Introduction to The CImg Library

C++ Template Image Processing Library (v.1.2.5)

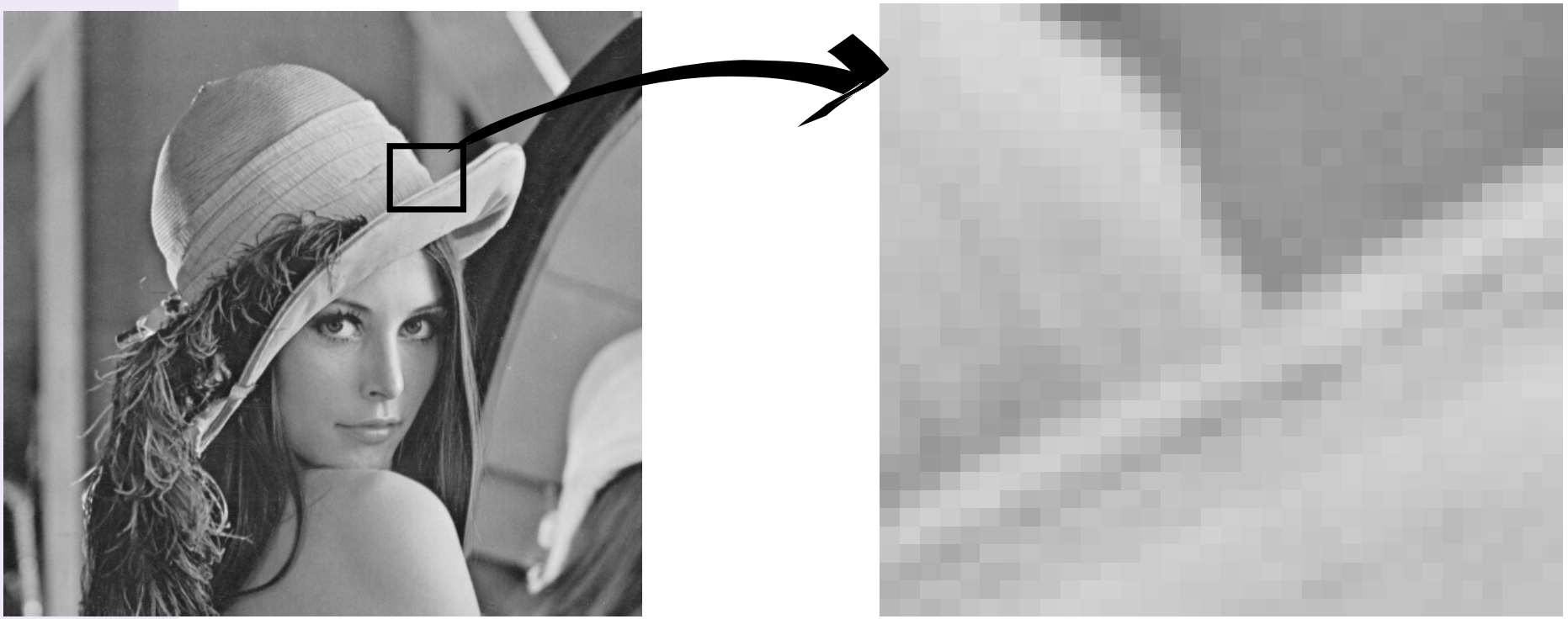
```
// Bouncing bubble
//------------------
CImg<unsigned char> back(320,256,1,3,0),img;
cimg_forXY(back,x,y) back(x,y,2) = (unsigned char)((y<2*ba
CImgDisplay disp(back,"Bouncing bubble",0,1);
const unsigned char col1[3]={40,100,10}, col2[3]={20,70,0}
double u = std::sqrt(2.0),  cx = back.dimx()/2, 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
```

David Tschumperlé

CNRS UMR 6072 (GREYC) - Image Team

Context

- Digital Images.



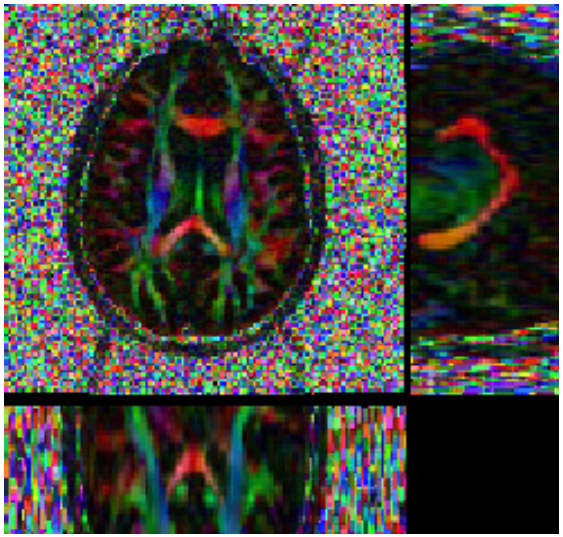
- On a computer, image data stored as a discrete array of values (pixels or voxels).

Context

- 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).

Context

- 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 ?

Context

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.

$\Rightarrow$ How to design a reasonable and useable programming library for such people ?

Context

- 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, Vigra, 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.*

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 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 CImg (.zip) package : approx. 4.2 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 1.2Mb of sources (approximately 23000 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”.
  - 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.
- 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.

$\Rightarrow$ 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.

- 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++, VC++ 6.0, Visual Studio .NET, Borland Bcc 5.6, Intel ICL, Dev-Cpp.

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("Hello World",80,80,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("Hello World",80,80,white,0,32);
  img.display();

  return 0;
}
```

Hello World step by step

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...).

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.get_rotate(20).blur(3); // ’img’ is not modified
```

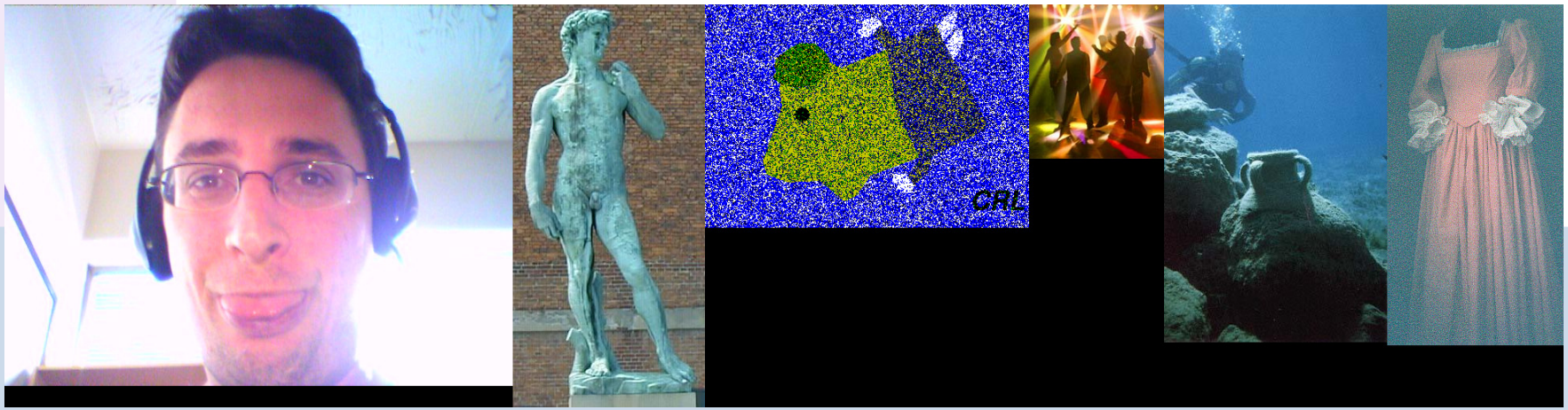
⇒ **Almost all CImg methods are declined into these two versions.**

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>` ($\neq$ STL) : `blur()`, `resize()`, `convolve()`, `erode()`, `load()`, `save()`...
- Method implementation aims to handle the most general case (3D volumetric hyperspectral images).

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.



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 own pixel buffer. It does not store any references to the `CImg<T>` or `CImgList<T>` passed at the last call to `CImgDisplay::display()`.

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++)
```

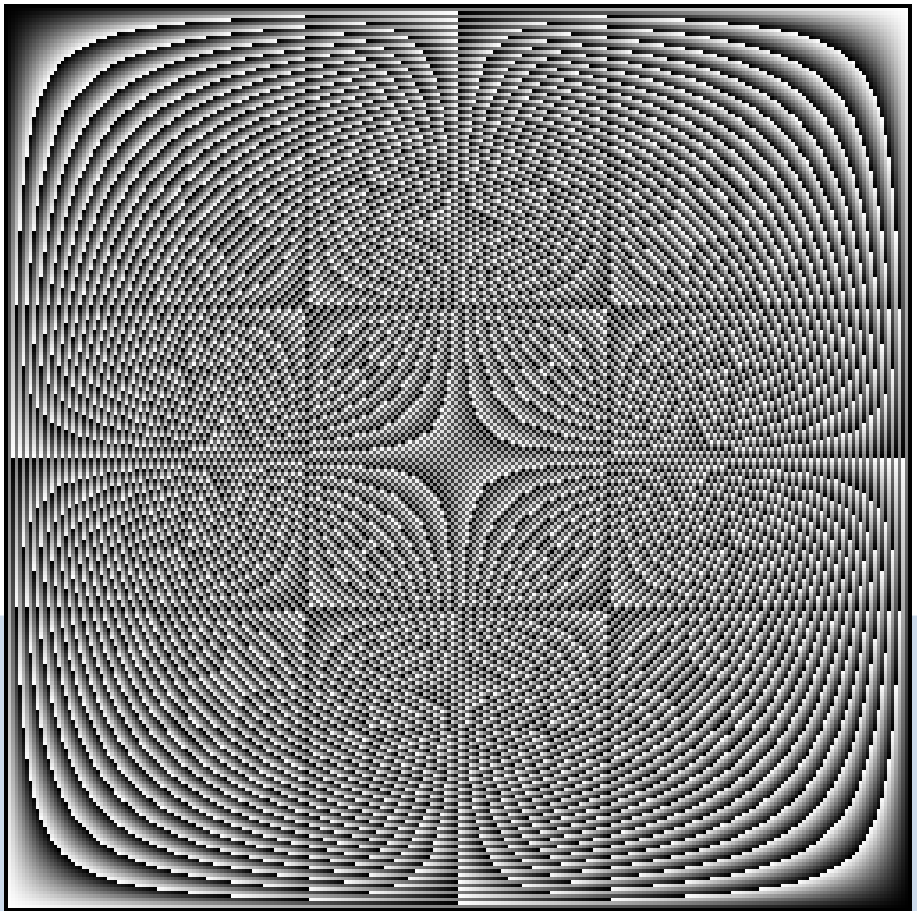
- CImg also defines :

```
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)%256; }
```



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).`

- 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);
}
```