

Comparison of some C++ image processing libraries

Giovanni Palma and Niels van Vliet

LRDE seminar, November 24, 2004, revision 515

<http://www.lrde.epita.fr/>



Copying this document

Copyright © 2004 LRDE.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being just “Copying this document”, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is provided in the file COPYING.DOC.

Table of Contents

Functionalities vs. Capabilities	3
Libraries presentation	10
Comparison	15
Ideas	33

Functionalities vs. Capabilities

Functionalities [1/2]

Functionalities = what is available?

Example of functionalities:

- 2D images (method: *image2d::number_of_rows()*),
- conversion from a color space to another (function: $RGB \rightarrow XYZ$),
- rotation (function or method?).

Functionalities [2/2]

The goal of some libraries is to provide the functionalities needed by most of the users:

- image 1 / 2 / 3 D,
- *bool / unsigned char / int / float / double / complex /* and vectors of these types,
- a lot of algorithms.

Capabilities [1/3]

But sometimes, users need a very *specific* functionality.

For example:

- circular 1D images,
- another type of storage for very large images,
- color images based on a color look up table.

Capabilities [2/3]

Capabilities = is it easy to *adapt* the library for a particular use?

Easy means:

- limited amount of work (< 1 day),
- good programmer (not expert),
- backward compatibility,
- avoid drawbacks (slow down, obfuscate bug reporting).

Capabilities [3/3]

Several techniques exist to allow *extensions* such as Object Oriented Programming and Generic Programming.

Example of Generic Programming (For_each in the STL):

```
template<typename It, typename Fun>
Fun
for_each(It first, It last, Fun f)
{
    // concept requirements
    function_requires(InputIteratorConcept<It>)
    for ( ; first != last; ++first)
        f(*first);
    return f;
}
```

Design

Compromise between capabilities and:

- complexity of the core,
- complexity to add a new feature (algorithm, data structure, . . .),
- efficiency,
- environment (compilation time),
- specific and very large field (image processing).

Key point: the design of the libraries limits their capacity of adaptation.

Libraries presentation

Comparison of four libraries:

- Cimg
- Vigna
- Horus
- Olena

Cimg

- stands for *Cool Image*,
- started by D. Tschumperlé in 1999 at INRIA.
- aims at being highly portable and simple,
- deals with 4D images templated by data type,
- under GPL.

Vigra

- stands for *Vision with Generic Algorithms*,
- developed by U. Köthe since his PhD thesis,
- aims at:
 - being adaptable to user's needs,
 - not being slowed down by genericity.
- written in a *STL* style,
- works on 2D images,
- under *Perl Artistic License* (more liberal than GPL).

Horus

- developed by the Intelligent Sensory Information Systems (ISIS) Group,
- designed for image and video analysis,
- efficient by a heavy use of the C++ template mechanism,
- CORBA (Common Object Request Broker Architecture) to allow interaction,
- non free software.

Olena 0.10

- started in 1999 at LRDE,
- generic with respect to data type, image dimension and data storage,
- based on A Static C++ Object-Oriented Programming Paradigm [[Burrus et al. \(2003\)](#)],
- under GPL version 2.

Comparison

- ability to use n D images,
- value type of pixels,
- accessors and iterators,
- specialization of algorithms,
- storage,
- grid,
- easiness of adding a new algorithm,
- bug tracking.

n D images [1/2]

1. A dimension k is chosen, and a default value is used for smaller dimensions. Ex: Vigna(2D), Horus(2/3D), Cimg(1/2/3/4D)

```
operator()(uint x, uint y=0, uint z=0, uint v=0)
```

2. The dimension is a part of the type. Ex: Olena

```
template<unsigned Dim, class T, class Impl, class Exact = mlc::final>  
class image;
```

n D images [2/2]

1. A single dimension k implies:

- intuitive (1D and 2D are special cases of 3D),
- less source code and less meta code,
- efficient only with iterators.

2. The dimension is as a parameter implies:

- any dimension can be used,
- efficient in all cases (iterators and (x, y) operators),
- static information.

Value type. Limited set [2/3]

Is a limited set of types a bad idea?

- the properties of these types are well known,
- less meta code has to be written,
- compatibility is easier.

The set can be extended by the maintainers, under given constraints.

Possibility to have more value types than arithmetic types:

```
int max(int, int); works with char, short, ...
```

Value type using a parameter [3/3]

Parameter \neq any type:

- need to define the expected members functions,

Is it possible to multiply two gray values?

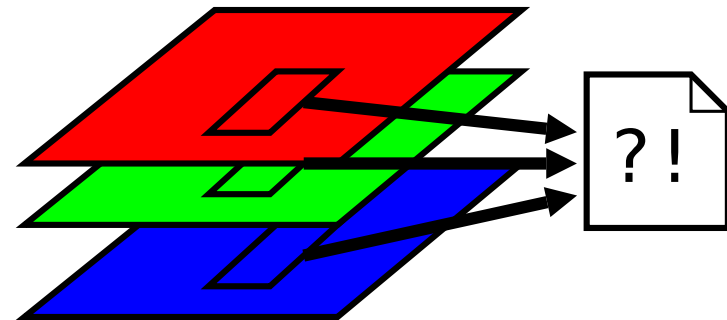
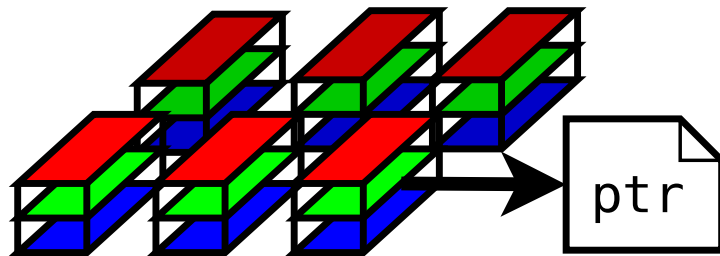
- need to define traits (ex: zero of the value type),
- limited by the design. Ex: Olena 0.10:

```
value_type&                // <-- Reference!  
operator[](const abstract::point<P>& p)
```

Accessors and iterators [1/4]

C and STL approach: pointers (Horus).

Problems with multi-band images.



Sometimes, the proper pointer does not exist.

Accessors and iterators [2/4]

A proxy can be used but [[Köthe \(1998\)](#)]:

- slow (optimized by the compiler?),
- difficult to have a safe proxy,
- several small problems, such as in conversions or specializations.

An image is a function that returns:

- the address of a value?
- the value itself (by copy)?

Accessors and iterators [3/4]

Accessors (Vigra) [[Köthe \(1998\)](#)].

```
template <class SrcIter, class SrcAcc,
          class DestIter, class DestAcc>
void copy(SrcIter src, SrcIter srcend, SrcAcc srcacc,
          DestIter dest, DestIter destacc)
{
    for(; src != srcend; ++src, ++dest)
        destacc.set(srcacc(src), dest);
}
```

Accessors and iterators [4/4]

A succession of points (Olena).

```
template <class Is, class Id>
void copy(const abstract::image<Is> &src,
          abstract::image<Id> &dst)
{
    oln_iter_type(Is) it(src);
    for_all(it)
        dst[it] = src[it];
}

void main()
{
    image2d<int> f(42, 16), g(64, 51);
    copy(f, g);
}
```

Storage

- The storage is handled by the image: `struct image { T *data; };` (Cimg).
- The storage is a parameter: `template<class s> image<s>{}`.

Hard to make an interface independent from the image.

- advanced task needed (border),
- efficiency,
- open to any optimization.

Specialization of algorithms [1/3]

Methods to specialize a function for a given set of types:

- overloading (Olena),
- dispatch by signature (Horus),
- template specialization: STL approach (Vigra).

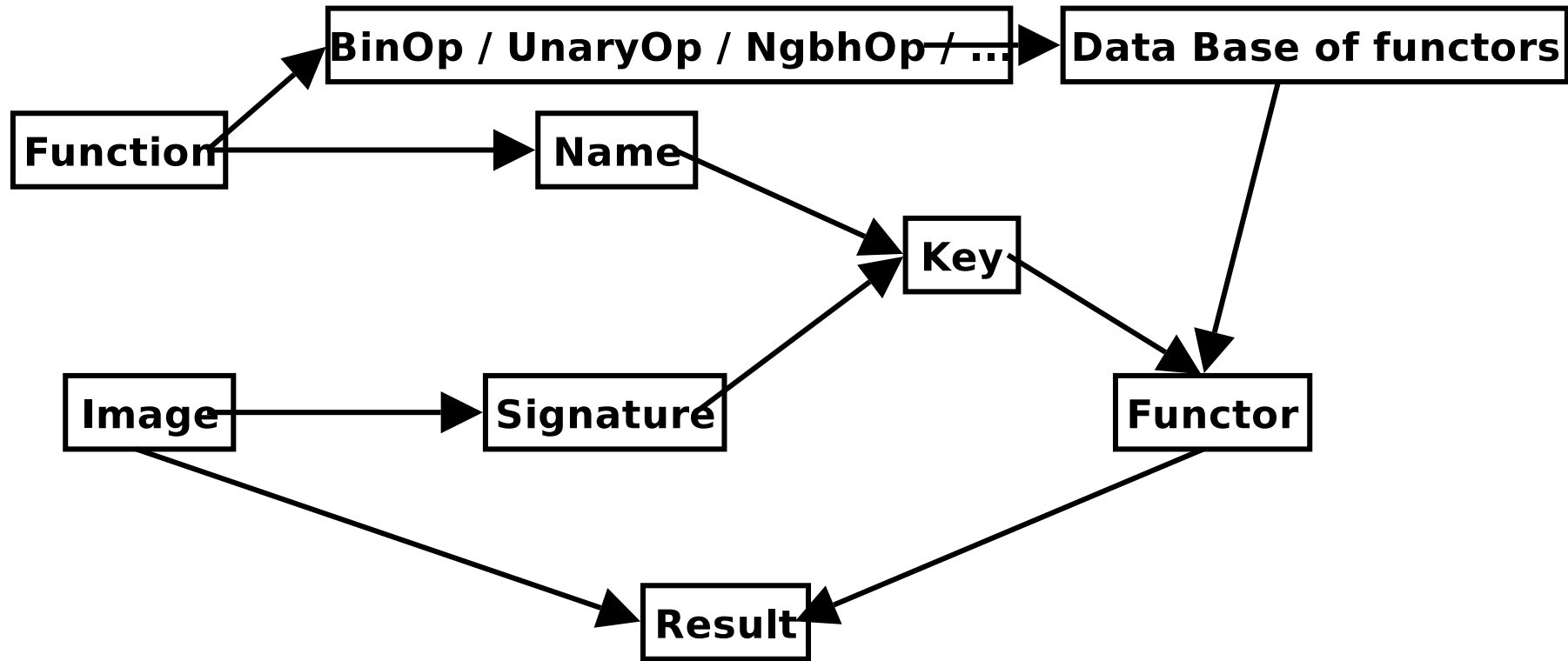
Specialization of algorithms [2/3]

```
namespace impl {
    template <typename I>                // Basic.
    void algo(abstract::image<I> &im) {...}
    template <typename I>                // Specialized.
    void algo(binary_image<I> &im) {...}
}

template <typename I>
void algo(abstract::image<I> &im) // Dispatcher.
{
    impl::algo(im.exact()); //Call the algorithm on im
}                                //casted to its real type.
```

Static specialization using Olena.

Specialization of algorithms [3/3]



Dynamic specialization using Horus.

Bug tracking

Error localization:

- restrictive generic prototypes (Olena),

```
template<class Exact>
void algo(const abstract::scalar_image<E> &im);
```

- dynamic assertions (Cimg, Horus, Vigna, Olena),

```
precondition(im[p] > 0);
```

- static assertions (Olena).

```
template <class T>
void foo() { mlc_is_a(T, int); }
```

Adding a new algorithm [1/2]

- Horus: Functor + *instanciator* (to register the algorithm in the data base),
- Vigna: STL like, non explicit error messages,
- Cimg: macros,
- Olena: intensive usage of traits (hidden by macros) \Rightarrow human readable *AND* generic code.

Adding a new algorithm [2/2]

```
template <class DstValueType, class Src1ValueType, class Src2ValueType>
class HxBpoSqrDst
{
    DstValueType doIt(const Src1ValueType &a, const Src2ValueType &b)
        { return sqr(a - b); }
    static HxString className() { return HxString("sqrDst"); }
};

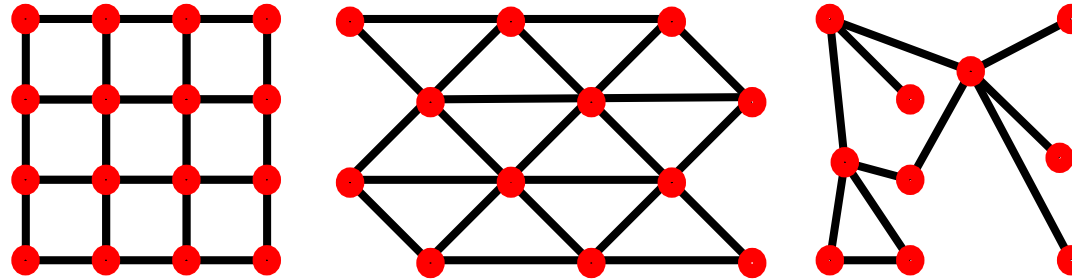
template <class Sig>
struct HxInstantiatorSqrDst
{ HxImgFtorBpo<Sig, Sig, Sig,
                HxBpoSqrDst<typename Sig::ArithType, typename Sig::ArithType,
                            typename Sig::ArithType> > f; };

static HxInstantiatorSqrDst<HxImageSig2dByte> f001;

HxImageRep HxSqrDist(HxImageRep im1, HxImageRep im2)
{ return ima1.binaryPixOp(im2, "sqrDst"); }
```

Adding a new algorithm in Horus.

Grid



- It changes points, vectors, neighborhood, iterators, images.
- non-rectangular grids are rarely implemented.

Ideas

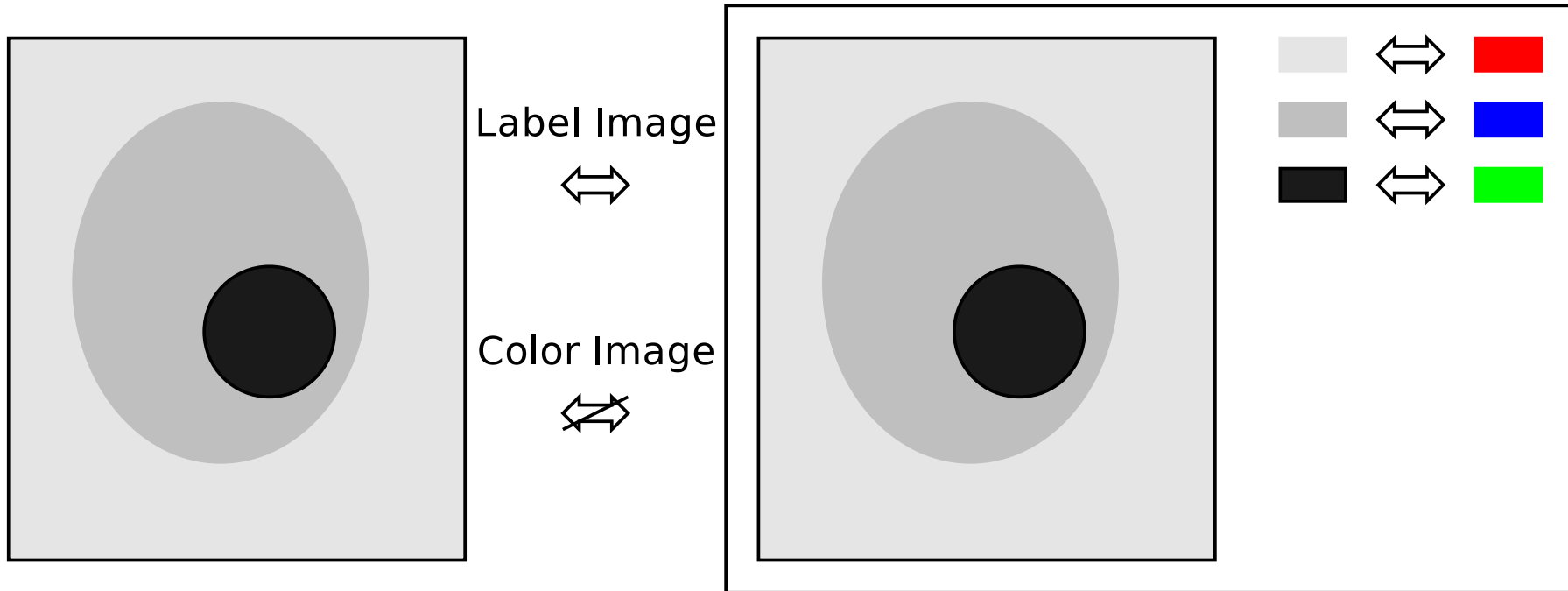
How should Olena 1.0 look like?

- data types,
- accessors and iterators,
- n D images,
- grid.

Value type of the pixels [1/2]

- Adding of a semantic layer (gray-scale, color, . . .) for internal types,
- different implementation for each semantic (float, integer, . . .),
- ability to the user to use external types,
- image inheritance follows the properties of the value type.

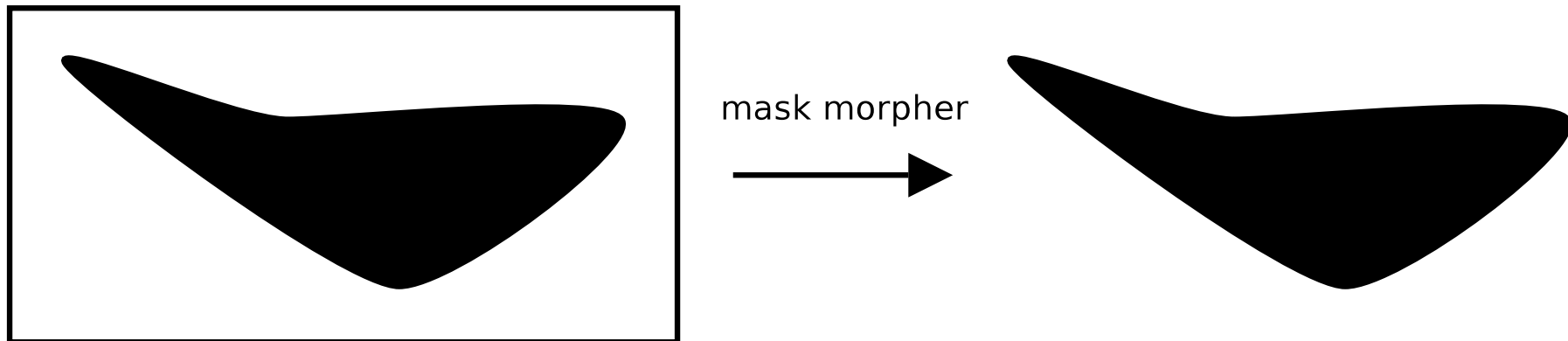
Value type of the pixels [2/2]



Label and color image with a color palette.

Accessors and iterators

- Separation between reading and writing a value,
- use existing iterators \Rightarrow can be used on several images,
- different kinds of iterators (skip some points for example).



Domain of the image defined by a mask.

Storage

- Easy way to change the implementation,
- morphers,
- implementation using types of external libraries (FFT).

Environment and new algorithms

- Specialization of algorithms: implicitly handled by overloading,
- generic algorithms: improve the way to write them,
- reduce compilation time,
- enhance interoperability (change IO).

A library should remain generic and simple. It is possible!

n D images

- real n D implementation,
- specialization of critical methods.

Grid

- factorization of the code (point2d works with rectangular or hexagonal pixels),
- rectangular grids and graph,
- bindings with other grids using morphers.

Conclusion

- The design of Olena is relevant,
- some advanced functionalities can rely on morphers,
- some code needs to be factored,
- concepts must be clearly defined,
- olena should be kept simple.

References

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. (1999). *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition.

Burrus, N., Duret-Lutz, A., Géraud, T., Lesage, D., and Poss, R. (2003). A static C++ object-oriented programming (scoop) paradigm mixing benefits of traditional oop and generic programming. In *Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL'03)*, Anaheim, California.

Eichelberger, H., Wolff, J., and v. Gudenberg (2000). UML description of the STL. In *First Workshop on C++ Template Programming, Erfurt, Germany*.

References

Koelma, D. and other ISIS members (2003). Horus user guide, version 2.0.

Köthe, U. (1998). On data access via iterators.

Köthe, U. (2000). *Generische Programmierung für die Bildverarbeitung*. PhD thesis.

Köthe, U. (2004). Vision with generic algorithms (vgra).

Tschumperlé, D. (2004). The cimg library.