Design of graphical interfaces for biomedical applications

Abstract

The visualization in biomedical applications includes diverse objects in a wide rank of scale, from molecules and cells to physiological, biomechanical and biophysical parts of the body, their anatomy and properties. The design of intuitive graphical interfaces that allow users to fastly select parameters is a key factor for the usability of visualization applications. Nevertheless, this task is often relegated to a secondary plane in software development. This largely contributes to the fast lapsing of these applications. In this article, we described the design of the library *BioMedIGU*, a tool conceived to make easier the development of biomedical applications interfaces and reinforces their reusability. An example of its use for the visualization platform *HipoVis* it is also shown.

1 Introduction

The technological advances in biomedical 3D acquisition equipment like scanners (CT), magnetic resonances (MR, fMR, MRA) and nuclear devices (PET, SPECT) have opened an alternative approach to traditional biomedical exploration based on dissections and observation of 2D images. From the images produced by these equipment in parallel planes, the acquired volume is reconstructed and then rendered with three main techniques:

- 2D image visualization: the original image slices or optimal sections and multiplanar cuts [13]
- Indirect Volume Rendering (IVR): Projection of surfaces extracted from the volume with the Marching Cubes algorithm.
- Direct Volume Rendering (DVR): projection of the whole volume using emission and volumetric absorption models, as well as local superficial reflection to emphasize regions of interest [7].

These techniques have extended to the *multimodal* (by opposite to *unimodal*) visualization that integrate volumes coming from different acquisition devices [3] and the visualization of time-varying datasets [10].

The exploration of unimodals, multimodals and time-varying biomedical models is an iterative process where the user experiments with the visualization parameters. The correct selection of parameters is fundamental to show semantically significant information of the data. The difficulty of the process takes roots in the specification of these parameters because they are very numerous and diverse. In general, the greater complexity of the input data, the more sophisticated the visualization methods and consequently, the larger the number and diversity of parameters. Hence, to produce an image, users must specify the characteristics of the camera, plus the transfer functions and the attributes of the visualization algorithms such as activation an early termination criterion, distance between samples and number of rays per pixel.

The design of graphical user interfaces that ease the interactive selection of parameters is key to take profit of the last improvements of acquisition and visualization techniques [13]. For that reason, in the last years, research in graphical interfaces for scientific visualization in general and biomedical in particular has intensified [9]. The classical WIMP (Windows, Icons, Menus and Picking Device) has evolved incorporating other modalities of interaction, sensorial-based, using sound and tact [6]. In addition, new exploration widgets have been proposed such as *image graphs* [9] and *spreadsheet* [5] that show the evolution of the iterative process of visualization, allowing users to compare images and the parameters with which they have been generated. Moreover, new techniques for the semi-automatic computation of transfer functions have

been proposed, based on interfaces that allow users to interactively move clipping planes onto 3D visualizations and to sketch the appearance of the final render onto cross sections [16].

The professional visualization applications like AVS, and Analyze [15] incorporate these new interaction paradigms since they dedicate an important part of their development to the interface design (more than 40% according to some authors). Nevertheless, in the biomedical research context, it is often necessary to develop specific visualization applications with a more experimental focus in order to test and compare new techniques. In these applications, the interface design is often relegated to a secondary plane; less effort than necessary is dedicated to it because it does not constitute a research target by itself. For that reason, these interfaces are poor, with a low usability, they quickly become obsolete and they are difficult to reuse.

This work is framed in an investigation project aimed at developing tools for the creation of graphical interfaces for biomedical applications. The requirements of these tools are to be easy to use, reusable, adaptable to the concrete needs of each application and easily extensible. In this paper, we discuss the problems found in the design of these tools, we describe our solution to these problems and we show an example of interface designed using these tools, the interface of the visualization plataform *HipoVis*.

2 Background

There are various libraries and toolkits for creating interfaces. The structure of an application based on their use is represented in Figure 1. The management of the application interaction automaton, that relates interactions to procedures, is centralized in the so-called *intermediate layer*. This layer communicates on one hand with the toolkit to make the interaction tasks, and on the other with the specific procedures of the application, which are completely independent of the interface.

The design of the intermediate layer depends on the features that the toolkits offer. These can be divided in two main groups: the ones bound to the development platform, like the Microsoft Foundation Classes [1] and those of more general character like Tcl/Tk [11], Qt [2], Gtk [4] and Java Swing [8].

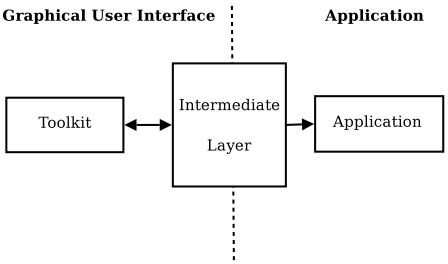

Figure 1: Structure of an application

Table 1 compares the benefits of these systems according to the following criteria:

- if they can be developed in different types of platform (Windows, Linux, UNIX,..)
- an example of an application developed using it
- support of threads for parallel programming that allows processes' interruptibility
- existence of visual programming tools
- ease of use
- restrictions on the programming language of the application
- cost
- type of license: proprietary or free (BSD, QPL, GPL and LGPL)

All the libraries provide basic widgets such as file selectors and advanced ones like 2D visualization areas (canvas), but none of them offer specific widgets for biomedical applications. Except the MFC, all are multiplatform and free, except Qt for commercial applications development. Gtk and Tcl/tk have LGPL and BSD licences, that give freedom to developers to decide on the type of license of their programs. As far as the ease of use, it is always difficult to evaluate. The MFC offers numerous widgets and a predefined style that makes them easier to use, although they are much less flexible. The syntax of Tcl/Tk is complex, it generates long and difficult to debug codes, a disadvantage solved by

	MFC	Tcl/Tk	Qt	Java Swing	Gtk
Multiplatform	No	Yes	Yes	Yes	Yes
Free of charge	No	Yes	Yes for comercial use, No for non comercial use	Yes	Yes
Developed software	Ms Office	VTK	Kde	ArgoUML	Gnome
Threads	Yes	Yes	Yes	Yes	Yes
Visual tools	Yes	Yes	Yes	Yes	Yes
Difficulty of use	Low	Middle-High	Middle	Middle	Middle
Programming languages	Basic, C, C++, C#	Tcl	C++, Ruby, Java, Perl, C#	Java	C, C++, C#, Tcl, Python, Perl, Ruby, ...
License type	Prop.	Free (BSD)	Free (QPL, GPL) and Propietary	Propietary	Free (LGPL)

Table 1: Comparative of interface creation libraries

the classes of Qt and Gtk. In all cases, the programming of the intermediate layer with any of these libraries requires a deep knowledge and an important time investment.

The objective of the system described in this paper is to overcome this drawback by offering an additional layer over the toolkit that provides the elements of interaction common in all biomedical applications, avoiding programmer users to enter in the particularities of the toolkit.

Other toolkits, like VTK, have not been included in this comparative since their primary aim is not the graphical user interfaces but the rendering procedures. VTK, for example, delegates the interface to Tk and Java classes, for which it offers plug-ins.

3 Design

3.1 Structure

The developed system *BioMedIGU* consists of an extension of Gtk that offers the basic functionalities to create the interface of a biomedical visualization application. It has been developed on Gtkmm, a wrapping of Gtk in C++. We have chosen Gtk according to the previous analysis because it embodies the advantages of portability, freeness, good structuring, possibility of threads and freedom when choosing the development license. In addition, the object orienting of Gtkmm provides the advantages of extensibility, easy adaptation and maintenance. Qt is also object oriented, but it does not provide classes to represent signals and slots which constitute important communication mechanisms between widgets.

The structure in three layers described in the previous section has thus been modified as indicated in Figure 2. The different modules are represented with boxes, and the uses between modules with arrows.

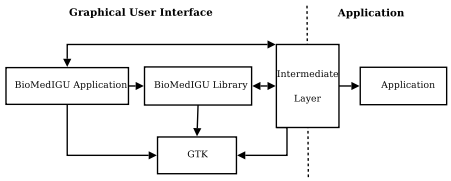

Figure 2: Layers of an application based on BioMedIGU

The module *BioMedIGU library* offers the basic functionalities to create the interface: widgets of visualization and selection of parameters as well as window management and control. It communicates with the intermediate layer and Gtk. The intermediate layer is the one in charge of the communication of the interface with the logic of the *Application*. It realizes the conversions necessary to make the calls to the methods of the *Application* and it modifies the interface according to the results. It also accesses directly to Gtk to instantiate advanced widgets like messages dialogues or file selectors, already provided by Gtk. The fact that Gtk is not totally wrapped allows advanced users to create their own widgets directly.

3.2 BioMedIGU classes

The common elements to biomedical applications have been characterized, in order to design the main classes that compose the library *BioMedIGU*, that

are expandable to future necessities. The main identified elements are windows, that have been typified in four groups: (i) the menu windows, that usually constitute the main window of the application, (ii) the options windows, through which users specify numerical and alphanumerical parameters of simulations and visualizations, (iii) the graphical options windows through which users specify graphical parameters, like curves or color scales for the transfer functions and, finally, (iv) the visualization windows in which the graphical models, biomedical images or volumes are drawn.

Based on this characterization, the BioMedIGU library defines three types of classes,

- the abstract Window class from which the concrete types of window are derived
- the Window Manager class that allows the management and communication between windows
- the External Creator class that provides advanced window creation mechanisms, as described in section 3.3.

Figure 3 shows a simplified diagram of the main classes of the library.

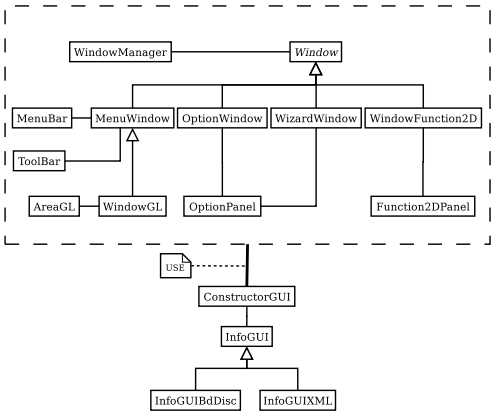

Figure 3: Simplified Class Diagram

The *WindowManager* is responsible for keeping the dependencies between windows. The dependencies have a tree structure when they are created by means of a hierarchical process, but since independent windows can also exist, the structure is actually a forest of trees. When the user of the application or some event closes a given window, it is the *WindowManager* who is in charge of warning the dependent windows in order to close them. In addition, the *WindowManager* provides a communication mechanism for sending messages from a window to its dependent windows and for receiving answers.

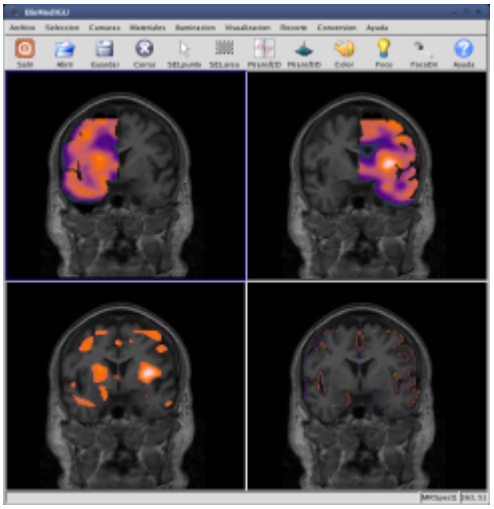

Figure 4: Graphical visualization window

As it can be observed in Figure 3, the library provides the following window types:

- a window with a menu bar and a tool bar(*MenuWindow*, see Figure 4). This window allows us to insert options in the menus and the tool bar, simply calling a method of the class. The options can be associated to states that represent the contexts in which the window can be (for example visualization or edition). These states activate, or deactivate automatically the options according to the context in which the window is at a given moment. In Section 4 an example of this functionality can be seen.
- a window with graphical visualization areas (*WindowGL*, see Figure 4). It is a *MenuWindow* for the visualization of graphical images by means of *GL* in the different areas. Some events of interaction with the user are captured (movement of the mouse, pressure of a button

of the mouse, etc.). A function or method of a class can be associated to each of these events. Different functions for a same event can coexist because each one is associated to a different state, that, similarly to *MenuWindow*, defines the context of the area. The function called when an event takes place is determined by the context or state that the area has at that precise moment.

- a window to introduce or to modify options (*OptionWindow*, see Figure 8). Given a list of options, the window is able to show them and to let users interact with them. Once the user has validated or cancelled the window the options are returned back with modified values, or with the original values if there has been a cancellation. Mechanisms are provided to activate/deactivate options and to modify the value of an option during the interaction with the user.

- Sometimes, the amount of parameters that users must introduce is large, and showing all of them simultaneously can be confusing, even separating them in tabs. Moreover, these parameters must often be introduced following a specific order and there can be dependencies between them. For these cases a special type of option window has been created (*WizardWindow*, see Figure 5) that allows programmers to show groups of options either sequentially or sorted according to a criterion computed during the interaction depending on users input. This way, it is possible to show some options or other ones according to the user's input.

- a window for visualizing and editing 2*D* functions such as transfer functions (*WindowFunction2D*, see Figure 6). These transfer functions can be used, for example, to design scales of colors, to classify the existing materials in a volume of data and to specify the combination of visible properties in a multimodal study. The values of the functions can be real or integer in any axis. It is the programmer who maps the values of its own functions to the (*WindowFunction2D* function values and to map back the window values to its function values. This provides flexibility for the programmer to design any type of 2D function. An example of this flexibility is discussed in Section 5.

  The same window can show different functions and the user can edit them graphically inserting, deleting and modifying values. The actions of the user are controlled automatically allowing him to undo and to remake the modifications. By means of a signal activated by a button, the changes made by the user can be applied with no need to close the window.

Figure 5: WizardWindow

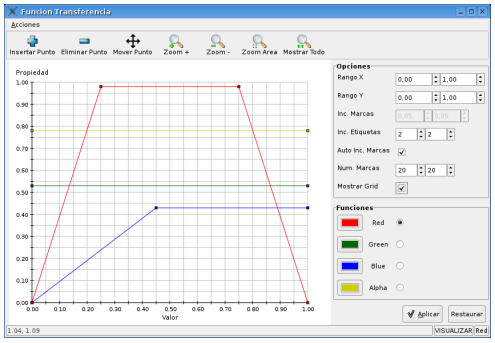

Figure 6: Window for the edition of 2*D* functions

3.3 Window creation mechanism

Two different methods exist to create windows:

- Reading a configuration file
- Instantiating or deriving a predefined window.

Using the first method, a window of the library can be created, or even one that inherits from one of those, by simply indicating in a configuration file the characteristics of the window. For instance, in order to create a *MenuWindow* it is necessary to indicate its menus, the options that each menu has, what function is called when the option is pressed, etc. The file is ASCII, with a very simple syntax XML style.

This method simplifies very much the window creation because it neither requires a deep knowledge of the library nor programming skills. However, it offers less reusability and security than the second method. The possibility of use the inheritance mechanism to create a window from another is not supported by this method. On the other hand, since all the information on the interface is in a text file, if a user modifies the file, it can invalidate the interface.

The second method offers more versatility to the programmer. By means of this method, it is possible to inherit from a class provided by the library and to apply the modifications needed by the application. This class is available for its reuse in other applications, and, since all the interface description is contained in the code, users cannot modify the interface.

The first method, based on configuration files, is convenient for inexpert users and to make prototypes of applications, whereas the instantiating and inheritance mechanism is more convenient for expert users who want flexibility and in order to extend and to adapt the library to specific requirements.

4 Implementation and results

A couple of examples will illustrate the facility with which an interface can be created with *BioMedGUI*.

We first show how to create an options window by means of the two available methods. In order to create it from a configuration file, the characteristics of the window must be provided in a suitable format. In this case, the minimum necessary information is the title of the window, the number of options that it has, and, for each option, the name, the group, the text that will be in the label of the option and the type of option.

With this information in a file, to instantiate the window it is necessary to create an object of the class *ConstructorGUI*, indicating the location of the file, and next, to create the window calling a method of this object with the name of the window as a parameter.

```
ConstructorGUI cons("bd.cfg");
SmartPtr<OptionWindow> refVen =
   cons.createOptionWindow("winName");
```

In order to create this same window by means of the second method described in Section 3.3, we must do a list with all the desired options and instantiate the window giving to it this list as a parameter.

```
InputOptionList lops;
lops.addOption(InputString ("Object",
   "object","cylinder1"),"Main View");
lops.addOption(InputNInteger("Color",
   "color",3,0.75),"Main View");
lops.addOption(InputBool("Centered",
   "centered",true),"Main View");
FramedOptionWindow ven(lops,"Options");
```

Figure 7 shows the length of the code that must be written to create a simple option window by means of these two method, and the code required to create a *simplified* version of that same window using directly Gtk. The length of the code is not only much shorter using *BioMedIGU*, it is also simpler. Figure 8 shows the resulting window, created with the three codes.

The second example shows how to add options to one *MenuWindow* and how these options are activated/deactivated automatically based on the context in which the window is.

Different methods exist for adding menus and options to a window. They can be classified into two groups, (i) those that use the objects as parameters (*Menus* or *Options*) and (ii) those that receive the information needed to create these objects. All the elements (options, menus and submenus) can have associated states. To indicate the states in which the elements are active, the first method (parameter based) must be used. In this example, we used the first method for the options and the second for the menus, since no state is added to the menus.

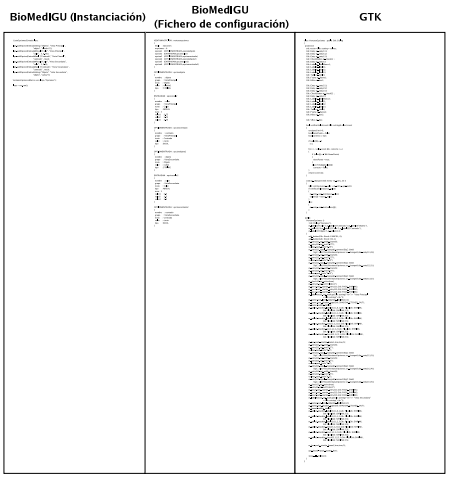

Figure 7: Comparison of sizes of the different codes

First, it is necessary to define the possible contexts of the window by means of an enumeration.

```
enum VISUALIZATION3D, VISUALIZATION2D;
```

Once the window is instantiated, the *changeState* method is called to establish the initial context.

```
MenuWindow ven;
//Initial context
ven.changeState(VISUALIZATION2D);
```

In order to add the menus, the method *addMenu* of *menuWindow* is used. Only the text of the menu must be provided.

```
ven.addMenu("Visualization 3D");
ven.addMenu("Visualization 2D");
```

Next, the options are created with the suitable parameters, the states in which the options will be active are added with the *addState* method. Finally, the options are added to the window's menu calling the method *addOption* and indicating the option and the menu to which this has to be added.

```
Option op("Option1",
   sigc::ptr_fun(onOption1),
   "icon1.png",true,"<ctrl>p");
op.addState(VISUALIZATION3D);
ven.addOption("Visualization 3D",op);

Option op2("Option2",
   sigc::ptr_fun(onOption2),
   "icon2.png",true,"<ctrl>s");
op2.addState(VISUALIZATION2D);
ven.addOpyion("Visualization 2D",op2);
```

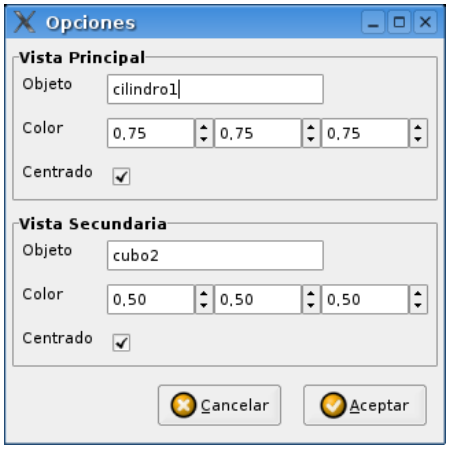

Figure 8: Option Window

We already have the window created and ready to display. In order to change the window context, it is simply necessary to call the *changeState* method of *MenuWindow*. Automatically, the options will activate and deactivate according their list of associated contexts.

```
ven.changeState(VISUALIZATION3D);
```

In Figure 9 two images of a menu in different contexts are shown.

5 Use of *BioMedIGU* in the visualization platform *HipoVis*

Our research group has been developing for ten years [**?**] a data visualization platform that constitutes a test bed to implement and to compare representation models and visualization algorithms. It incorporates surface, volume and hybrid models, such as the voxel model extended to multimodal data [12] and the model of extreme vertices [14]. At

Figure 9: A menu in different contexts

the moment, it is being adapted to support the temporary dimension. The rhythm of research at the university, that requires results in very short terms, as well as the lack of human resources had led us to develop a minimum interface with Tcl/Tk that allowed us to visualize in a graphical window using OpenGL but that did not offer any mechanism of interaction. The multiple parameters of the visualization were specified by means of files in a notation similar to XML. This operative was very little effective and, actually, it limited the use of the platform to its own developers. For that reason, we decided to design a new interface using *BioMedIGU*. Figure 10 shows an example of this interface.

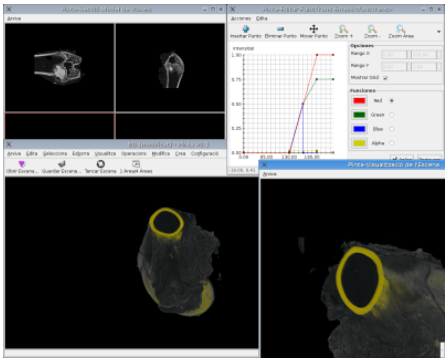

Figure 10: Screenshot of the application *HipoVis*

The structure of the application follows the three layers model described in Figure **??**. The automaton of the application is implemented in the intermediate layer that receives the interface events, manages the state of the application, processes the data and provides the communications between windows. In order to make these tasks, the intermediate layer is based mainly on two own classes: the descriptor window class *(infovis)* and the scene manager *(catalogue)*. The descriptor of graphical windows *infovis* maintains information of the entities that take part in the process of visualization in a window (models, cameras, lights and materials), as well as selection lists and visualization preferences. It also keeps the last rendered image, which allows us to redraw very fast, simulating the mechanism of *backingstore*.

One of the greatest advantages of *WindowGL* is that the mouse events can be associated to different procedures depending on the state of the application. Thus, for example, the movement of the mouse with the left button pressed is interpreted as a camera movement or a geometric transformation of the selected objects depending on the current context.

Different graphical windows can share information, for example, different views of an object. In order to avoid destroying a window entity being used by another window, and to facilitate the management of the life cycle of the entities, control is centralized in the *catalogue* class. The *catalogue* is based on the design pattern <abstract factory>, but, by opposite to the factory, it does not create the entities, these are created first and then registered in the catalogue, who is in charge of maintaining and destroying them when they are no longer necessary. The catalogue also maintains the relations <referenced-by> and <reference-to> between entities and it offers various methods to make queries on the entities or to access to them. All the windows have access to this catalogue and access to the entities through it. When a window uses an entity, it sets a reference in the catalogue that prevents the entity from being destroyed. The unreferenced orphaned entities are destroyed and, recursively, the entities only referenced by these ones, freeing memory.

On the other hand, the intermediate layer, using *BioMedIGU*, constructs the windows, adds the menus and connects each event with the function that must treat it, using the mechanism of signals and slots. All the classes of *BioMedIGU* are used, specially the class *MenuWindow* for the management of the parameters and the class *WindowGL* for the visualization. For the visualization of voxel models, different types of transfer functions are

used: the selection function, that allows users to select the ranks of visible property, the function of volumetric illumination, that allows users to assign an emission and opacity to the different values from voxels and the function of surface illumination that allows users to assign coefficients of reflection and color to the isosurfaces depending on voxel values.

For the edition of these functions, the class *WindowFunction2D* of *BioMedIGU* is used. This one is a clear example of how the use of the user interface library simplifies the development of the application. The programmer must only worry to update the automaton, to generate and to recover the transfer functions and to construct the edition window. The pseudo code for the generation of the 2D functions is shown: it consists simply of mapping the information of the axes of the transfer function onto values that the *WindowFunction2D* widget can process. The recovery of the values once modified is done similarly. The visualization of the widget and its edition are totally automated by the library, the intermediate layer only has to provide and recover the data.

```
EditFuncTransWindow::CreateFunction(...)
{
   SmartPtr<Function2D> func =
      Function2D::create(name,tipX,
         tipX,name);
   func->setRangeY(minY,maxY);
   func->setRangeX(minX,maxX);
   FillFunction(func);
   addFunction2D(func);
}
EditFuncTransWindow::FillFunction
   (SmartPtr<Function2D> func)
{
   for i = 0 to i = N-1 do
      valX =
         GetValueAxeFuncTrans(axeX,i)
      valY =
         GetValueAxeFuncTrans(axeY,i)
      ValorXY v =
         CalcCoordinate2D(valX,valY)
   func->addValueXY(v);
   endfor
}
```

The platform *HipoVis* is not in any case restricted by the classes provided by *BioMedIGU*, and it has different mechanisms to obtain specific functionalities:

- It can access directly to the underlying Gtkmm toolkit, in order to invoke a file selector or to modify the size or aspect of a widgets, for instance.
- It can extend *BioMedIGU* library classes with specialized classes adapted to specific needs. As an example, *EditFuncTransWindow* derives from *WindowFunction2D* provided by the library, and also new types of input options have been created.

6 Conclusions

The design of the library *BioMedIGU*, described in this paper, is framed in a project whose objective is to facilitate the development of graphical interfaces in biomedical applications. *BioMedIGU* has already been applied to the development of the interfaces of diverse applications. In this paper, we have described, as an example, the development of the interface of the visualization platform *HipoVis*, that has allowed us to validate the scope and the effectiveness of the features of *BioMedIGU*. The result is very satisfactory. The time of development of the interface has been relatively short and the platform has improved a lot in usability.

The investigation that starts from this project is centered in the study of advanced methods of interaction and data selection allowing users to manipulate large volumes of multimodal and multisensorial information. We wish to extend the features of *BioMedIGU* to support this type of interactive manipulation.

References

[1] K. Bugg. *Building Better Interfaces With Microsoft Foundation Classes*. John Wiley & Sons, 1999.

[2] M. Dalheimer. *Programming with Qt, 2nd Edition*. O'Reilly & Associates Co,, Inc., 2002.

[3] M. Ferré, A. Puig, and D. Tost. Rendering techniques for multimodal data. *Proc. SIACG 2002 1st Ibero-American Symposium on Computer Graphics*, pages 305–313, 2002.

[4] T. Gale. *GTK+ 2.0 Tutorial*. http://www.gtk.org, 2004.

[5] T. J. Jankun-Kelly and K.L. Ma. A spreadsheet interface for visualization exploration. In *Proceedings of the 11th IEEE Visualization 2000 Conference (VIS 2000)*. IEEE Computer Society, 2000.

[6] E. Jovanov, K. Wegner, V. Radivojevic, D. Starcevic, MS. Quinn, and DB. Karron. Tactical audio and acoustic rendering in biomedical applications. *IEEE Transactions of Information technology in Biomedicine*, 3(2):109–118, 1999.

[7] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics & Applications*, 8:29–37, May 1988.

[8] M. Loy, R. Eckstein, D. Wood, J. Elliott, and B. Cole. *Java Swing, 2nd Edition*. O. Reilly & Associates Co,, Inc, 2002.

[9] K.L. Ma. Image graphs: a novel approach to visual data exploration. In *Visualization '99*, pages 81–88. IEEE, IEEE CS Press, 1999.

[10] N. Neophytou and K. Mueller. Space-time points: 4D splatting on efficient grids. In *Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, pages 97–106. IEEE Press, 2002.

[11] J.K. Ousterhout. *Tcl and the Tk toolkit*. Addison-Wesley Longman Publishing Co., Inc., 1994.

[12] A. Puig, D. Tost, and M. Ferré. Design of a multimodal rendering system. *Proc. 7th International Fall Workshop Vision, Modeling and Visualization 2002, Greiner G,. Niemann, H., Ertl, T., Girod, B. and Seidel, HP. Editors*, pages 488–496, 2002.

[13] R.A. Robb. *Three-Dimensional Visualization in Medicine and Biology*, chapter 42, pages 685–712. Academic Press, 2000.

[14] Rodriguez, D. Ayala, and A. Aguilera. Complete solid model for surface rendering. *Geometric Modeling for Scientific Visualization,Springer Verlag*, pages 259–274, 2004.

[15] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3-D Graphics (2nd Edition)*. Prentice Hall, 1998.

[16] F.Y. Tzeng, E. Lum, and K.L. Ma. A novel interface for higher dimensional classification of volume data. In *Visualization 2003*, pages 16–23. IEEE Computer Society, 2003.