Capabilities of some C++ image processing libraries

Giovanni Palma, Niels van Vliet

Technical Report n^o0414, November 2004
revision 572

This report compares several image processing libraries. The comparison of libraries are generally based on their functionalities. The functionalities are the tools which are directly available to the user. For example, in image processing, some libraries propose 3D binary images, or the rotation of 2D images. On the contrary, this report compares some C++ libraries on their capabilities. It is their capacity to be adapted by a user to his particular need. For example, if circular images are not present in the library, is it simple to add this feature ? The goal of this comparison is to improve the design of Olena, the LRDE C++ image processing library.

Ce rapport compare plusieurs bibliothèques de traitement d'images. On compare généralement les bibliothèques selon les fonctionnalités proposées. Par fonctionnalités nous entendons des outils directement utilisables par l'utilisateur. Par exemple dans le cadre des bibliothèques de traitement d'images, certaines d'entre elles proposent des images 3D binaires, ou la rotation d'une image en 2D. Au contraire ce rapport compare les bibliothèques en fonction de leur capacité à s'adapter à des utilisations originales. Par exemple s'il n'y a pas d'image circulaire, est-il facile pour l'utilisateur d'ajouter cette fonctionnalité ? L'objectif étant d'améliorer l'architecture de la bibliothèque Olena développée au sein du LRDE, les bibliothèques comparées sont écrites en C++.

Keywords
Image processing libraries, C++, Design, Capabilities, Genericity

Laboratoire de Recherche et Développement de l'Epita
14-16, rue Voltaire – F-94276 Le Kremlin-Bicêtre cedex – France
Tél. +33 1 53 14 59 47 – Fax. +33 1 53 14 59 22
lrde@lrde.epita.fr – http://www.lrde.epita.fr/

Copying this document

Contents

Introduction

Since 1999, an image processing library is developed at LRDE[1]. This library, Olena[2], aims at being a generic one. It provides algorithms which work as well on one dimension gray images as on three dimensions color images.

An important criterion for potential users of a library is the number of functionalities provided. Some image processing libraries implement a large number of algorithms, or propose different ways to store an image in memory. Our goal today is to release a stable version of Olena. Thus the functionalities are not a priority. Instead we want to build a core which provides interesting capabilities, and which is easy to extend. For example, we will not implement 4 dimensions images, or sparse images yet, but such extensions must be easy to create either by us or by a user.

This technical report compares the capabilities of several existing C++ libraries. First we explain what is the goal of a library and the difference between the functionalities and the capabilities. Then the libraries used in the comparison are presented. In the third part, the comparison of their capabilities is made. Finally we focus on what can be improved in Olena.

[1]EPITA Research and Development Laboratory

[2]http://www.lrde.epita.fr/olena

Chapter 1

Functionalities vs. Capabilities

A library is a collection of tools which are useful in a given domain. For a given field, most of the time different libraries exist and are not compatible. The user has to choose among them. This chapter explains two different criteria. The first one is the number of functionalities, the second one is the facility to adapt the library to a particular usage.

1.1 Functionalities

Most of the time, the user chooses the library that provides the maximum of functionalities. The library factors the work needed by several users. The functionalities are what is directly available; without any piece of work.

The algorithms provided by libraries are functionalities. For example, it is the case for the rotation or the copy of images. Reading JPEG images, or having conversion between several color spaces are also common functionalities. The quality of the algorithm to compute a functions is also important. Functions must be efficient.

The functionalities are not only the functions provided but also the types which are directly available. For example some image processing libraries provide RGB images[1], other do not.

Because most of the choices of users are often based on the number of functionalities, one approach is to try to cover all functionalities that the user may need. That is why the libraries usually provide:

- 1 to 3D images,
- all the built-in data types (*bool, short, int, float*...), and vectors of these types (*float[3]*...),
- a large number of algorithms.

But sometimes, users need to adapt the library for a particular usage. All the functionalities can not be added to every libraries. The reason is that the amount of time is too high to build such a library. Even if it were possible, the library would be huge, and it would certainly yield to problems.

[1]RGB stands for Red Green Blue, RGB images are color images.

1.2 Capabilities

A capability is the availability to extend more or less easily a library. If a functionality is not available, most of the time this new feature can be added. In some cases, it can be limited by the conception of the library. That is why it is needed to take a look at the capabilities of libraries.

The user may need to use existing algorithms with his own types. For example, he might appreciate a library, but he might need to change the storage of images because the default storage loads the whole image in memory whereas his own images do not fit into this one. Another example is circular images: the user might need to have the lefts pixels connected to the right pixels in an image. As you can see, the user might want to use his own types with the existing tools, but he might also want to use the existing types with his new algorithms.

The capabilities of the library are linked to the limitations imposed by its design. Some libraries provide an easy and elegant way to extend this one without knowing all about the technical part of them. It is always possible to add any feature to a library, but sometimes you just need to re-write the library from zero. Thus in this report, only the capabilities that are easy to write by a user are considered. A new functionality is easy to add if:

- the amount of work is limited,

  It means that is takes less than one day to adapt the library.

- the programmer is not an expert,

  The programmer must be good. He does not have to be able to decipher unusual error messages, or to know tricks such as the Barton and Nackman (1994) one[2], but he must have the level and the will to learn one or two things.

- it keeps the backward compatibility,

- the adaptation does not lead to drawbacks.

  For example, functions must not run much slower on home-made images (circular ones for example) than on built-in images. Furthermore, the modification of the library must keep its homogeneity. Error messages must not be obfuscated.

Several techniques exist to allow extensions such as Object Oriented Programming and Generic Programming. This report does not deal with technical aspects [see Burrus et al. (2003) and Eichelberger et al. (2000)]. We will focus on the design of the libraries. It is hard to propose a good design, because it is a compromise between the capabilities and the:

- complexity of the core,

  It is easier to write a library which works only with 2D images than several different dimensions. The complexity of the core might also increase the complexity of the error messages.

- complexity to add a new feature (algorithm, data structure, . . .),

  So as to be generic, the user might need to write some functions. For example, in the case of STL, in order to use a *std::map* on a new type, a specialization of the *std::less* class must be written.

[2] This trick is used to have static hierarchies.

- efficiency,

  Some methods can be used to ease the adaptation of a library, but lead to poor performances. It is the case of virtual functions.

- environment,

  Genericity often obfuscates error messages, and leads to compilation that are time consuming.

- specific and very large field.

  The library must not be too general. It must focus on its field in order to be user friendly.

In this report, we compare the design of different libraries. We do not compare libraries in term of functionalities, but rather in term of capabilities. The reason is that we want to be sure that the core of our library, Olena, has a suitable design. Later, additional functionalities will be added to please users.

Chapter 2

Image processing libraries

In this chapter some C++ libraries are going to be compared. First a quick overview of these libraries is given, then their capabilities are compared.

2.1 Overview of the libraries

The compared libraries are: Cimg, Vigra, Horus and Olena. Several good libraries are written in Java, OCaml or other languages, but we focus on libraries written in C++. The reason is that the goal of our comparison is to improve our C++ library.

There are many C++ image libraries. A large part of them is actually written in a C-style. Because of their lack of genericity, it is hard to enlarge their capabilities. Some image libraries are generic, but their paradigm seems too far from our library to improve it. An example is Pandore[1] which uses a new preprocessor and macros to have genericity.

The capabilities of several libraries are going to be presented in the next section. Here is the list and a quick overview of these libraries.

- Cimg: the simplicity.

  Cimg stands for *Cool Image*. It has been started by David Tschumperlé during its PhD thesis in 1999 at *INRIA Sophia Antipolis*. This library is under GPL (Gnu General Public License).

  Cimg has been designed to be simple to use, and efficient [Tschumperlé (2004)]. It consists only of a single huge file, which provides as well input output functions, based classes or advanced algorithms. It is highly portable.

  Its design is inspired from STL. All the tricks that can be useful but that obfuscate the library are avoided. All images are 4D and the data type is a parameter of the template.

- Vigra: the genericity 'a la STL'.

  Vigra stands for Vision with Generic Algorithms. It is under a license "which is modeled after the Perl Artistic License and thus more liberal than the GPL", and has been developed by Köthe (2000) during its PhD thesis.

  The goal of this library is to be easily adapted to the needs of the user, without a cost at execution time.

[1]http://www.greyc.ensicaen.fr/ regis/Pandore

It uses template techniques similar to those used in the C++ Standard Template Library [Köthe (2004)].

This library is made for 2D images. As STL does, Vigra proposes containers, and basic functions, but does not offer to the user many complex algorithms.

- Horus: a mature and commercial software.

  Horus is developed by the Intelligent Sensory Information Systems (ISIS) Group.

  It is designed for image and video analysis. CORBA (Common Object Request Broker Architecture) is used to allow interaction in a pool of computers.

  Horus wants to be efficient by a heavy use of the C++ template mechanism,

  It is not a free software.

- Olena: the genericity.

  Olena is the *LRDE* generic image processing library. Olena is free software: it is released under the conditions of the GNU General Public License version 2.

  It has been developed since 1999, and aims at being generic with respect to the data types, but also to the dimension of the image.

  Furthermore, performances have not to be forgotten, that is why it is based on the *SCOOP* [Burrus et al. (2003)] paradigm which proposes an efficient usage of static hierarchies.

2.2 Capabilities comparison

In this section, the previously presented libraries will be compared on several criteria. These criteria have been chosen in order to show what can be wanted, in an extensibility point of view, when a library is used.

2.2.1 n Dimensions

Most of images are 2D, or 1D (a 1D image is a signal). 3D images are also used in medical image processing applications. Some libraries work only for a given dimension. Other libraries use a high dimension, and use the default value 0, for the dimensions that are not used. Like it will be shown, a better approach seems to have the dimension within the type of the image.

Cimg

Cimg provides only 4D images. By default, the depth and the hight are set to 1, for example in the constructor or the *operator()*. Adding a new dimension would change a large number of function prototypes and the core of the function. Moreover it rises problems of typing. 1 to 4D images are an instance of common class *Cimg<V>*. A list of image (Cimg) can hold as well 1D image and 4D images.

The good point is that the code is well factored in macros. So as to avoid writing 4D loops or 4D manipulations of data, macros are used. The figure 2.1 presents an example of a piece of code extracted from the function *get_erode*.

Cimg_mapV is used to run the algorithm on all the channels (for example the red one, the green one, and so one). *cimg_map3x3x3 (*this,x,y,z,k,I)* loops on the image using x, y, and z. It

```
CImg dest(*this);
cimg_mapV(*this,k)
   cimg_map3x3x3(*this,x,y,z,k,I)
    if (!Iccc && (Incc || Ipcc || Icnc || Icpc || Iccn || Iccp))
      dest(x,y,z,k) = tmax;
```

Figure 2.1: Neighborhoods and loops in Cimg.

assigns to many variables a point in the image. For example Icnc corresponds to the point that has the following position[2]: $(x+0, y-1, z+0)$.

These macros can be easily changed, but each time a dimension is added, most algorithms become slower. In the previous example, in 1D the points of the structuring element that are not on the Y and the Z axis appear in the loop. It slows the computation.

But the cost of this design is compensated by the simplicity of the code. Knowing the number of dimensions allows the library to unroll some loops using macros.

Vigra

Vigra is made for 2D images. It does not provide function or data types for other dimensions. But we think that its design enables a programmer to add these features. All the types, such as *PixelType*(point), *Iterator*, or *size_type* are defined by a typedef in the image class, and it should be easy to make 3D image types compatible with this paradigm. Unfortunately, because the library is 2D only, most of the algorithms do not use the typedef, but use directly the underlying type. For example *difference_type* is often replaced by *Diff2D* (simpleSharpening, differenceOfExponentialEdgeImage, differenceOfExponentialCrackEdgeImage, moveDCToCenter are some concrete examples). In addition, algorithms are written using double loops (rows and columns) and should be changed to work in another dimension.

Even if this version of Vigra only works with 2D images, its design provides enough tools to allow to write a core that works in any dimension. But the algorithms are written only for 2D images, using for example a loop on the columns, and a loop on the rows. Thus it is difficult to adapt and modify Vigra so as to use 3D or 1D images.

Horus

Horus is designed to work on 1D, 2D and 3D images. An extension of the library to support nD images may be difficult because of the complexity of the hierarchy. Furthermore, even if the maintainers try to do it, it seems very intrusive in the core of the library.

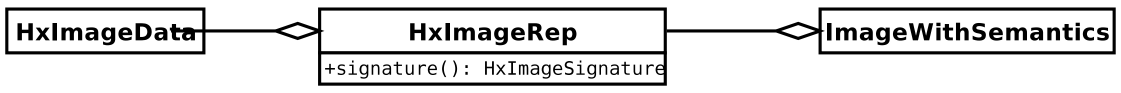
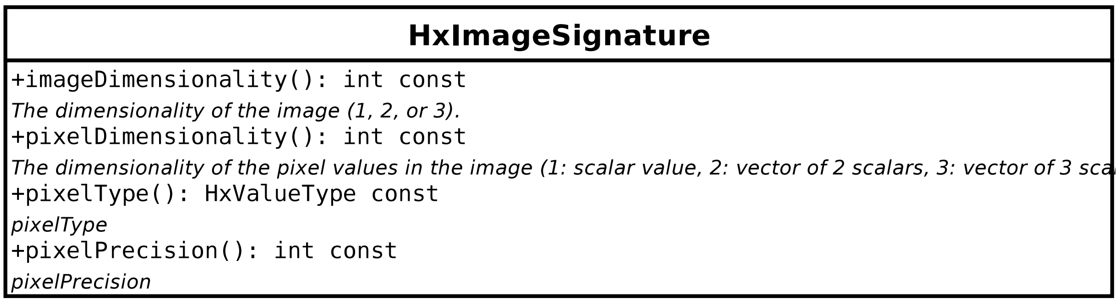


Figure 2.2: Image modeling in *Horus*.

As a matter of fact, all the images in Horus are represented by the *HxImageRep* class (Fig. 2.2). All the algorithms take this type in input, make dynamic checks using its *signature* to be sure the

[2] c stands for center and n for negative.

image has good properties before doing their job. This *signature* aims at providing informations about what is really the image (Fig. 2.3).



HxImageSignature

```
+imageDimensionality(): int const
The dimensionality of the image (1, 2, or 3).
+pixelDimensionality(): int const
The dimensionality of the pixel values in the image (1: scalar value, 2: vector of 2 scalars, 3: vector of 3 sca
+pixelType(): HxValueType const
pixelType
+pixelPrecision(): int const
pixelPrecision
```

Figure 2.3: *Image signatures* in *Horus*.

It can be difficult to add new signatures that are compatible with the existing code: for example the code of *factories* used to create images should take into account the new signatures. Furthermore, the greatest part of the library source code is not available (because of the license), it is a problem when you have to be intrusive in the code.

Olena

Olena has been designed to be generic with the data type used, but also with the dimension of the image. Thus 1D, 2D and 3D images are natively supported.

Nonetheless, the nD image type is not implemented. Actually, there is a specific implementation for each dimension of the image, but none for dimension greater than three. It would have been more interesting to provide an implementation for nD and to use it for the common dimensions (1, 2 and 3). Moreover, the specialization could have been used to provide optimized version for critical method of these dimensions. But this is not really a problem, because most of people do not use images with a dimension greater than 3.

Despite this aspect, algorithms are written to be generic with the value and the dimension. Thus, even if 4D images, are not implemented, if someone does it, the existing algorithms will work with this new type. Doing such a thing is not really difficult because of the design: it is one of its goals.

The approach of Olena has several advantages. First some errors can be detected at compile time. For example, the piece of code of the figure 2.4 will not compile:

```
image3d ima;
ima(x, z) = 4; //error, ima is not a 2d image.
```

Figure 2.4: Invalid assignment in Olena.

If the dimension is part of the type, it is also possible to statically specialize a function. It is possible to write a rotation which works only for 2D images. Furthermore, if a high dimension is used such as in Cimg, only the access based on iterators are efficient in small dimensions. The access based on operators such as (x, y) are slow, because what is actually written is $(x, y, 0, 0)$. If the dimension is a part of the type, the access is efficient in all cases.

2.2.2 Value of the pixel

Depending on what you are working on, different types of value are used. Some libraries propose only a few pre-defined types, such as *unsigned char* or *float*. Other libraries let the user use his own types. You will see how difficult it is to introduce a new type in the different libraries.

Cimg

The type of value of pixels is the parameter of the Cimg class (the image class of the library). There is no trait that uses the type. It enables a good portability, and it is not necessary to write traits in order to use the library with a new type.

But the library assumes that many operations can be called on the type of the value of the pixels. For example the addition, the 0 and 1 values are expected in many algorithms. These operations have no prior meanings for some type: what does *true - true* means?

There is no specialization, and no assertion. For example, the erosion is an algorithm which works with gray level images or binary images. In this library, if it is called on a color images, it is ran on all channels, one after the others. Once again, Cimg provides a simple solution, not too far from MatLab.

Vigra

The images are templated by the value type. Some traits allows to promote the types. For example if a mean is done on a large set of data, it is useful to make sure it is done using a floating point type to compute the sum. It is easy to add traits, and to add your own types.

As it will be shown in the subsection 2.2.4, the access to a pixel is suitable. It separates the reading and the writing of values. This is useful in some cases. For example, it is possible to have different assertions in the two cases.

Here are two examples of situations in which it useful. If a user needs to write a data type that has the following behavior: it is equal to 0 by default, but the user can not set it to 0. Another example is to automatically threshold the value set by the user to a pixel.

Horus

Horus separates the semantic, the representation, and the implementation of objects [Koelma and other ISIS members (2003)]. The implementation is the storage of the data. The representation provides a neutral view of a concept. It provides functionalities to deal with a concept. The semantic gives properties to the pixels. For example, a representation of an image of unsigned integer, can have several meanings: an image of intensity of a monochrome light, or a distance in a distance map. The following items are presented by Koelma and other ISIS members (2003) in their web site as being semantics in Horus:

- intensity images: the pixel value indicates a monochrome light intensity,
- color images: the pixel value represents a color (RGB, HSI, etc.),
- X-ray, ultrasound, or electron microscope images: a pixel value depends on object density or another physical phenomena,
- satellite images: the pixel value represents a recording of up to 7 spectral bands,
- range images: the pixel value indicates a distance,

- characteristic images: the pixel value indicates whether the pixel is element of a set,
- flow fields: the pixel value represents a motion vector,
- complex images: FFT domain.

The separation of semantic and representation can prevent some bugs. For example it avoids mixing range images and intensity images.

The representation of a pixel value in Horus is a scalar value or a vector of n scalar values. A scalar value is represented by one of the following [Koelma and other ISIS members (2003)]:

- a k bits integer value (bit, byte, short, int, ...),
- a k bits floating point value (float, double, ...),
- a complex number.

Horus separates data types and arithmetic data types. It provides more data types than arithmetic data types (16 vs. 7). For example, the arithmetic data type *HxScalarInt* is used for several data types, such as *short*, *int* or *unsigned char*. Having small data types such as *unsigned char* or *float* allows to use images that require less memory. Having less arithmetic data types reduces the number of template that must be instantiated. Furthermore, the operations with large data types (for example *int*) do not significantly cost more CPU time than smaller one (for example *short*).

Cast are available for all data types. For example it is possible to cast a *Complex* $d = (d1, d2)^T$ to a *Vec3Double*. In this case, the *Vec3Double* is equal to $(d1, d2, 0)^T$.

A lot of operations are defined for all arithmetic data types. For example *square root* or *tangent* are defined even for vectors.This makes the creation of a generic function easier. Only the *complement* is defined only for a set of type (integer types).

It seems hard to add a data type to Horus. If it is not compatible with an existing arithmetic data type, a new arithmetic data type must be added. It might not be a good idea to have a lot of arithmetic data types, because it increases the number of instantiations.

Olena

Olena aims at being generic with the data types of the images. That is why there is no problem to use it with either pre-defined data types, or user's own types. In the last case, the user may have to define some traits to make them compatible with the existing code. For example, if an algorithm needs to know the *max value* of the data type, the user will have to define the corresponding traits (*optraits<user_type>*).

Furthermore, Olena comes with a library of types named Integre. This library provides safe basic types that go from int_u8 (integer coded on eight bits) to rgb_u8 (vector of three int_u8 components). The main difference with the C++ pre-defined types (unsigned char, integer, float, etc.) is that these types provide checks. For example if there is an overflow, the user will be warned. This may help him to debug his program. Moreover, the behavior of these types can be changed: you can choose to work on a modular type ($\mathbb{Z}/\mathbb{Z}^{128}$), or saturated ones (when the limit is reached, it will stay to the extremum value).

2.2.3 Grids

Most of the time images are based on rectangular grids. It explains why most libraries do not give other implementations. But some none rectangular grids exist such as hexagonal ones.

More common needs are neighborhood relationships that are different from the standard ones. For example, it is often useful to have circular images.

Horus

The abstraction of the algorithms should allow to use other kinds of grid. Horus has algorithms such as 'for all points, for all neighbor do F'.

Vigra

Vigra is done for 2D rectangular grids. It is possible to add another kind of grid, but like for the dimension, the algorithms must be re-written. Furthermore, some modifications have to be done to the core, and a large amount of additional code has to be written.

Olena

Adding new grid in Olena can be done by implementing *images*, *points*, *dpoints* compatibles with this new grid. It should be easy to insert it into the existing hierarchy, and thus to make the existing algorithms work with. Nonetheless, this is not a trivial task since a large amount of code have to be written in the core of the library. Even if it can be easy to a developer familiar with Olena, if a simple user wants to do it, he may have many difficulties to do it cleanly.

2.2.4 Iterators

Another critical step is the way data are retrieved from the image. Most of the time several kinds of iterators are proposed, such as forward and backward ones. It can be interesting to let users propose new iterators. For example an iterator which goes only through the points that match a criterion.

Cimg

The *operator()* can be used to read or write the data of an image. There is no structures for points. A direct access to the underlying buffer is possible using an unsigned integer.

The Cimg library avoids a class of iterators and points. It makes the library more simple. It avoids all problems linked to iterators, for example invalid pointers.

To iterate on images, a lot of macros are used. For example the macro *cimg_mapXZ(img,x,z)* (Fig. 2.5) is used to iterate on the x and z axis:

```
#define cimg_mapX(img,x)     for (int x=0; x<(int)((img).width); x++)
#define cimg_mapZ(img,z)     for (int z=0; z<(int)((img).depth); z++)
#define cimg_mapXY(img,x,y)  cimg_mapY(img,y) cimg_mapX(img,x)
```

Figure 2.5: Cimg macros used to traverse an image.

It makes it hard to run an algorithm with a new iterator.

Vigra

The iterators used in Vigra are described by Köthe (1998). This paper explains that the STL iterators are not suitable for several kinds of storage of images. A solution is proposed to solve the problem of the references to the pixels.

The first problem, the difficulty to return a value by reference, will be explained using the example of Köthe (1998). This example deals with color images. Let us define the RGB color type (Fig. 2.6) which represents the red, green and blue layer.

```
struct rgb
{
  rgb(float r, float g, float b): r(r), g(g), b(b) {}
  float r, g, b;
};
```

Figure 2.6: RGB data structure.

There are two different ways to store an image of RGB pixels as shown in figures 2.7 and 2.8.

```
struct image_of_rgb
{
  vector<rgb> vec;
  [...]
};

struct image_of_rgb
{
  vector<float> red;
  vector<float> green;
  vector<float> blue;
  [...]
};
```

Figure 2.7: Different implementations of a color image.

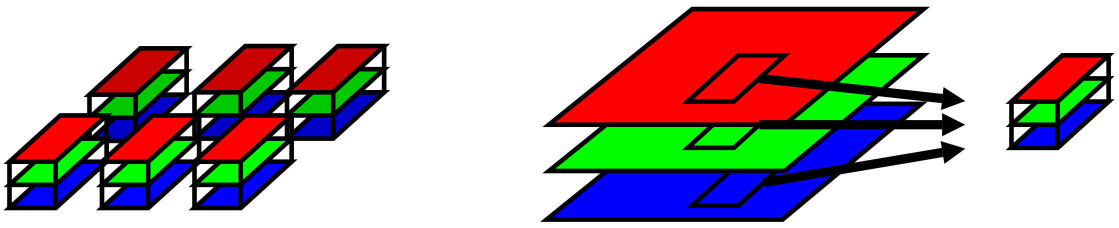


Figure 2.8: Graphical representation of data structures of the figure 2.7.

Our goal is not to discuss in which case one is more efficient than the other, we focus on the data access. In the first case, it is easy to return values by reference (Fig. 2.9).

```
struct image_of_rgb
{
  vector<rgb> vec;

  [...]
public:
  rgb                         //by value
  operator[](index i) const {return vec[i];}
  rgb&                        // by reference
  operator[](index i)       {return vec[i];}
};
```

Figure 2.9: Pixel accessor returning a valid reference.

But in the second case, the RGB value is computed on the fly. After the call to the method, the object is destroyed. Thus the reference points on an invalid address as it can be seen in the figure 2.10.

```
struct multi_band
{
  vector<float> red;
  vector<float> green;
  vector<float> blue;
  rgb operator[](index i) const {
    return rgb(red[i], green[i], blue[i]); //By value. OK.
  }

  rgb& operator[](index i) const {
    return rgb(red[i], green[i], blue[i]); //By reference. FAIL.
  }
};
```

Figure 2.10: Pixel accessor returning an invalid reference.

It is possible to use a *set* method and a *get* method to avoid the return by reference. It would also be possible to return by copy a proxy that holds a reference on the data. But it has many drawbacks: it slows down the function, it is difficult to write and hardly usable by the user [Köthe (1998)].

Vigra proposes a solution that allows to change the way the value are *set* or *gotten*: the accessors. The figure 2.11 presents an example.

In this example, iterators src and $dest$ are used to iterate on the image. But in contrary to STL, the iterators are not used directly to read or write the data. They are used through the accessors $destacc$ and $srcacc$. This method is powerful, because the same algorithm can be used to copy all the values of the pixels or to copy only a component of a color image: only the accessors

```
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);
}
```

Figure 2.11: Copy algorithm in Vigra.

must be modified.

Olena

Olena provides iterators to traverse an image. These are quite different form STL ones because they do not behave in the same way. Here the iterators iterate through a set of points, a domain. Thus they do not have a reference to the data. They hold the domain of the image, and the current point instead. This current point is used to read or write the data in the image. An iterator can be used for several images, with a restriction: the images must have a domain which includes the domain of the iterator. This is an interesting thing because an iterator can be used to iterate through several images which have the same size.

It has similarities to the Vigra approach, with the accessors, but in Olena, the accessor is the *operator[]* of the image. Vigra allows to use different accessors. For example it is possible to use an accessor which returns only the red channel of a color image. In Olena, this can be done by changing the *operator[]*. This can be done using morphers.

The Vigra approach is interesting because the access to the data and the image are not correlated, which is not the case here. In Olena, a reference to a pixel is returned. Most of the time it is fine, but when the data do not physically exist, it might be a problem like it has been seen previously. For example, a RGB image can be viewed as a gray scale one, if a function which convert RGB to gray scale is defined. This encapsulation can be seen in figure 2.12. A shell translates every access to the data into the output format. If you want to assign a gray value to a pixel (each of the RGB components has to be set to this value), you will not be able to easily use pointers or references. A solution to this problem, is to return a proxy or to use *set* and *get* methods.

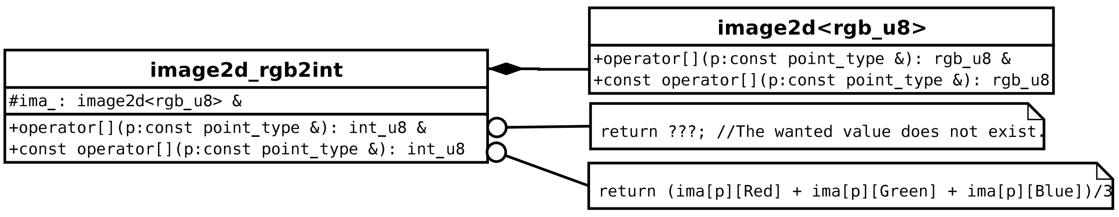


image2d<rgb_u8>
+operator[](p:const point_type &): rgb_u8 &
+const operator[](p:const point_type &): rgb_u8
image2d_rgb2int
#ima_: image2d<rgb_u8> &
+operator[](p:const point_type &): int_u8 &
+const operator[](p:const point_type &): int_u8
return ???; //The wanted value does not exist
return (ima[p][Red] + ima[p][Green] + ima[p][Blue])/3

Figure 2.12: Point access problem for Olena.

2.2.5 Storage

The common storage is a vector, but different implementations exist, for example implementations based on hash tables. It might be interesting to use the storage of another library, for example the containers of STL or an image type of another library.

Cimg

The grid is based on a vector. Even if images are 4D, the *operator[]* can be used with an integer that is the index in the buffer. The vector is a "T*". This is simple, and it can be easily binded with the LAPACK (Linear Algebra PACKage) library [Anderson et al. (1999)].

Once again, the library is simple and provides an efficient access to the image, but it is hard to change it.

The Cimg has the approach of MatLab or LAPACK. The user know what the data represent. For example an integer can represent a frequency. The user knows the goal of the algorithms and the data that it expects. For example only positive values. The user is also aware of the numerical problems, for example integer overflows. In exchange, the code is simple, and prototypes can be written quickly. It has no cost at execution time.

Vigra

The storage in Vigra is done using a buffer in the *BasicImage* class. The allocator is a parameter of the class like the STL containers. Thus, this library does not provide any tools to change the storage. You must write a new image class for each different storage. This code has to implement some functions that are not specific to your storage. An example is *size*, which does not really depends on the underlying storage. But the good point is that it does not modify neither the core nor the algorithm[3]. It is the same using STL, for example std::sort can be easily used on a container that has only little to do with STL containers (no inheritance from a STL container).

Horus

The storage mode used in Horus is very basic. Actually, it is a simple array of the value type used. Furthermore, even if the way to iterate through the data does not assume anything about its real representation in memory, it is not an easy task to make the library support another way to store data. Like it has be seen for the nD images, such changes would result in rewriting some piece of existing code.

Olena

In Olena, the storage and the interface of the images are separated. Thus the design allow to easily implement a new image type based on another storage mode.

2.2.6 Specialization

The specialization is useful to have different implementations of an algorithm depending on the input images. For example, if the image is stored in a linear buffer, the copy of this image can be done using *memcpy* which is efficient. The user might want to use specializations on any

[3]Of course, the implementation must have the same interface than *BasicImage*

criterion of the image. He might want to use several criteria as well, for example a version of a function which works on images that are 2D and binary.

The specialization can be used to restrict the entry types of a function, but this point will be discussed in the next section.

Horus

The choice of the algorithms that have to be called in Horus is done at the execution time. This principle is quite unusual and is presented in the figure 2.13. To call a function on an image, you will have to know what kind of operation you are working on[4], then you will have access to the database of the concerned functors. In another hand the image will give its signature[5], which combined to the function name will generate a key. This key is used to get the good functor in the previously retrieved database. Then this functor can be called on the image to get the result of the process. Thus to give a specialized version of an algorithm, you have to refer it in the good database with a key corresponding to a signature.

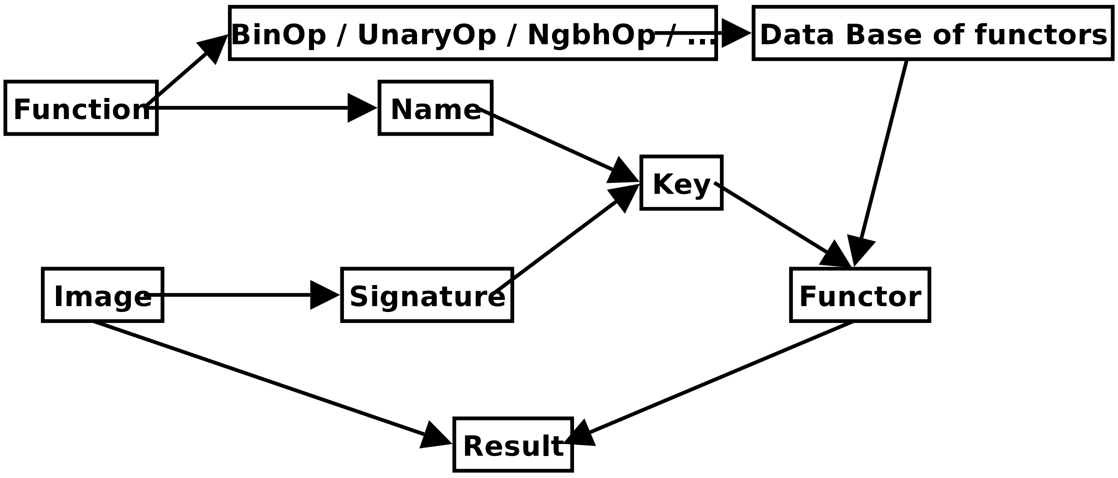


Figure 2.13: Dynamic specialization using Horus.

Vigra

The specialization with Vigra, can be done in the same way it is done with STL. Like in STL, it is possible to specialize a function using template specialization.

Olena

Olena proposes a hierarchy of images with diamond inheritance. This kind of inheritance is used to have several orthogonal discriminants in the hierarchy. Thus, *image*, *vectorial image*, *image 2D* are abstract classes that can be used to restrict the prototype of a function. Furthermore, the *SCOOP* paradigm [Burrus et al. (2003)] used in Olena allows to reinterpret statically the

[4]The authors of Horus give an exhaustive list of every kind of operator (meta-algorithms) that can be used in image processing: binOp, UnaryOp, . . .

[5]Every information that is related to the image: its dimension, information on its pixels, . . .

type of an object. It is then easy to use the *C++ overloading* to choose the optimized version of an algorithm.

An example of such a procedure is presented in the figure 2.14. The user calls *algo* on his image, the function will then retrive the real type of the image (go to the leaf of the image hierarchy), to call the implementation corresponding to that type. In the example, if *algo* is called with an *image2d<int_u8>*, the called version of the algorithm will be the first one. On the contrary, if an *image2d<bin>* is used, the second version will be called.

```
template <typename Exact>
void impl_algo(abstract::image<Exact> &im)
{
  // Basic version of the algorithm (slow)
}

template <typename Exact>
void impl_algo(binary_image<Exact> &im)
{
  // Specialized version of the algorithm (fast)
}

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

Figure 2.14: Algorithm specialization in Olena.

2.2.7 Bug reporting / strong typing

Often, the genericity obfuscates error messages produced by the compiler.

Another problem are bugs in templated functions. Some of them are not detected by the developers, and appears when the user calls the function with his own data type.

You will see in this section how the restrictions on algorithm prototypes are done.

Cimg

The Cimg is not strong typed. It is difficult to make some checks at compile time. But Cimg is simple. Only a few base classes are available. Thus, the user easily understand the library.

Olena

The *SCOOP* paradigm [Burrus et al. (2003)], which is used in Olena, aims at providing more restrictions than the STL way of programming. Actually, there is a representation of the image hierarchy which is used to restrict the input of generic algorithms. On contrary to STL, the user will not be able to call an algorithm with something that is not an image. This check is useful, to avoid some wrong function calls that would result in obfuscating error messages.

Furthermore, It has been said in the subsection 2.2.6 that the image hierarchy of Olena takes into account several orthogonal discriminants. Thus, this property can be used to check the input of a function. If the image does not verify some properties, there will be no matching function prototype, and the error message will be *human-readable* (it will appears in the function call, and not in the core of the function).

An example of such a prototype is presented in the figure 2.15. Here the compiler will not complain about not finding a method on the data type. Instead, it will just yield that there is no matching function called opening that takes a *vectorial image*. This last type has been deduced automatically from the *rgb_u8* type.

```
template<class I, class E>
oln_concrete_type(I)
opening(const abstract::non_vectorial_image<I>& input,
        const abstract::struct_elt<E>& se);

int main()
{
  oln::image2d<ntg::rgb_u8> im;

  opening(im, win_c4p());
}
```

Figure 2.15: Example of type restriction using Olena.

2.2.8 Adding new algorithms

If the image processing library works only with 2D images of *unsigned char*, it is intuitive for a developer to write a new function. If the library is generic, the developer has to do some effort to change the way he writes algorithms. This capability is subjective but is one of the most important.

Cimg

Cimg focuses on the user, and is designed to let him write easily a new algorithm. Even if it is hard to change the image class, it is easy to add a new algorithm. Macros are provided to factor the code.

Most users are satisfied by 1 to 4D images that support float, integer, arrays, and other simple types. Cimg provides these functionalities, and is fast.

Horus

Horus uses several meta-algorithms to write its algorithm. The meta-algorithms of Horus are [Koelma and other ISIS members (2003)]:

- Unary pixel operation. An unary function is applied to each pixel (ex: threshold).

- Binary pixel operation. An operation combines two pixels at the same point in two images (ex: sum).
- Reduce operation. A function takes the value of the pixels and return a value (ex: maximum of an image).
- Neighborhood operation. At each point computes a function that depends on a neighborhood (ex: median).
- Generalized convolution. It is a neighborhood operation that can be written using two binary functions (ex: gauss).
- Operation on the domain (ex: rotation).

So as to add a new algorithm, a functor must be written. This functor is linked to one of these meta-algorithms (inserted in the corresponding database), and instantiated for the set of types used in Horus. For more details on how to write practically an algorithm in Horus, you can see the user's guide [Koelma and other ISIS members (2003)].

Olena

To add an algorithm to Olena, you can write it either in a generic way or in a specialized way. The second way to process is the simpler one but is also the less interesting. Actually, it can be assimilated as using the library.

To write a generic algorithm, you will need to have knowledge about the way to do it. It is easy write a *semi-generic* piece of code. The figure 2.16 presents such a case: the function *foo* aims at setting an image to *zero*. The problem here is the *zero value* (understood as a neutral element) may vary depending on the data type the function is instantiated for (for example, an user type representing a monoid with a neutral element not equal to zero).

```
template <typename E>
void foo(abstract::image<E> &im)
{
  oln_iter_type(E)  it(im);

  forall(it)
    im[it] = 0; // <-- There is a problem here.
}
```

Figure 2.16: Wrong implementation of a generic algorithm in Olena.

Instead of using directly *0* in the code, a good implementation (Fig. 2.17) get this value by asking it to the type. In this case, the previously introduced user type will be usable by the algorithm (the trait called by *ntg_zero_val* has to be written before).

Furthermore, the error messages you may encounter can be more obfuscating than the one a simple user may have.

```
template <typename E>
void foo(abstract::image<E> &im)
{
  oln_iter_type(E)  it(im);

  forall(it)
    im[it] = ntg_zero_val(oln_value_type(E));
    //             ^
    //             |-- There is no problem anymore.
}
```

Figure 2.17: Good implementation of a generic algorithm in Olena.

Chapter 3

What can be improved in Olena

The goal of this chapter is to give some idea on what can be improved in our image processing library, Olena.

3.1 Value type of the pixels

Olena provides many data types that make possible to have strong typed value types. It is possible to express that the value of the pixels are between 0 and 100 for example. In our opinion this is fine, and doing better is not a priority.

The important thing is the semantic layer that stands on the top of these value types. An example is given in the figure 3.1. Generally, a color image is not a label image. The type of the components can be a *float* for example. But it is also possible that an image associates a label to a color. This is the case for some file formats such as GIF, which have a palette of colors. We want to express in the type that this image is an image of color an also an image of label. Thus the user can save it like a color image, and can run an algorithm that returns the number of occurrences of each label.

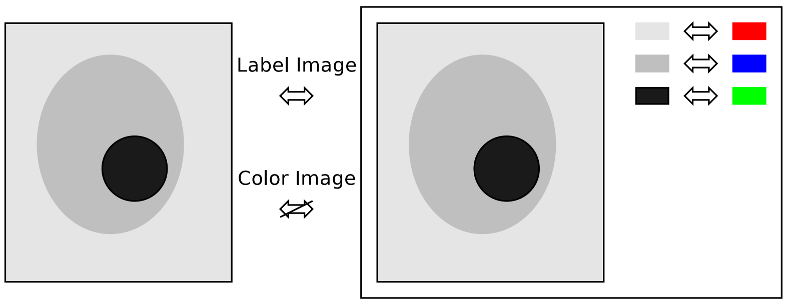


Figure 3.1: Label and color image with a color palette.

In Olena 0.10, if a new value type is used, it is easy to make it belongs to a given semantic type. For example, if the user wants to use the *std::vector<bool>* as a value type, he just needs to write a specialization of a trait which specifies that the *std::vector<bool>* is a vectorial type. The image of *std::vector<bool>* will automatically inherits from the vectorial image type. But in Olena 0.10, it is quite intrusive to add a new kind of semantic. The header $image\_with\_type\_with\_dim.hh$ must be changed. It is not a huge problem, but is could be a good thing to ease this process.

Right now, the semantics are general such as vectorial images, or scalar images. It could be interesting to add more specific semantics. Among these new types of images, gray tone images or color images could be useful.

3.2 Accessors and iterators

The accessors of Olena are the most intuitive of those presented in our comparison. They are safe and can be use on different images just like points.

The main problem with the accessors is the reference given by the non-const *operator[]* and *operator()* operators. It could be possible to return a proxy, but this method could yields to many problems [Köthe (1998)]. Using *set* and *get* operator makes everything very simple and solves the problem. But it is less human readable.

The morpher design pattern will be used in Olena 1.0 to re-use a given type of image, changing only a few properties, such as the type of the iterators. This can be used for example to have iterators which iterate only on the points given by a binary mask (figure 3.2).

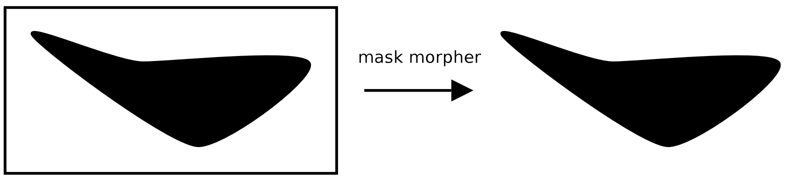


Figure 3.2: Domain of the image defined by a mask.

3.3 Storage

Several kinds of storage must be proposed to link our library with other ones. For example, it could be interesting for many users to have a binding with MatLab. The method used in Cimg could be used. In fact, we just have to check that the type of the pixels and the dimension is compatible with MathLab, to use a buffer for the storage, and to have access to the address of this buffer. The storage can also be used to wrap on external libraries such as fftw[1].

Furthermore, some morphers can be seen as images using other images as implementation. Because Olena 1.0 relies on a heavy use of such structures, an easy way to change the data storage has to be found.

3.4 nD images and Grids

There is no real nD images in Olena 0.10. Actually, the design of Olena 0.10 allows nD structures but they have not been implemented yet. Furthermore, it would be a good way to factor the code for images, points and vectors (*dpoint*) for the existing dimensions (1D, 2D and 3D) in a nD type.

[1]The fftw library can be used to compute a discrete Fourier transform. It is a free software.

In the current version of Olena, to use hexagonal grids, you have to specify a neighborhood that varies depending on the location of the point. For a given 2D image, it is possible to call two times an algorithm, the first time with a rectangular neighborhood, the second time with a hexagonal one. In other words, the neighborhood does not depend on the image. This approach can be assimilated to the float approach for data types (an image of *float* can be considered as a gray scale image or a percentage image). Olena aims at being a strong typed library, thus images and their possible neighborhoods should be correlated.

Nonetheless, so as to implement hexagonal images, it is possible to re-use the code of rectangular images, changing the neighborhood type using a morpher. The resulting image is strong typed, which is not the case in Olena 0.10.

Olena 1.0 should implement rectangular grids and grids based on graphs, to check that the design support non-rectangular grids.

3.5 Environment and new algorithms

Olena 0.10 provides already meta-algorithms such as Horus. They are rarely used. But in fact, it is because the iterators of Olena are simple. It is shorter to write a loop using the iterators of Olena than to write a functor. The documentation must explain clearly the way these meta-algorithms works with examples in order to push developers to use them.

The specialization of algorithms is implicitly handled by overloading. It works well in Olena 0.10 and it gives human readable error messages; which is not the case for STL.

Olena uses meta-code and many templates to be generic. This yields to two problems. The first one is the high compilation time. Several tricks can be used to solve this problems. Today, the compilers can compile the header once, and thus avoid to parse huge headers many times. It could be a good thing to use this pre-processed header trick. The second problem is that Olena works only with recent compilers. For example, Olena does not compile using Microsoft Visual 6. Once again it is not really a problem because Olena uses standard C++, and today the compiler manufacturers are working to release compilers which parse standard C++.

Conclusion

Two different ways to compare libraries have been presented: the functionalities and the capabilities. Our goal is to improve the core and the capabilities of the image processing library Olena. Thus, we have compared the facility to adapt several libraries to particular needs. Several points of view have been discussed. Cimg proposes a limited genericity but is simple. Vigra takes the drawbacks and advantages of STL. It also proposes clever solutions to retrieve the data. Horus clearly defines each concepts, and is built to be user friendly. But adding a new functionality may be a quite difficult task.

The design of Olena is relevant in comparison to other libraries. Some difficult functionalities can rely on morphers, which provides a way to modify the types without being intrusive. The core of Olena is big in comparison to other libraries, and the core must be factored. Because Olena is generic in regard of many concepts, it is important to clearly defined each of them like Horus does. For example, the user must know if an iterator can go through a point more than once in an iteration. Finally, Olena 0.10 offers a lot of genericity, without obfuscating error messages but also a complex core. It is a good thing and Olena must be simple, in order to be easily developed and accepted by the users.

Chapter 4

Bibliography

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.

Barton, J. J. and Nackman, L. R. (1994). *Scientific and Engineering C++*. Addison-Wesley, Reading, MA.

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*.

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

Köthe, U. (1998). On data access via iterators. Technical report. This technical report explains well the design pattern used in Vigra to access to the data, and why this design has been chosen.

Köthe, U. (2000). *Generische Programmierung für die Bildverarbeitung*. PhD thesis. In this thesis the design of Vigra is described; but we did not read it.

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

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

List of Figures