ATTI DEL CONVEGNO

8° convegno internazionale e mostra sulle applicazioni della computer graphics nella produzione, progettazione e gestione
A C++ Application Framework to Integrate X Windows and 
graphHIGS in a coherent object-oriented environment

G. Chiaveri, G. Ghersi, S. Pantanoni, M. Tiecci,
IBM Scuola Scientifico e Tecnologica Center
Università degli Studi di Milano
e-mail: tiecci@rncsce.vene.iben.com

Abstract

Under the AIIX operating system, many tools and libraries are widely available for applications in computer graphics. However, they cannot be easily integrated in a coherent programming environment. This paper presents an object-oriented approach to the construction of a framework for developers who need to get start from both X Window and graphHIGS. X Window strength stands in its capability, residing in tool for development graphical user interfaces (GUI) with a standard look and feel, but it lacks in poor graphical functions. On the other hand, graphHIGS is a powerful language for 2D or 3D graphics programming. This paper presents an object-oriented approach to the construction of a framework for developers who need to get start from both X Window and graphHIGS.

1 Introduction

Many tools and libraries are widely available for applications in computer graphics, under the Unix operating system and its IBM implementation, AIIX. They usually can be classified into two distinct classes: on the one hand, X Window based tools are commonly used to build graphical user interfaces, with a standard look and feel, but they lack in poor graphical functions. On the other hand, advanced graphics are achieved by libraries such as X11, graphHIGS and GL) that do not provide high level primitives for the user interface, such as buttons, menus, X Window, and graphHIGS are difficult to be mastered and are too much different to be integrated in a modern application (a tutorial about this subject can be found in [9]).

Graphical applications usually consist of two main parts: a front end to manage the user interface, and a second one to process commands and data [4]. Most of the code needed to manage the interaction and the graphical display is application independent. Standard procedures and algorithms can be implemented to accommodate the specific elements of the user interface, such as buttons, menus, windows, and graphical objects. The graphical interface is subject to be manipulated interactively. An object oriented approach seems to be the natural and obvious solution.

Starting from a proper set of classes providing selection of user interface, complete applications can be inherited, adding new functions and keeping the behavior of existing objects. This is what we call an Application Framework [2]. This paper presents a C++ Application Framework that encapsulates both X Window and graphHIGS in a coherent programming environment. All the features described here have been implemented and used to develop several applications. The package runs on IBM Risc System/6000 workstations, equipped with X/Windows and graphHIGS.

2 Design of the Application Framework

This project originated within the IBM Italian Scientific and Technical Center, to serve some specific needs. The basic issues were to provide a framework both to build 2D graphic user interfaces and to serve as a development and debugging tool in an industrial research environment. At the same time, users of the library had no experience in Object Oriented Programming and were not accustomed to the development of user interfaces with X Window or advanced graphical applications with graphHIGS.

Then we identified a set of primary goals. Rapid prototyping was the first, to allow a quick development of debugging tools and application prototypes, with a standard look and feel. All the user interfaces must have a similar style and comply Motif [7] and IBM SAA CUA [9] guidelines. At the same time, a gradual migration to Motif was required, to allow Motif programmers and X programmers to migrate smoothly toward Object Oriented Programming and reuse their old code. Last, but not least, we wanted to reduce learning time, i.e. reduce the effort required to learn how to develop user interfaces with X Windows and effective graphical contexts with graphHIGS.

All these issues led to a multi layers architecture. At the lowest level, two wrappers provide an object-oriented interface to the basic X/Windows and graphHIGS libraries. In this way, we obtain a coherent environment, the users of both the libraries can easily learn how to develop applications within the new environment; actually, the same concepts are preserved in both cases, but their implementation is extremely simplified. A set of classes, called the Motif helper, consist of functions to be called, and other features of the environment.

Using the first level classes and functions, objects of higher complexity are built. They include tools to implement dialog boxes, to fill in forms, or menus, to retrieve graphical objects, and to draw something interactively. At the highest level, skeleton application classes are the core of the Application Framework. In order to develop a new program, the user must choose the most similar sample object and inherit a new class from it.

These layers will be described in greater details in the following section.

3 The X Windows wrapper

Motif and the X extensions are widely used for developing user interfaces in the Unix world. They support an object-oriented architecture, through the concept of widget [9, 10], even in a non object-oriented language, like C.

We decided to develop a set of classes wrapping widgets into C++ objects. We chose this approach even though object-oriented languages based on X Windows are available. In fact, most of them, such as Motif libraries, are built over Xlib, the lowest X Windows layer, that does not include widgets. This means that the experienced Motif programmer is not lost in the need to approach the new library in a different way. We want to save skills. Moreover, the library developer should re-implement many tasks already accomplished by Motif. This was beyond our scope. Last, Motif look and feel and the use of widgets has become a de facto standard.

We also preferred to build such library instead of using other available sets of classes built over Motif, such as the Widget Wrapper Library [9]. They do not fit well our needs in particular, they do not address the problem of the integration with graphHIGS.

3.1 The class tree

The tree of the "widget objects" (Figure 1) has its root in the Class O class, that encapsulates the core widget, and covers all Motif widgets. The hierarchy includes classes obtained in three different ways:

- encapsulation of a standard widget: this is done by deriving a class from the object that corresponds to its parent in the widget hierarchy. Motif makes such process straightforward. This approach ensures full reusability of already developed widgets;
- classes with extended functions: they are obtained by inheritance from the class that is to be extended. For example, the StrongFieldO is derived from the TextO and handles an input field linked to a data variable. The programmer has no need of reading the field to update the variable;

- classes with extended functions: they are derived from the class that is to be extended. For example, the StringFieldO is derived from the TextO and handles an input field linked to a data variable. The programmer has no need of reading the field to update the variable;
3.2 The Application_O object

Every user interface developed with this library requires at least one instance of the class Application_O; this class hides the programmer X-Windows initialization, display handling, and a number of lower procedures. Using only the first level classes mentioned above, the traditional Hello-World program has the following form:

```c++
#include <Core.O.h>

main()
{
  Application_O app("app");
  Label_O lbl("Hello World", app);
  app.Start();
}
```

Figure 1: Some classes of the Core.O tree

where the BIF object is a child of app, an instance of Application_O class. The main limit of our library is that it does not overlap all the functions of X-Windows. However, this target is well above our possibilities. In any case, each class is related to the main widget hierarchy. We developed conversion operators that extend the widget in a way transparent to the programmer. As a consequence, whenever the need arises, it is possible to use X-Windows and X-Windows Toolkit functions easily.

We also provide methods to add new resources to the classes, thus fully supporting the X-Windows Resource Database Manager.

4 The gPHIGS wrapper

PHIGS is a powerful graphics language [6]. It virtually answers all the needs of a graphics programmer (except the user interface development). Due to its complexity, however, it takes much time before being used properly and effectively. The concepts related to workstations, windows, views, and structures, are not readily mastered. The number of routines and parameters is very high, but most applications actually use only a subset of PHIGS routines and functions.

We choose the most suitable subset for our purposes (2-dimensional primitives). We took advantage of the fact that the gPHIGS architecture is suited for an object-oriented implementation. A similar approach can be found in [11], where the problem of integration with X-Windows, however, is not fully addressed.

4.1 The gPWorkstation_O object

The first step in the integration of gPHIGS and AIX workstations has been the development of a gPWorkstation_O object to manage a PHIGS workstation: the gPWorkstation_O object.

This is fully integrated in the Core.O tree and adds a number of specific resources to its ancestor, the Drawing_O. It provides a graphic area, where the drawing is performed with gPHIGS. For example, it implements methods for creating and connecting workstations, associating views to them, and dealing with coordinate transformations.

4.2 The other main objects

The encapsulation of the graphic language consists of a set of classes, besides the gPWorkstation_O:

- gView : in PHIGS philosophy, a view is simply a way to display a portion of the world (the window) into a rectangular area of the workstation (the viewport). The methods associated with this class make these concepts intuitive and the implementation of panning, zooming, etc., quite easy.

- Struct : the main advantage of PHIGS approach is that the basic graphic element is not the pixel (as in X-Windows), but the structure element (line, polygon, text, etc.). These elements are assembled into complex objects, called structures. They are modifiable at any time, but the programmer does not have to take care of redrawing the screen. The Struct class implements these concepts, and its methods include all the 2-dimensional graphic primitives;

- Input : PHIGS offers powerful input techniques. Different input devices can be associated to the same physical device. The user can directly return the input position in the world coordinate system (Locator), or the structure element classed with the pointer (Pick). Classes derived from the Input meta class implement these input devices. They are fully integrated with the X-Windows input system: all X-Windows and PHIGS events are driven by a single queue, managed by the X Server. The application simply waits for input events from the X Server. Then, it issues the callback mechanism [10]. This event driven model is the best solution for multi-tasking environments, since an idle application (one with no empty event queue and waiting for input only) can be suspended by the system, thus preventing waste of CPU time.

4.3 The functional approach

All the graphic primitives, besides being methods of the Struct object, appear in the library as functions. Compared to gPHIGS functions, they are much more easily mastered, thanks to a heavy use of useful C++ features, such as "function overloading" and "default parameters".

We chose this approach for a number of reasons: (i) to maintain a "solid" approach to the library for those programmers not familiar with object orientation; (ii) to make the porting of already-developed graphic
5 High-level tools

On top of these two wrappers, we developed a wide set of classes. They make it easy to implement tasks, which are common to almost every program. Among these classes, Dialogs, Menus and Fills in panels are of fundamental importance; they implement context of the application user-dialogue. In this way, the programmer does not need to handle low-level events.

All the above classes are of general use. The library also has a set of classes, explicitly designed for specific applications. In particular, the field of map generating and recognition [9] was chosen as a "test" example of applications for our framework. In this case, the main class is a general purpose map viewer with navigation functions and the ability of displaying multiple views of the same map. It also provides navigational tools, such as zooming and panning, making the implementation of a complete map editor an easy task.

6 Application classes

The lowest layers of the library provide a much higher-level interface than Motif and PHIGS. However, most applications contain a surprising amount of duplicate code.

Considering only the minimum features common to all our applications, it is true that all programs need:

- declare an instance of Application.O;
- create one or more widget objects defining the user interface;
- create the g*WorkStation.O instance;
- initialize the input devices;
- handle events starting the Application.O by calling its Start() method.

Most programs perform all these tasks with only minor changes. A commonly used approach is to prepare a template with these steps and use a copy of it as the basis of every new program.

A more powerful approach is to use in a class an application skeleton that performs these steps. Programmers can reuse the common code in a very simple way: an instance of the class must be declared, or a new class must be derived from the original one.

Several advantages stem from this approach. In fact, when a new class is derived and new features are added, it is easy to go back to the original behavior, because changes are made only to a derived class. In this way, programmers are unlikely to introduce accidental errors in already-existing code, because changes are only made to the derived class, and the original code is safely encapsulated in its own class. Enhancements and fixes of the base class are automatically extended to the derived ones.

The power of this approach becomes evident when we apply it to a specific category of complex applications.

In such a case, it is easy to identify a wide set of common features that can be encapsulated in an application class.

The library itself provides only a basic set of application classes, but every development group can build its own specific set in a short time. Some of the ready-made classes are:

SimpleAppication: a basic skeleton that provides an empty window. It takes care of most of the initialization. The programmer must put all the widget objects needed in the window, and then define the actions he will perform in response to events.

MenuAppication: it provides an application with standard menu handling (based on the IBM SAA CUA specifications), error management, help, and message system.

GraphApplicatlon: adds to the preceding one a g*WorkStation.O object, with a standard implementation of input handling, hiding all initializations.

Conclusions

In this paper, we have discussed our object-oriented approach to the development of applications that require both an effective user interface and advanced 2D graphic capabilities.

We developed a tool that pays attention to the requirements of an industrial research environment: reusability of all the code already developed and fast migration of programmers to the new techniques are of fundamental importance.

The Application framework is in use by our group since June 1991. Other groups of developers are using it too. These C programmers now work in a well-structured way, and their backtracks have shown a number of extensions and changes to the architecture.

In this way, we have been able to fulfill the most important original goals. In particular, the application classes and the complex objects provide standard look and feel and uniformity between applications. The high-level class also simplifies the development of graphic code: this allows rapid prototyping of new applications.

Our experience has demonstrated that a C programmer usually needs three days of work, side by side with an expert use of the library, to become productive. After a few weeks, he fully masters the library and stays most of the features of C++. The possibility of a gradual migration to C++ is of primary importance for this result.

More work can be done to solve the issues identified throughout the paper. In particular, we feel the need for a better documentation of our library with the programming tools available in the IBM REXX environment. A complete step-by-step and the usability of the library would be an interactive user interface builder, mostly for simple interfaces. This will be the subject of future work.

References