A programming environment for the development of modular 3D biomedical image processing applications

Release 0.00

Julien Montagner[1], Alice Villéger[1], Frédéric Flouvat[2], Jean-Yves Boire[1]

July 1, 2006

[1]ERIM, Auvergne University, Clermont-Ferrand, France
{julien.montagner, alice.villeger, j-yves.boire}@u-clermont1.fr
[2]LIMOS, Blaise Pascal University, Clermont-Ferrand, France
flouvat@isima.fr

Abstract

ImageLib is a free object-oriented development environment destined to biomedical image processing. The software package contains a programming library (C++ language) and an MDI user interface (kernel). The major interest is to allow the creation of fully modular applications. Both user processing functions and image read/write facilities are separately compiled as DLLs, and the resulting plugin objects are dynamically imported into the kernel. The library provides all the required wrapping material to incorporate all find of image processing functions into user-written components. A rapid prototyping application is provided to communicate with the kernel communication interface. No graphics programming skills is required to manage advanced user interactions, image display (*e.g.* OpenGL©), and generic image processing functions based on C++ templates. The component-oriented software architecture makes it possible to share image processing and handling resources for light Windows©processing applications.

Contents

1 Introduction

Many software applications have been proposed for image processing in the field of biomedical imaging [1] [2] [3] [4]. Most are composed of processing functions integrated into a graphic user interface for image handling and display, but do not allow the final user to incorporate his own processing procedures within this environment. The ability to write scripts often represents an intermediate solution to versatility [4], but the user code is then limited to a sequence of existing processing functions. Image processing environments that can dynamically import the required processes are far more common in commercial products, whether these modules are written by the user [6] or sold by the software editor [1].
We propose a free image processing package called ImageLib, which contains both a prototyping tool to support the creation and assessment of new processing functions and a dynamic environment allowing user module sharing. ImageLib is not a new image processing library, but a development package helping in the creation of image processing applications, possibly including functions from existing libraries in a coherent user environment.

This new version of the development tool initially proposed by Collin et al. [5] was written in C++ (C++Builder©environment, Borland©, Scotts Valley, USA) and designed to be highly modular. It has multiple goals: besides supporting the conception of image processing functions, as in the first version, the upgraded ImageLib aims at allowing users to easily share and select the processing modules they need, thus providing a framework for creating user-friendly software applications. User procedures are organized in independent and easy-to-maintain compilation units called ”plugins”. A central software component, called the ”kernel”, provides a unique graphic interface for dynamically importing processing modules according to specific user needs. ImageLib was initially designed to handle biomedical images, but is generic enough to be used as a framework for general image processing.

In this paper, we present both the general functioning of the ImageLib package and some of the technical elements that contribute to the modular architecture. An overview of the general architecture is given first, including a presentation of some technical solutions adopted for the modular decomposition. The next part focuses on communication between user plugins and the kernel application, through a detailed example of plugin creation. Finally, we address issues related to the currently available ImageLib version and shared plugins.

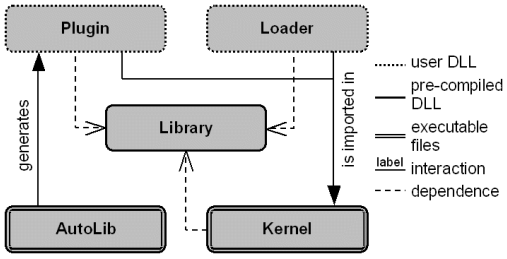

Figure 1: Global view of the ImageLib package

2 Software architecture

As in the first version of ImageLib, the development package contains both a library and a code generation program (called AutoLib). The first visible difference concerns the kernel application: this user interface is now compiled independently of all processing functions, and consists in an executable file provided in the package (see Fig. 1).

The ImageLib library has been compiled as a dynamic link library (DLL). It contains all the procedures and data structures required to implement the image handling functions of the kernel. It also provides software material for the creation of user processing functions.

The ImageLib kernel (see Fig. 2) is a multiple document interface (MDI) application based on the library, in which the user interface is made up of the parent window (the `TFormPrincipal` class see Fig. 3), and where MDI child windows are used to display open images (`TWnd` class). Both 2D and 3D biomedical images are displayed as simple flat pictures (slice-by-slice presentation for 3D images) in gray scale, and color palette management is a possible option. The parent window contains a static menu, allowing the user to access classical image handling functions (image opening/recording, display control, *etc.*), as well as other more specific functions (plugin loading/release, regions of interest, *etc.*). The menu bar also contains a dynamic part that, initially empty, fills when processing modules are loaded. Moreover, an image control dialog box is available to change the current slice (3D images) and adjust display (contrast and zoom).

2.1 A software application based on ImageLib

The object-oriented implementation and the architecture in several compilation units all contribute to the modular design of both the library and processing applications based on ImageLib. From a functional point of view, the tasks of such a software application can be divided into logical units:

- basic image handling and display
- management of image file formats
- advanced image processing functions

The first is common to all image processing applications based on ImageLib, and contains computing primitives available for the two other groups. Image handling and display functions are implemented in the library and reachable through the pre-compiled kernel. Processing functions are defined by the user in plugin files compiled as independent DLLs. Image read/write routines are also implemented in special plugin modules, called ”loaders” (see Fig. 1). Each loader object is dedicated to the management of a given image file format, and can be dynamically imported into the software application according to context and user needs.

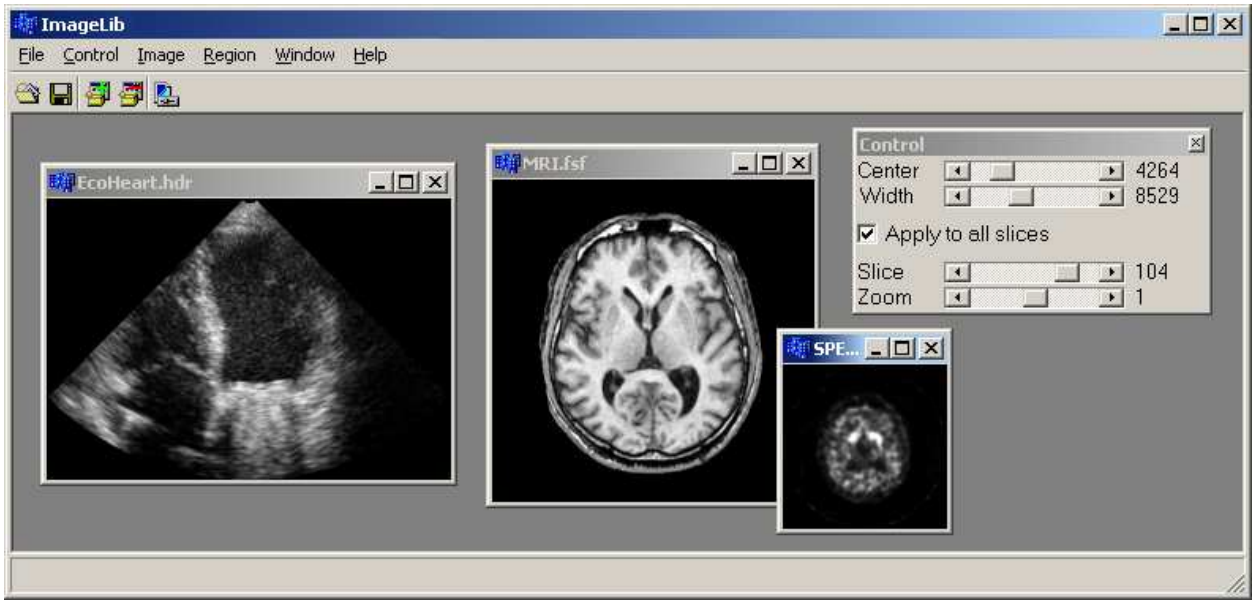

Figure 2: Window of the central application (kernel); three images are opened in MDI child windows, and the image control box is visible

Structure of the programming library

The central ImageLib component is the pre-compiled library. Besides the base functions supplied for user procedures, the C++ classes included in the library define the software interfaces required to ensure communication between all ImageLib modules (see Fig. 3).

The `TFormPrincipal` class inherits its graphical behavior from the Borland VCL©`TForm` class, and implements all the elements required by a classical MDI application using this base class. Moreover, it also allows to manage the particular MDI child windows associated with the `TWnd` class (management of the displayed `TWnd` list, creation of a `TWnd` object to display a new image, *etc.*). `TWnd` inherits from the `TForm` class too, and adds a specific image display function for the image data contained in the associated `TImg` objects.
The `TPluginLoader` and `TImageLoader` classes cooperate with the `TFormPrincipal` class, and are used

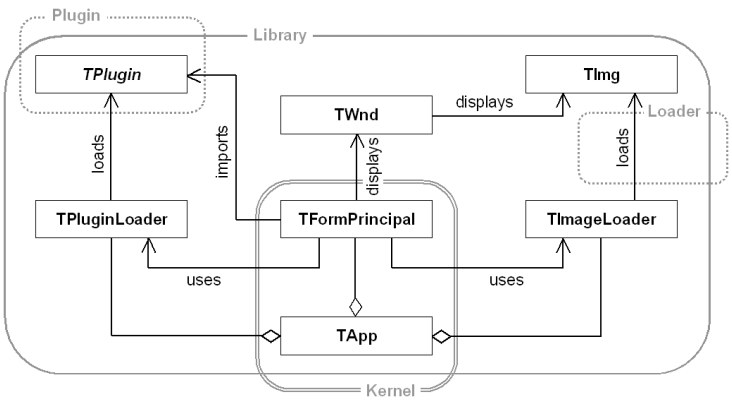

Figure 3: Simplified UML class diagram of the ImageLib library, with superimposition of the most concerned package components

as software interfaces dedicated to the management of plugin DLL and image files. The `TPluginLoader` class builds `TPlugin` objects from the user DLL and associates them with the kernel. The `TImageLoader` class receives all the image opening/saving orders from the kernel, and transmits it to the appropriate loader object, depending on the image file format.

Use of the library to create the kernel application

The executable file of the user interface is simply obtained by instantiating: a `TFormPrincipal` object a `TPluginLoader` object a `TImageLoader` object. For a given instance of the kernel, each object in the above list is unique. The operations they provide have to be accessible from each part of the user code, and not only through the user interface. Objects of this essential set are in fact gathered within a data structure that is reachable from the entire user code as a construction parameter of `TPlugin` objects. This global structure is an instance of the `TApp` (for ”application”) class (see Fig. 3).

2.2 Data structures

The object-oriented design is the first modularity feature introduced in ImageLib to cope with the problems mentioned in the introduction. The use of abstract data types leads to a program structure centered on data, and encapsulated processing instructions are easier to maintain since they are clearly located in the code and relatively independent from other parts of the program. Moreover, the inheritance mechanism introduced by the object implementation allows a generic approach to image processing. Indeed, both the nature of image data and contextual information linked to the image (*e.g.* patient or acquisition system) may be considered as specialization parameters for the general concept of image.

Generic handling of the images

The `TImg` class describes the structures which contain image data. `TImg` objects are characterized by the numerical type of the data array they contain (8/16-bit integer or 32-bit floating point value in ImageLib). Whatever the data type, they also contain information about the spatial distribution of image data (number of voxels) in the two (or three) spatial dimensions, and the anisotropy factor (3D images).

The template mechanism proposed by the C++ language provides a suitable solution to the problem of specialization by a simple data type. This technical solution makes it possible to apply processing routines written for general image objects to all kinds of data. However, C++ language is unable to handle pointer parameters on generic objects for which the instantiation type is unknown. This problem appears when trying to generate a prototype for a generic processing function, since main input and output parameters are `TImg` objects. The abstract `BaseTImg` class has thus been inserted as a `TImg` parent class in a standard inheritance relationship. `BaseTImg` is used as a non-template part of image objects and contains all the attributes that are independent of the image data type.
The second specialization of image objects by context dependent data is implemented by simple aggregation of an additional data structure of a user-subclass of `TSaveParams` within the instance of `TImg` (actually in the generic part of the object, *i.e.* the `BaseTImg` parent class). The `TSaveParams` object can be attached to the relevant image and filled by the loader of the relevant file format during the loading operation.

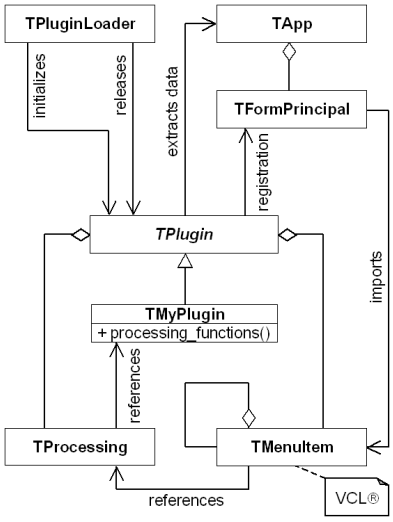

Figure 4: UML class diagram of the plugin communication system, from the plugin point of view

Data/visualization independence

The first ImageLib function using this generic mechanism is the image display. In order to respect the data/visualization principle, all the related procedures were separated from the `TImg` class, and assigned to the MDI child windows themselves, *i.e.* to the `TWnd` class. Since this graphical control class is unique, it displays all kinds of image data, and thus manages them through a `BaseTImg` object (the ”displays” relation between `TWnd` and `TImg` on Fig. 3 in fact heads toward the `BaseTImg` class).

2.3 Built-in components and programming resources

The library also includes other useful programming resources. These are proposed either to help the programmer writing C++ code, or to allow the integration of user interactions in processing functions. Besides built-in implementations of the loader abstract classes (FSF [5], raw and bitmap formats), ImageLib provides software material to handle graphical events on `TWnd` windows, manage regions of interest (ROIs), and others. A text-mode debugging tool is also available.

2.4 Technical solutions for modular decomposition

The decomposition follows the same scheme in each case: for each modular element, the library includes both a controller class (*e.g.* `TPluginLoader`) and an abstract base class (*e.g.* `TPlugin`) which acts as an interface between the controller and the user-written code (Fig. 4).

Interface of processing modules

Since the DLL technology is initially designed for the C language [7], and does not allow to export C++ classes. It remains possible to bypass this problem using a wrapping method. The DLL thus exports two C functions, called from the `TPluginLoader` object:

- `initPlugin()`: builds an instance of the `TMyPlugin` class, which inherits from `TPlugin` and contains the specific user code, and then calls the plugin registration method of the `TFormPrincipal` object of the kernel
- `closePlugin()`: calls the plugin release method of `TFormPrincipal` and deletes the `TMyPlugin` object.

Both the `initPlugin()` and the `closePlugin()` methods accept a `TApp` object as an input parameter. This object, thereafter used as construction parameter for the plugin classes, contains all the information required on the library objects built into the kernel application. It is thus used as the principal means of communication with the ImageLib library.

The base code of the `TMyPlugin` class is generated by the AutoLib tool from user specifications. Its methods are written by the user and, which forbids this class to be declared in the library. `TMyPlugin` inherits from `TPlugin`, but a polymorphic call to processing methods remains impossible, since their number, name, and prototype is different in all user plugins.
Plugin functions are associated with a menu, which is built directly in the constructor of the `TMyPlugin` class using `TMenuItem` (VCL©class) objects, and then stored in its generic part. References to processing methods are stored in the related leaf object of the menu tree. The entry point of each processing function is then transmitted to the kernel application when the registration method adds the plugin menu to the user interface.

Plugin structure and processing functions

All the methods of `TMyPlugin` possibly called from the kernel have the following prototype (P1):

```
void funcName( BaseTImg ** inImg, int nbIn, BaseTImg ** outImg, int nbOut );
```

The `inImg` and `outImg` parameters are the arrays of input and output images, respectively, containing `nbIn` and `nbOut` images structures. Both the input and the output arrays contain `BaseTImg` type data in order to allow the kernel to send images of all numerical types. Input images are selected by the user before the function call. Output images are automatically displayed in `TWnd` windows when the processing function returns.

The ability to write fully generic image processing functions requires to propose user methods with the modified prototype (P2):

```
template< class T >
void funcName( TImg<T> ** inImg, int nbIn, TImg<T> ** outImg, int nbOut );
```

The template parameter `T` defines the numerical type of image data contained in the input/output `TImg` objects. The compulsory prototype of methods runnable from the kernel requires the introduction of an intermediate function, with prototype (P1). This non-template method is called instead of the user-defined generic function. Its code (automatically generated by AutoLib) builds the appropriate instance of the processing method according to the type of input images, and runs it in a way that is transparent for the user.

3 Plugin creation and integration

The aim of this section is to illustrate the practical application of technical elements explained in the previous sections, from the user's point of view. The development process using ImageLib is of course influenced by both the software architecture and the chosen data structures, but many technical elements are hidden and embedded to make the programmer's task as easy as possible. We propose to detail this development process by studying the practical example of a basic image processing operation (3D morphological erosion on a grayscale set), implemented as a standard processing function (prototype (P1)), with a 3D/16-bit integer input image, and producing an image of the same size and numerical type. The generation of other kinds of prototypes will also be mentioned.

3.1 AutoLib

The AutoLib application is composed of an executable file and a set of resource code files used by the program to generate the user code. The executable file displays a graphical interface which allows the user to input all the parameters related to both the plugin entity and each processing function. The joined directory contains the files which compose the plugin skeleton, *i.e.* both the files required by the compilation step (Borland C++Builder©files), and the header/linkage files of ImageLib libraries.

User interface

The user interface of the code generation application is split into two parts (see Fig. 5). The upper part is dedicated to global plugin management, and is centered on the plugin menu. The name "Standard erosion", given to the processing procedure and the related menu item, refers to the lower part of the user interface. The lower part allows to input parameters of the selected processing function. The name of the processing procedure is modified in the "Name" field of this second area (see Fig. 5.b). This name only has a descriptive/graphical interest, and is associated with the string "`standardErosion`". This string is the name which will be given to the associated method, when the code of the `TMyPlugin` class will be generated. This name thus has to respect C++ code conventions. Since we are working under the hypothesis that the image to process will always have 16-bit integer data, the chosen method type is "standard", corresponding to prototype (P1). For this kind of method, as for the "generic" type (P2), the number of required input and output images is indicated in the two next fields (possibly 0). ImageLib also allows to write C processing functions, making such prototypes compatible with the previous version of ImageLib.

The second tab of the processing function description area is used to manage a list of tests performed on input images before the method is called by the kernel (*e.g.* the numerical type of input images, set to 16 bits in the example).
The third tab of the processing function area allows to choose the image generation mode (*e.g.* automatically or manually created in the body of the processing function) in the same way as the input conditions. In contrast to validity tests, this choice is compulsory for each image. In the case of the `standardErosion` method, both the type of the unique output image and its size depends on the input image. The output image is thus created with a "Relative size" creation mode.

Each plugin is managed as an independent project. AutoLib allows to save plugin projects to disk, and thereafter to open project files either to add new functions or to modify them, following the same process as the one presented here. After each modification, the plugin code has to be regenerated.

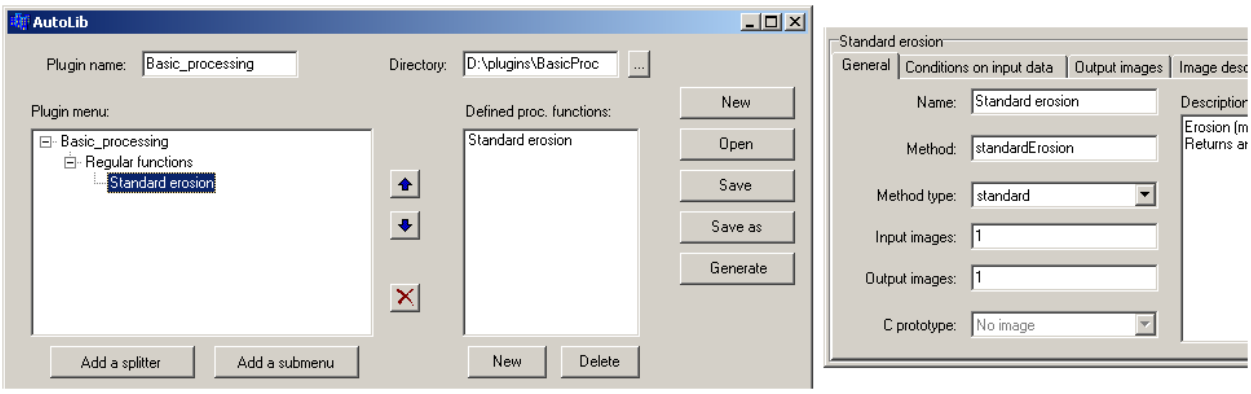

Figure 5: The graphical user interface of the AutoLib code generation application, split in two parts: the upper area (a) and a part of the lower area (b); example of a segmentation plugin specification

Generated files and user code

After the plugin is generated, the selected directory (see Fig. 5.a) contains a complete Borland C++Builder©project, ready to be compiled. Paths to the header files of ImageLib classes are configured in the project, and ImageLib libraries are also included. The majority of the plugin files has been copied from the plugin skeleton associated with AutoLib (and renamed according to the plugin name).

Only three files are entirely generated by AutoLib:

- `TMyPlugin.h`: declaration of the plugin user class, including the prototypes of processing functions
- `TMyPlugin.cpp`: body of the C wrapping functions of the DLL file, of the class constructor (building of both `TProcessing` objects and plugin menu), and of intermediate methods for prototypes (P2) and (P3)
- `processing.h`: skeleton of all the user functions.

User code is fully separated from all the technical classes. Thus, only one file has to be opened to write a processing function, making this task accessible even to programmers not well versed in object-oriented programming.

3.2 Modules importation

The visible result of a plugin loading is the addition of the related item to the menu of the kernel window (see Fig. 6 the example plugin has been modified to show the standard erosion function in a complete menu tree; the related menu item is disabled, since no image is opened, while the first function takes no input parameter). This item is dynamically deleted in the same way, when the plugin is released. Leafs of the menu tree structure are the entry points of the processing functions.

The example function only needs one input image. This image is thus automatically selected from the active `TWnd` window when the menu item is clicked. When the function needs more than one input image, a selection dialog box is automatically opened that displays the description strings of the images to select. The example function produces one output image, which is automatically displayed when the method returns. Multiple output images would be displayed the same way, whether automatically or manually created. If the

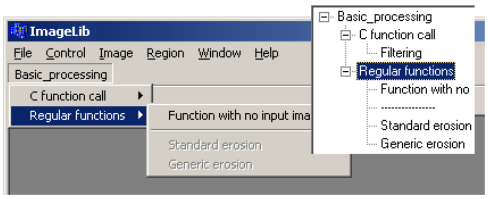

Figure 6: Plugin menu item and submenus available when the plugin is loaded; example of the modified segmentation plugin, with superimposition of the modified AutoLib area

data of an input image are modified, the linked `TWnd` window is also refreshed.
Instead of adding a menu item in the MDI parent window, importing a loader modifies the list of supported formats in the file open and file save dialog boxes.

4 Current version

The new ImageLib development package is commonly used by researchers of our lab as a shared development environment. The new version was only recently released, and is still less widely used than the first C version within the image and biomedical image processing research community. However, the modular structure of ImageLib allows us to share processing tools within our lab, not only with other ImageLib users, but also via the integration of processing modules into other environments. ImageLib has thus become the major collaboration tool with our research (*e.g.* French "Institut National de la Recherche Agronomique"), biomedical and industrial partners. All the known bugs have been fixed in the current release, which is the version we are using for now. However, ImageLib specifications are completed as and when new development needs appear in the field of imaging research, and when new ideas emerge for improving the proposed panel of development possibilities. The programming of new ImageLib versions is thus in progress, maintaining the same development principles while introducing changes, *e.g.* in data structures, but no fully stable version is yet available.

Availability

The whole ImageLib package is available on the website of our research lab[1]. This package contains the AutoLib application, the library DLL files, the executable file of the kernel application, and a document designed to help with the development of ImageLib-based applications. The released version has been extensively tested under Windows©2000 and Windows©XP, running on a large range of hardware configurations (desktop and laptop computers). All the software applications based on ImageLib were developed using the Borland C++Builder©6.0 environment, as was ImageLib itself.
Neither the library code nor processing plugin are publicly available on the website for now, but it remains possible to obtain some of them by contacting the authors.

5 Conclusion

In this paper, a new version of the ImageLib development package is presented. The main contribution of this work is to propose a fully new software architecture for image processing applications based on

[1] http://www.u-clermont1.fr/erim/

ImageLib. The C++ programming library and the code generation application make it possible to develop advanced image processing functions and easily integrate them into a common user interface managing all the graphical aspects of 2D/3D image handling. The simple development principle makes this task accessible even to programmers not well versed in neither object-oriented nor graphical programming. Moreover, this new version of ImageLib is fully compatible with formerly written C functions.
Excluding the central kernel application, all the components of an image processing application may be written by the programmer: processing plugins, image loaders and dedicated context data storage objects, new ROI shapes and user event handlers associated with image display windows. The inner data structures of ImageLib are designed using the C++ template mechanism. Consequently, user processing functions can be written as generic code. This new feature, while not affecting running performances, provides a convenient development principle leading to user-friendly applications.
ImageLib thus makes it possible to design convenient image processing applications with a professional feature and running even on low-end PCs, while it remains a fully free programming tool. We hope that a wide use of this common environment within the biomedical image processing research community will contribute to improve the resource sharing, possibly leading to propose the creation of an image processing module and code database.

Acknowledgements

The authors would like to thank Mr. David Couderc, engineer student, who was in charge of a large part of the ImageLib development. We also thank Mrs. Laurent Sarry, Vincent Barra and Christophe Tilmant for the help they have provided for both this work and the redaction of this paper.

References

[1] V. Barra, P. Briandet, and J.-Y. Boire. Fusion in medical imaging: theory, interests and industrial applications. In *Proceedings of the 10th Medinfo congress (London-UK 10)*, page 896900, 2001. 1

[2] M. Bosc, T. Vik, J.-P. Armspach, and F. Heitz. Imlib3d: an efficient, open source, medical image processing framework in c++. In *Proceedings of the 6th MICCAI conference (Montreal-Canada)*, volume 2879 of *Lect. Notes in Comp. Sci.*, page 981983. Springer, 2003. 1

[3] R. Clouard, A. Elmoataz, C. Porquet, and M. Revenu. Borg: a knowledge-based system for automatic generation of image processing programs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(2):128–144, 1999. 1

[4] Y. Cointepas, J.-F. Mangin, L. Garnero, J.-B. Poline, and H. Benali. Brainvisa: software platform for visualization and analysis of multi-modality brain data. *Neuroimage*, 13(6):S98, 2001. 1

[5] A. Colin and J.-Y. Boire. A novel tool for rapid prototyping and development of simple 3d medical image processing applications on pcs. *Comput. Meth. Prog. Bio.*, 53:8792, 1997. 1, 2.3

[6] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK software guide: the Insight segmentation and registration toolkit (version 1.4)*. Kitware Inc., New-York-USA, 2003. 1

[7] J. Richter. *Programming applications for Microsoft Windows (4th edition)*. Microsoft Press, Redmond-USA, 1999. 2.4