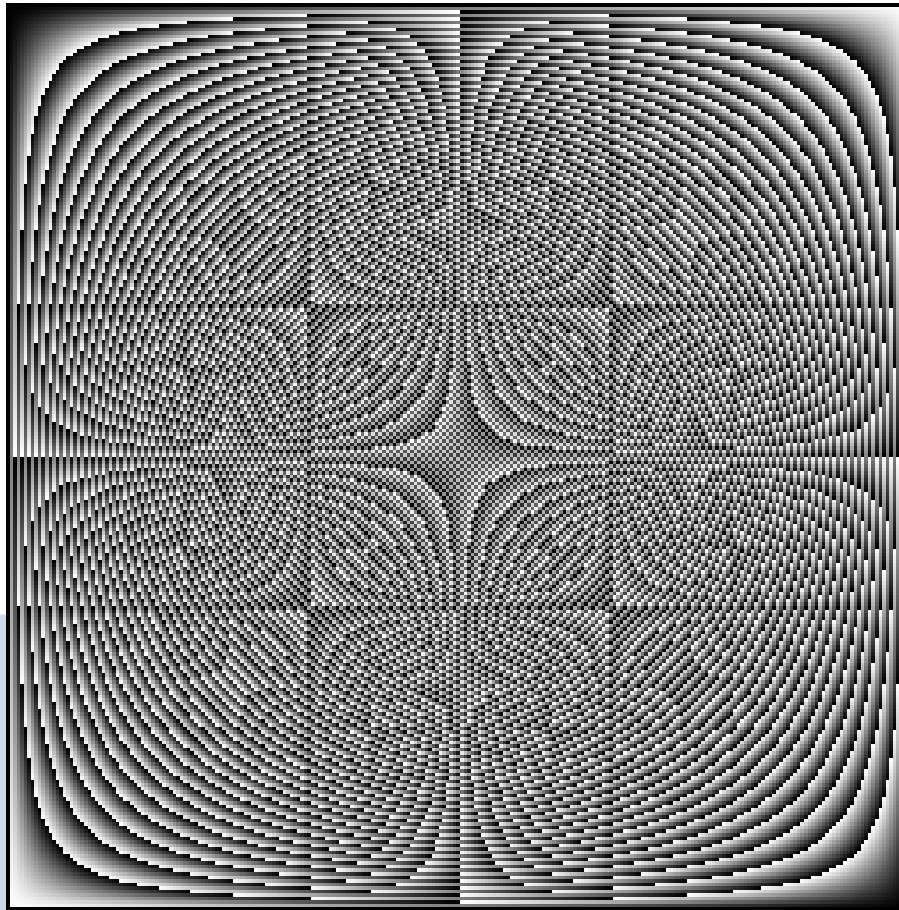
- These loops lead to natural code for filling an image with values :

```
CImg<unsigned char> img(256,256);  
cimg_forXY(img,x,y) { img(x,y) = (x*y)%256; }
```

Simple loops (2)

- These loops lead to natural code for filling an image with values :

```
CImg<unsigned char> img(256,256);  
cimg_forXY(img,x,y) { img(x,y) = (x*y)%256; }
```

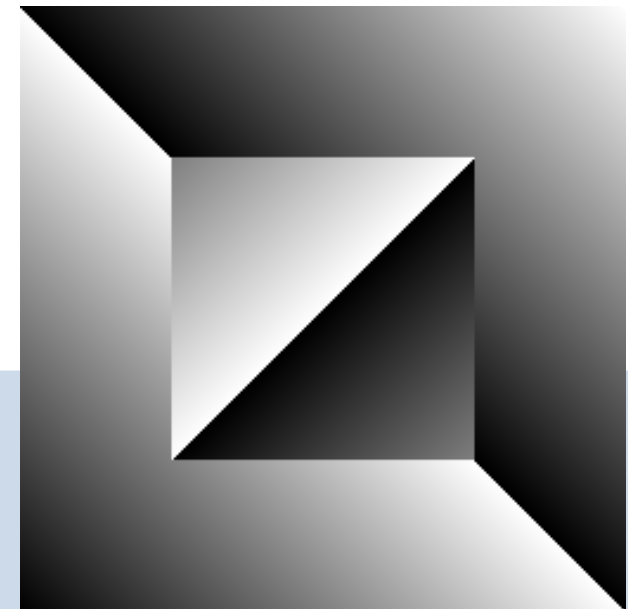


Interior and Border loops

- Slight variants of the previous loops, allowing to consider **only interior or image borders**.
- An extra parameter n telling about **the size of the image border**.

`cimg_for_insideXY(img,x,y,n)` and `cimg_for_borderXY(img,x,y,n)` (same for 3D volumetric images).

```
CImg<unsigned char> img(256,256);  
cimg_for_insideXY(img,x,y,64) img(x,y) = x+y;  
cimg_for_borderXY(img,x,y,64) img(x,y) = x-y;
```



- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - Morphomaths - Median Filter.
 - Anisotropic smoothing.
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - ⇒ **Neighborhood loops.**
- The plug-in mechanism.
- Dealing with 3D objects.
- Shared images.

Neighborhood-based loops

- Very powerful loops, allow to loop **an entire neighborhood** over an image.
- From 2×2 to 5×5 for $2D$ neighborhood.
- From $2 \times 2 \times 2$ to $3 \times 3 \times 3$ for $3D$ neighborhood.
- Border condition : **Nearest-neighbor**.
- Need an external neighborhood variable declaration.
- Allow to write **very small**, **clear** and **optimized** code.

Neighborhood-based loops : 3×3 example

- Neighborhood declaration :

```
CImg_3x3(I,float).
```

Neighborhood-based loops : 3×3 example

- Neighborhood declaration :

`CImg_3x3(I, float).`

- Actually, the line above defines 9 different variables, named :

Ipp	Icp	Inp
Ipc	Icc	Inc
Ipn	Icn	Inn

where $p = \text{previous}$, $c = \text{current}$, $n = \text{next}$.

Neighborhood-based loops : 3×3 example

- Neighborhood declaration :

`CImg_3x3(I,float).`

- Actually, the line above defines 9 different variables, named :

Ipp	Icp	Inp
Ipc	Icc	Inc
Ipn	Icn	Inn

where *p* = *previous*, *c* = *current*, *n* = *next*.

- Using a `cimg_for3x3()` automatically updates the neighborhood with the correct values.

```
cimg_for3x3(img,x,y,0,0,I,float) {  
    .. Here, Ipp, Icp, ... Icn, Inn are updated ...  
}
```

Neighborhood-based loops

- Example of use : Compute the **gradient norm with one loop**.

```
CImg<float> img('milla.jpg'), dest(img);  
CImg_3x3(I,float);  
cimg_forC(img,v) cimg_for3x3(img,x,y,0,v,I,float) {  
    const float ix = (Inc-Ipc)/2, iy = (Icn-Icp)/2;  
    dest(x,y) = std::sqrt(ix*ix+iy*iy);  
}
```



Example : Modulo Filtering

- What if we want to define the following filter ???

$$\forall p \in \Omega, \quad J(x, y) = \sum_{i,j} \text{mod}(I(x - i, y - j), M(i, j))$$

Example : Modulo Filtering

- What if we want to define the following filter ???

$$\forall p \in \Omega, \quad J(x, y) = \sum_{i, j} \text{mod}(I(x - i, y - j), M(i, j))$$

- Simple solution, using a 3x3 mask :

```
CImg<unsigned char> img('milla.jpg'), mask(3,3);  
CImg<> dest(img);  
CImg_3x3(I,float);  
cimg_forV(img,v) cimg_for3x3(img,x,y,0,v,I)  
    dest(x,y) = mask(0,0)%Ipp + mask(1,0)%Icp + mask(2,0)%Inp  
               + mask(0,1)%Ipc + mask(1,1)%Icc + mask(2,1)%Inc  
               + mask(0,2)%Ipn + mask(1,2)%Icn + mask(2,2)%Inn;  
}
```

- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - Morphomaths - Median Filter.
 - Anisotropic smoothing.
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.

⇒ **The plug-in mechanism.**

- Dealing with 3D objects.
- Shared images.

- Sometimes an user needs or defines **specific** functions, either very specialized or not generic enough.
- Not suitable to be integrated in the CImg Library, but interesting to share anyway.

- Sometimes an user needs or defines **specific** functions, either very specialized or not generic enough.
- Not suitable to be integrated in the CImg Library, but interesting to share anyway.

⇒ **Integration possible in CImg via the plug-ins mechanism.**

```
#define cimg_plugin 'my_plugin.h'  
#include 'CImg.h'  
using namespace cimg_library;  
  
int main() {  
    CImg<> img('milla.jpg');  
    img.my_wonderful_function();  
    return 0;  
}
```

- Plugin functions are directly added as member functions of the CImg class.

```
// File ‘‘my_plugin.h’’  
//-----  
CImg<T> my_wonderful_function() {  
    (*this)=(T)3.14f;  
    return *this;  
}
```


- Plugin functions **are directly added as member functions of the CImg class.**

```
// File 'my_plugin.h'  
//-----  
CImg<T> my_wonderful_function() {  
    (*this)=(T)3.14f;  
    return *this;  
}
```

- **Very flexible system**, implemented as easily as :

```
class CImg<T> {  
    ...  
    #ifdef cimg_plugin  
    #include cimg_plugin  
    #endif  
};
```

- Advantages :
 - Allow creations or modifications of existing functions by the user, without modifying the library source code.

- Advantages :

- Allow creations or modifications of existing functions by the user, without modifying the library source code.
- Allow to specialize the library according to the user's work.

- Advantages :

- Allow creations or modifications of existing functions by the user, without modifying the library source code.
- Allow to specialize the library according to the user's work.
- Allow an easy redistribution of useful functions as open source components.

⇒ A very good way to contribute to the library.

- Advantages :

- Allow creations or modifications of existing functions by the user, without modifying the library source code.
- Allow to specialize the library according to the user's work.
- Allow an easy redistribution of useful functions as open source components.

⇒ A very good way to contribute to the library.

- Existing plugins in the default CImg package :

- Located in the directory CImg/plugins/
- `cimg_matlab.h` : Provide code interface between CImg and Matlab images.
- `nlmeans.h` : Implementation of Non-Local Mean Filter (*Buades et al*).
- `noise_analysis.h` : Advanced statistics for noise estimation.
- `toolbox3d.h` : Functions to construct classical 3D meshes (cubes, sphere,...)

- Plug-ins variables :

- #define cimg_plugin : Add functions to the CImg<T> class.
- #define cimglist_plugin : Add functions to the CImgList<T> class.

- Using several plug-ins is possible : #define cimg_plugin ‘‘all_plugins.h’’.

```
// file ‘‘all_plugins.h’’  
#include ‘‘plugin1.h’’  
#include ‘‘plugin2.h’’  
#include ‘‘plugin3.h’’
```

⇒ With the plugin mechanism, CImg is a very open framework for image processing.

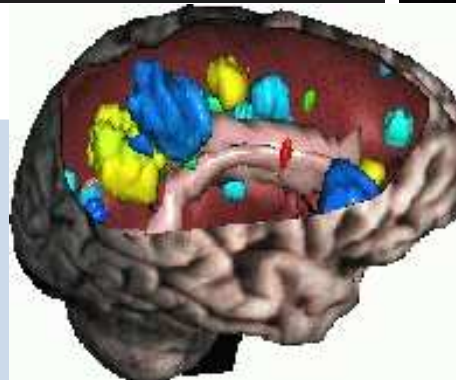
- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - Morphomaths - Median Filter.
 - Anisotropic smoothing.
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.
- The plug-in mechanism.

⇒ **Dealing with 3D objects.**

- Shared images.

3D Object Visualization : Context

- In a lot of image processing problems, one needs to **reconstruct 3D models** from raw image datasets.
 - 3D from stereo images/multiple cameras.
 - 3D surface reconstruction from volumetric MRI images.
 - 3D surface reconstruction from points clouds (3D scanner).

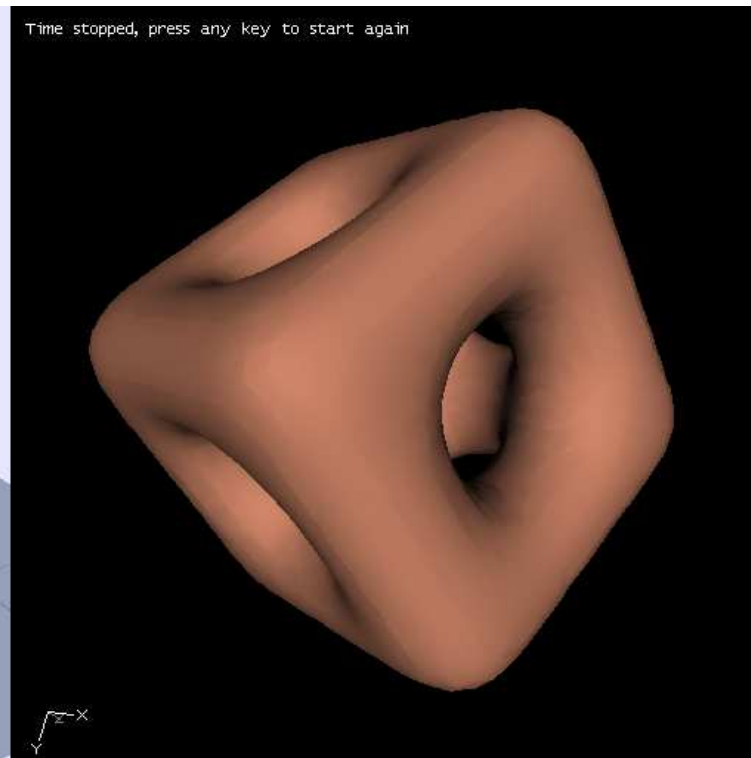
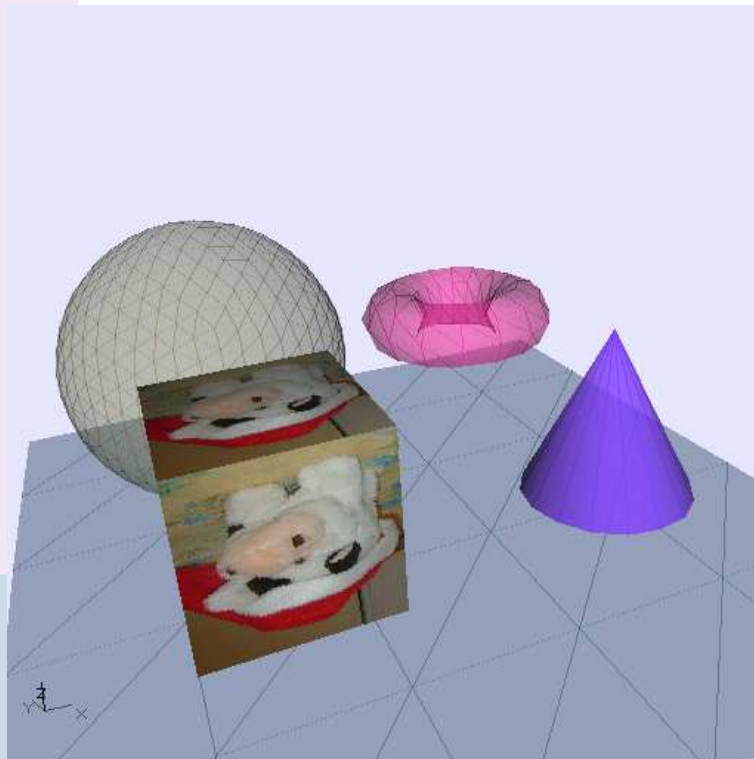


3D Object Visualization : Context

- ⇒ Basic and integrated 3D meshes visualization capabilities may be useful in any image processing library.
- ... but we don't want to replace complete 3D rendering libraries (OpenGL, Direct3D, VTK, ...).
 - CImg allows to visualize 3D objects for punctuals needs.
 - Can displays a set of 3D primitives (points, lines, triangles) with given opacity.
 - Can render objects with flat, gouraud or phong-like light models.
 - Contains an interactive display function to view the 3D object.
 - Texture mapping supported.
 - No multiple lights allowed.
 - No GPU acceleration.

3D Object Visualization : Live Demo

- Mean Curvature Flow.
- Image as a surface.
- Toolbox3D.



3D Object Visualization : How does it works ?

- CImg has a `CImg<T>::draw_*()` function that can draw a projection of a 3D object into a 2D image :

`CImg<T>::draw_object3d()`

3D Object Visualization : How does it works ?

- CImg has a `CImg<T>::draw_*()` function that can draw a projection of a 3D object into a 2D image :

```
CImg<T>::draw_object3d()
```

- High-level interactive 3D object display :

```
CImg<T>::display_object3d()
```

⇒ All 3D visualization capabilities of CImg are based on these two functions.

3D Object Visualization : How does it works ?

- CImg has a `CImg<T>::draw_*`() function that can draw a projection of a 3D object into a 2D image :

`CImg<T>::draw_object3d()`

- High-level interactive 3D object display :

`CImg<T>::display_object3d()`

⇒ All 3D visualization capabilities of CImg are based on these two functions.

- Needed parameters :

- A `CImgList<tp>` of 3D points coordinates (size M).
- A `CImgList<tf>` of primitives (size N).
- A `CImgList<T>` of colors/textures (size N).
- A `CImgList<to>` of opacities (size N) (optional parameter).

Display a house : building point list

```
CImgList<float> points(9,1,3,1,1,  
                      -50,-50,-50, // Point 0  
                      50,-50,-50,  // Point 1  
                      50,50,-50,   // Point 2  
                      -50,50,-50,  // Point 3  
                      -50,-50,50,  // Point 4  
                      50,-50,50,   // Point 5  
                      50,50,50,    // Point 6  
                      -50,50,50,   // Point 7  
                      0,-100,0);   // Point 8
```

⇒ List of 9 vectors (images 1x3) with specified coordinates.

Display a house : building primitives list

```
CImgList<unsigned int> primitives(6,1,4,1,1,  
                                0,1,5,4,  // Face 0  
                                3,7,6,2,  // Face 1  
                                1,2,6,5,  // Face 2  
                                0,4,7,3,  // Face 3  
                                0,3,2,1,  // Face 4  
                                4,5,6,7); // Face 5  
primitives.insert(CImgList<unsigned int>(4,1,2,1,1,  
                                           0,8,  // Segment 6  
                                           1,8,  // Segment 7  
                                           5,8,  // Segment 8  
                                           4,8))); // Segment 9
```

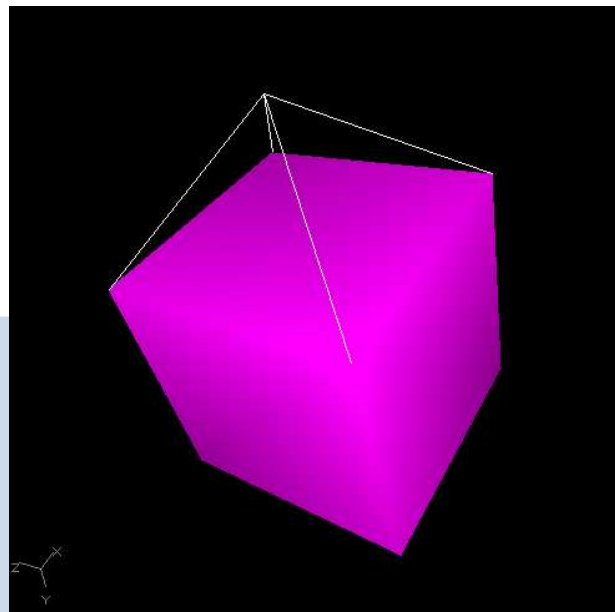
⇒ List of 10 vectors : 6 rectangle + 4 segments.

Display a house : building colors

```
CImgList<unsigned char> colors;  
colors.insert(6,CImg<unsigned char>::vector(255,0,255));  
colors.insert(4,CImg<unsigned char>::vector(255,255,255));
```

- Then,.... visualize.

```
CImg<unsigned char>(800,600,1,3).fill(0).  
display_object3d(points,primitives,colors);
```



Display a transparent house : setting primitive opacities

```
CImgList<float> opacities;  
opacities.insert(6,CImg<>::vector(0.5f));  
opacities.insert(4,CImg<>::vector(1.0f));
```

- Then,.... visualize.

```
CImg<unsigned char>(800,600,1,3).fill(0).  
display_object3d(points,primitives,colors,opacities);
```

- Other parameters of the 3D functions allow to set :
 - Light position, and ambient light intensity.
 - Camera position and focale.
 - Rendering type (Gouraud, Flat, ...)
 - Double/Single faces.

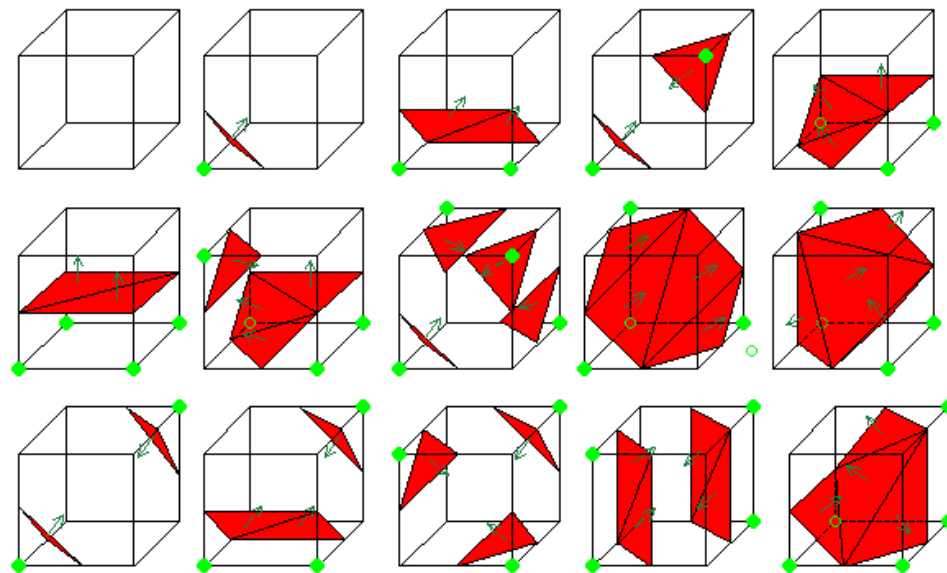
How to construct 3D meshes ?

- **Plugin** : `CImg/plugins/primitives.h` contains useful functions to retrieve classical meshes.

`CImg<T>::cube()`, `CImg<T>::sphere()`, `CImg<T>::cylinder()`, ...

- **Library functions** : `CImg<T>::marching_cubes()` and `CImg<T>::marching_squares()`.

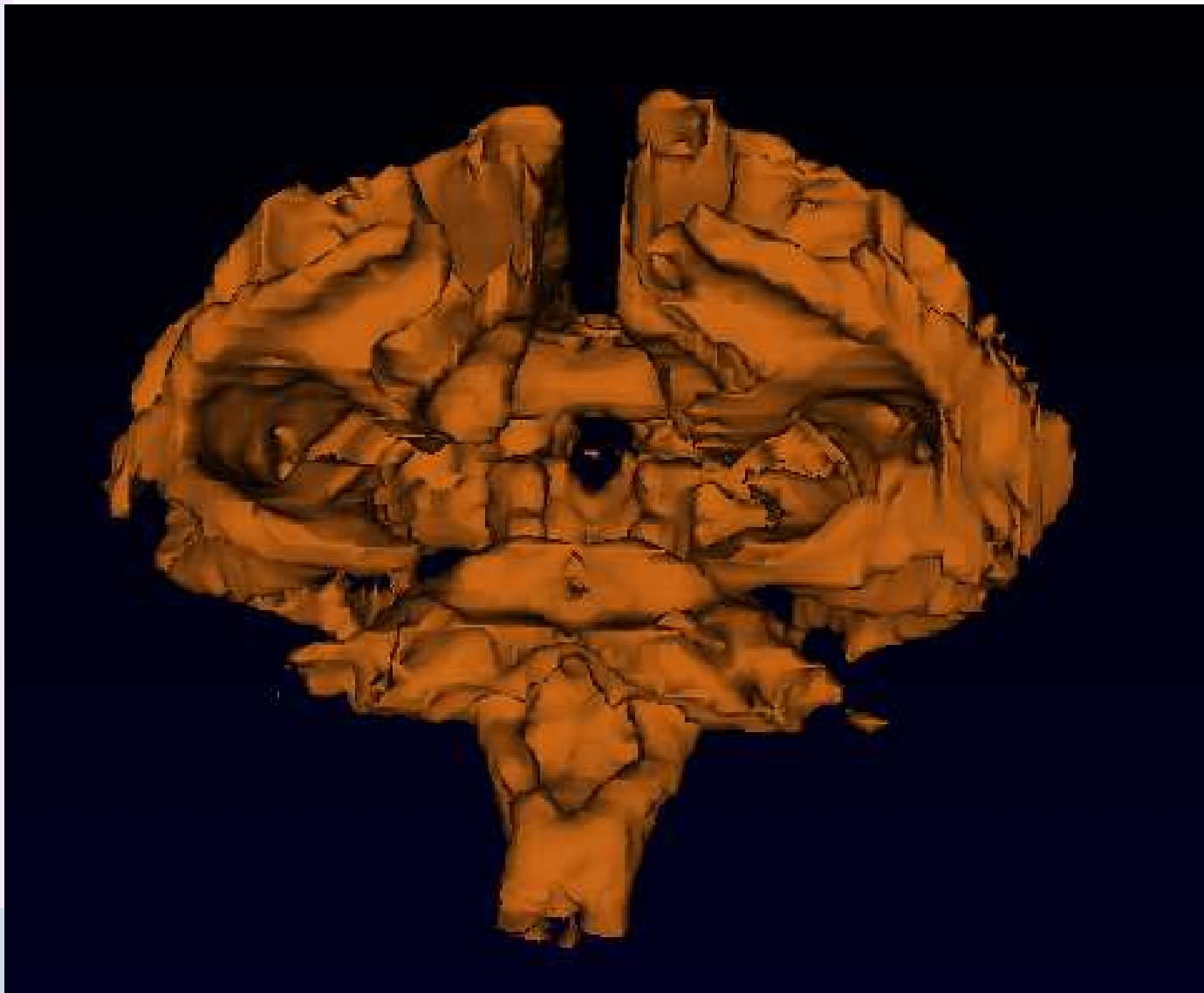
⇒ **Create meshes from implicit functions.**



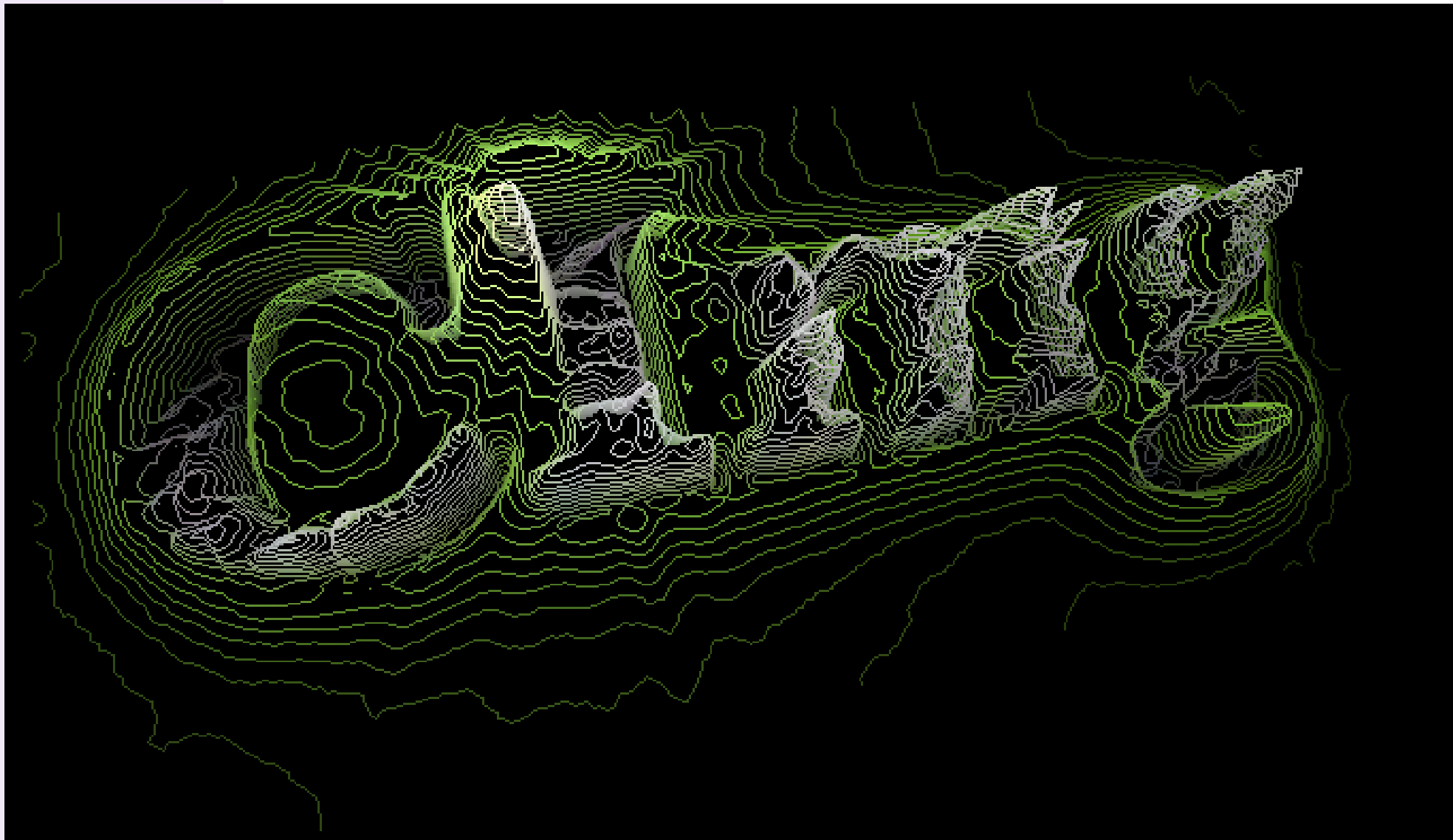
Example : Segmentation of the white matter from MRI images

```
CImg<> img('volumeMRI.inr');  
CImg<> region;  
float black[1]={0};  
img.draw_fill(X0,Y0,Z0,black,region,10.0f);  
(region*=-1).blur(1.0f).normalize(-1,1);  
  
CImgList<> points, faces;  
region.marching_cubes(0,points,faces);  
CImgList<unsigned char> colors;  
colors.insert(faces.size,CImg<unsigned char>::vector(200,100,20));  
  
CImg<unsigned char>(800,600,1,3).fill(0).  
display_object3d(points,faces,colors);
```

Example : Segmentation of the white matter from MRI images



Example : Isophotes with marching squares



- **Image Filtering** : Goal and principle.
 - Convolution - Correlation.
 - Morphomaths - Median Filter.
 - Anisotropic smoothing.
 - Other related functions.
- **Image Loops** : Using predefined macros.
 - Simple loops.
 - Neighborhood loops.
- The plug-in mechanism.
- Dealing with 3D objects.

⇒ **Shared images.**

- Two frequent cases with undesired image copies :

1. Sometimes, we want to pass **contiguous parts** of an image (but not all the image) to a function :

```
const CImg<> img('milla.jpg');  
CImgList<> RG = img.get_channels(0,1).get_split('v');
```

- Two frequent cases with undesired image copies :

1. Sometimes, we want to pass **contiguous parts** of an image (but not all the image) to a function :

```
const CImg<> img('milla.jpg');  
CImgList<> RG = img.get_channels(0,1).get_split('v');
```

2. ..Or, we want to modify **contiguous parts** of an image (but not all the image) :

```
CImg<> img('milla.jpg');  
img.draw_image(img.get_channel(1).blur(3),0,0,0,1);
```


- Two frequent cases with undesired image copies :

1. Sometimes, we want to pass **contiguous parts** of an image (but not all the image) to a function :

```
const CImg<> img('milla.jpg');  
CImgList<> RG = img.get_channels(0,1).get_split('v');
```

2. ..Or, we want to modify **contiguous parts** of an image (but not all the image) :

```
CImg<> img('milla.jpg');  
img.draw_image(img.get_channel(1).blur(3),0,0,0,1);
```

⇒ ... But we also want to avoid image copies for better performance...

- **Solution :** Use shared images :

1. Replace :

```
const CImg<> img('milla.jpg');  
CImgList<> RG = img.get_channels(0,1).get_split('v');
```

by

```
const CImg<> img('milla.jpg');  
CImgList<> RG = img.get_shared_channels(0,1).get_split('v');
```

- **Solution :** Using shared images :

2. Replace :

```
CImg<> img('milla.jpg');  
img.draw_image(img.get_channel(1).blur(3),0,0,0,1);
```

by

```
CImg<> img('milla.jpg');  
img.get_shared_channel(1).blur(3);
```

- Regions composed of **contiguous pixels in memory** are candidates for being **shared images** :
 - `CImg<T>::get_shared_point[s]()`
 - `CImg<T>::get_shared_row[s]()`
 - `CImg<T>::get_shared_plane[s]()`
 - `CImg<T>::get_shared_channel[s]()`
 - `CImg<T>::get_shared()`
- Image attribute `CImg<T>::is_shared` tells about the shared state of an image.
- Shared image destructor does nothing (no memory freed).

⇒ **Warning : Never destroy an image before its shared version !!**

- Inserting a shared image CImg<T> into a CImgList<T> makes a **copy** :

```
CImgList<> list;  
CImg<> shared = img.get_shared_channel(0);  
list.insert(shared);  
shared.assign();           // OK, 'list' not modified.
```

- Function CImgList<T>::insert() can be used in a way that it forces the **insertion of a shared image into a list**.

```
CImgList<unsigned char> colors;  
CImg<unsigned char> color = CImg<unsigned char>::vector(255,0,255);  
list.insert(1000,colors,list.size,true);  
color.fill(0);           // 'list' will be also modified.
```

Conclusion

Conclusion and Links

- The CImg Library eases the coding of image processing algorithms.
- For more details, please go to the official CImg site !

<http://cimg.sourceforge.net/>

- A 'complete' inline reference documentation is available (generated with doxygen).
- A lot of simple examples are provided in the CImg package, covering a lot of common image processing tasks. It is the best information source to understand how CImg can be used at a first glance.
- Finally, questions about CImg can be posted in its active Sourceforge forum :
(Available from the main page).

Conclusion and Links

- Now, you know almost everything to handle complex image processing tasks with the CImg Library.

⇒ **You can contribute to this open source project :**

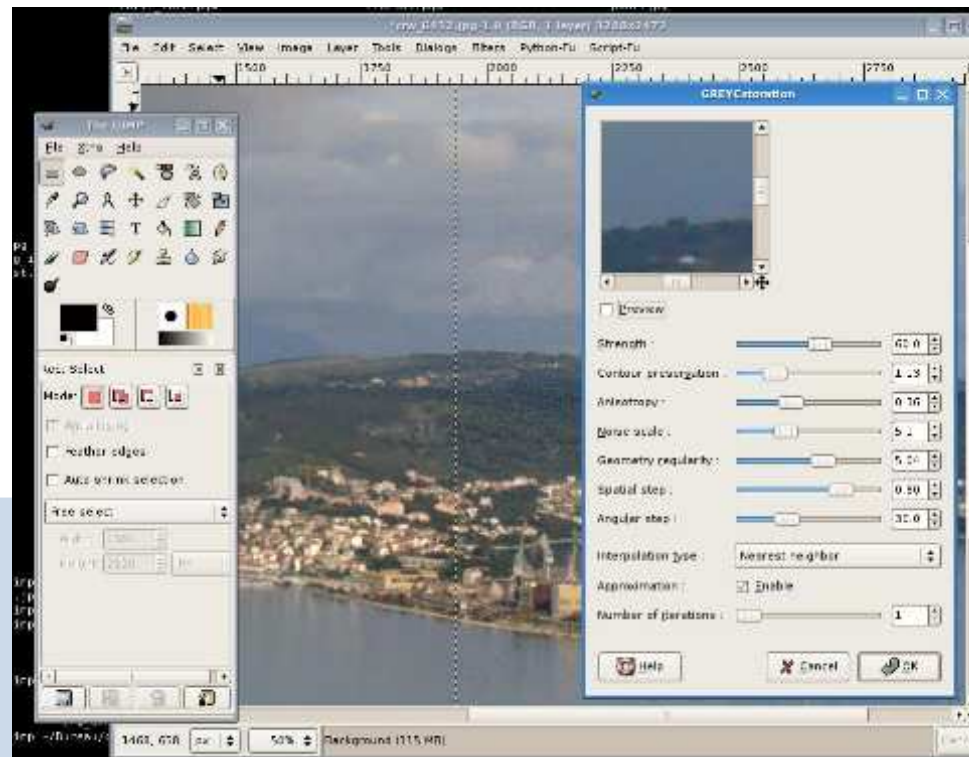
- Submit bug reports and patches.
- Propose new examples or **plug-ins**.

Used in real world : “GREYCstoration”

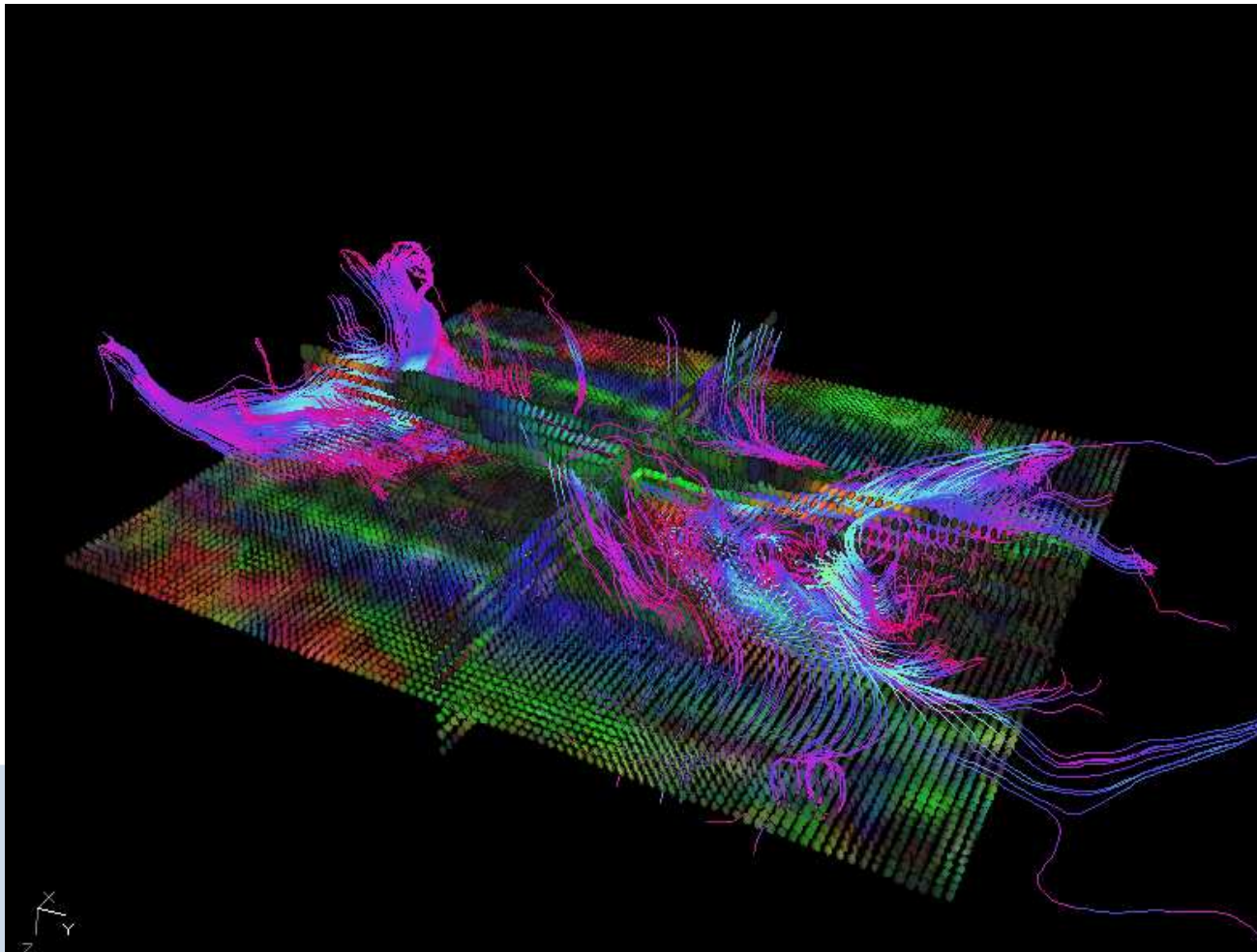
- This anisotropic smoothing function has been embedded in an open-source software : **GREYCstoration**.

⇒ Distributed as a free command line program or a plug-in for GIMP.

⇒ <http://www.greyc.ensicaen.fr/~dtschump/greycstoration/>



- DTMRI dataset visualization and fibertracking code is distributed in the Clmg package (File `examples/dtmri_view.cpp`, 823 lines).



Corpus Callosum Fiber Tracking

The end

Thank you for your attention.

Time for additional questions if any ..

