Introduction to The CImg Library

C++ Template Image Processing Library (v.1.1.8)

```
CImg<unsigned char> back(
cimg_mapXY(back,x,y) back
CImgDisplay disp(back,"Bou
const unsigned char col1[3
double u = sqrt(2.0),  cx
while (!disp.closed && dis
  img = back;
  int xm =(int)cx, ym = (i
  float r1 = 50, r2 = 50;
```

David Tschumperlé

CNRS UMR 6072 (GREYC) - Image Team

Internet web course, February 2007.

Outline - PART I of III : General Overview

- Context of C++ Image Processing Libraries
- Overview of the CImg<T> Class
  - Image construction, data access, math operators
  - Basic image transformations
  - Drawing on Images
- Overview of the CImgList<T> Class
- Overview of the CImgDisplay Class

Outline - PART II of III : Filtering and Loops

- Context of Image Filtering
- Convolution - Correlation
- Morphomaths - Median
- Anisotropic smoothing
- Other related functions
- Using Loops in CImg

Outline - PART III of III : Other things to know

- CImg plugins
- 3D object creation & visualization
- Shared images
- Conclusion

PART I of III

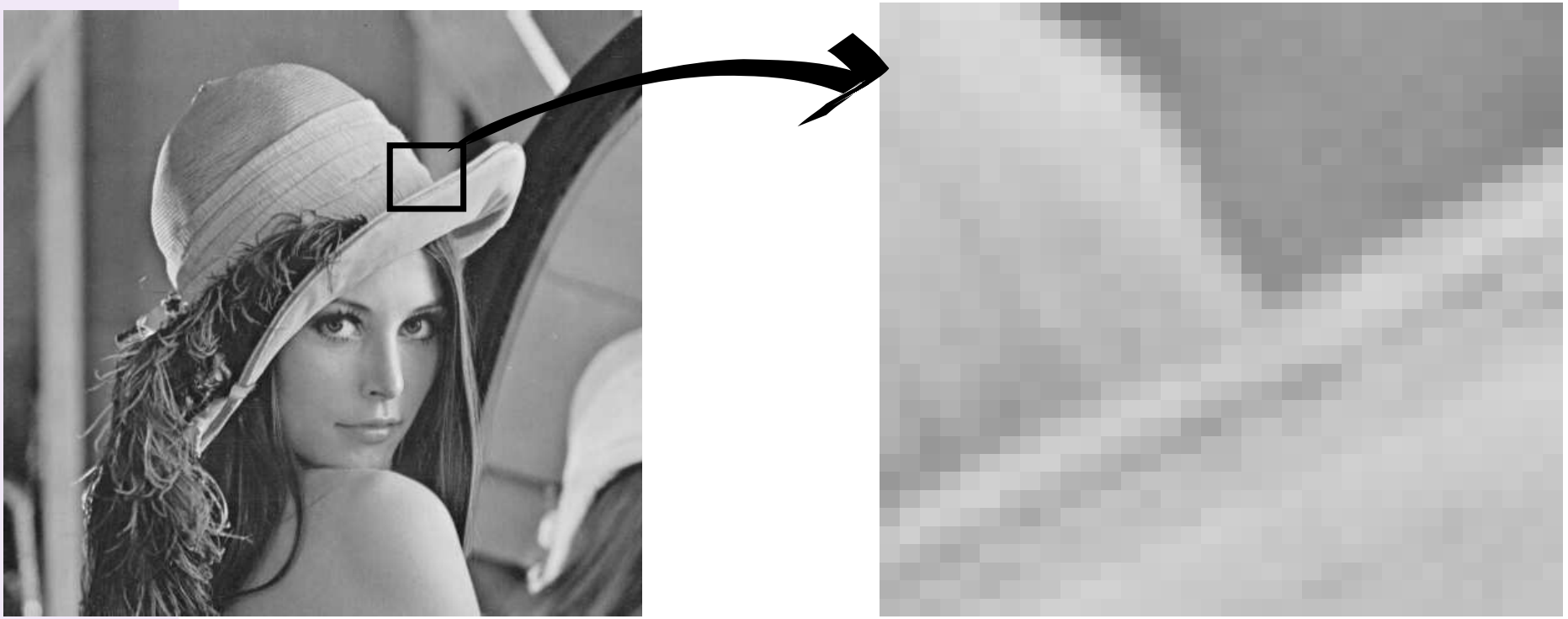
Outline - PART I of III : General Overview

⇒ **Context of C++ Image Processing Libraries**

- Overview of the CImg<T> Class
  - Image construction, data access, math operators
  - Basic image transformations
  - Drawing on Images
- Overview of the CImgList<T> Class
- Overview of the CImgDisplay Class

Context

- Digital Image Processing.



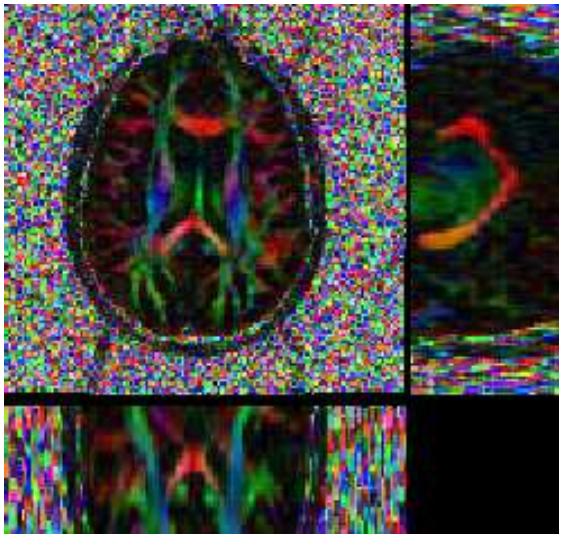
- On a computer, image data stored as a discrete array of values (pixels or voxels).

Context

- Acquired digital images may have 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, ...
- Different types of images are digitally stored with different image 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 bits).

Context

- But most of advanced image processing techniques are “image type independant”.
- e.g. : binarization of an image $I : \Omega \rightarrow \Gamma$ by a threshold $\epsilon \in \mathbb{R}$.

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

- Implementing an image processing algorithm should be independant on the image format and coding.

$\Rightarrow$ Generic Image Processing Libraries :

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

- Genericity in C++ is possible and elegant.
- But too much genericity may lead to unreadable code.

Too much genericity...(Olena).

Olena

- Main Page
- File List
- Class List
  - ntg::internal::_from_float< n, ncomps, qbits, color_system >
  - ntg::internal::_to_float< n, ncomps, qbits, color_system >
  - oln::topo::combinatorial_map::internal::alpha< U >
  - ntg::any< E >
  - oln::topo::combinatorial_map::internal::any< Inf >
  - ntg::any_ntg< E >
  - ntg::internal::any_ntg_< E >
  - oln::topo::combinatorial_map::internal::anyfunc< U, V, Inf >
  - oln::io::internal::anything
  - oln::morpho::attr::attr_traits< ball_parent_change< I, Exact > >
  - oln::morpho::attr::attr_traits< ball_type< I, Exact > >
  - oln::morpho::attr::attr_traits< box_type< I, Exact > >
  - oln::morpho::attr::attr_traits< card_full_type< I, T, Exact > >
  - oln::morpho::attr::attr_traits< card_type< T, Exact > >
  - oln::morpho::attr::attr_traits< cube_type< I, Exact > >
  - oln::morpho::attr::attr_traits< dist_type< I, Exact > >
  - oln::morpho::attr::attr_traits< height_type< T, Exact > >
  - oln::morpho::attr::attr_traits< integral_type< T, Exact > >
  - oln::morpho::attr::attr_traits< maxvalue_type< T, Exact > >
  - oln::morpho::attr::attr_traits< minvalue_type< T, Exact > >
  - oln::morpho::attr::attr_traits< other_image< Dad, I, Exact > >

Too much genericity...(GIL).

```
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)));
```

⇒ Definitely not simple and suitable for non computer scientists !!

The CImg Library

- An open-source C++ library aiming to simplify the development of image processing algorithms for (quite) generic datasets.
- 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 generic images.
- Started in 2000, the project is now hosted on Sourceforge since December 2003 :

http://cimg.sourceforge.net/

THE CIMG LIBRARY

C++ Template Image Processing Library.

Main characteristics

CImg is **lightweight** :

- Total size of the full package : approx. 3.5 Mb.
- All the library is contained in an unique 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
```

- So, the library itself only takes 1Mo of sources (approximately 20000 source 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 header file ?
  - Simplicity “à la STL”.
  - Used template functions and structures know their type only during the compilation phase.
  - Compilation time is bigger than using a classical pre-compiled library, but only used functions are actually compiled here.

  $\Rightarrow$ Small executable code.

- Main drawback : Compilation time when optimization flags are set.

Main characteristics

CImg is (sufficiently) **generic** :

- CImg implements static genericity by using the template mechanism.
- CImg defines an image class that can handle hyperspectral volumetric 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.

$\Rightarrow$ CImg covers 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 the standard C libraries.

- Advanced tools or libraries may be used with CImg (ImageMagick, XMedcon, libpng, libjpeg, libtiff...), but these tools are also multi-platform.

- Successfully tested platforms : Win32, Linux, Solaris, FreeBSD, Mac OS X.

- It is also “multi-compiler” : g++, VC++ 6.0, Visual Studio .NET, Borland Bcc 5.6, Intel ICL, Dev-Cpp, DMC.

Main characteristics

And finally, ... CImg is **simple to use** :

- Only 1 single file to include.
- Only 5 C++ classes to know.
- Very basic low-level architecture, simple to apprehend.
- Enough genericity, allowing complex image processing tasks.

Main characteristics

And finally, ... CImg is **simple to use** :

```
#include ‘‘CImg.h’’
using namespace cimg_library;

int main(int argc, char **argv) {
  CImg<float> img(‘‘input3d.hdr’’);
  CImgList<float> grad = img.get_gradientXYZ();
  grad[0].pow(2);
  grad[1].pow(2);
  grad[2].pow(2);
  CImg<float> norm = (grad[0] + grad[1] + grad[2]).sqrt();
  norm.normalize(0,255);
  norm.save(‘‘output3d.hdr’’);
  return 0;
}
```

Main characteristics

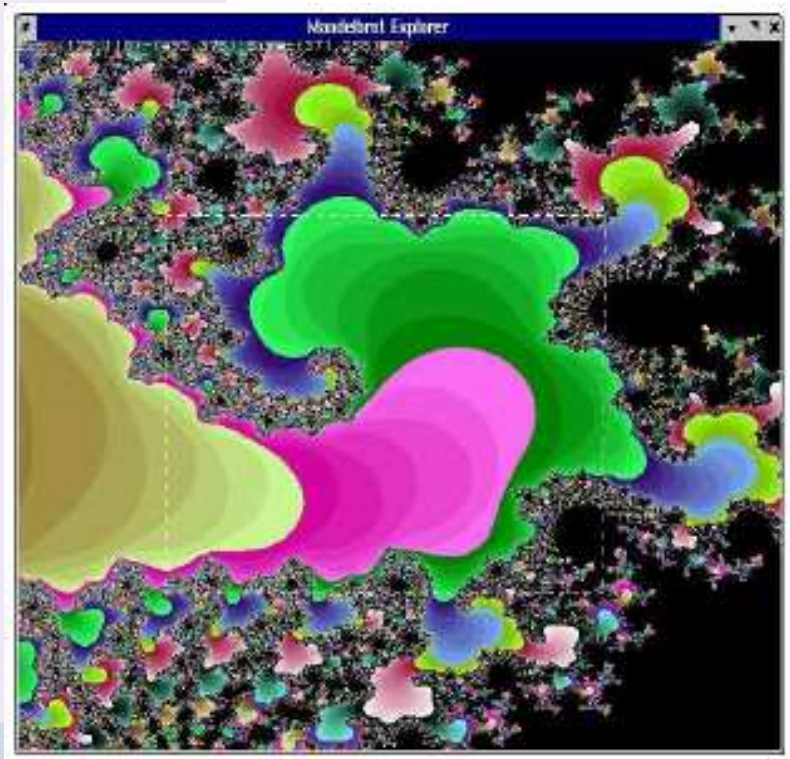
And finally, ... CImg is **simple to use** :

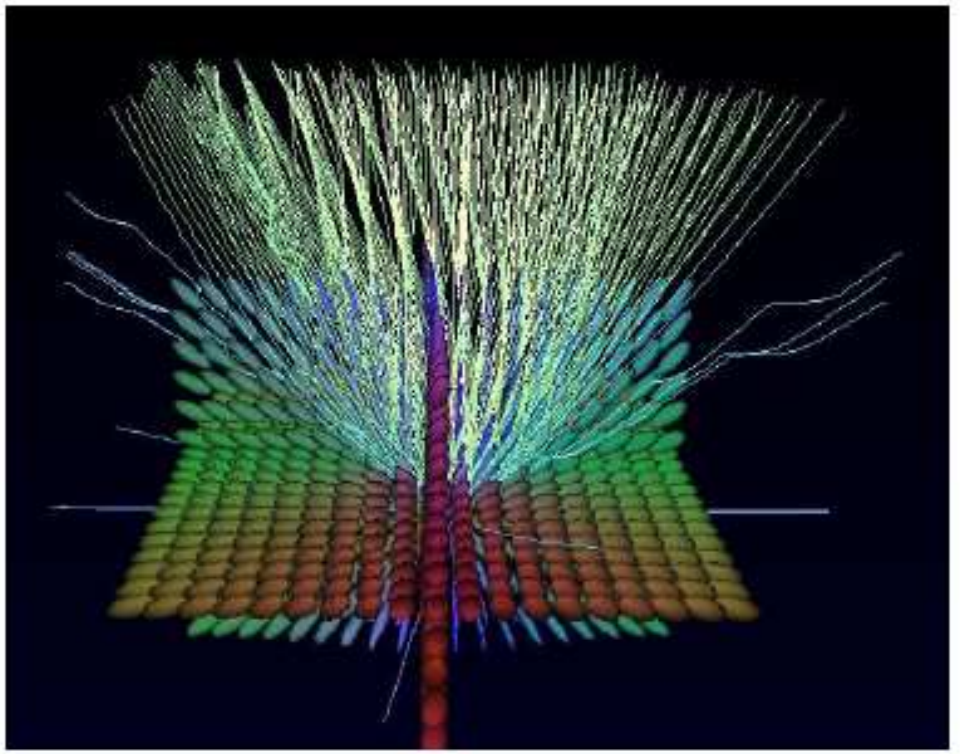
```
#include ‘‘CImg.h’’
using namespace cimg_library;

int main(int argc, char **argv) {
  CImgList<float> grad = CImg<float>(‘‘input3d.hdr’’).get_gradientXYZ();
  (grad[0].pow(2) + grad[1].pow(2) + grad[2].pow(2)).sqrt().
   normalize(0,255).save(‘‘output3d.hdr’’);
  return 0;
}
```

- Reassuring : Can produce eventually as compact and unreadable code as its other competitors...

Main characteristics

- 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.

- The library is composed of only five C++ classes :
  - `CImg<T>`, that represents an image with pixels of type `T`.
  - `CImgList<T>`, that represents a list of images `CImg<T>`.
  - `CImgDisplay`, that represents an image display window.
  - `CImgStats`, that represents statistics of an image.
  - `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...).

CImg methods

- 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.blur(2);          // ’img’ is modified
```

- Almost all CImg methods are declined into these two versions.

Typical CImg code

- The code below opens an image, rotate it, blur it, and save the result in an output filename.

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

int main(int argc, char **argv) {
  const char *file_i = cimg_option(``-i'',``foo.jpg'',``Input image'');
  const float angle = cimg_option(``-angle'',0.0f,``Rotation angle'');
  const float blur = cimg_option(``-blur'',0.0f,``Blur strength'');

  const CImg<unsigned char> img(file_i);
  const CImg<unsigned char> dest = img.get_rotate(angle).blur(blur);
  dest.save(``result.jpg'');
  return 0;
}
```

Outline - PART I of III : General Overview

- Context of C++ Image Processing Libraries
- Overview of the CImg<T> Class

  ⇒ **Image construction, data access, math operators**
  - Basic image transformations
  - Drawing on Images
- Overview of the CImgList<T> Class
- Overview of the CImgDisplay Class

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 vectors.
- Image processing algorithms are methods of `CImg<T>` : `blur()`, `threshold()`, `rotate()`, `resize()`, `convolve()`, `erode()`, `load()`, `save()`...
- Most of the methods are implemented to apply for the most wide case (3D volumetric and hyperspectral images).

CImg<T> : Low-level Architecture

- 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 independant : it has its own pixel buffer.

- CImg functions (destructor, constructors, operators,...) handle memory allocation/desallocation efficiently.

CImg<T> : Constructors

- `CImg<T>()`
  Default constructor, constructs an empty image.

- `CImg<T>(unsigned int, unsigned int, unsigned int, unsigned int)`
  Constructs a 4D image with specified dimensions. Omitted dimensions are set to 1.

```
CImg<float> img(100,100);  // 2D scalar image.
CImg<unsigned char> img(256,256,1,3);  // 2D color image.
CImg<bool> img(128,128,128);  // 3D scalar image.
CImg<short> img(64,64,32,16); // 3D hyperspectral image, with 16 bands.
```

- `CImg<T>(const char *filename)`
  Construct an image by reading an image filename.

```
CImg<float> img(``toto.jpg'');
CImg<bool> img(``volume3D.dcm'');
```

CImg<T> : Access to data informations

- `int dimx() const`

  Get the dimension along the X-axis (width). Same for `dimy(),dimz() and dimv()`.

```
int W = img.dimx();
```

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

  Get the pixel value at specified coordinates. Omited coordinates are set to 0. Out-of-bounds coordinates are not checked !

```
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;
```

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

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

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

CImg<T> : Copies and assignments

- `template<typename t> CImg<T>(const CImg<t>& img);`
  Construct an image by copy. Perform pixel type cast if necessary.

  `CImg<float> img_float(img_double);` (lossy cast will be done)

- `template<typename t> CImg<T>& operator=(const CImg<t>& img);`
  Assignement operator. Replace the instance image by a copy of `img`.

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

CImg<T> : Math operators and functions

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

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

- Operators returns images that have the “best” types.

```
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 $*$ operator corresponds to a matrix product !

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

- 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);
float det = img.det();

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

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

Outline - PART I of III : General Overview

- Context of C++ Image Processing Libraries
- Overview of the CImg<T> Class
  - Image construction, data access, math operators

  ⇒ **Basic image transformations**
  - Drawing on Images
- Overview of the CImgList<T> Class
- Overview of the CImgDisplay Class

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);
```

- `normalize(), cut(), quantize(), threshold()`.
  Apply basic global transformations on pixel values.

```
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(256,256);
img.rotate(45);
```

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

```
CImg<> img(256,256);
img.resize(-100,300);
```

⇒ Border conditions and interpolation types can be chosen.

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()`,....

- Look at the reference for all the functions available....

`http://cimg.sourceforge.net/reference`

Outline - PART I of III : General Overview

- Context of C++ Image Processing Libraries
- Overview of the CImg<T> Class
  - Image construction, data access, math operators
  - Basic image transformations

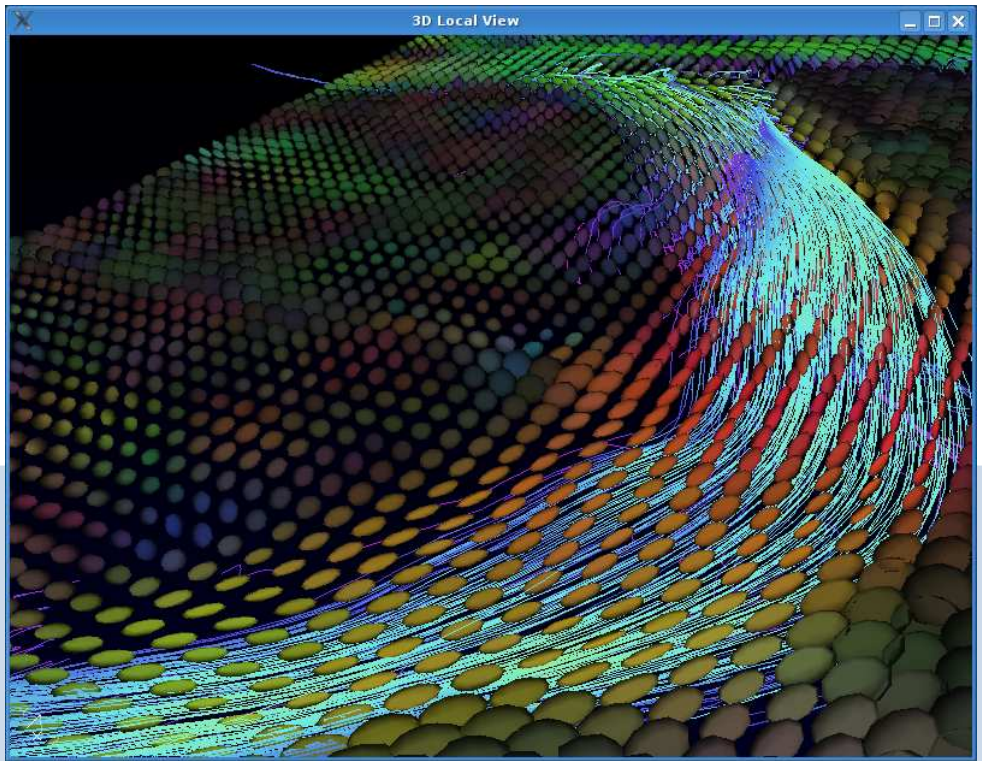
  ⇒ **Drawing on Images**
- Overview of the CImgList<T> Class
- Overview of the CImgDisplay Class

CImg<T> : Drawing functions

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

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

- All drawing functions begin with `draw_*()` and draws features on the instance image.



CImg<T> : Drawing functions

- They are easy to use. They just need an image, user-specified coordinates for the feature, and a color pointer (and optionnaly an opacity).
- They return a reference to the instance image, so they can be pipelined.
- They clip objects that are out of image bounds.

⇒ `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);
```

- `CImg<T>::draw_object3d()` can draw 3D parametric objects in an image (mini OpenGL!)

Outline - PART I of III : General Overview

- Context of C++ Image Processing Libraries
- Overview of the CImg<T> Class
  - Image construction, data access, math operators
  - Basic image transformations
  - Drawing on Images

⇒ **Overview of the CImgList<T> Class**

- Overview of the CImgDisplay Class

CImgList<T> : Overview

- A `CImgList<T>` represents a list 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.
- Used as a flexible and ordered set of images.

CImgList<T> : Main functions

```
// Create a list of 20 color images 100x100.
CImgList<float> list(20,100,100,1,3);

// Insert a scalar image 50x50 at the end of the list.
list.insert(CImg<float>(50,50));

// 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).

Outline - PART I of III : General Overview

- Context of C++ Image Processing Libraries
- Overview of the CImg<T> Class
  - Image construction, data access, math operators
  - Basic image transformations
  - Drawing on Images
- Overview of the CImgList<T> Class

⇒ **Overview of the CImgDisplay Class**

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.
- A `CImgDisplay` has its main 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` fields are updated.
  - Keyboard events : `key` is updated.
  - Window events : `is_resized`, `is_closed` and `is_moved` are updated.
- 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 Win32 and X11 graphics libraries.
- Display can handle automatically image normalization to display float-valued images correctly.

CImgDisplay : Useful functions

- Construction :

```
CImgDisplay disp1(img,``Mon premier display'');
CImgDisplay disp2(640,400,''Mon deuxième display'');
```

- Display :

```
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`

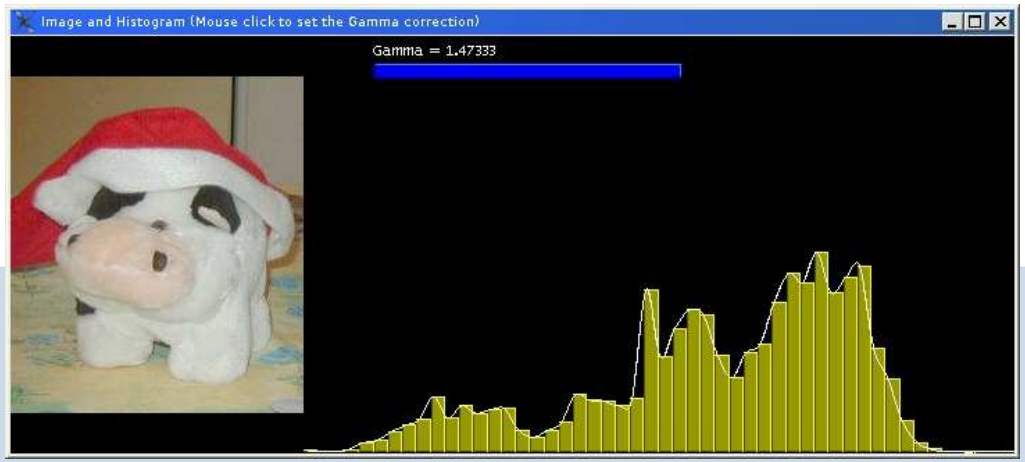
```
#include "CImg.h"
using namespace cimg_library;
int main() {
  CImgDisplay disp(256,256,"Mon Display");
  while (!disp.is_closed) {
    if (disp.button&1) {
      const int x = disp.mouse_x, y = disp.mouse_y;
      CImg<unsigned char> img(disp.dimx(),disp.dimy());
      unsigned char col[1] = {255};
      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);
CImgl<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_forXYZV(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,white,NULL,1,"Gamma = %g",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.dimx()+50 && disp.mouse_x<=img.dimx()+450) gamma = (disp.mouse_x-img.dimx()-50)/150.0;
  disp.resize(disp).display(visu).wait();
}
```

Result :

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



PART II of III

Outline - PART II of III : Filtering and Loops

⇒ **Context of Image Filtering**

- Convolution - Correlation
- Morphomaths - Median
- Anisotropic smoothing
- Other related functions
- Using Loops in CImg

Context : Image Filtering

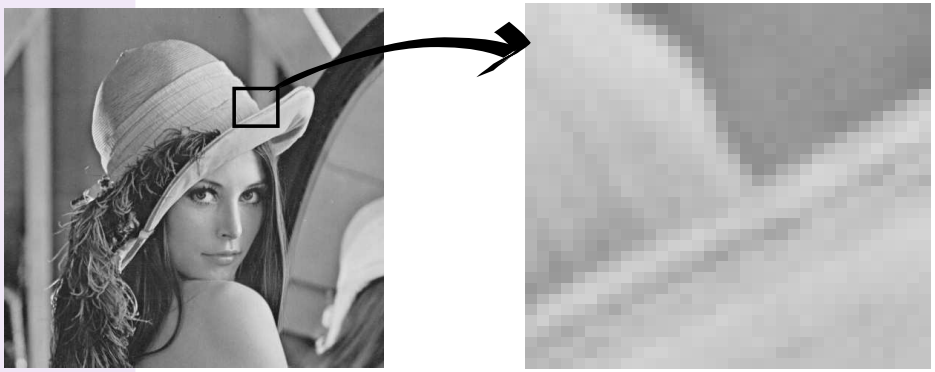
- Image filtering is one of the most common operations done on images in order to retrieve informations.

- Filtering is needed in the following cases :
  - Compute image derivatives (gradient) $\nabla I = \left(\frac{\partial I}{\partial x} \quad \frac{\partial I}{\partial x}\right)^T$.
  - 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 $M = [0.5 \quad 0 \quad -0.5]$
- Erosion obtained with $\bullet = \min()$.

Outline - PART II of III : Filtering and Loops

- Context of Image Filtering
- ⇒ **Convolution - Correlation**
- Morphomaths - Median
- Anisotropic smoothing
- Other related functions
- Using Loops in CImg

Linear filtering

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

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

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

- `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()`
- Two common linear filters are already implemented.
- Gaussian kernel for image smoothing : `CImg<T>::get_blur()` and `CImg<T>::blur()`
- Image derivatives : `CImg<T>::get_gradientXY()` and `CImg<T>::get_gradientXYZ()`

$\Rightarrow$ 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)

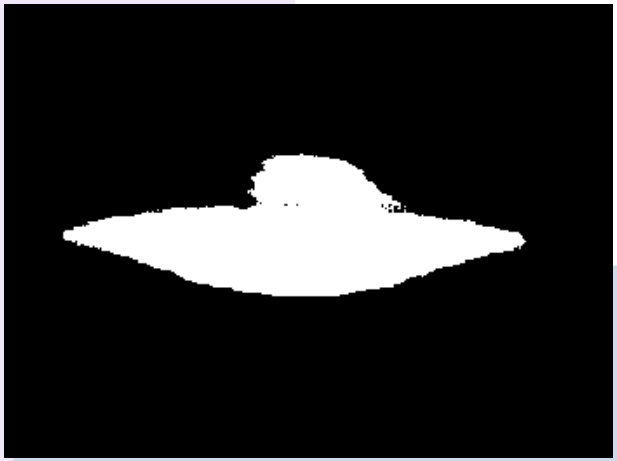
- When mask size is big, you can efficiently convolve the image by a multiplication in the Fourier domain.
- `CImg<T>::get_FFT()` returns a `CImgList<T>` with the real and imaginary part of the FT.
- `CImg<T>::get_FFT(true)` returns a `CImgList<T>` with the real and imaginary part of the inverse FT.
- `CImg<T>::mul()` and `CImg<T>::div()` can be used to multiply or divide two images (pointwise operation).

Outline - PART II of III : Filtering and Loops

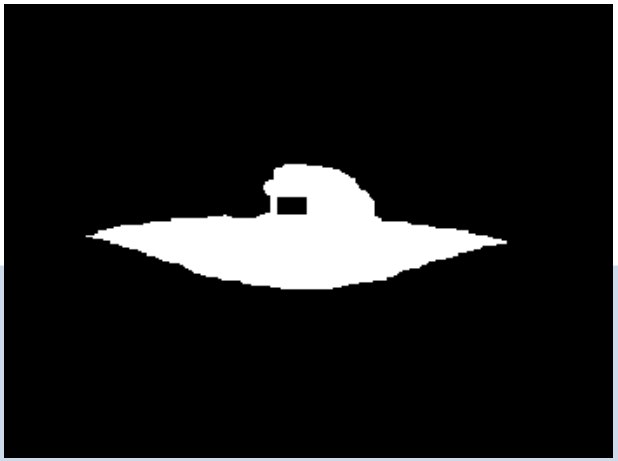
- Context of Image Filtering
- Convolution - Correlation
- ⇒ **Morphomaths - Median**
- Anisotropic smoothing
- Other related functions
- Using Loops in CImg

Morphomaths

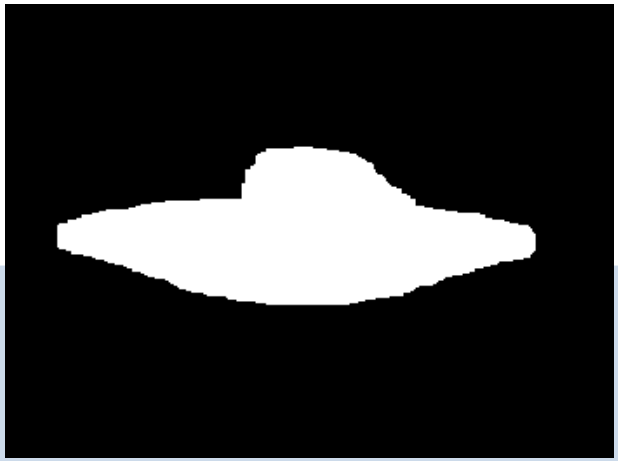
- Nonlinear filters.
- Erosion : Keep the mininum value in the image neighborhood having the same shape than the structuring element mask.
  `CImg<T>::erode()` and `CImg<T>::get_erode()`.
- Dilatation : Keep the maximum value in the image neighborhood having the same shape than the structuring element 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

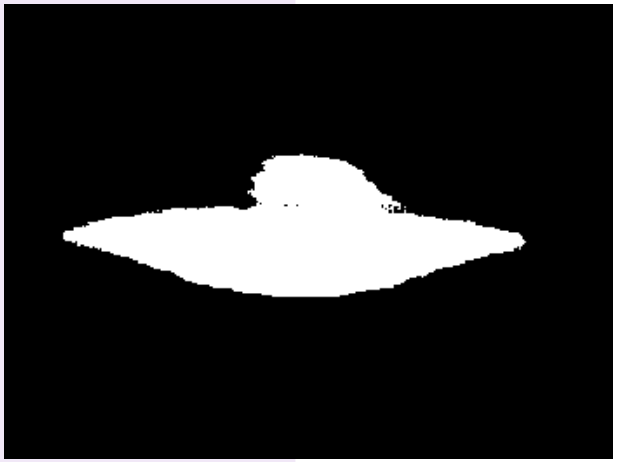
Morphomaths (2)

- Opening : Erode, then dilate :

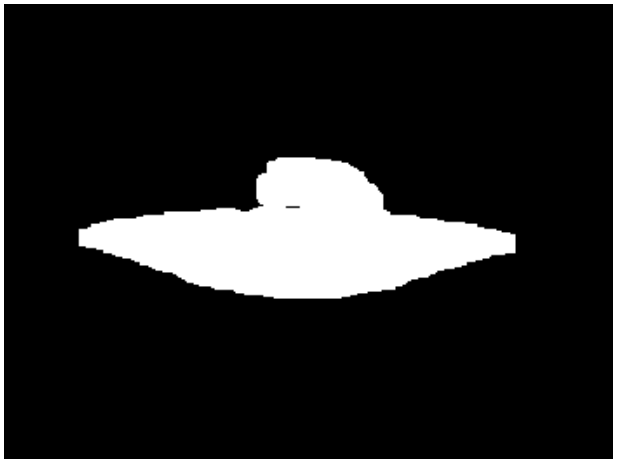
```
img.erode(10).dilate(10);
```

- Closing : Dilate, then erode :

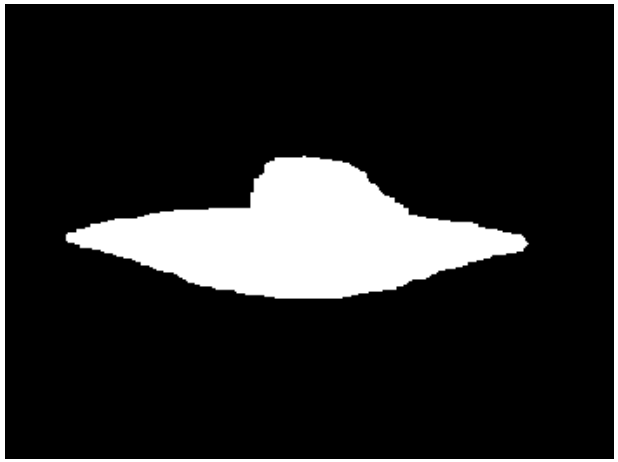
```
img.dilate(10).erode(10);.
```



(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.



Outline - PART II of III : Filtering and Loops

- Context of Image Filtering
- Convolution - Correlation
- Morphomaths - Median
- ⇒ **Anisotropic smoothing**
- Other related functions
- Using Loops in CImg

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, inpainting, resizing.

Anisotropic smoothing

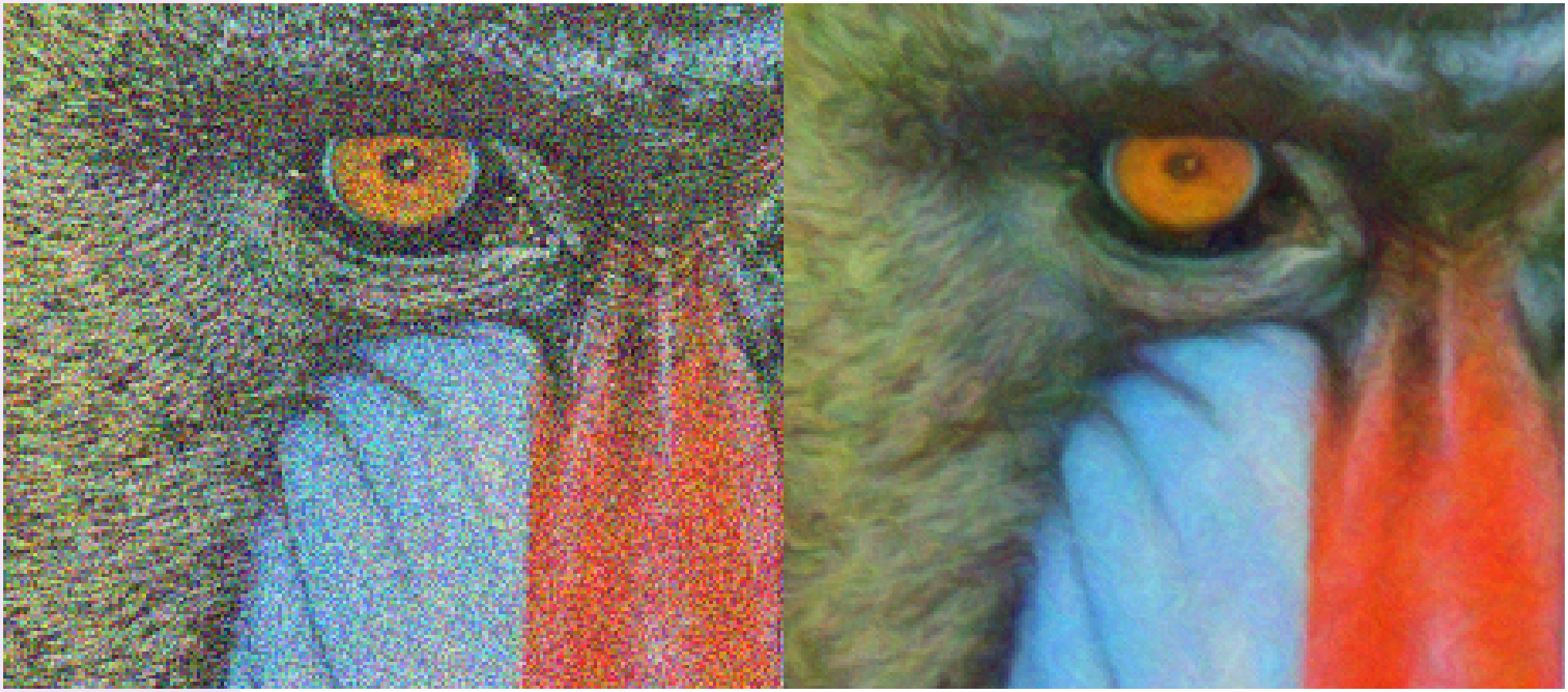
- `CImg<T>::blur_anisotropic()` implements the following diffusion PDE :

$$\frac{\partial I_i}{\partial t} = \text{trace}\left(\mathbf{w}\mathbf{w}^T\,\mathbf{H}_i\right) + \nabla I_i^T \mathbf{J}_\mathbf{w}\mathbf{w}$$

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

- Image smoothing while preserving discontinuities (edges).
- One of the most 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.)

Outline - PART II of III : Filtering and Loops

- Context of Image Filtering
- Convolution - Correlation
- Morphomaths - Median
- Anisotropic smoothing

⇒ **Other related functions**

- Using Loops in CImg

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.




Comparing images

- 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)`

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

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

Filtering in CImg : Conclusions

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

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

$\Rightarrow$ There are smart ways to define your own nonlinear filters, using neighborhood loops.

Outline - PART II of III : Filtering and Loops

- Context of Image Filtering
- Convolution - Correlation
- Morphomaths - Median
- Anisotropic smoothing
- Other related functions

⇒ **Using Loops in CImg**

Simple loops

- 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.dimx(); x++)
cimg_forY(img,y)  ⇔  for (int y=0; y<img.dimy(); y++)
cimg_forZ(img,z)  ⇔  for (int z=0; z<img.dimz(); z++)
cimg_forV(img,v)  ⇔  for (int v=0; v<img.dimv(); v++)
```

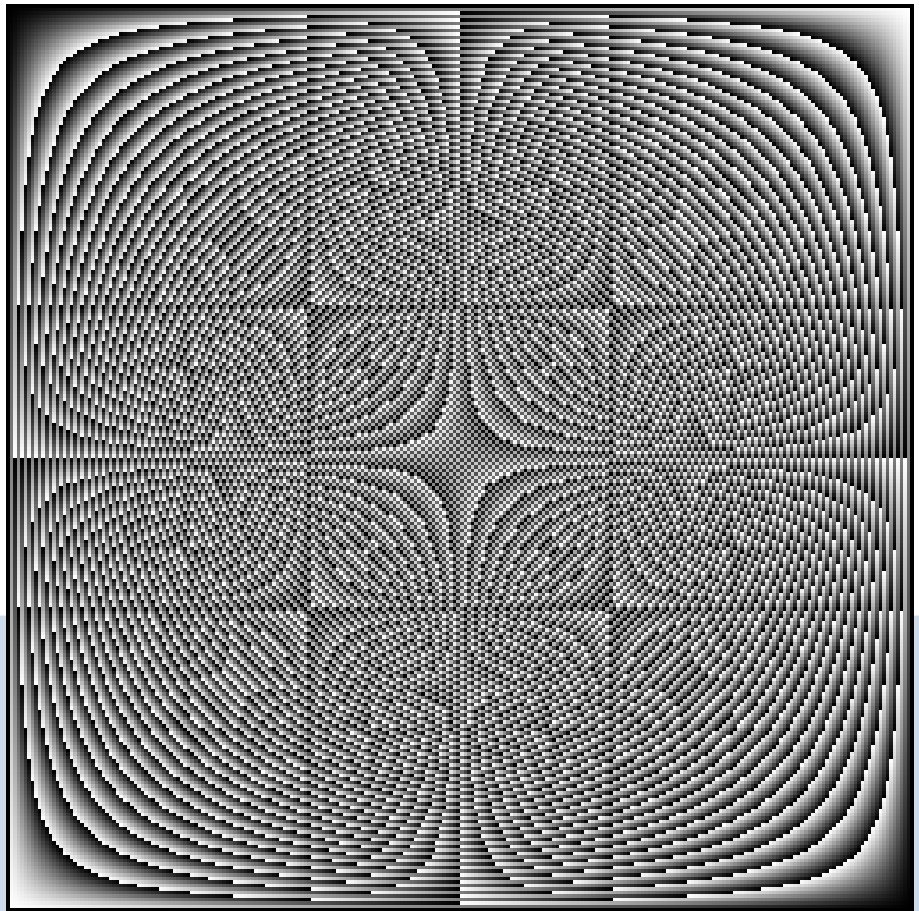
- CImg also defines :

```
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_forXYZV(img,x,y,z,v)  ⇔  cimg_forV(img,v) 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;
```

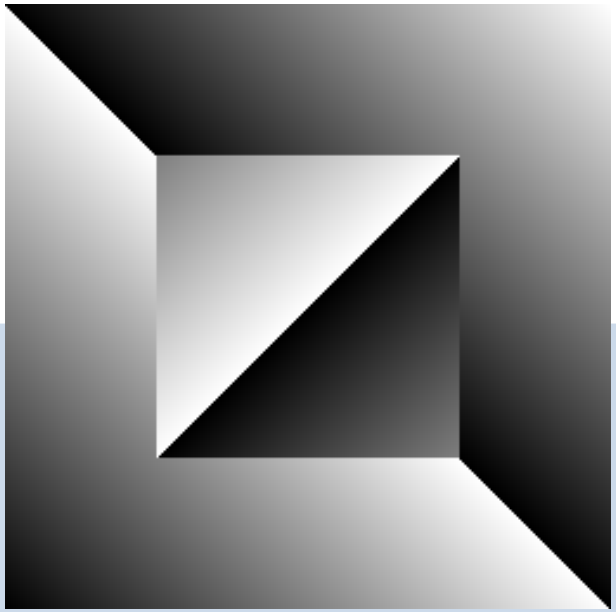


Interior and Border loops

- 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)` and for all other dimensions.
- Example :

```
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;
```



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 and comprehensive code.

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) {
  .. Here, Ipp, Icp, ... Icn, Inn are accessible ...
}
```

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_forV(img,v) cimg_for3x3(img,x,y,0,v,I) {
  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 to 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;
}
```

PART III of III

Outline - PART III of III : Other things to know

⇒ **CImg plugins**

- 3D object creation & visualization
- Shared images
- Conclusion

CImg plugins

- Sometimes an user needs or defines specific functions, either very specialized or not generic enough.
- Not suitable for integration 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;
}
```

CImg plugins

- Plugin functions are 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
};
```

CImg plugins

- Advantages :
  - Allow slight 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 etal*).
  - `noise_analysis.h` : Advanced statistics for noise estimation.
  - `primitives3d.h` : Functions to construct classical 3D meshes (cubes, sphere,...)

CImg plugins

- Plug-ins variables :
  - `#define cimg_plugin` : Add functions to the `CImg<T>` class.
  - `#define cimglist_plugin` : Add functions to the `CImgList<T>` class.
  - `#define cimgstats_plugin`, `#define cimgdisplay_plugin`.
- 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 this simple plugin mechanism, CImg is a very open framework for image processing.

Outline - PART III of III : Other things to know

- CImg plugins

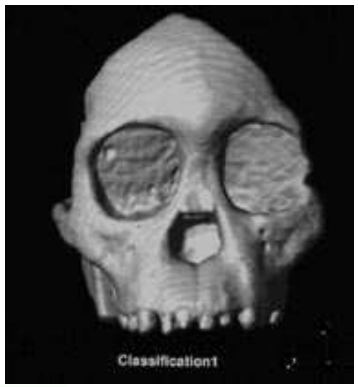
⇒ **3D object creation & visualization**

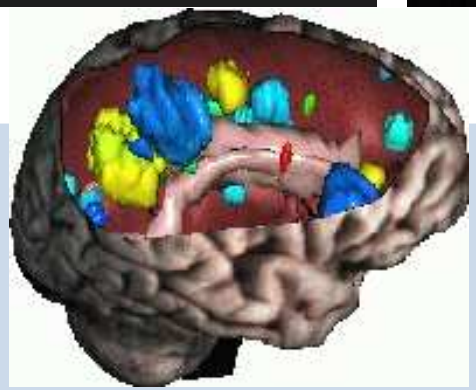
- Shared images

- Conclusion

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 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.
  - No multiple lights allowed.
  - Texture mapping partially supported.
  - No GPU acceleration.

3D Object Visualization : Live Demo

- Double torus.
- Image as a surface.

3D Object Visualization : How 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()`

- Higher-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 (size M).
  - A `CImgList<tf>` of primitives (size N).
  - A `CImgList<T>` of colors (size N).
  - A `CImgList<to>` of opacities (size N) (optional parameter).

Display a house : building point list

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

Display a house : building primitives list

```
CImgList<unsigned int> primitives;
primitives.insert(CImg<unsigned int>::vector(0,1,5,4));
primitives.insert(CImg<unsigned int>::vector(3,7,6,2));
primitives.insert(CImg<unsigned int>::vector(1,2,6,5));
primitives.insert(CImg<unsigned int>::vector(0,4,7,3));
primitives.insert(CImg<unsigned int>::vector(0,3,2,1));
primitives.insert(CImg<unsigned int>::vector(4,5,6,7));
primitives.insert(CImg<unsigned int>::vector(0,8));
primitives.insert(CImg<unsigned int>::vector(1,8));
primitives.insert(CImg<unsigned int>::vector(5,8));
primitives.insert(CImg<unsigned int>::vector(4,8));
```

Display a house : building colors and visualize

```
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 ambiant light intensity.
  - Camera position and focale.
  - Rendering type (Gouraud, Flat, ...)
  - Double/Single faces.

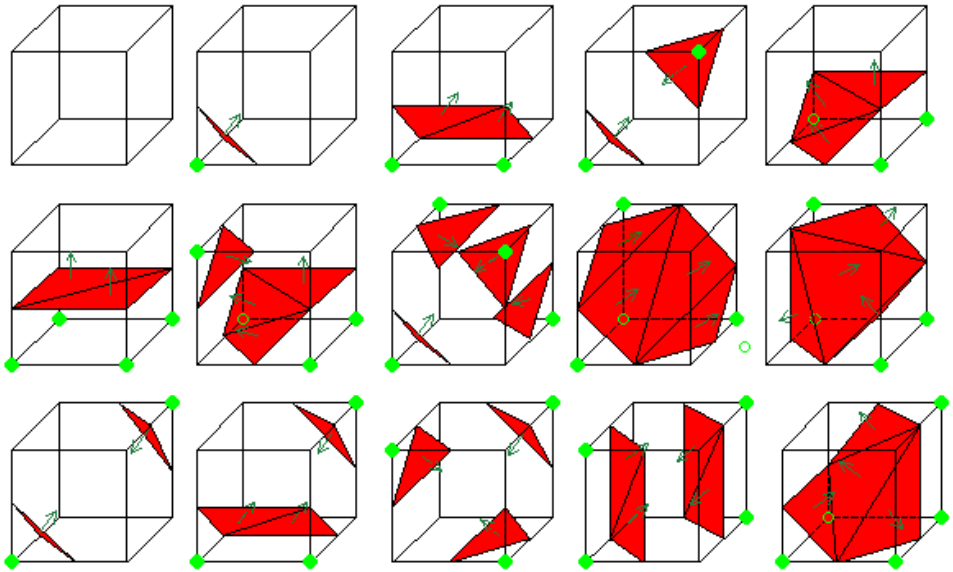
How to construct 3D meshes ?

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

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

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

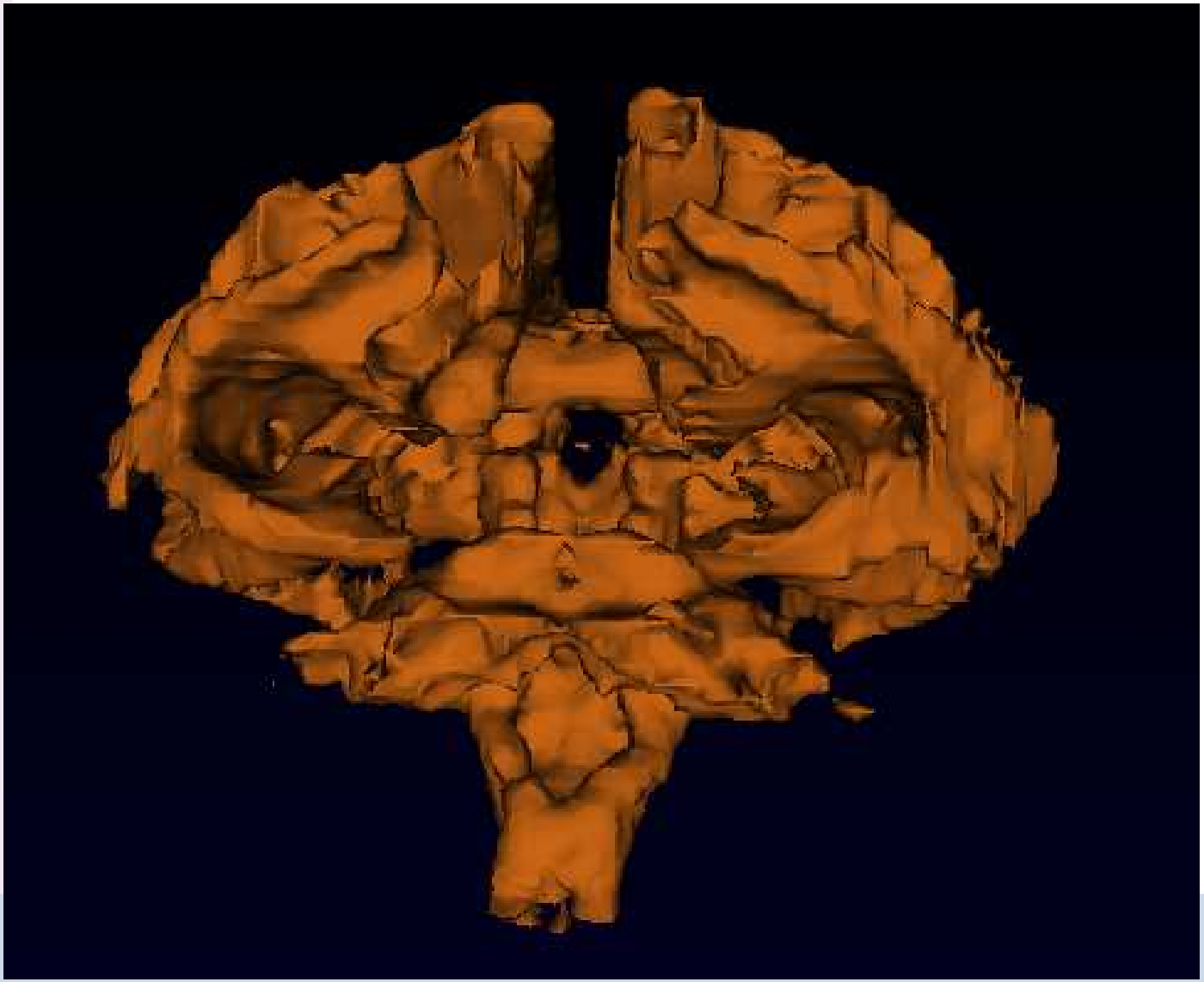
⇒ Create meshes from implicit functions.



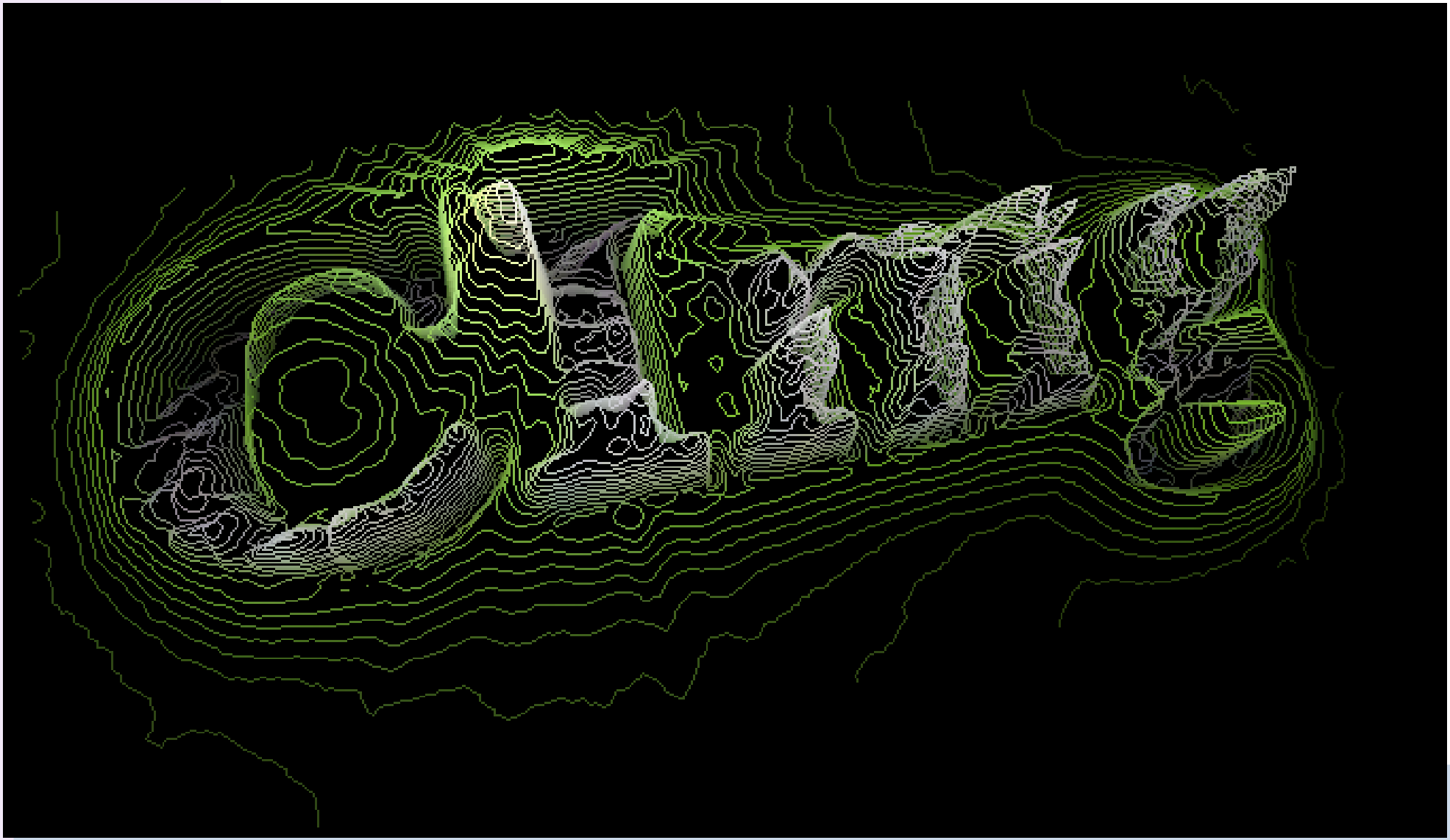
Example : Segmentation of the white matter from MRI images

```
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



3D Visualization : Related functions

- `CImg<T>::vector()` : Return a vector.
- `CImg<T>::matrix()` : Return a square matrix.
- `CImg<T>::rotation_matrix()` : Return a rotation matrix from a 3D axis and an angle.

$\Rightarrow$ Easy-to-use 3D visualization capabilities.

Outline - PART III of III : Other things to know

- CImg plugins
- 3D object creation & visualization
- ⇒ **Shared images**
- Conclusion

Shared images : Context

- 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_append(’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...

Shared images

- Solution : Using shared images :

1. Replace :

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

by

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

Shared images

- 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);
```

Shared images

- Regions composed of contiguous pixels in memory are candidates for being shared images :
  - `CImg<T>::get_shared_point[s]()`
  - `CImg<T>::get_shared_line[s]()`
  - `CImg<T>::get_shared_plane[s]()`
  - `CImg<T>::get_shared_channel[s]()`
  - `CImg<T>::get_shared()`
- Image attribute `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 !!

Shared images and CImgList

- 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, no problem.
```

- Function `CImgList<T>::insert_shared()` forces 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_shared(1000,colors);
```

Shared images and CImgList : example

- How to revert channels of a RGB images ?
- [1.] Simple but unefficient (in terms of memory usage) :

```
CImg<> img(‘‘milla.jpg’’);
CImg<> res = img.get_split(’v’).reverse().get_append(‘v’);
```

- [2.] More complicated but also more efficient (no copy!) :

```
CImg<> img(‘‘milla.jpg’’);
CImgList<> list;
list.insert_shared(img.get_shared_channel(2));
list.insert_shared(img.get_shared_channel(1));
list.insert_shared(img.get_shared_channel(0));
CImg<> res = list.get_append(’v’);
```

Outline - PART III of III : Other things to know

- CImg plugins
- 3D object creation & visualization
- Shared images

⇒ **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.
- 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 processing tasks with CImg.
- You can contribute to this open source project with :
  - Submitting bug reports and patches.
  - Propose new examples of use or interesting plug-ins.
- Remember that CImg try to constantly evolve (converge?) into something stable and coherent.