Annales UMCS Informatica AI 3 (2005) 159-167

Annales UMCS
Informatica
Lublin-Polonia
Sectio AI

http://www.annales.umcs.lublin.pl/

Development of the cross-platform framework for the medical image processing

Marcin Denkowski*, Michał Chlebiej, Paweł Mikołajczak

Laboratory of Information Technology, Maria Curie Skłodowska University, pl. M. Curie-Skłodowskiej 1, 20-031 Lublin, Poland

Abstract

This paper presents the development process of a platform for image processing with a focus on the medical imaging. Besides general image processing algorithms and visualization tools, this platform includes advanced medical imaging modules for segmentation, registration and morphological analysis. It allows fast addition and testing of new algorithms using a modular structure. New modules can be created by using a platform-independent C++ class library and can be easily integrated with a whole system by a plug-in mechanism. An abstract, hierarchical definition language allows the design of efficient graphical user interfaces, hiding the complexity of the underlying module network to the end user.

1. Introduction

Our image processing framework *SemiVis* was developed as a general platform for a medical pipeline, including data import, image processing and visualization. The main features of this system are its portability and ease of extension. Platform independence is achieved by using freely available cross-platform libraries like *Qt Toolkit* for user interface development and file handling [1] and *Kitware Visualization Toolkit (Vtk)* for image processing and rendering [2]. Extensibility is achieved by a plug-in mechanism. Developers can add functionality to *SemiVis* by implementing their own plug-ins and registering them with the framework. This can be done without recompilation of the source code. Since all core and plug-in classes are implemented in C++ the code can be used to generate executable programs on many systems.

The whole system is intended to be open source software published under a GPL license [3]. For that reason all file and structure formats used by this system must be open standards (for example XML [4], DICOM [5]).

* Corresponding author: *e-mail address*: denmar@goblin.umcs.lublin.pl

This defines the system as the universal platform for image processing and only the plug-in modules designate the use of this system in medical imaging. Together with the system core in the framework there are included modules for medical image processing for such purposes as:

1. data managing;
2. 2D and 3D visualization with interaction tools;
3. basic image processing [6]:
   – segmentation,
   – transformation,
   – filtering,
   – 2D and 3D editing,
   – analysis;
4. image and movie generation for demonstration purposes.

2. System project

Functionality of this framework is divided between two main parts: system core and modules. Even though structure of this system is complicated, its usage must be intuitive and conventional. Therefore both main parts, the core and modules, must not have too many use cases that interact with the user, or existing complicated use cases must be partitioned to form a hierarchical structure of simpler use cases [7]. Figure 1 presents the most general use case diagrams for two perspectives for both main parts of the system.

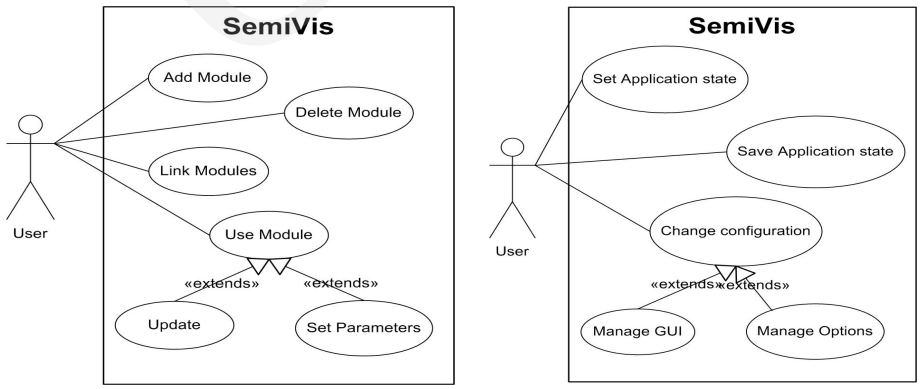


Fig. 1. Two use case perspectives for *SemiVis* framework

System Core perspective consists of three main use cases that are responsible for setting and saving application state as well as managing the application configuration. Module perspective involves four main use cases for managing modules, i.e. for adding and removing modules from the application, linking two modules and using modules. The last use case is extended by two other use cases: Update use case that releases module processing and Set Parameters use

case for changing the module configuration. Of course every module has its own use case diagram that extends the shown general diagram, but that diagram depends on module purposes and its designer.

The whole framework was created according to the paradigms of object-oriented design, and defines abstract, hierarchical and modular collections of packages, classes and components with all the essential communication protocols between them [8,9] (see package diagram in Figure 2). It is generally divided into five main packages responsible for specific functions:

1. *Main* Package – core management, application configuration and safety. This package is the major building block for the whole system, which is fundamental for an object-oriented framework.
2. *Frames* Package – user interaction for Main Package, i.e. all docking frames for logging, configuration, and information panels.
3. *ProcessObjectMan* Package – module management. This package is responsible for modules (plug-ins) loading, proper functioning and removal.
4. *Workspace* Package – maintaining graphical interface between user and modules, graphical representation of modules and connections between them.
5. *Modules* – all modules included into framework or added by the user, built according to the rules presented in the next chapter.

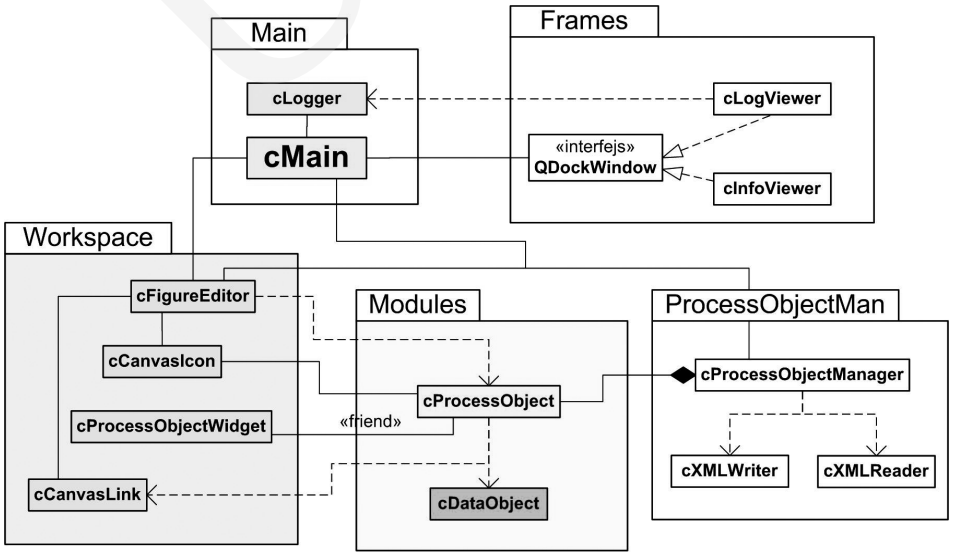


Fig. 2. General *SemiVis* framework package and class diagram

3. Modules

The whole framework is strongly module oriented and can not work without at least one module loaded. The interface between the system core and modules has been defined in a *ProcessObjectMan* package through a *cProcessObject* abstract class. This class offers operations common for all modules. Every module must derive from this class directly or indirectly. Every module class is also associated with a specific class derived from a *cProcessObjectWidget* class, which implements an interface for GUI representation of a module. The main principle of module function can be described as collecting data from its inputs, processing them and passing to its outputs. The module can have any number of entries and any number of exits. The modules can be connected into pipelines by connection exits from one module to the entry for other module(s), but there are some limitations:

- one exit can propagate to any number of entries,
- one entry can be linked only to one exit.

Data processed by modules are also encapsulated into classes derived from the common ancestor *cDataObject* class. The important feature of this class is the ability for sharing internal data between the objects of the same type. For that purpose a *reference count*, *deep copy* and *shallow copy* mechanisms are used. In the deep copy case a new copy object is an exact copy with all internal data of the original object and is completely independent. On the other hand, for a shallow copy only a small part of internal data is copied but larger data parts are shared by both the original and copy objects. This mechanism was implemented for memory saving reasons. It is easy to imagine how much memory is needed if processing of an object takes up 50 MB (average medical image data set), each module has its own copy and there are about 30 modules in a pipeline.

Figure 3 presents a simplified project perspective of module classes and data objects classes [7]. Class *cProcessObject*, as an ancestor for all module classes, defines operations that enable connecting output (*getOutput()* operation) from one module to inputs (*setInput()* operation) of other modules and operations that force data processing (*update()* operation). This class depends on the *cDataObject* class and is associated with the *cProcessObjectWidget* class by *«friend»* stereotype [7].

Modules can also be arranged into packages according to their specified functions:

1. input-output – modules for loading, saving, converting and maintaining various medical image standards (DICOM, TIFF, RAW [5]), other digital images (BMP, JPEG, PNG) and encapsulating these images into internal objects representation. An example of GUI representation is shown in Fig. 4a.
2. Information, statistics – modules responsible for various informational and statistical outcome generation (see Fig. 4b).

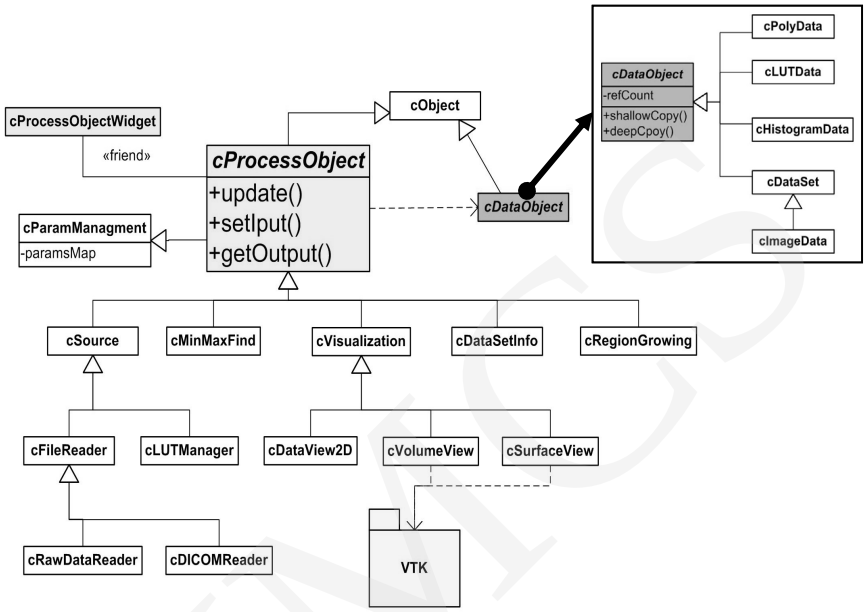
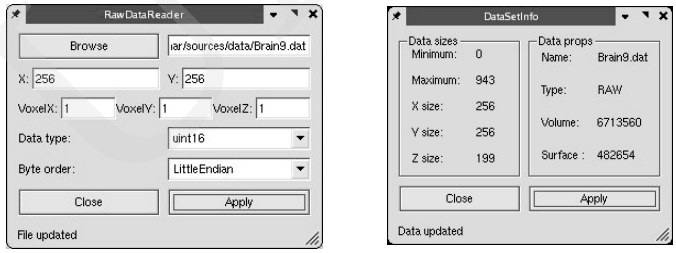


Fig. 3. *SemiVis* framework package and class diagram



a) b)

Fig. 4. Example of GUI representation of input-output a) and information b) modules

3. Visualization – modules for graphical representation of dataset:
   a) 2D – allowing two-dimensional visualization of volumetric data from the three orthogonal perspectives (transversal, sagittal and coronal), providing selection from a color lookup-table (LUT) and supplying the masking mechanism for the dataset (Figure 5).
   b) 3D – providing a set of three-dimensional visualization tools like volume rendering (raycasting, texture mapping), surface rendering (isosurfaces rendered as triangle meshes) rendering together with a wide range of manipulation mechanisms providing a good spatial orientation in volumetric datasets (Figure 6). These modules rely heavily on OpenGL library [10].

c) Graph visualization – histogram of dataset, scatter-plot, and graphical representation of various data relationships.
4. Segmentation – including modules for standard 3D image segmentation mechanisms (thresholding, region based, contour based, fuzzy logic).
5. Filtering – providing a set of morphological, convolution and diffusion filters.
6. Transformation – translation, rotation, rigid body, affine transformations, manual registration of two datasets.

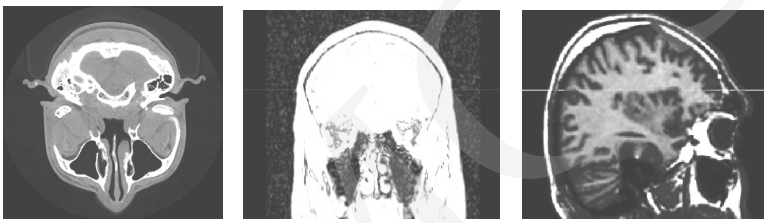


Fig. 5. Examples of two dimensional visualizations from three orthogonal perspectives

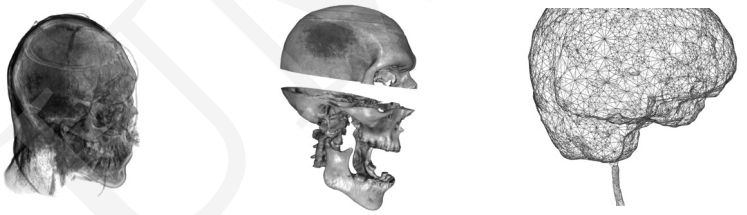


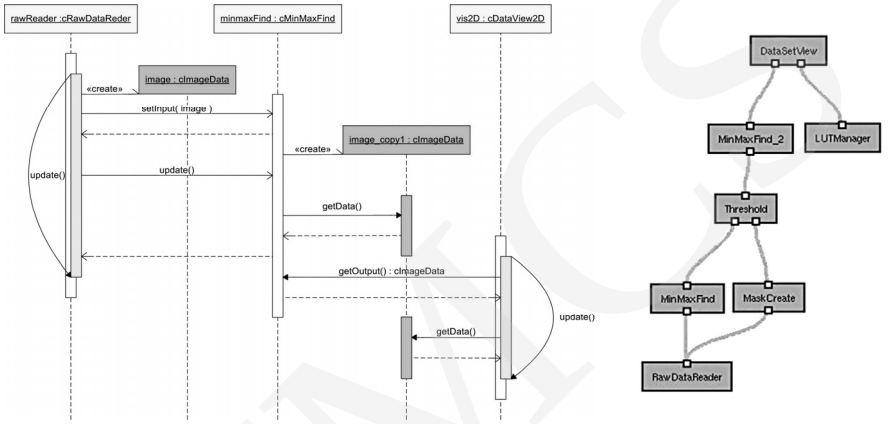
Fig. 6. Examples of three dimensional visualizations, from left: volumetric rendering, textured isosurface rendering, and isosurfaces rendered as triangle meshes

4. Data processing pipeline

Every module is responsible for one, specific function performed on entry data. To achieve an aggregation of functions of several modules we can connect them into pipeline. Modules can be linked either in series or in parallel. The aggregate action starts from the bottom module and propagates to upper modules as far as the current processed module is configured to be auto updated.

An examplary action of 2-dimensional visualization of a data set is shown in the sequence diagram in Fig. 7a. The operator *User* starts the sequence by updating (*update()* operation) *rawReader* module. This module creates a data object *image («create»* signal*)*, loads data set from file and converts it into an internal format. In the next step the *rawReader* puts a properly prepared data object into the entry of the connected module and forces it to update (the *AutoUpdate* flag of this module is set to *true* value). This module creates its own shallow copy of the data object and processes these data. The third module in the shown pipeline is *vis2D* and has the *AutoUpdate* flag set to *false* so that its

update must be released by the user. In this case the module gets the data from its entry and processes it i.e. visualizes the data according to the module configuration. See Fig. 7b for a module processing pipeline using a graphical interface.



a) b)

Fig. 7. Example of sequence diagram for the data processing pipeline a), graphical UI representation of pipeline b)

5. Graphic user interface

The main factor that usually decides whether the application is functional or not is the way the program controls the data flow and communicates with the user.

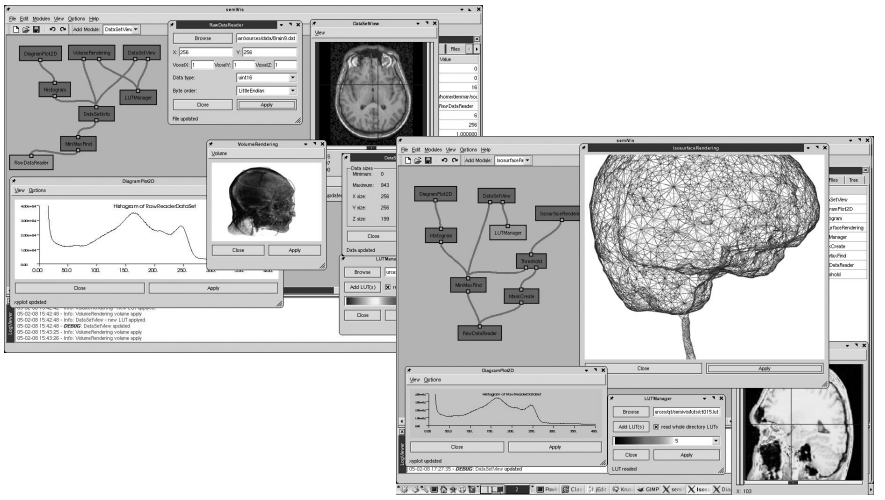


Fig. 8. GUI examples of *SemiVis framework* with simple visualization pipelines and windows of involved modules

So the creation of an intelligible, flexible and useful graphical user interface is in principle a very important task. For this reason, the authors decided to use a multiple-document-interface (MDI) concept [9] with dockable windows from *Frames Package* and independent *ProcessObjectWindows* for each module. The central part of the application occupies the *Workspace* – the panel for defining data processing pipelines from module blocks. All application Frames can be docked to the *Workspace* or flattened around as independent windows. All modules have their own controlling main window designed independently for each module to fulfill their functionality. See Fig. 8 for the best illustration of graphical user interface.

6. Physical structure

The created system consists of several elements. The main one is the activating component *SemiVis* Core, which includes all the essential logical dependence mechanisms, GUI interface, and managing mechanisms. The modules making up the full functional system are delivered separately as dynamic shared system-dependent libraries (Linux – so, MS Windows – dll). These libraries need to harmonize with some internal components (i.e. OpenGL library). Configuration files are also included in the whole system package.

An individual, and also very important, task is the problem of delivering the fully functional API interface allowing creation of personal components. The API consists of a static library including binaries and the header files package. The API documentation is also included as a set of html files. See Fig. 9 for the diagram of system components.

The system requires at least a Pentium III processor. The lower limit of operating memory is 256 MB, but this may be insufficient for most medical datasets and long task pipelines. Installing at least 1 GB of RAM is recommended. For high quality and efficient visualization, the OpenGL compatible graphics card is indispensable. The application has been compiled and is currently being tested on a Linus system (kernel v. 2.4+) and MS Windows (2000, XP) [11].

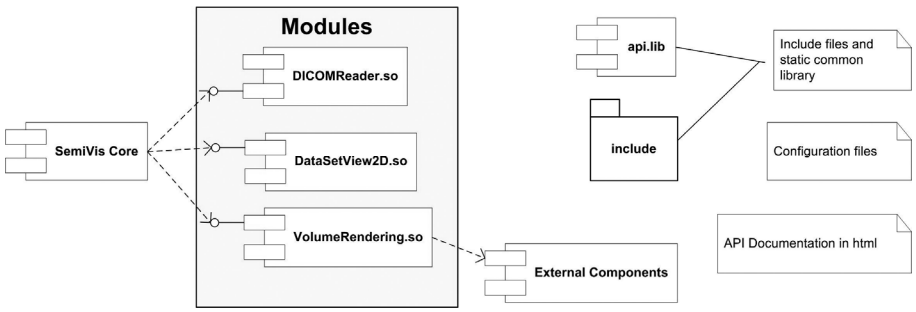


Fig. 9. *SemiVis* components diagram

7. Conclusions

The *SemiVis framework* and its plug-ins provide a tool for visualization, manipulation and processing of medical datasets of various types. Its extensibility and adaptability allow the user to tailor and modify the systems capabilities to suit his/her own context. Included modules are sufficient for typical medical image processing but add-on tools give almost infinite possibilities of implementing user functions. Though the system is still in the early beta phase, it is developing rapidly.

References

[1] http://www.trolltech.com
[2] http://www.kitware.com
[3] http://www.gnu.org/copyleft/gpl.html
[4] http://www.w3.org/XML/
[5] http://medical.nema.org/
[6] Gonzalez R.C, Woods R.E., *Digital image processing*, Addison-Wesley Publishing Company, Inc., (1992).
[7] Booch G., Rumbaugh J., Jacobson I., *UML: przewodnik użytkownika*, Wydawnictwa Naukowo-Techniczne, Warszawa, (2002), in Polish.
[8] Sommerville I., *Inżynieria oprogramowania*, Wydawnictwa Naukowo-Techniczne, Warszawa, (2003), in Polish.
[9] Hamlet D., *Podstawy techniczne inżynierii oprogramowania*, Wydawnictwa Naukowo-Techniczne, Warszawa, (2003), in Polish.
[10] Neider J., Davis T., Woo M., *OpenGL Programming Guide: The official guide to learning OpenGL*, Release 1, Addison Wesley, (1993).
[11] Mitchell M., Oldham J., Samuel A., Linux. *Programowanie dla zaawansowanych*, Wydanie I, Wydawnictwo RM, Warszawa (2002), in Polish.