A Motif-Like Object-Oriented Interface Framework Using PHIGS

by

Scott A. Woyak

thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Mechanical Engineering

APPROVED:

[illegible]

Dr. Arvid Myklebust, Chairman

S Jayaram

Dr. S. Jayaram

[illegible]

Dr. J. R. Mahan

September 25, 1992

Blacksburg, Virginia

C.2

LD
5655
V855
1992
W693
c.2

Abstract

Graphical user interfaces (GUIs) are defining what the interface to a program should look like. Unfortunately for programmers who need to use three-dimensional graphics found in environments like PHIGS, there is no convenient way the to use the features found in existing GUI's. This thesis describes the implementation of an interface framework running totally under PHIGS. The framework is designed using object-oriented principles and is coded in C++. The tools provided by the interface emulate the look and feel of the objects found in Motif. The interface is designed in such a way that new objects (such as windows and menu items) can be added in the future without modification to existing code. This flexibility allows programmers to specialize interfaces for their programs while maintaining complete compatibility with the base code. This also allows the framework to emulate other GUIs (such as OS/2 or Macintosh) in addition to Motif. The interface software is therefore referred to as an interface framework rather than a tool kit. The windows and menu items provided in this implementation are samples of the objects that can be created with the framework.

Acknowledgments

During undergraduate school, I was unsure of what graduate school to attend or what type of research to pursue. I talked to many people and visited several schools. It wasn't until I talked to Dr. Sankar Jayaram that I began to get enthusiastic. I would like to thank Dr. Jayaram for introducing me to CAD and for sparking my interest in computer programming.

I would also like to thank Dr. Arvid Myklebust (my advisor) for allowing me to enter the CAD program. With very little previous exposure to me, Dr. Myklebust took a chance and allowed me to prove myself. He provided funding and a place to work when many other students were having trouble just getting into graduate school.

Ultimately my funding came from a grant from IBM. I would like to thank IBM for providing my funding and for providing the CAD lab with equipment for me to work on.

Even though I was financed through IBM, it was my wife Laura who provided most of my support. I thank her for temporarily putting aside her career goals while I started mine. Without her financial and moral support, I would have undoubtedly had to have left the program long ago.

As members of my committee, Dr. Myklebust, Dr. Jayaram and Dr. Mahan have provided me the opportunity to do my research. I thank all of them for taking there time to guide me through my work.

Lastly, I would also like thank the other students in the CAD lab for the releases they provided me from my work. I enjoyed the many basketball, softball and walleyball games we competed in.

Table of Contents

List of Illustrations

Glossary

Base class - In C++ a base class is the class inherited from.

C++ - An object-oriented programming language.

CAD - Computer aided design.

Class - In C++ a class is an encapsulation of data and the functions needed to manipulate that data.

Derived class - In C++ a derived class is the class inherited to.

Device independent - A program that is device independent can run on multiple platforms ranging from PCs to workstations.

graPHIGS - The IBM implementation of PHIGS

GUI - Graphical user interface. Examples are Microsoft Windows, OS/2 and Motif.

Inheritance - In C++ inheritance is the process by which one class assumes the properties of another class by inheriting from it.

Linked list - A method of maintaining a group of items. Each item contains a reference to a previous and next item. The items are linked together to form a chain of items.

Menu item - An object that a user can interact with in a menu. Examples are push buttons and check boxes.

Motif - An interface tool kit that runs under X windows.

Object - In C++ an object is an instance of some class.

Object-oriented - A design process which focuses on subdividing the problem by data rather than procedures.

PEX - PHIGS extensions to X.

PHIGS - Programmers Hierarchical Interactive Graphics System. A programming environment which provides facilities for displaying three-dimensional geometry.

Virtual function - In C++ a process by which a function declared in a base class can be defined in a derived class. This allows class specific definitions of functions for classes inherited from a common base class.

Window - In reference to a GUI, a window is a section of the display that is used to display information. A GUI consists of several windows that overlap and can be raised above one another.

X windows - A network protocol that allows one computer to graphically display the output of a program running on another computer.

Thesis Organization

This thesis presents the work related to the creation and usage of the interface framework. The thesis is divided into many sections. Listed below is a description of what is contained in each of the sections.

Introduction - Provides an introduction into the world of workstation interfaces

Problem Definition/Derivation - Provides a history of the tools used by CAD workstation programmers and explains the need for a three-dimensional device-independent interface framework

Goals - Outlines the goals for the interface framework

Literature Survey - Describes some previous work relating to PHIGS and interfaces

Tools Used - Lists the tools that were used for the interface and explains why each was chosen

Program Development - Documents the evolution of the interface code

Class Overview - Presents the model used to create the interface framework

Menu Item Protocol Specification - Explains how to add new menu items to the framework

Window Protocol Specification - Explains how to add new windows to the framework

Sample Usage - Presents a simple sample program and an explanation of the code required to create it

Results - Describes a sophisticated demo program created using the interface code

Conclusions - Summarizes the work accomplished and draws conclusions

In addition to the above sections, the thesis also contains several appendices which describe how to use the interface framework. Notes on the exact syntax of function calls and examples of those functions are provided.

Introduction

The objective of this thesis is to create a set of modern interface tools using PHIGS. The word modern is used in reference to the recent appearance of environments such as Motif, OS/2, and Microsoft Windows. These environments use interactive tools such as push buttons and check boxes to liven up and create a more visually appealing interface.

The interface is often the most time consuming part of a project - consisting of up to 60% of the total code [Bass89]. Since the interface is usually not the focus of a project, the quality of the interface suffers. This leads to a program that is difficult to use and not well suited for future expansion. Higher quality interfaces can be created by providing the programmer with a GUI (Graphical User Interface) tool kit.

While the PC is blessed with a wide range of sophisticated GUI's, the workstation market is still in its infancy. One of the reasons for this is the lack of standards for workstations. Each manufacturer creates its own operating system with its own peculiarities. This makes it very difficult for software companies to create products that run on multiple platforms.

As noted in a recent survey of Fortune 500 companies, many companies develop their own software in-house [Penn91]. Without a common set of standard tools, programmers tend to develop code that becomes very dependent on the hardware it is running on. This not only limits the reuse of the code, but also makes it difficult to port the code to other systems.

The most widely used interface tool kit for workstations is Motif. The popularity of Motif largely lies in its multi-platform existence and the aesthetic interfaces that it can create. Motif owes its portability to X windows. Because Motif runs on top of X, it shares all the benefits of X.

Although Motif is extremely popular and very successful, in terms of many engineering programs, it is not very useful. Many engineering programs are built around environments such as PHIGS which provide strong three-dimensional rendering facilities. The major drawbacks that Motif has are:

- Motif is limited to two-dimensional graphics
- Users of Motif are forced to program in an X style
- Motif is not designed to be easily expanded by the user

Since Motif is basically an extension of X, it exhibits the same properties of X. This includes working in a 2D environment only. X displays are created in terms of pixels and bitmaps. This makes it very difficult for engineering programs, which rely on three-dimensional graphics, to use Motif.

The usual work-around is to use Motif for the interface, and PHIGS for the display. This is difficult to do because of the restrictions placed on programs by the two environments. For example, Motif is an event mode only environment. The entire system is an action-reaction environment. A program largely consists of a group of callbacks which are each called in response to user input. On the other hand, PHIGS provides sample, request and event modes. Tying the two environments together means that one environment must sacrifice its principles.

The most recent solution to this problem is PEX - PHIGS Extensions to X. While PEX brings a 3D environment to X, it also introduces many problems. The conflicts between PHIGS and X are well documented [Sung90].

Even if it were easy to integrate Motif into a PHIGS program, Motif would still not be suitable for engineering projects. Motif is fixed in that it provides a set of standard interface tools that cannot easily be modified or expanded on. Although these tools are satisfactory for most applications, they do not provide all the features that would be useful for an engineering application. Often engineers need specialized tools for a specific application.

The aforementioned problems can usually be worked around if they are acknowledged from the start of a project. Adding a Motif interface to an existing PHIGS program, though, is much more complicated. The entire input system usually must be rewritten to accommodate the Motif input model. A better solution is to use an interface that is more closely tied to PHIGS.

By building an interface environment around PHIGS, the portability that PHIGS enjoys is brought to the interface. Programs created using the interface framework will inherit multi-platform existence while not sacraficing flexibility. Existing PHIGS applications can be given facelifts without major modifications to underlying code. Specialized tools utilizing PHIGS capabilities can also be created.

Problem Definition/Derivation

This section presents a background into the problems that affect human interaction with computers. The section first describes the rudimentary conflicts that affect humans and computers. From the engineering standpoint, the solutions developed to bridge the gaps are then discussed. This then leads into proposed research for this thesis.

Computers are powerful instruments. They are powerful because they can perform thousands of calculations in the blink of an eye, simultaneously display millions of colors, and store billions of data values. They are, however, still instruments that, if used improperly, are no better than the common calculator.

The problem with most computer systems is in the difficulty of getting them to do what you want them to do. It is not that the computer cannot do what you want it to, rather it is that it is difficult to express your needs in a way that the computer can interpret.

This difficulty is acknowledged in a model created by Fischer [Fisc89]. The model is displayed in Fig. 1 . The model defines two channels for human - computer interaction:

> **Implicit Channel** - If knowledge is based on the same structures, then it is not necessary to explicitly exchange all information. Some information is implicitly exchanged.
>
> **Explicit Channel** - This is a huge, largely unstructured and poorly understood gray area that is inadequately supported by existing tools.

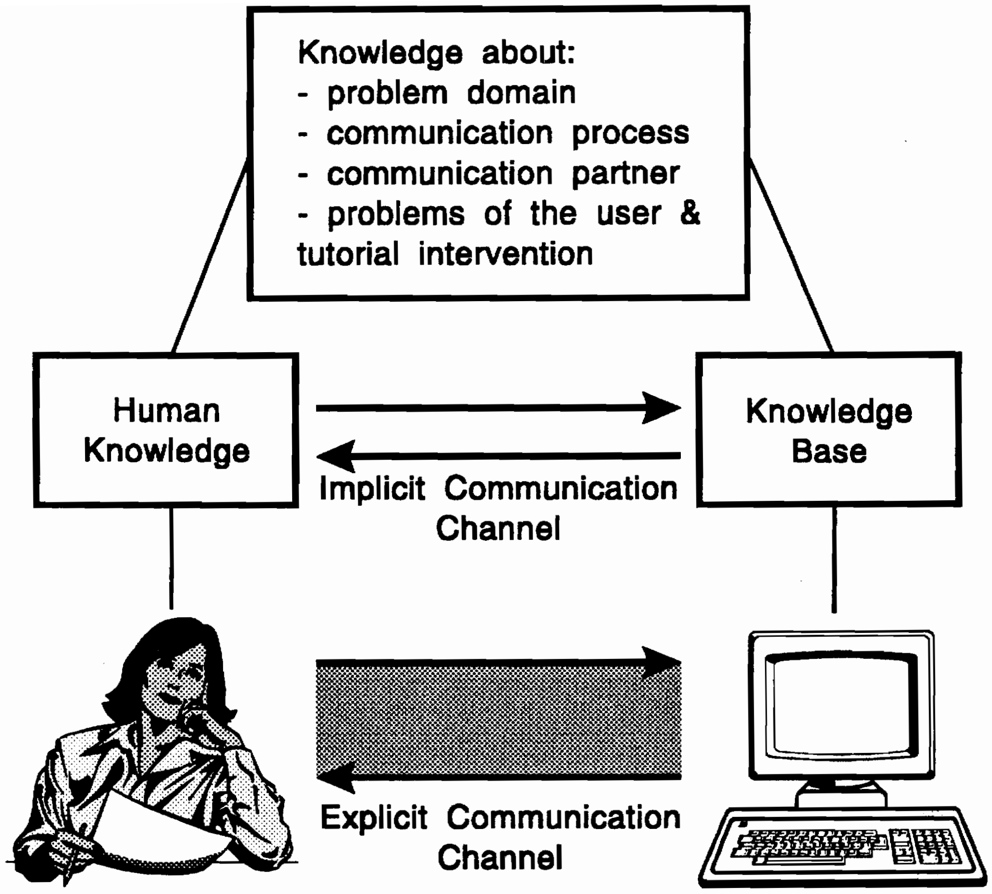


Figure 1 - Human - Computer Communication Channels [Fisc89]

Fischer defines the following knowledge domains for implicit communication:

Problem domain - knowledge of the problem constrains the number of possible actions and describes reasonable goals and actions.

Communication processes - dialogue on a shared display should be used instead of issuing isolated messages.

Communication partner - the system should adapt to the needs (or level) of the user.

Common problems and instructional strategies - the system must teach the user - it must interrupt when the user does something wrong.

The computer program (and hardware) should attempt to remove the explicit communication channels and increase the implicit communication. This will allow the user of the program to focus on the problem at hand rather than wasting time trying to figure out a way to express the problem so that the computer will understand.

Fischer notes that to reduce the explicit channel, human computer interaction software should acknowledge human weaknesses such as limited short-term memory and execution errors. The software should instead exploit human strengths such as a powerful information-processing and visual system.

This problem is in part being addressed with the migration of computer systems from character mode displays to graphical user interfaces. In terms of the workstation market, this transition can be marked by the appearance of the X protocol. X provides network transparent operation. This means that one computer can perform calculations while another displays the results.

While the ability to run applications across networks opened the door for X as a defacto standard, it has been its generous use of graphics that has helped to keep it around. X was designed with future improvement and expansion in mind. This allowed graphical user interface environments to build onto X. These GUI's grew in popularity and because they were based on X, assured the continued existence of X.

The most common interface tool developed for X is Motif. Motif allows programs to add advanced features such as push buttons, check boxes, and radio buttons. Motif also provides the capability of displaying multiple overlapping windows.

Since X (and therefore Motif) is based on two-dimensional bit-mapped displays, it was the business community that first adopted its use. Business applications such as word processors, spreadsheets, and databases worked well with X and Motif. Engineers, whose main focus was with three-dimensional graphics and true-color displays, had to look elsewhere for programming environments. The first three-dimensional standard to appear was PHIGS - the Programmers Hierarchical Interactive Graphics System. PHIGS allowed engineers and scientists to display their models in a very realistic fashion.

As computer systems developed and matured, the paths of PHIGS and X began to cross. Sophisticated PHIGS applications began to require the features found in Motif, while many X programs began to need to display 3D graphics. The immediate solution was to implement PHIGS inside an X window. This is the approach taken by products such as graPHIGS and FIGARO - two implementations of PHIGS. This solution is limiting to PHIGS because it limits the display capabilities to those of X.

Figure 2 shows the problem that programmers are faced with. Application programs that wanted to use both Motif and PHIGS are forced to integrate the two. This integration is handled differently for each program and usually means limiting the functionality of one of the two tools.

A better solution to the integration of Motif and X is PEX - PHIGS Extensions to X. PEX brings 3D functionality to X. The advantage of using PEX is that an application program only has to connect to one tool kit. The problem is that PEX does not completely encapsulate all the features of both PHIGS and Motif. This is shown in Fig. 3.

The reason that some feature of both PHIGS and Motif cannot simultaneously be integrated into PEX is that they conflict. Sung points out some of these problems [Sung90]. Some of the most prominent are:

- Where are structures stored?
- How are input prompts and echo output stored and displayed?
- How are the request and sample modes handled?
- How are color table conflicts handled?
- How are redraw events performed?
- How are variable devices sizes handled?
- How are formats assumptions handled across a network?
- How are X and PHIGS error handling integrated?

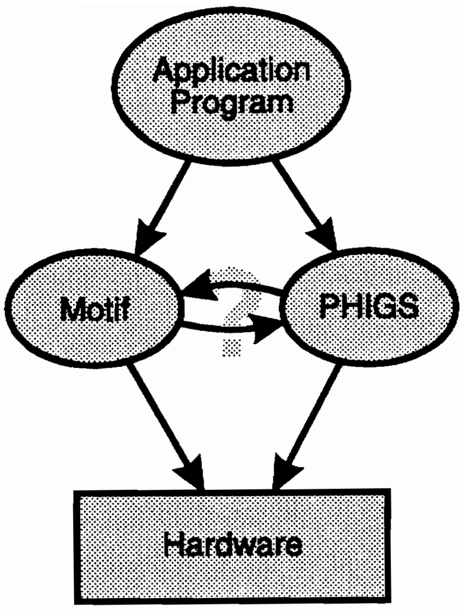


Figure 2 - Motif - PHIGS Integration

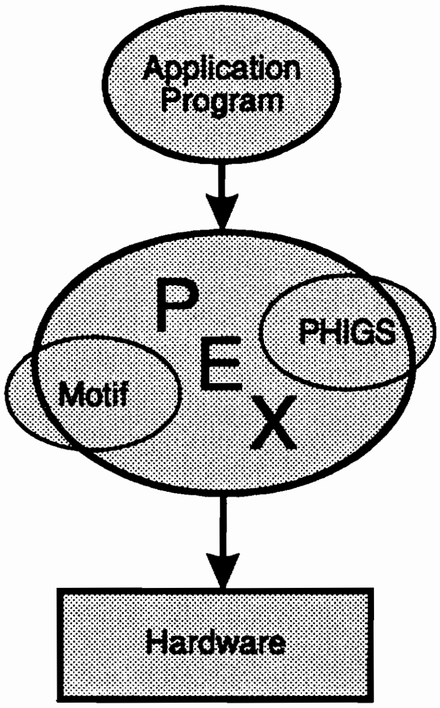


Figure 3 - Motif - PHIGS Integration Using PEX

While several sample implementations have been created, they all do so at the expense of performance. This is because most of the current implementations exist on top of other tool kits rather than as stand-alone products. The problems that exist must be resolved at the root, not with a work-around front end. Until those problems are resolved, PEX will remain a developing standard.

A more immediate solution is to develop an interface framework around PHIGS. Using PHIGS as the base code will allow PHIGS programmers to easily integrate their code with the interface. It will also allow a windowing environment to be brought to existing hardware that does not support X but does support PHIGS. Support for future standards is also assured. In the future, if PEX becomes a mature product, the interface code will be able to easily to migrate to PEX because the code is base on PHIGS. The interface framework described is the topic of this thesis. This framework will allow PHIGS programmers to use features of Motif, but now in a 3D environment.

Goals

Motif brings the workstation market a defacto standard for interface tools. These tools, as described in the previous two sections, are not perfect. The interface environment created in this project is designed to be a more viable alternative to Motif for PHIGS programmers. More specific goals are discussed in the following sections.

Emphasis on engineering needs

The interface framework must focus on the needs of engineers. This includes such things as heavy use of 3D graphics, advanced rendering, and sophisticated input models. Current interface environments put more focus on such things as font rasterizers or network communications - things of secondary interest to an engineering program. Engineers have been forced to squeeze their ideas through the tools and environments provided by computer scientists and business professionals. The design of these interface tools should alleviate those restrictions and allow the engineer to focus on the components of his program.

Compatibility with PHIGS

One of the major tools engineers can use to program on workstations is PHIGS. The PHIGS environment provides sophisticated and mature 3D display facilities. The interface framework should utilize PHIGS as much as possible. This includes using the PHIGS input devices for all input and PHIGS output for all display purposes.

Ease of use

The interface should be easy to use. If the tools provided are difficult to use, then PHIGS programmers will code their own interface tools, as they probably already do. The tools must be powerful, and yet simple enough that engineers will want to include them in their applications.

Ease of implementation

The programmer must be able to use the interface framework with a minimal amount of modification to existing code. This means that the interface framework must be as undemanding (in terms of resources and requirements) of the programmer as possible. It also means that the interface framework must not force the programmer to implement code in any specific way.

Object-Oriented

Object-oriented languages are quickly becoming very popular programming languages. This is because an object-oriented approach allows programmers to create applications that more closely model the real world. This permits programmers to think on a higher level and create much more sophisticated and reliable code. In addition to being sounder code, the code is also much more reusable.

Extensible

In addition to the set of menu items and windows provided by the interface framework, programmers must be able to implement new objects if they need to. The ability to create and implement new menu items and windows without modifying existing code must exist. This will not restrict a programmer to the set of already existing tools.

Literature Survey

This section is subdivided into the following sections:

General Literature on Interfaces - Presents some literature that describes what constitutes a good interface

Notes on X - Discusses references that describe what X is and how it works

Motif - Lists references that describe Motif

PHIGS-Based Programs - Describes several programs that have been coded using PHIGS

Previous Motif Emulation Using PHIGS - Describes two programs that provide Motif-like interfaces

CAE Environments Using PHIGS - Describes some environments created with PHIGS that are created specifically for engineers

General Literature on Interfaces

User interfaces are a highly debated topic. As such, there is a lot of literature describing what constitutes a good interface. Several of the more general articles are listed below. These sources describe what an interface should include in terms of human needs (such as visual feedback) rather than listing application-specific needs (such as check boxes or three-dimensional appearances).

Fischer documents some of the problems that exist between humans and computers [Fisc89]. Wilson describes research into new methods that improve user interaction with computers [Wils91]. Myers describes some techniques that are used to create interfaces and their advantages[Myer89]. Lee describes some tools that help build interfaces [Lee90]. Myers also describes what constitutes a window manager [Myer88].

Notes on X

A good source of information on the workings of X is provided by Scheifler [Sche86]. This article describes the components of the X window system and how they work. A more complete guide to X is provided by O'Reilly and associates [Quer90][Nye90]. These guides explain X anywhere from how to use it, to how to program with it, to how it works. These guides also provide an introduction to Motif and several other window managers.

Motif

Power Programming Motif provides an introduction to Motif concepts by guiding the user through many example programs [John91]. These programs demonstrate the different components of Motif. Visual Design with OSF/Motif describes the components of Motif in terms of their dimensions, functions, and states [Koba91]. The reader familiar with X who wants to take a look at Motif and widgets from a lower level should look at The X Window System, Programming and Applications with Xt, OSF/Motif Edition [Youn90].

PHIGS-Based Programs

VIEW (Virtual Integrated Engineering Workbench) is a graPHIGS-based (the IBM implementation of PHIGS) graphics framework that provides an integrated graphics solution for circuit designers [Rubi91]. The program provides simple menu facilities that include icon-based menus. The strength of the code lies in the ability of end users to customize an application. Users can assign functions to PF keys or mouse buttons, and specify the location of menus and views on screen. The configuration process is handled by a file that is read when the program is first executed.

Three codes developed here at Virginia Tech in the CAD lab are ACSYNT (AirCraft SYNThesis) [Wamp88a][Wamp88b][Jaya92a], Mechin (an input interface for spatial-mechanism design) [That87][That88], and MECSYN (MEChanism SYNthesis) [Mans87a][Mans87b]. All of these codes provide menu facilities that are typical of PHIGS based programs - interaction is provided through a hierarchical menu system. Selecting a menu item causes either a new menu to appear or an action to be performed.

An alternate approach to developing a PHIGS interface has recently been demonstrated by Cannon and Tuel [Cann91]. Release 2.2 of graPHIGS introduced a Motif widget called a gPWorkstation widget. This widget allows a Motif program to work with graPHIGS. Graphical output is displayed in the widget, while graPHIGS input is treated as an X event and therefore handled using callbacks. While this approach works fine for IBM machines running graPHIGS, it does not work for any other combination.

Previous Motif Emulation Using PHIGS

Previous to the work presented in this thesis, there were two attempts to emulate the features found in Motif. Neither was an attempt to create an interface framework. Nor was either project an attempt to create a full-featured product.

The first project was created by Ken Davies [Davi91]. Davies, using C and FORTRAN, created a simple menu system. The system that Davies created supported cascade menus. These menus could either remain fixed on screen or act as pop-up menus. The system provided acceptable performance, but was inflexible with respect to future expansion. The main purpose of the program was to ease the transition from character-based programs to graPHIGS-based programs.

CADAM Inc. has also created Motif emulation code [Corn91]. This code was created as part of a test to determine which was better for building interfaces, Motif or PHIGS. CADAM created code that emulated push buttons, check boxes, radio buttons, sliders, and text cells. CADAM concluded that the Motif interface was better in terms of both performance and appearance. They also noted that in programs where 3D rendering was required, Motif could not be used. As a result CADAM was waiting for the appearance of PEX, which in their minds will be the easiest way to allow Motif to display 3D objects.

Both of the Motif emulation codes developed were designed with a single purpose in mind. Neither code was created to be a general purpose interface tool kit. What both projects did demonstrate though, was that it is feasible to create an interface framework that has reasonable performance around the PHIGS graphics standard.

CAE Environments Using PHIGS

Chloé is a CAE application development environment [Guib91]. It is an environment that not only sits on top of PHIGS, but encompasses it. Chloé incorporates features such as pop-up menus, scrolling lists of options, and forms. Chloé provides a good base from which to start a graphics program, but since a user does not actually access PHIGS functions, it is really a redefinition of PHIGS. This means that considerable work would be required to use Chloé with an existing PHIGS application.

Jayaram proposed an engineering environment called CADMADE (Computer-Aided Design and Manufacturing Applications Development Environment) in his doctoral dissertation [Jaya89][Jaya90][Jaya92b]. Jayaram proposes an environment that incorporates interface code, artificial intelligence, and design and modeling tools among other things. At this time, the prototype code for the interface tools consists of code written in FORTRAN that incorporates such features as hierarchical menu structure and communication areas.

Tools Used

All work presented in this thesis was developed on IBM RS/6000's. These machines are the workstation entries of IBM. The operating system for the workstations is UNIX, more specifically AIX version 3.2.

The programming language chosen for this project was C++. This language was chosen because of its growing dominance among object-oriented languages. C++ is an extension of C. It provides all of the features essential to an object-oriented language, namely encapsulation, inheritance and polymorphism, in addition to the features present in C.

C++ was also chosen because of its availability on IBM machines. Hunt, at IBM Kingston, has created a satellite simulation environment using C++ [Hunt91]. Wampler, also at IBM Kingston, has started work on an object-oriented interface for graPHIGS - the IBM implementation of PHIGS [Wamp91b]. If this work is later adopted as a standard, then the current use of C++ will greatly facilitate later integration with the object-oriented PHIGS.

Version 3.0 of the AT&T CFRONT C++ standard was used for coding the project. As the ultimate word on C++ syntax, the <u>Annotated C++ Reference Manual</u> was used [Elli90]. As a quick reference and introductory material, the Borland C++ manuals were used. For a deeper insight into object-oriented programming, <u>The C++ Programming Language</u>, by Bjarne Stroustrup (the original creator of C++), proved invaluable [Stro91].

PHIGS was chosen as the graphics language for the project. PHIGS is an international standard and is widely available on many different platforms. Since the initial development environment was on IBM machines, IBM's version of PHIGS (graPHIGS) was used for actual coding. Some of the strengths and weaknesses of using graPHIGS for CAD programs are pointed out by Fleming [Flem91].

Program Development

The current code is the third generation of code. As acknowledged in several highly acclaimed programming books, software development is an evolutionary process [Booc91][Stro91]. The waterfall method as show in figure 4, previously considered to be the preferred programming method, has proven inadequate. The reason this model needs to be modified is that it rests on the assumption that problem requirements can be stated precisely at the beginning of a project and that complete specifications and an implementation can be derived from them through formal manipulations [Fisc89].

Object-oriented design does not follow the same rigid, formal creation process that procedural programming does. As noted by Stroustrup, the development process is characterized by three stages [Stro91]:

Analysis: Defining the scope of the problem to be solved

Design: Creating an overall structure for a system

Implementation: Writing and testing the code

These stages do not exist as separate consecutive processes. The development process is instead iterative. Each of the stages coexist and evolve as the process proceeds. Stroustrup also notes that the following five operations permeate the above three stages:

Experimentation

Testing

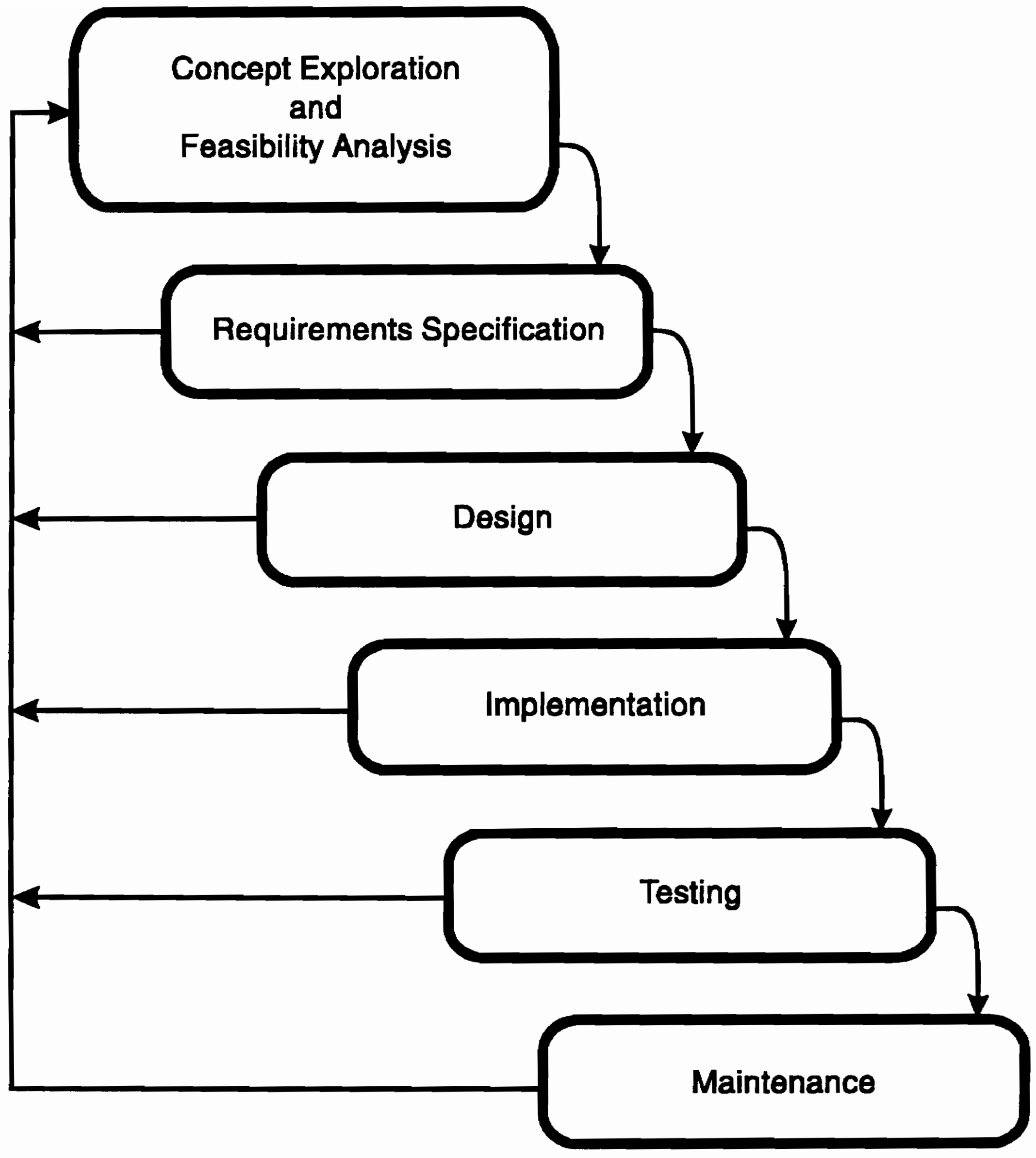


Figure 4 - Waterfall Development Cycle

Analysis of the design and the implementation

Documentation

Management

The first generation project was simply an attempt to verify the viability of interface tools using PHIGS. Internal testing by companies such as CADAM had given rise to the question of whether PHIGS performance was good enough for interfaces [Corn91].

The first generation code only implemented push buttons, check boxes and pop-up menus. The code was written entirely in C. No attempts were made to make the code extensible and so the code was not used after it was completed. Results clearly showed that acceptable performance could be achieved using PHIGS. The only place where performance became a question was on slow X-stations where PHIGS performance in general is questionable.

The second generation code was a more serious attempt at creating interface code. The goals for this phase of the project were to emulate all the major features found in Motif. This code was also developed in C. The design process was to think of all the features that menu items need and to develop a menu manager that would control those features. It was determined that all menu items exist in one of four states: active, ready, selected, or inactive. The menu items (or widgets as they are called in Motif) found in most major environments could all be controlled by switching them among these states.

At this stage in the development, the code began to be used by other people in the CAD lab. Very quickly it became evident that each individual had his own ideas of what constituted a menu item. It became apparent that tying the menu manager and menu items together by something as specific as the menu item state was too specific and limiting. New, creative menu items developed using the PHIGS 3D environment did not always conform to the previously defined states. In order to create new menu items, many work-arounds had to be developed. These shortcomings led to the third and current generation of interface tools.

The current generation tools remove as many specifics from menu item definitions as possible by allowing the menu item to control itself. While this requires more code on the menu item side, it allows the menu manager to be reused. The following section describes the design developed to model the interface framework for this third generation of code.

Class Overview

Since one of the goals of object-oriented programming is to allow programmers to more closely model the real world, the interface framework was created in terms of objects and their relationships. This is in contrast to the traditional method of structured programming, which defines relationships first, then the data passed between them. Designing in terms of objects allows a programmer to match a real world model to a computer model. This allows programs to be created at a higher conceptual level than traditional programming methods.

The first step in designing an object-oriented program is to identify the classes. This can usually be accomplished by finding all the nouns that are needed to describe the project. Once the objects have been found, the next task is to create the interface for each class - the procedures needed by an outsider to control the data inside the class.

There are five major groups of classes that define the interface framework: windows, interface managers, menu managers, menu items, and menu item managers. A window is an individual channel of communication between the application and the user. A window can be represented by a menu, some geometry, or a dialogue box. The interface manager is the central processing object. It handles the interaction between the user and the windows. A menu manager maintains a group of menu items and, combined with a window, forms a menu. Menu items are objects the user can interact with in menus. Examples are check boxes, push buttons and radio buttons. Some menu items perform specialized functions that affect other menu items. In addition to being controlled by a menu manager, these menu items must also be managed by a menu item manager.

In this thesis there is an ambiguity in the use of the word interface. Interface framework is a reference to the entire project - the menus, windows, menu items, etc. If the word interface is used alone, then the reference is to the communication channel between classes. The interface of a class consists of the functions which allow a programmer access to the data contained in the class.

There is also an ambiguity with the use of the word window. Window with a capital W is a reference to the base class called Window. Window with a lower case W is a reference to any separate pane displayed on screen. Thus pop-up menus, simple views, and geometry managers are windows derived from the Window base class.

Windows

The central focus of any GUI is windows. Their advantages and popularity are seen in the success of X Windows, OSF/Motif, Microsoft Windows, and OS/2. A window is usually represented by a rectangular pane on the computer screen. This pane can be dynamically moved and scaled during program operation. Panes can overlap, and can be raised above or lowered below one another. The information contained in a pane is dependent on the type of window it is.

Geometry managers are windows that display and maintain a PHIGS view. Pop-up menus are windows that are created and destroyed during program operation and are used to display menu items. Dialogue managers are windows that are used by a program to send messages to a user.

The powerful C++ features of *inheritance* and *virtual functions* allow the interface framework to implement an infinite variety of windows. Inheritance allows a base class called Window to be used by all classes wishing to display something on screen. Virtual functions allow all classes derived from the base class Window to perform functions specific to the window. These two features allow the interface manager to communicate with windows through a well defined channel. This guarantees that *future classes derived from the Window class will operate properly in the framework without any modification to the pre-existing framework code* - an extremely important feature.

Inheritance is a basic feature of any object-oriented language. Inheritance allows a base class of common operations to be created. Any class that inherits from this base class assumes all the properties of the base class in addition to the properties it defines. This is an important feature because it allows a programmer to think about the specific details of a class that separate it from other classes rather than the repetitive components it shares with other classes.

In terms of the interface framework, the Window class is the base class from which other window classes are derived. The Window class is a specialized base class called an abstract class. An abstract class is a class that is not physically realizable itself, and whose sole purpose is to act as a base class to be inherited by other classes. This means that no object can be declared a Window. Instead, all the different windows displayed in the framework are derived from the Window base class.

By using inheritance, the interface framework gains four advantages over applications programmed in traditional, non object-oriented languages:

- The avoidance of repetitive programming in window classes
- Application programmers can remain ignorant of implementation details
- Modifications to the Window base class will not require changes to the derived classes
- A consistent interface to each window is guaranteed to exist

Because all window classes must be derived from the base Window class, all the derived window classes contain a common set of features. These common features need not be recreated each time a new window class is implemented. This not only saves time, but more importantly eliminates the chance of bugs being introduced in the re-coding of repetitive code.

An implementation detail of the Window class is that each window is a member of a doubly linked list. This feature is required by the interface and means that each window, among other things, must be able to add and remove itself from a linked list. This is a task a programmer interested in creating a window may not know how or care to do. Fortunately the programmer does not have to worry about these details because any class derived from the Window class will already contain these functions. This allows the programmer to focus on the conceptual parts of the window rather than the minor details required to make it fit into the interface framework.

Because each derived window class is ignorant of the doubly linked list previously mentioned, the interface framework is free to change implementation details. For instance, if it were determined that it would be more efficient to change the linked list to a singly linked list, then modifications would only need to take place in the Window base class. All classes derived from the Window class would then reflect the change without any modification to their code. This not only allows for future optimizations of code, but also facilitates the porting of code to new systems, where it may be necessary to modify implementation specifics.

Since each window class contains a common set of features and procedures, it is possible for the framework to communicate with each of the windows through this common interface. By interacting with only the functions declared in the base Window class, the interface framework is guaranteed to be compatible with any class derived from the

Window class. This important feature allows future window classes to be created without any modification to the existing framework code.

Figure 5 shows the communication channels within the interface framework. The diagram consists of four objects: an interface manager, a dialogue manager, a geometry manager, and a pop-up menu. The latter three classes are each windows and therefore are each derived from the Window base class. This is signified by the Window object inside each of the objects. The arrows represent C++ pointers from one object to another. A line with arrows at both ends means that each object contains a pointer to the other object.

The interface manager contains a single reference to one window. This window is the first window in a doubly linked list of windows. Since it cannot be known at compile time how many windows will be present in the interface framework, the linked list methodology must be utilized. By referencing only the first window, the linked list can grow and shrink dynamically during program operation without affecting the interface manager. The drawback to this system is that to communicate with a window other than the first window, the interface manager must traverse the linked list until the window is found.

Each window, in contrast with the interface manager, contains a pointer back to the interface manager. This can be hard coded because each window will never have more than one interface manager. To communicate with the interface manager, each window needs only to reference its pointer.

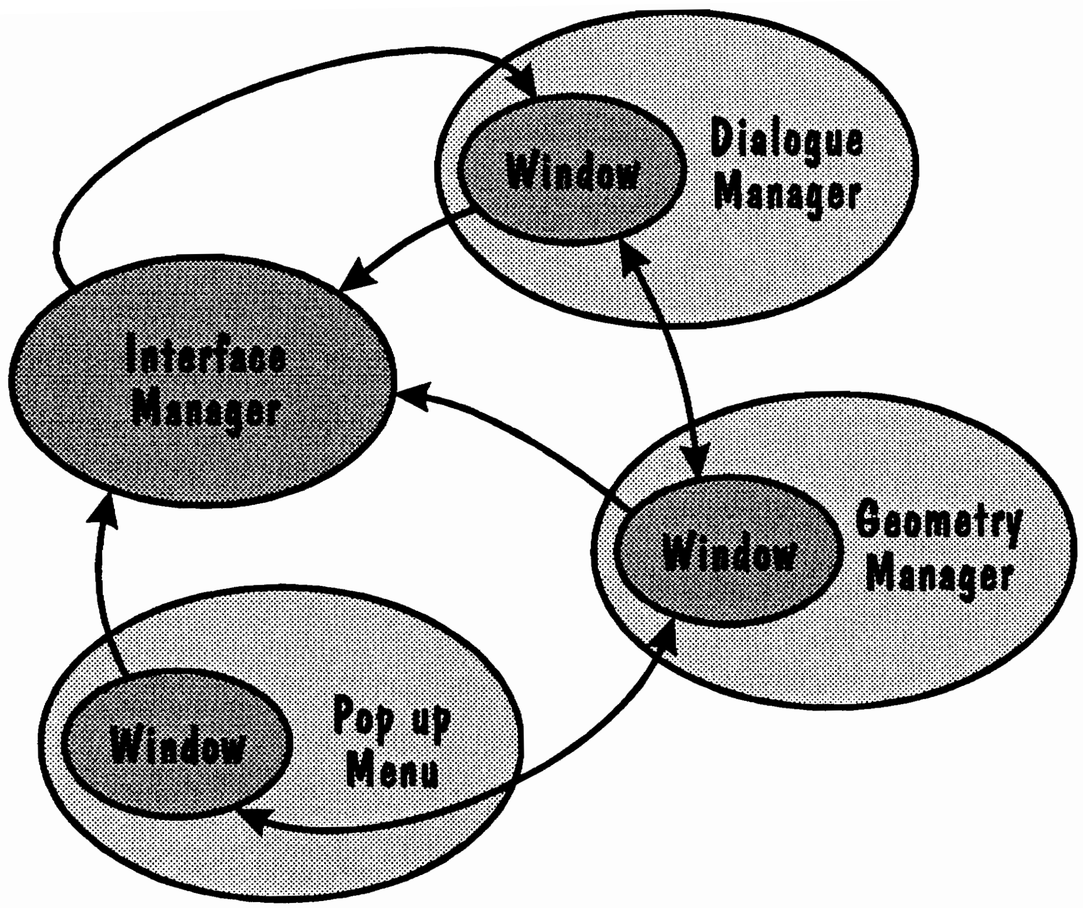


Figure 5 - Interface Manager Communication Channels

The significance of the diagram is that each of the pointers is either originating or terminating in a Window base class. The actual class (dialogue manager, geometry manager, or pop-up menu) that is being communicated with is irrelevant. It can be easily imagined that if a new window class is developed, it will be easy to integrate into the existing system as long as it inherits from the Window base class.

The question arises that if the interface manager can only communicate with Window base classes, how does the interface framework direct the derived window to perform functions specific to the window. The answer is that functions not provided by the Window base class are handled using virtual functions.

Each window performs its own specialized functions. A menu must control a set of menu items, while a geometry manager is responsible for displaying geometry and a dialogue manager must display text. No matter how insightful the Window base class is, it is impossible to devise enough functions to meet all possible needs. It is inevitable that somebody in the future will desire a function not currently provided. This is where virtual functions come in.

A virtual function is a function that is declared in a base class and defined in a derived class. This means that if the interface manager wants to raise a window, it will request the action from a Window base class and a window derived class will perform the action. Instead of enforcing the rules in the Window base class, control is instead passed onto the derived class. This allows the Window base class (and therefore the interface manager) to retain conceptual control of windows without needing to know about any implementation details.

By declaring all functions that require actions specific to a window as virtual functions, compatibility with just about any type of window is assured. Now when a new window class is created, the implementation specifics are left to the window. The framework does not assume anything related to how the window should perform its actions.

Each window must be able to provide the following virtual functions:

```
is_raisable()
lower()
raise()
is_locator_in_window()
process_from_mouse()
process_from_keyboard()
manage()
unmanage()
```

The is_raisable() function is used by the interface manager to ask a window if it can be raised. The raise() and lower() operations are requests from the interface manager to move the window on top of or below the other windows. Using PHIGS this is usually accomplished by changing view priorities. The is_locator_in_window() function is a request to the window to determine if the locator is currently over the window. The process_from_mouse() and process_from_keyboard() functions are each called when it is determined that user input is directed to one of the windows. The manage() and unmanage() calls are used to tell the window to create and destroy itself.

Interface Managers

The interface manager is the connection between the application program, the windows used by the program, and the user of the program. The interface manager is responsible for two major tasks:

- Maintaining the priorities (the raising and lowering) of the windows displayed.
- Channeling input from the user to the proper window, and if necessary, from the window back to the application program.

The interface manager performs these functions by communicating with each of the windows through the functions that the windows inherit from the base Window class and by maintaining two variables: a current window pointer and a raise-operation flag. The current window pointer is a pointer to the last window to be raised. This is the window that is currently on top of all the other windows and, in general, the window the user is currently operating with. The raise-operation flag indicates if windows can currently be raised. If the flag is set to NO, then no windows can be raised above the current window. This effectively forces a user to interact with a single window until the raise-operation flag is set back to YES.

An outline of the main processing routine of the interface manager is given in Fig. 6.

```
• wait for input for a user specified amount of time
• if no interaction took place in the specified amount of time, then
    • return
• else if a choice class (mouse or keyboard) input was detected, then
    • if the choice device was the mouse, then
        • if the window is not the current window, then
            • if the raise_operation_flag = YES, then
                • determine which window the cursor is in by
                running through the linked list of windows until one of
                the windows determines that the cursor is in it or the
                end of the linked list is reached
                • if a window is found, then
                    • if the window can be raised
                        • raise the window
                    • redirect the input to the window
                    • return
                • else the cursor was not in any window
                    • return
            • else the raise_operation_flag = NO
                • return
        • else the window is the current window
            • redirect the input to the window
            • return
    • else if the choice device was the keyboard, then
        • defer processing to the current window
        • return
• else input is from some other device class
    • return
```

Figure 6 - Interface Manager Processing Outline

The outline in Fig. 6 is very nonspecific for a reason - it is not possible for the interface manager to assume anything about the windows it manages. The philosophy is that the interface manager tells each window what to do, but not how to do it. This allows the interface manager to remain very conceptual in nature.

By using the virtual functions provided by each window, the interface manager can request actions to be performed by the windows without any knowledge of how those actions are being performed. The interface manager effectively communicates with each window without knowing or caring what kind of window it is. As far as the interface manager is concerned, it is always communicating with the Window base class. This is the key feature that makes future expansion easy to achieve. As long as the window is derived from the Window base class, it can communicate with the interface manager.

Referring back to Fig. 5, this relationship is once again acknowledged. It can be seen in the figure that the interface manager only has contact with the Window base class. The interface manager never communicates directly with any of the derived window classes, rather only with the Window base class in each of the windows. Of course, through the use of virtual functions, each Window base class communicates with the derived windows, but the interface manager is not aware of or affected by this.

Menu items

Menu items are the individual entities that users interact with in menus. They are roughly the equivalent of Widgets in Motif. A menu item may take the form of a push button, a radio button, a check box, etc.

The menu item to menu manager analogy is roughly the same as the window-to-interface manager analogy. All menu items inherit from the Menu item base class and menu managers only communicate with the base Menu item class. As with the window classes, when referring to Menu items with a capital M, the reference is to the base class. Menu item with a lower case M is a reference to a class derived from the Menu item base class.

Again, just like the window classes, it is the C++ features of *inheritance* and *virtual functions* that provide the flexibility in menu items. Inheritance allows the base Menu item class to be included in each menu item class. Virtual functions allow each menu item to individualize how it performs its functions. Together, virtual functions and inheritance let menu managers communicate with just about any kind of menu item.

By using inheritance for menu items, the interface framework gains the same four advantages that window classes do over applications programmed in traditional, procedural languages:

- The avoidance of repetitive programming in menu item classes
- Application programmers can remain ignorant of implementation details
- Modifications to the Menu item base class will not affect the derived classes

- A consistent interface to each window is guaranteed to exist

By inheriting from the Menu item base class, implementing new menu items involves only the creation of the menu item specific elements. The features similar to all menu items will be provided by the Menu item base class. Because each menu item always inherits from the Menu item base class, the application programmer does not need to know how the functions in the base class are performed. Since the programmer remains ignorant of the Menu item base class details, they are free to be changed without affecting the application programmer. This need may arise when the code needs to be either optimized or modified to include new features. Communicating with menu items is very easy because they each contain the interface provided by the Menu item base class. This allows a menu manager to communicate with a menu item in a very predictable manner.

Figure 7 shows the communication channels between the menu manager and the menu items. The diagram consists of four objects: a pop-up menu (which is inherited from both a menu manager and a window), a push button, a radio button, and check box. The latter three classes are each menu items and therefore are each derived from the Menu item base class. This is signified by the Menu item object inside each of the objects.

The menu manager contains a single reference to one menu item. This menu item is the first menu item in a doubly linked list of windows. Since each menu will manage a different number of menu items, the linked list methodology must be utilized. By referencing only the first menu item, the linked list can grow and shrink dynamically during program operation without affecting the menu manager. The draw back to this system is

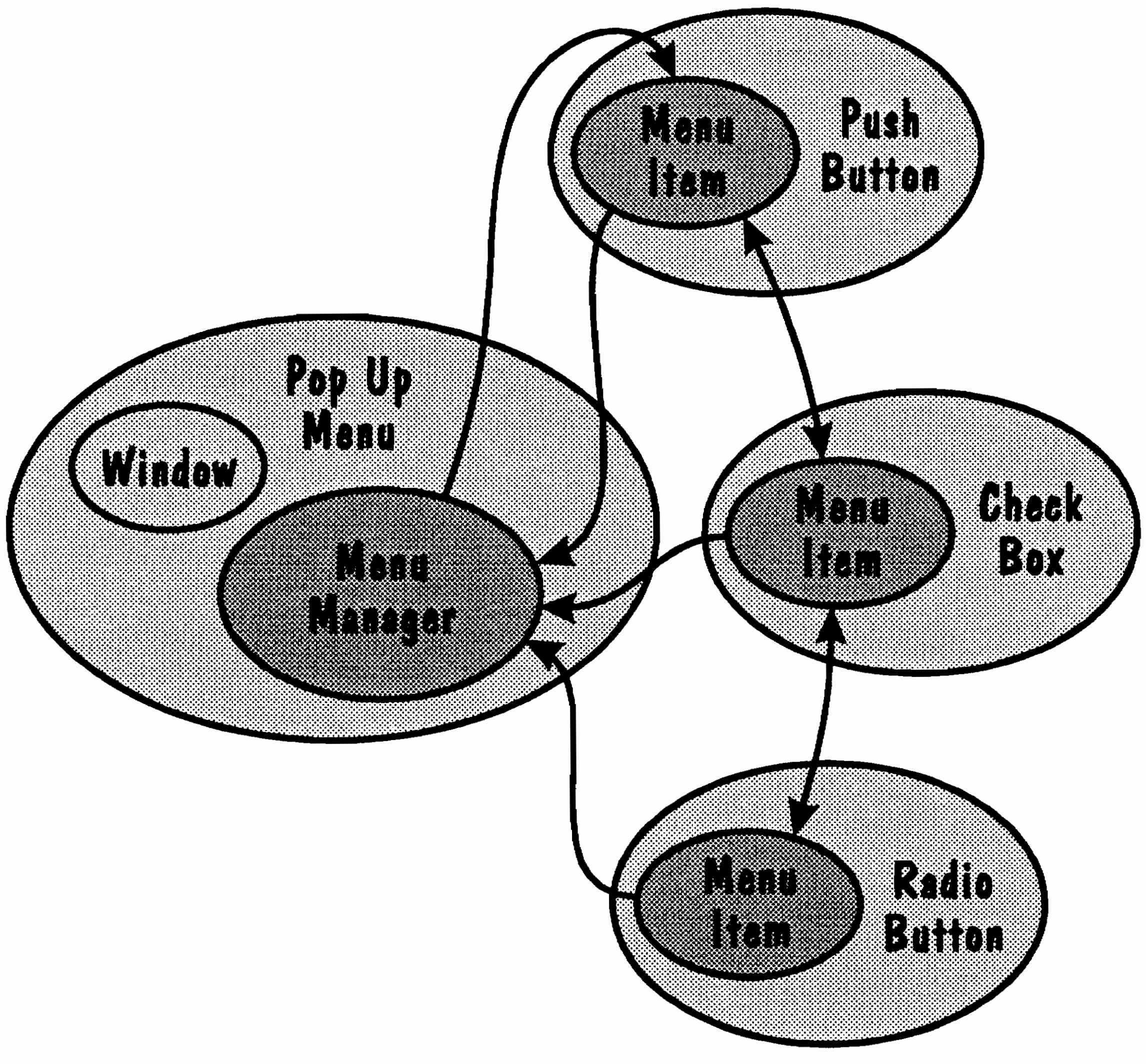


Figure 7 - Menu Manager Communication Channels

that to communicate with a menu item other than the first menu item, the menu manager must traverse the linked list until the menu manager is found.

Each menu item, in contrast with the menu manager, contains a pointer back to the menu manager. This can be hard coded because each menu item will never have more than one menu manager. Therefore to communicate with the menu manager, each menu item needs only to reference its pointer.

The significance of the diagram is that each of the pointers in the menu items is either originating or terminating in a Menu item base class. The actual class (push button, radio button, or check box) that is being communicated with is irrelevant. It can be easily imagined that if a new menu item class is developed, it will be easy to integrate into the existing system as long as it inherits from the Menu item base class.

Similar to the way the interface manager communicates with windows, the menu manager communicates with the menu items through the use of virtual functions. These functions are declared in the Menu item base class and defined in the derived menu item classes. This allows the menu manager to tell the menu items to perform certain actions without actually telling them how to perform the actions.

The importance of using inheritance and virtual functions shows itself in the creation of new menu items. While the need for new windows will probably be minimal, the need for new menu items will almost always exist. Environments such as Motif and Microsoft Windows provide a fixed set of interface tools. These tools perform functions that allow a user to turn an option on or off or allow the user to pick from a list of items. These tools are fine in the business environment, but in the engineering environment where visualization is a premium, more sophisticated tools are needed in addition to those tools. Menu items that show the rotation of an axis, or the Gouraud shading of a sphere could be used in many places, but are very difficult to create using existing tools.

By inheriting from the Menu item base class and interfacing with the virtual functions, an unlimited number of new menu item types can be easily created. In all cases, the menu manager thinks it is communicating with the Menu item base class. Thus by utilizing the virtual functions in the base class, any new menu item will appear to the menu manager as an object of the type of the base Menu item class. The virtual functions in the Menu item base class are:

```
manage()
unmanage()
is_cursor_over_menu_item()
process_from_mouse()
process_from_keyboard()
is_highlightable()
highlight()
unhighlight()
get_highlight_condition()
save_state()
revert_to_saved_state()
generate_delayed_events()
```

The first two functions, manage() and unmanage(), are used to tell the menu items to create and destroy themselves. This usually means creating and deleting the PHIGS structure used to display the menu item. The manage() and unmanage() calls are called by the menu manager when the menu items need to be added to the display and removed from the display.

The is_cursor_over_menu_item() call is used by the menu manager to determine if the cursor is currently over the menu item. By deferring this function to the menu item for processing, the menu item can be unrestricted in its shape. If the testing were performed in the menu manager, then the menu manager would have to assume something about the shape of menu items, i.e., that they are all circular or they are all rectangular. Deferring the function also allows a menu item to decide if it can be picked. For instance, a menu item may be in an inactive state. Even though the cursor may be over the menu item, the menu item may tell the menu manager that it is not because it is inactive.

The process_from_mouse() and process_from_keyboard() calls allow the menu item to perform its own processing. This not only allows each menu item to perform differently, but also bypasses interaction with the menu manager, which reduces overhead.

The process_from_keyboard() procedure is called for the highlighted menu item when a key press is detected. Highlighting is necessary to distinguish which menu item is currently being interacted with, and to determine which menu item to redirect keyboard input to. The highlight() and unhighlight() functions are called by the menu manager to add and remove highlights. The get_highlight_condition() is called to check to see if a menu item is currently highlighted. The is_highlightable() function is called to determine if a menu item can be highlighted. An example of a menu item that cannot be highlighted is a text label. The label will exist on the menu, but since mouse or keyboard input will never be directed towards the label, there is no need to highlight it.

Most menu items exist in several states. For example, a push button can either be pressed or raised. Changing the state of a menu item causes an event to be generated. Events can be generated at two different times: when the interaction with the menu item occurs, or when the menu that the menu item is in is closed. The second option usually occurs in a pop-up menu when several options are picked, and then told to perform when the pop-up menu is closed. If after picking the options, the user changes his mind, an option to cancel the choices is usually also available. This will cause the menu items to revert to their saved states using the revert_to_saved_state() call. The state of each menu item is always saved each time a menu is opened using the save_menu_item_state() call. If the menu is not closed with the cancel option, then each menu item is told to perform its delayed events with the generate_delayed_events() call. This function usually compares the saved state to the current state and generates the proper events if they differ.

Menu managers

Menu managers control the operation of a set of menu items. Operation is similar to the interface manager-window control scheme. The menu manager contains a pointer to one of the menu items. The remaining menu items are joined in a doubly linked list. To reference a menu item other than the first menu item, the menu manager traverses the linked list until the menu item is reached.

Unlike the interface manager, a menu manager is not a stand-alone class. Using the C++ multiple inheritance mechanism, the menu manager base class and Window base class are usually inherited together to form a new menu class. Examples of this type of class are pop-up menus and static menus. Both classes exhibit the properties of windows and menu managers.

The existence of the menu manager as a distinct and separate class allows future menu classes to be created. Currently both the static menu class and the pop-up menu class are both based on 2D views. It is conceivable that in the future, it may be desirable to have a menu based on a 3D view. By inheriting the menu manager class, the new class will not have to recreate the routines that control the menu items.

Creating a new menu class is usually straightforward. Most of the display properties are handled by the Window base class, while the input handling routines are taken care of by the menu manager. User input is directed by the interface manager to the proper window. If the window is a menu, then the input is redirected to the menu manager which in turn redirects the input to the proper menu item.

Figure 8 shows a typical user input interaction scenario. The interface setup consists of three windows, one of which is a menu that is managing three menu items. In the sample setup, input is first received by the interface manager. The interface manager first asks the first window (the dialogue manager) if it can use the input. If the window can use the input, then the window does so until the interaction is complete. If the window cannot use the input, then the interface manager checks the next window in the linked list of windows. This process continues until a window can use the input, or all of the windows have been checked.

A special case occurs with the processing of a class that inherits from the menu manager class. When input is sent to a menu, the menu manager checks each of the menu items to see if they can handle the input. If a menu item is found that can process the input, then it does so, otherwise control is passed back to the menu manager.

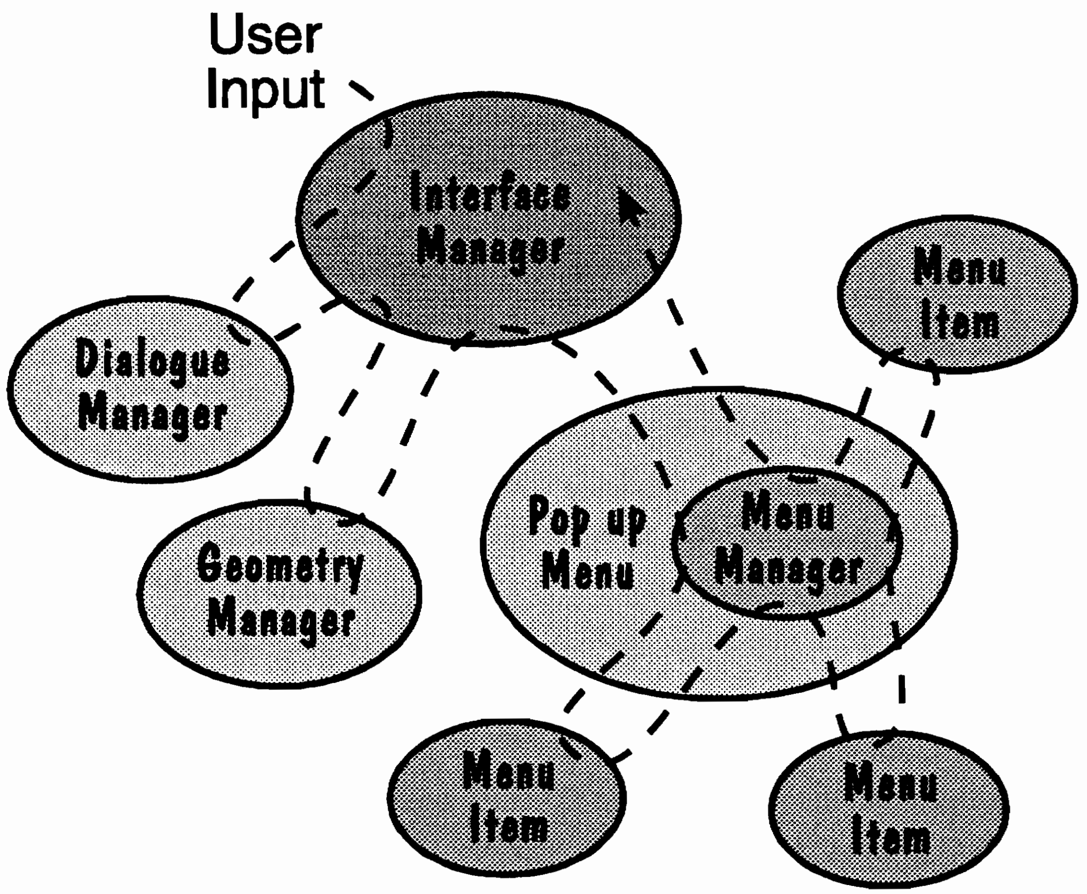


Figure 8 - User Input Path Trace

Menu item managers

Some menu items do not act independently of other menu items. An example of this is a radio button. In a group of radio buttons, only one can be selected at a time. Thus the selection of one radio button means that another radio button must be deselected. This means that there must be some object which both defines the group of radio buttons, and handles the selection process for the radio buttons. This object is a menu item manager.

The menu item manager class is similar to the menu manager class in that it contains a reference to a doubly linked list of menu items. This linked list is the menu items to be managed by the menu item manager. During processing of the menu item, if some sort of action occurs that affects the other menu items in the group, then control is passed to the menu item manager.

Figure 9 shows the communication channels that a menu item manager uses. The figure consists of five menu items, a menu manager and a menu item manager. The menu manager is actually a radio button manager - a specialized menu item manager designed to manage radio buttons. Arrows in the figure indicate pointers from one object to another. Lines with arrows at both ends mean that each object points to the other object.

Though not shown in the diagram, each pointer to and from the menu items actually comes from the Menu item base class that each menu item was derived from. This means that, like the menu manager, the menu item manager only communicates with the Menu item base class. This assures that the menu item manager class will be compatible with future classes derived from the Menu item base class.

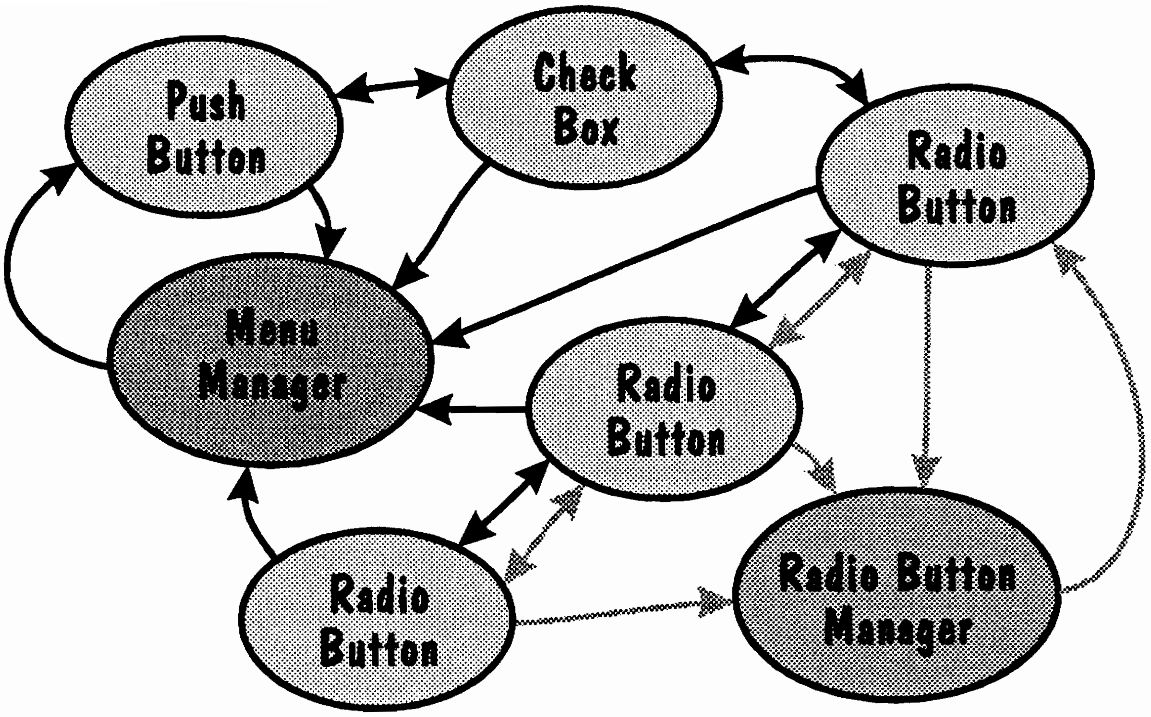


Figure 9 - Menu Item Manager Communication Channels

Although appearing complicated, the system in Fig. 9 mainly consists of only two doubly linked lists. The first list links all the menu items and is utilized by the menu manager. This is the linked list that the menu manager traverses when it needs to access one of the menu items. In addition to this linked list, each menu item also contains a pointer to the menu manager. The black arrows are the pointers associated with menu manager.

The second linked list connects all the menu items controlled by the menu item manager. These menu items also contain a pointer back to the menu item manager. The pointers associated with the menu item manager are shaded gray.

Class Organization

Figure 10 shows how all the classes described so far fit together. The figure uses two kinds of relationships to describe the organization of the classes: control and inheritance relationships. Two examples of how the diagram can be read are as follows:

1. A simple view is a window that is controlled by an interface manager which is a view manager.
2. A radio button is a) controlled by a radio button manager which is a menu item manager and, b) a menu item that is both a menu item linked list and a managed menu item linked list.

Several new, less important classes are introduced in Fig. 10. These additional classes are:

- Menu Item Linked List
- Managed Menu Item Linked List
- Window Linked List
- View Manager

The reason these classes are not discussed is because they are base classes to the classes of the interface framework that programmers will use. This means that these classes carry out implementation specifics and do not ever need to be accessed by the programmer. Also, since they are inherited by other classes, the features they possess are assumed by the inherited classes. Thus a user of the inherited class does not know if the functions are defined in the derived class or somewhere else in a base class. All functions appear to the user as if they are one consistent group belonging to the derived class.

An example of one of theses functions is the request_next_available_view() call from the View Manager class. This function allocates PHIGS view indices for the interface to use. Since the Interface Manager class inherits from the View Manager class, it assumes this function. The user of the Interface Manager class never knows that the class is inherited from another class. For this reason calls like the request_next_available_view() call are documented in the Interface Manager class, and the View Manager class is not formally documented.

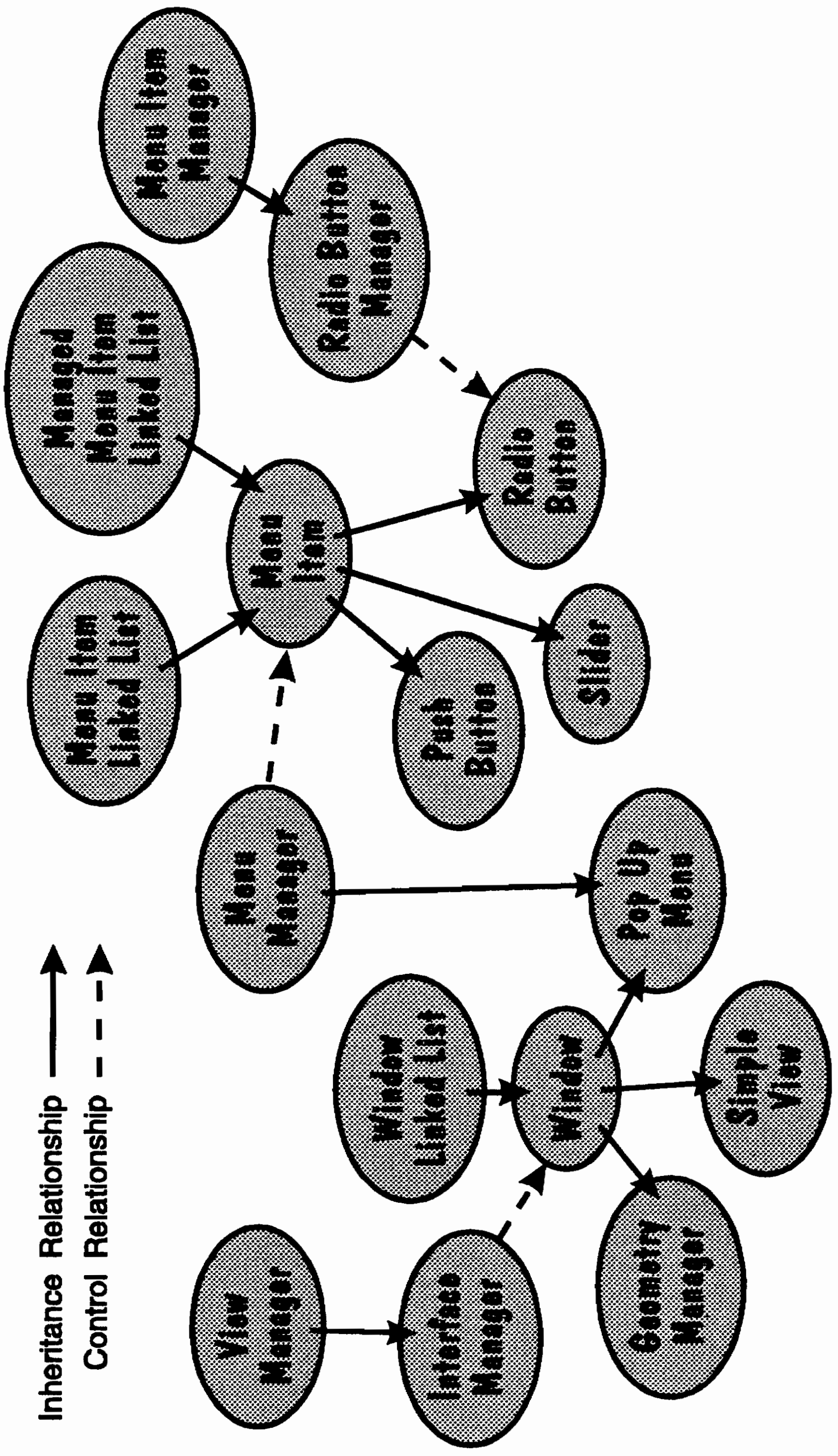


Figure 10 - Class Organization

Menu Item Protocol Specification

The menu item protocol specification defines the communication channel between menu items and the menu manager. This is accomplished through virtual functions in each menu item base class. This section provides a description of each of the virtual functions that must be defined for the menu item derived class.

Since the menu item base class is an abstract class, each of the following functions *must be defined* for the derived menu item. If one of the functions does not apply to a specific menu item, then the function can be defined to just return. Even though it does nothing, it *still must be defined*. An example of this is a label. Labels do not interact with users and so there is never a need to highlight or unhighlight them, but the highlight() and unhighlight() functions must still be defined.

This section and other sections define several numerical constants used by the interface. An example of two constants are the YES and NO values. These two values are equivalent to 1 and 0. Since it is much easier to confuse the meaning of 0 and 1, the YES and NO constants are used in their place. Any word spelled in all capitals is a constant that represents some numerical value. These values are stored in one location in a file called interface.h that can be accesses by programmers.

manage()

When a menu manager class is told to display itself by the interface manager, it does so by telling each menu item that it contains to manage itself. This is done with the manage() function. The function that each menu item should perform when it is told to manage itself is to associate its structure with the view specified. This means that the structure for the menu item should either have already been created or created as the first step of the function. There should be no workstation updating in this function - the workstation will be updated once all of the menu items have managed themselves.

```
void    manage( wsid, view_index, priority )
```

wsid - *integer*. This is the PHIGS workstation identifier.
view_index - *integer*. This is the view that the menu item will be displayed in.
priority - *float*. This is the priority value to use to associate the menu item structure with the view.

unmanage()

The unmanage() call is issued when a menu is closed. The menu item should disassociate itself with the view it is currently associated. A simple and convenient way to do this is to delete the structure used to hold the menu item. This function should also not contain any update workstation calls as that will be done once all menu items have been unmanaged.

```
void    unmanage()
```

is_highlightable()

There is always one highlighted menu item in a menu. This is the menu item that was last interacted with. The menu manager will try to highlight a menu item for two reasons:

1. The TAB key was pressed so the next menu item in the menu should be highlighted.
2. The user clicked the mouse over one of the menu items.

Keyboard input is redirected to the highlighted menu item.

Some menu items such as labels or separators do not interact with the user. These menu items do not ever need to be highlighted. The following function is called by the menu manager to determine if it can highlight the menu item.

```
int     is_highlightable()
```

Return Value - YES or NO.

highlight()

If a menu item can be highlighted, then the following functions will be called to highlight and unhighlight the menu item.

```
void    highlight()
```

unhighlight()

```
void     unhighlight()
```

Even if the menu item cannot be highlighted, the above functions must still be defined for the menu item. In those cases, the function can simply be defined to do nothing but return.

get_highlight_condition()

If the menu manager needs to determine if a menu item is highlighted, it will call the following function:

```
int      get_highlight_condition()
```

Return Value - YES or NO.

For menu items that cannot be highlighted, this function should always return NO.

is_cursor_over_menu_item()

This function is called by the menu manager when a mouse button is pressed. It is used to determine which menu item the mouse button was pressed over.

```
int      is_cursor_over_menu_item( x, y )
```

x, y - *floats*. These are locator values in the menu.

Return Value - if the coordinates are in the menu item then return YES, else return NO.

Menu items that do not interact with the user should return NO regardless of whether the cursor was over the menu item or not.

process_from_mouse()

These are functions that are used to generate events for the menu item. Even if a menu item cannot generate an event, these functions still must be defined.

Once the menu manager has determined that the user is pressing a mouse button over a menu item, it will call the following function for the menu item.

```
void    process_from_mouse( x, y, event )
```

x, y - *floats*. These are the coordinates in the menu that the locator triggered the menu item with.
event - *pointer to type Event*. This is a pointer to one of the members of the linked list of events. If the menu item generates an event, it should add it to this list.

The function is called when the mouse button is first pressed. The menu manager assumes that when the call returns, the mouse button will have been released. **It is the job of the menu item to wait for the user to release the mouse button**.

process_from_keyboard()

If a menu item is highlighted and the user presses a key on the keyboard, then this function will be called.

```
void     process_from_keyboard( key_press, event )
```

key_press - *integer*. This is the PHIGS key identifier for the key press.
event - *pointer to type Event*. This is a pointer to one of the members of the linked list of events. If the menu item generates an event, it should add it to this list.

save_state()

When a menu is first opened, the menu manager will send a signal to the menu item that it should save its state. This state will then later be used to either generate delayed events or to be reverted to.

```
void     save_state()
```

generate_delayed_events()

The following function is called for each menu item when a menu is closed. This function should compare the state of the menu item to the state that is was in when the menu was opened. If the state has changed, then the menu item should now generate an event.

```
void     generate_delayed_events( event )
```

event - *pointer to type Event*. This is a pointer to one of the members of the linked list of events. If the menu item generates an event, it should add it to this list.

revert_to_saved_state()

If a menu is cancel closed, then menu items in the menu will be requested to revert to the state they were in when the menu was opened. If the state of the menu item has not changed since the menu opened, then this function can just return.

```
        void      revert_to_saved_state()
```

Window Protocol Specification

The window protocol specification defines the communication channel between windows and the interface manager. This is accomplished through virtual functions in each window class. This section provides a description of each of the virtual functions that must be defined for the window class.

Since the Window base class is an abstract class, each of the following functions ***must be defined*** for the derived window. If one of the functions does not apply to a specific window, then the function can be defined to just return. Even though it does nothing, it *still must be defined*. An example of this is a simple view. Simple views do not interact with users and so there is never a need to process them from the mouse or keyboard, but the process_from_mouse() and process_from_keyboard() functions must still be defined.

Often the functions will require information from the interface manager. Access to the interface manager is provided when the window is added to the interface manager. At this time a pointer to the interface manager will be stored in the Window base class. This means that each derived window class will automatically have access to the interface manager; a special field in the window derived class does not need to be created.

manage()

This function is called when the window is added to the interface manager. The routine should perform the following functions:

1. Request any needed view indices from the interface manager with the request_next_view_index() call.
2. Initialize these views - set proper values and turn them on.

Immediately after the manage() call is performed, the interface manager will call the raise() routine for the window. Therefore the window should have completely initialized itself by the time the manage() call is finished. This function should not perform any workstation updating.

```
void    manage()
```

unmanage()

This function is called by the interface manager when a window is closed. The function should reverse the action performed by the manage() call. Namely, the routine should do the following:

1. Turn off the views used by the window.
2. Release the views back to the interface manager with the make_view_available() call.

As with the manage() call, this function should not perform workstation updating.

```
void    unmanage()
```

is_raisable()

Before a window is raised or lowered, the interface manager first asks the window if it can be raised or lowered. This function is useful to prevent the interface manager from raising windows that do not need to be raised. Raising a window requires updating the entire workstation and if the purpose of a window is solely for viewing and it is not obscured by another window, then there is no reason to raise it.

```
int      is_raisable()
```

Return Value - YES of NO.

raise()

When the interface manager needs to raise a window, it will call this function for the window. The function should raise the window above all other windows in both input and output priorities. No workstation updating should take place in this function.

```
void     raise()
```

lower()

Lowering a window is the opposite of raising a window. The function should lower the window below all other windows in both input and output priorities. No workstation updating should take place in this function.

```
void     lower()
```

is_locator_in_window()

When the mouse button is pressed, the interface tries to determine which window the cursor is currently over. This function is a request from the interface manager asking if the following values indicate that the cursor is in the window. This function is usually performed by comparing the window's view index and the view index passed in as an argument.

```
int      is_locator_in_window( x, y, view_index )
```

x, y - *floats*. These are locator values retrieved from sampling the locator when the mouse button was pressed.
view_index - *integer*. This is the view index retrieved from sampling the locator when the mouse button was pressed.

Return Value - if the arguments indicate that they are in the window then return YES, else return NO.

process_from_mouse()

Once the interface manager determines that the cursor was clicked over a window it will defer processing to that window. This function is designed to let the window perform its own specific functions. The only responsibility of this function (with respect to the interface manager) is to wait for the mouse button to be released before returning.

```
void     process_from_mouse( x, y, view_index, event )
```

x, y - *floats*. These are locator values retrieved from sampling the locator when the mouse button was pressed.
view_index - *integer*. This is the view index retrieved from sampling the locator when the mouse button was pressed.
event - *pointer to type Event*. This is a pointer to one of the members of the linked list of events. If the window generates an event, it should add it to this list.

process_from_keyboard()

When a key on the keyboard is pressed, the interface manager will pass on that key press to the currently active window (the window which is on top of the other windows).

```
void    process_from_keyboard( key_press, event )
```

key_press - *integer*. This is the PHIGS key identifier for the key press.
event - *pointer to type Event*. This is a pointer to one of the members of the linked list of events. If the window generates an event, it should add it to this list.

get_view_priority()

This function should return the view priority of the window. If the window contains multiple views, the priority of the highest priority view should be returned. The function is used by the interface manager to determine which window is currently on top of all the other windows.

```
int     get_view_priority()
```

Return Value - The priority of the window.

Sample Usage

This section describes a sample program. The program operation is first described. The steps needed to create a program are then discussed. Lastly the code for the sample program is listed.

Sample Program Operation

The sample program shown in Fig. 11 displays three simple views, a static menu, and a pop-up menu. The static menu contains two push buttons and a slider. One of the buttons exits the program and the other opens the pop-up menu. The pop-up menu contains one push button that causes the pop-up menu to close.

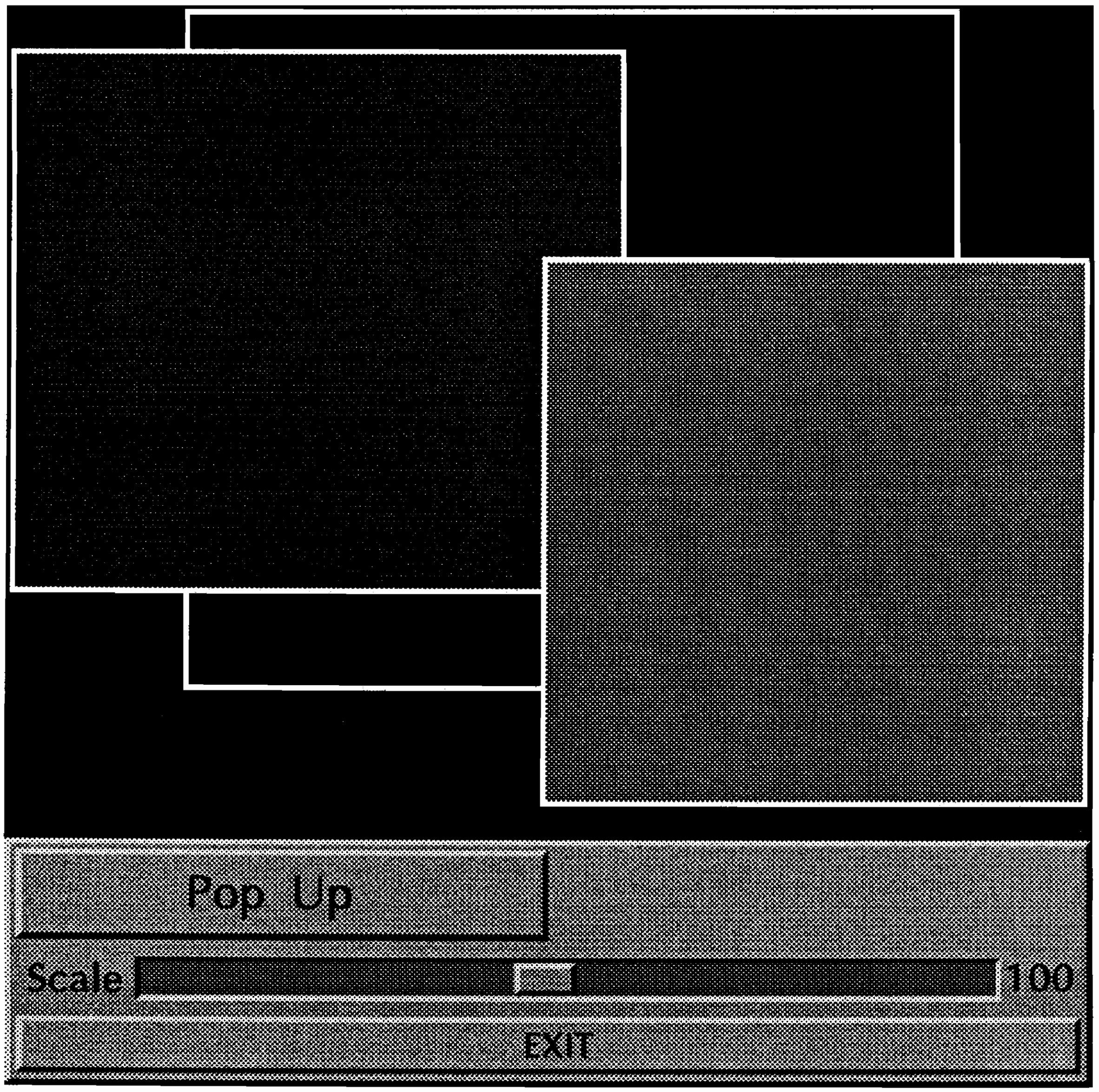


Figure 11 - Sample Program Display

Two of the three simple views can be raised by clicking on them. The third window cannot be raised by the user. Each time a window is raised, the program prints out a character string identifier associated with the window. The value of the slider is also printed out when it is changed.

Program Coding Steps

Each program using the interface framework performs the same basic operations. The steps are listed below:

- Create interface manager
- Create windows
- Create menu items

Although creating the interface manager is the first step in using the interface framework, creating windows and menu items can occur at anytime. For example, a pop-up menu and its menu items do not need to be created until they are used. The objects do not need to be created at the start of the program.

Creating the interface manager involves invoking an instance of the Interface manager class. This step can also include specifying such things as telling the interface manager which views not to use.

The create windows step consists of the following three phases:

phase 1 - Create. Instantiate one of the window classes.

phase 2 - Apply attributes. If needed, apply additional attributes to the window. This step can include things such as making a window raisable, or in the case of a menu, adding menu items.

phase 3 - Add to interface manager. Put the window under control of the interface manager.

Creating menu items is part of the second phase of creating a menu window. The phases involved in creating menu items follow the same pattern as creating windows and are:

phase 1 - Create. Instantiate one of the menu item classes.

phase 2 - Apply attributes. If needed, apply additional attributes to the menu item. This step can include things such as setting the initial value or state of a menu item.

phase 3 - Add to menu. Put the menu item under control of a menu.

Sample Code

```
#include <stdio.h>

// include files that are needed for this sample program
// for each different menu item or window type, you will have
// to include the appropriate header file
#include "/u/scott/c++/object_files/afmnc++.h"
#include "/u/scott/c++/object_files/color.h"
#include "/u/scott/c++/object_files/event.h"
#include "/u/scott/c++/object_files/interface.h"
#include "/u/scott/c++/object_files/interface_manager.h"
#include "/u/scott/c++/object_files/open.h"
#include "/u/scott/c++/object_files/pop_up_menu.h"
```

```
#include "/u/scott/c++/object_files/push_button.h"
#include "/u/scott/c++/object_files/simple_view.h"
#include "/u/scott/c++/object_files/slider.h"
#include "/u/scott/c++/object_files/static_menu.h"

// prototype procedures later defined and used in this file
void do_pop_up( Interface_manager* );

// define some structure numbers to be used later
#define PB1_STRUCT          1
#define PB2_STRUCT          2
#define S_STRUCT            3
#define MENU_STRUCT         4
#define POP_UP_STRUCT       5
#define POP_UP_UTIL_STRUCT  6
#define OK_STRUCT           7

void main()
   {
   // create an instance of the Color class and four color table
   // indices that can be used by the interface.  These indices
   // will be translated by the interface into the proper shades
   // of red.  Any object to be displayed in red can then use these
   // same indices.  Use of another color requires another 4 indices.
   Color    red( 1.0, 0.0, 0.0 );
   int      indices[] = {10, 11, 12, 13};

   // open the workstation number 1
   open_graPHIGS( 1 );

   // create the interface manager for workstation 1
   Interface_manager interface_manager( 1 );

   // tell the interface manager not to use certain views
   // these are views that you need to use elsewhere in your code
   interface_manager.make_view_unavailable( 1 );
   interface_manager.make_view_unavailable( 7 );
   interface_manager.make_view_unavailable( 11 );

   // create the windows - window phase 1
   // the first 6 arguments are:
   // x, y, horizontal_alignment, vertical_alignment, width, height
   Simple_view    win1( 0.5, 1.0,  CENTER, TOP,    0.8, 0.7, 100 );
   Simple_view    win2( 0.0, 0.95, LEFT,   TOP,    0.6, 0.5, 100 );
   Simple_view    win3( 1.0, 0.35, RIGHT,  BOTTOM, 0.5, 0.5, 100 );
   Static_menu    main_menu( MENU_STRUCT, 0.0, 0.0, LEFT, BOTTOM, 1.0,
                     0.27, &red, indices );

   // apply attributes to windows - window phase 2
   // setting the window name applies an identifier for the window
   // this name can later be used in identifying the window
```

```
win1.set_name( "Window 1" );
win2.set_name( "Window 2" );
win3.set_name( "Window 3" );
main_menu.set_name( "Main Menu" );

// by default Simple Views are not raised when a user clicks
// on them, so let interface manager raise the following
// windows
win2.turn_raising_on();
win3.turn_raising_on();

// turn shielding on in different colors indices
win1.turn_shielding_on( 0 );
win2.turn_shielding_on( 2 );
win3.turn_shielding_on( 3 );

// turn border on using white (color index 1)
win1.turn_border_on( 1 );
win2.turn_border_on( 1 );
win3.turn_border_on( 1 );

// create the menu items - menu item phase 1
// the first 7 arguments are:
// structure number, x, y, horizontal_alignment, vertical_alignment,
// width, height
Push_button    p1( PB1_STRUCT, 50,  1, CENTER, BOTTOM, 98,  6,
                  "EXIT", &red, indices );
Push_button    p2( PB2_STRUCT,  1, 15,   LEFT, BOTTOM, 48, 11,
                  "Pop Up", &red, indices );
Slider         s1( S_STRUCT,    50,  9, CENTER, BOTTOM, 76,  4,
                  10, 200, &red, indices );

// apply attributes to menu items - menu item phase 2
// in this case only the slider needs additional parameters
// to be specified
s1.add_number( RIGHT, 3.0 );
s1.set_initial_value( 100 );
s1.add_label( LEFT, "Scale" );

// add the menu items to the menu manager - menu item phase 3
main_menu.add_menu_item( &p1 );
main_menu.add_menu_item( &p2 );
main_menu.add_menu_item( &s1 );

// add the windows to the interface manager - window phase 3
interface_manager.add_window( &win1, DO_NOT_PERFORM );
interface_manager.add_window( &win2, DO_NOT_PERFORM );
interface_manager.add_window( &win3, DO_NOT_PERFORM );
interface_manager.add_window( &main_menu );

// create two variables:
```

```
// an event pointer which will be used to extract events from the
// the event queue and a flag to indicate when to exit
Event    *event;
int      exit = NO;

do {

   // this call will wait for up to 50 seconds for a user action
   // to occur.  If some sort of action occurs, the call will
   // build an event queue and return a pointer to the first event
   // event in the queue, otherwise NULL will be returned
   event = interface_manager.process( 50.0 );

   // start a processing loop that processes while the event
   // is not NULL
   while (event != NULL)
      {
      // switch on the id of the event
      switch( event->get_id() )
         {
         // an id of 0 means that the event was generated from
         // the interface manager
         case 0:
            if ( event->get_reason() == NO_WINDOW_SELECTED )
               {
               printf( "No window selected\n" );
               }
            else if ( event->get_reason() == WINDOW_RAISE )
               {
               printf( "Raised window is %s\n",
                  (event->get_window_ptr())->get_name() );
               }
            break;
         case PB1_STRUCT:
            exit = YES;
            break;
         case PB2_STRUCT:
            do_pop_up( &interface_manager );
            break;
         case S_STRUCT:
            printf( "slider = %f\n", event->get_float_value() );
            break;
         }

      // once an event has been processed, get the next event
      // from the event queue
      event = event->extract_next_event();
      }

   // continue processing until the exit flag changes
   } while ( exit == NO );
   close_graPHIGS();

   }

//------------------------------------------------------------------------
// this routine pops up a menu that displays a button to exit the menu
void do_pop_up( Interface_manager *interface_manager )
   {
   Color            blue( 0.0, 0.0, 1.0 );
   int              indices[] = {20, 21, 22, 23};

   // create a menu - window phase 1
   Pop_up_menu      menu( POP_UP_STRUCT, POP_UP_UTIL_STRUCT, 0.5, 0.5,
                       CENTER, CENTER, 0.4, 0.4, "Pop Up Menu", 2, &blue,
                       indices );

   // create a push button - menu item phase 1
   Push_button      ok_button( OK_STRUCT, 2, 2, LEFT, BOTTOM, 36, 36,
                       "OK", &blue, indices );

   // add the menu item to the menu - menu item phase 3
   menu.add_menu_item( &ok_button );

   // add the window to the interface manager - window phase 3
   interface_manager->add_window( &menu );

   Event            *event;
   int              exit = NO;

   do {

      event = interface_manager->process( 100 );

      while ( event != NULL )
         {

         switch( event->get_id() )
            {
            case OK_STRUCT:
               exit = YES;
               menu.ok_close( event );
               break;
            }

         event = event->extract_next_event();
         }

      } while ( exit == NO );

   }
```

Results

A complete implementation of the classes described in the "Class Overview" sections has been accomplished. This includes the following classes:

- Interface manager class
- Window base class
- Menu item base class
- Menu Item manager class
- Menu manager class

Together these classes represent the interface framework. This framework controls both windows and menu items. As sample implementations of windows and menu items, the following classes have also been created:

- **Menu items**: Check boxes, Frames, Labels, Numbers, Push buttons, Radio buttons, Separators, and Sliders.
- **Menu item managers**: Radio button manager.
- **Windows**: Geometry managers, Pop-up menus, Simple views, and Static menus.

The following pages show several photographs taken of a demo application created using the interface framework. The application displays three geometry managers, each containing a different geometric model. The application also displays a static menu. The static menu contains three sliders that control the scaling and translation of the objects in the top geometry manager. The menu also contains push buttons which allow the object in the geometry manager to be rotated. Another button opens a pop-up menu that allows the user to change the color of the object.

The pop-up menu shows a preview of the object as well as sliders that control the red, green, and blue components of the color. Although they don't perform any action, the menu contains three radio buttons which allow the user to select the color model used to display the object. After changing the color, the user can either accept the change or cancel it.

Figure 12 shows the program as it is when it first starts. Figure 13 shows the display after a background geometry manager was raised by clicking on it. Figure 14 shows the geometry manager after it has been resized by dragging one of the corners. Figure 15 again shows the geometry manager after it has been expanded to full view. Figure 16 shows the pop-up color selector menu. Figure 17 shows the pop-up menu being moved on screen. Lastly, Figure 18 shows the display after the color change has been accepted.

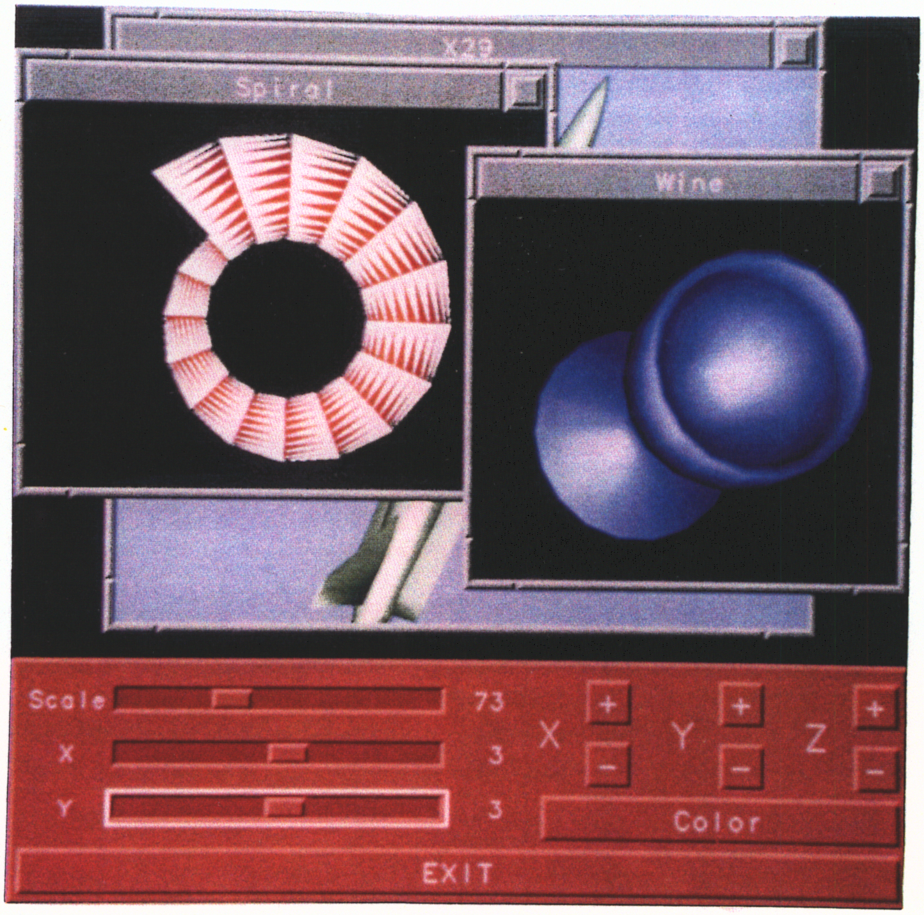


Figure 12 - Demo Program Photograph 1

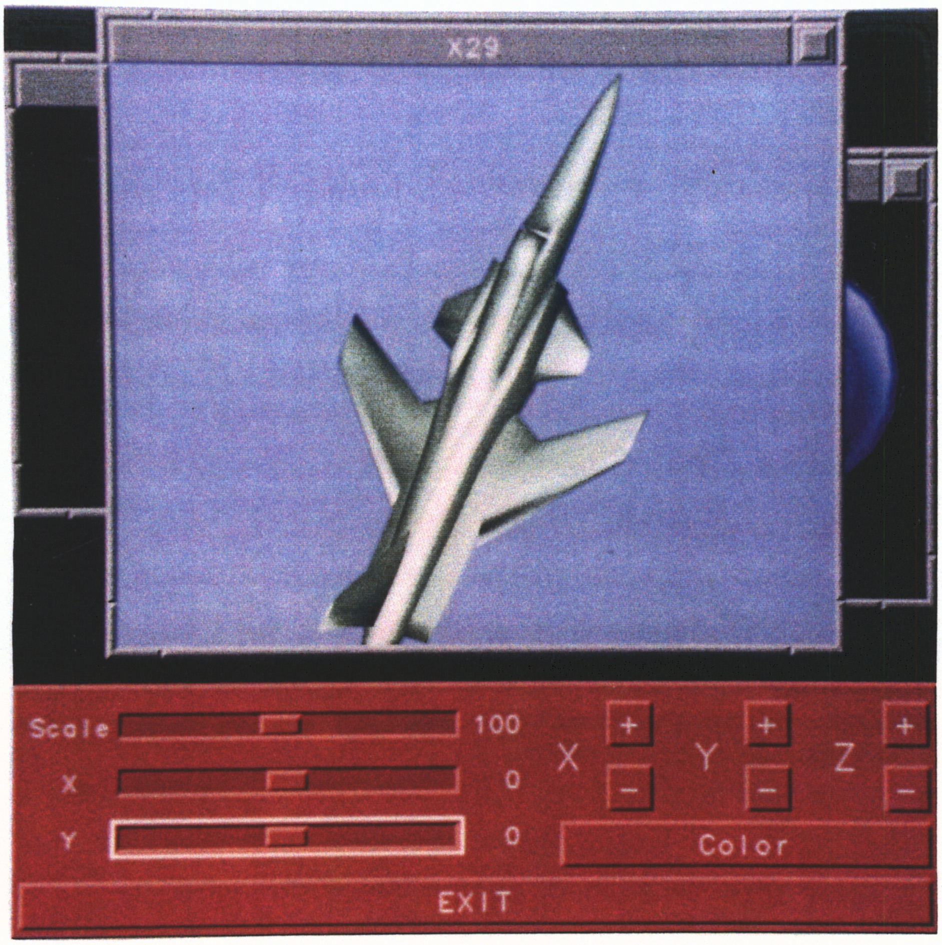


Figure 13 - Demo Program Photograph 2

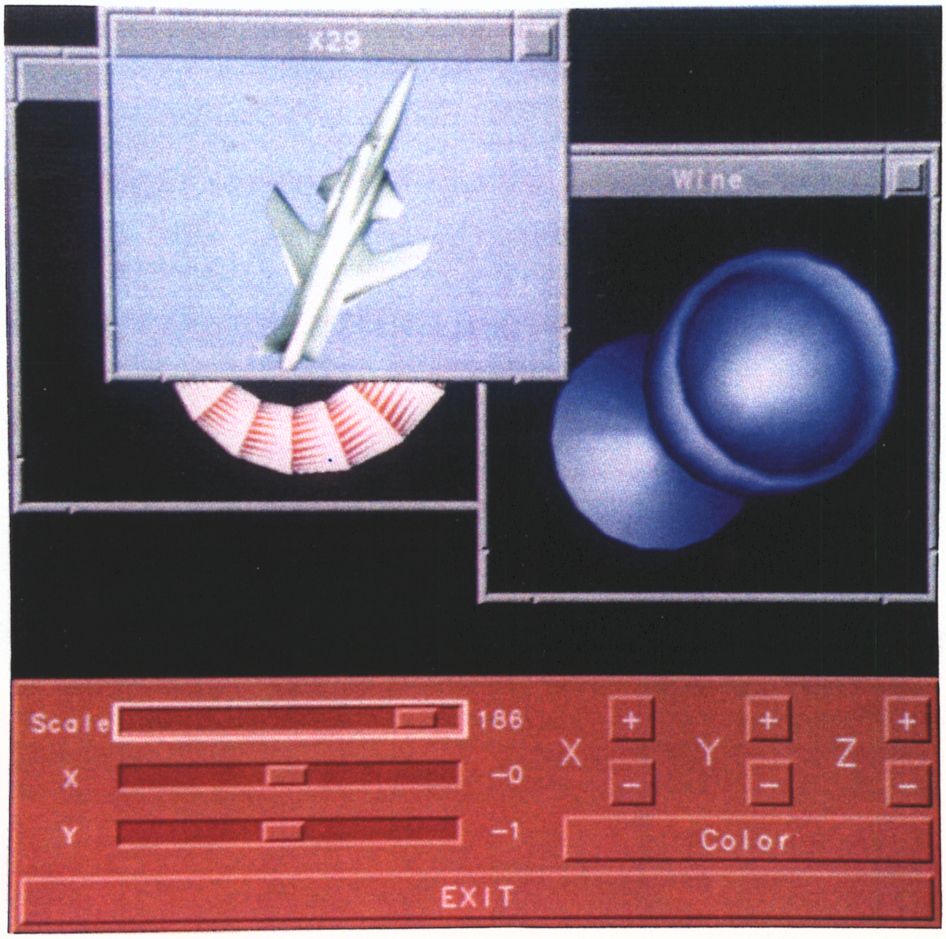


Figure 14 - Demo Program Photograph 3

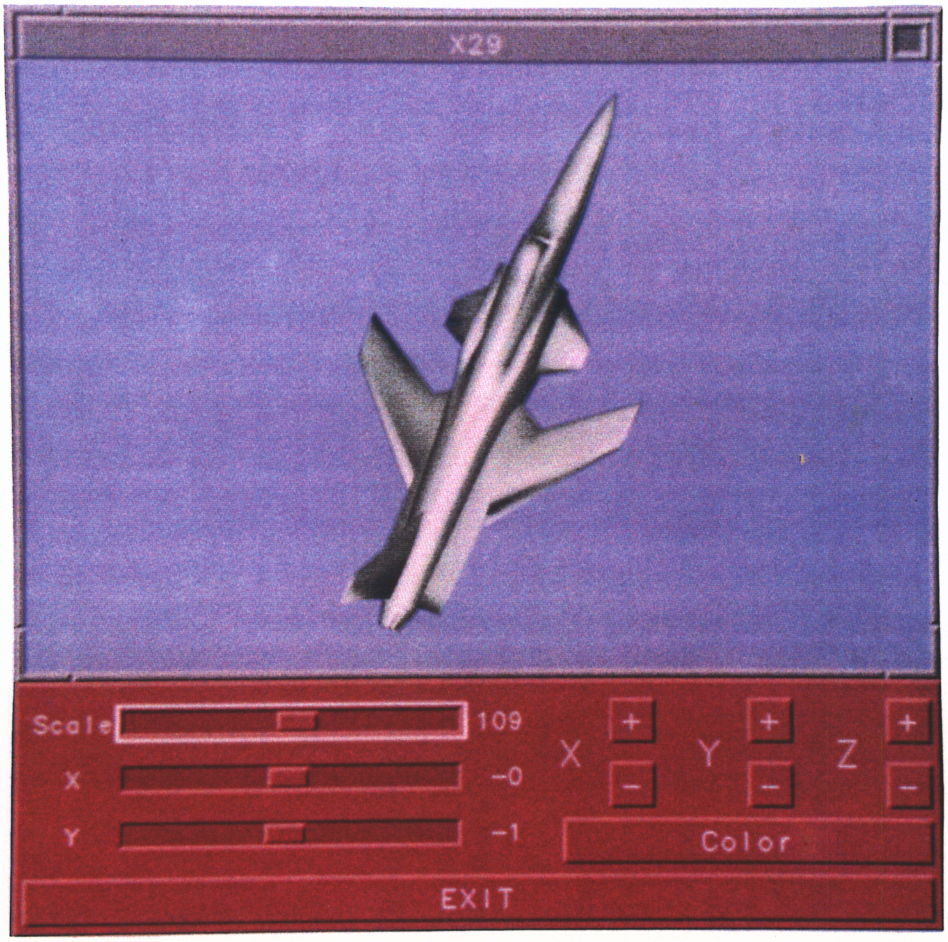


Figure 15 - Demo Program Photograph 4

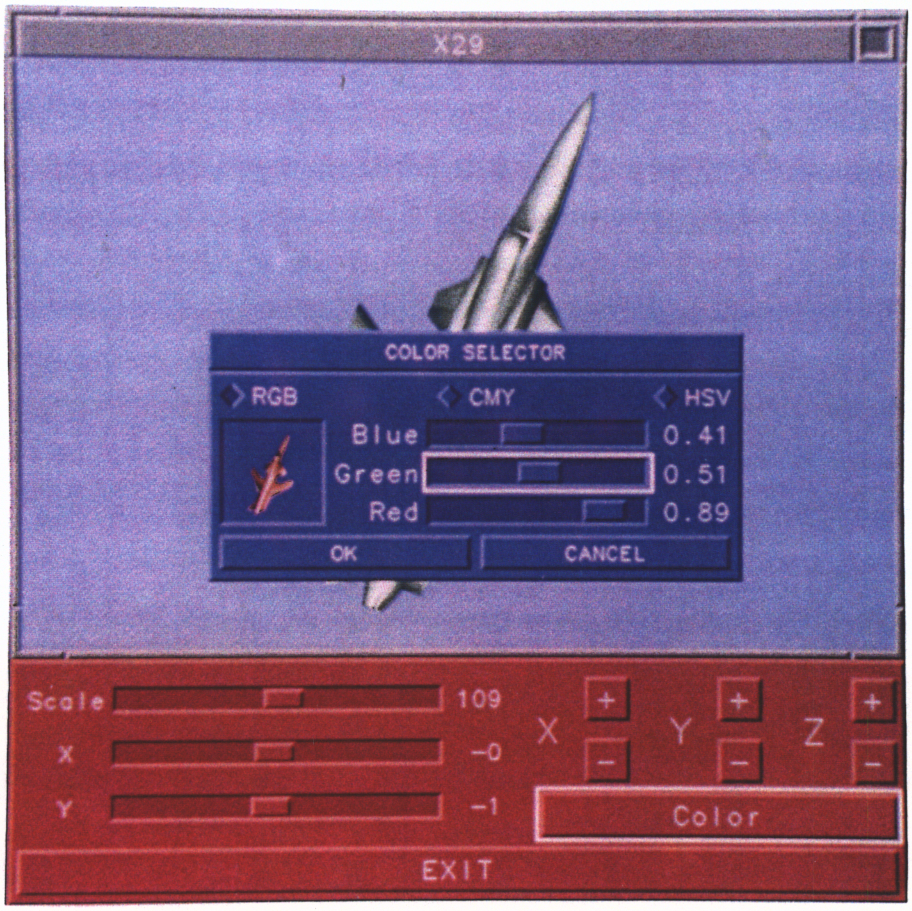


Figure 16 - Demo Program Photograph 5

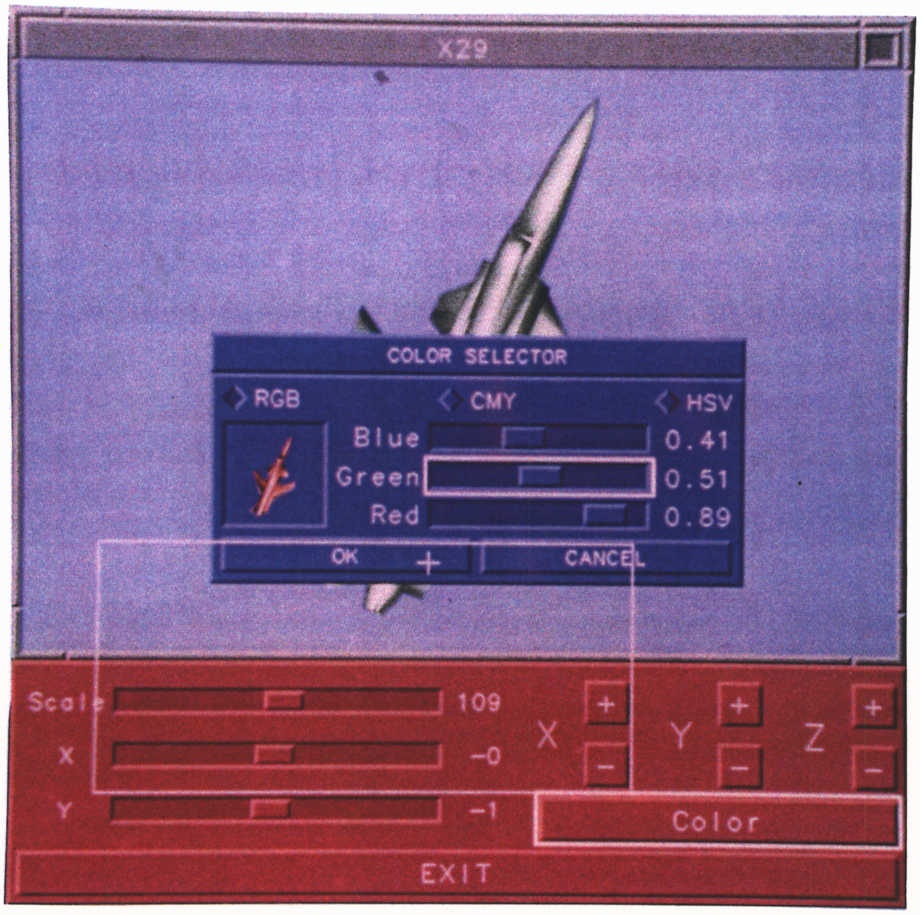


Figure 17 - Demo Program Photograph 6

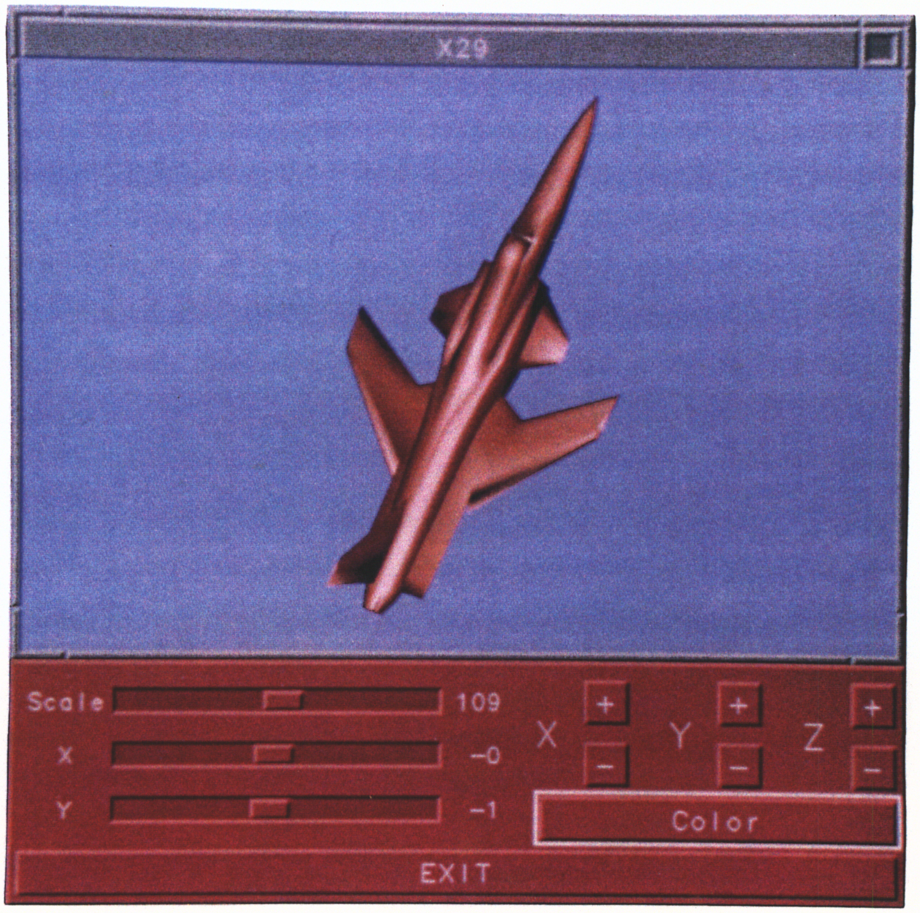


Figure 18 - Demo Program Photograph 7

Conclusions

The goal of this thesis was to create a framework that would allow engineers to easily use advanced interface tools. This meant creating a GUI in a 3D environment focusing on the following features:

- Emphasis on engineers needs
- Compatibility with PHIGS
- Ease of use
- Ease of implementation
- Object-oriented
- Extensible

These goals have been accomplished. The code is currently being used by several other graduate students in the CAD lab. In the future it is hoped that these students, as well as other programmers, will expand on the current set of menu item and window classes in addition to using the classes currently available.

In addition to positive response from students, members of the ACSYNT Institute have also expressed the desire to see some of the features implemented in the ACSYNT code. There is in general a strong desire to have menus that displayed a preview of components in the menu.

The interface framework has proven to be a valuable tool. Operation is surprisingly fast and adds greatly to the appearance of a PHIGS interface. Coding with the framework is straightforward and easy to do.

While a wide range of windows and menu items have been developed for this thesis using the framework, there are still several gaps. A dialogue manager window that displays messages to the user is an example. Menu items that allow string input is another.

During the final phase of coding for this thesis, a nonstandard PHIGS feature was experimented with - immediate mode rendering. Immediate mode allows graphics to be drawn on screen without updating the workstation. Using immediate mode, a push button can be raised and lowered without affecting anything else on the screen. This shows extremely fast operation.

Immediate mode was found to be a very powerful enhancement. It allowed the interface to react as fast as the user could press the mouse button. This made the interface very responsive and pleasing to use. Although immediate mode is not a standard PHIGS feature, it is under consideration by the ISO committee.

Further investigation into the benefits of using immediate mode is recommended. For the time being (while immediate mode is not a standard), immediate mode can be implemented as a feature that can be turned on for machines that support it. The current implementation controls immediate mode at the menu item level - it can be turned on or off for any individual menu item.

Attention should also be paid to the development of PEX. While the current interface uses only PHIGS call and will therefore run under PEX, advanced PEX features should also be integrated. Integration of immediate mode and PEX into the framework will assure the continued existence of the interface code for a long time.

The interface code developed has proven to be an effective tool. Users have responded positively to it. Engineers programming in a PHIGS environment can now implement interfaces quickly and easily. The efforts of this thesis will help to increase the quality of work that the CAD lab produces.

References

[Bass89] Bass, L., Hardy, E., Little, R., and Seacord, R., "Incremental Development off User Interfaces", *Proceedings of the IFIP TC 2/WG 2.7 Working Conference on Engineering for Human-Computer Interaction*, August, 1989, pp. 155-173.

[Booc91] Booch, G., Object-Oriented Design with Applications, The Benjamin/Cummings Publishing Company, Inc., 1991.

[Cann91] Cannon, R., and Tuel, W., "A User Interface for the NCAR Community Climate Model", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 53-58.

[Corn91] Corn, C., Morimizu, F., and Muha, J., "PHIGS Vs Motif: Which is Best for Designing a User Interface?", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 29-38.

[Davi91] Davies, K., "Developing 'Motif-Like' Menu Facilities for graPHIGS Applications", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 47-52.

[Elli90] Ellis, M. A., and Stroustrup, B., The Annotated C++ Reference Manual, *Addison-Wesley Publishing Company*, Reading, Massachusetts, 1990.

[Fisc89] Fischer, Gerhard., "Human-Computer Interaction Software; Lessons Learned, Challenges Ahead", *IEEE Software*, January, 1989, pp. 44-52.

[Flem91] Fleming, S., "Utilizing the graPHIGS API for CAD Applications", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 3-9.

[Guib91] Guibault, F., Magnan, R., and Camarero, R., "Chloé: A CAE Application Development Environment", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 39-45.

[Hunt91] Hunt, D., "IBM AIX AstroSIGHT/6000 Demonstration System", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 21-27.

[Jaya89] Jayaram, S., "CADMADE - An Approach Towards a Device-Independent Standard for CAD/CAM Software Development", Dissertation - Doctor of Philosophy in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1989.

[Jaya90] Jayaram, S., and Myklebust, A. "Towards A Standardized Environment for the Creation of Design and Manufacturing Software", Proceedings of the International Conference on Engineering Design (ICED), Dubrovnik, Yugoslavia, August, 1990.

[Jaya92a] Jayaram, S., Myklebust, A., and Gelhausen, P., "ACSYNT - A Standards-Based System for Parametric Computer Aided Conceptual Design of Aircraft", *Proceedings of the 1992 AIAA Aerospace Design Conference*, Irvine, California, February, 1992.

[Jaya92b] Jayaram, and S., Myklebust, A., "Device-Independent Programming Environments for CAD/CAM Software Creation", accepted for publication, Computer-Aided Design, 1992.

[John91] Johnson, E., and Reichard, K., Power Programming ... Motif, Management Information Source, Inc., 1991.

[Koba91] Kobara, S., Visual Design with OSF/Motif, *Addison-Wesley Publishing Company*, Reading, Massachusetts, 1991.

[Lee90] Lee, E., "User-Interface Development Tools", *IEEE Software*, Vol. 7, No. 3, May 1990, pp. 31-36.

[Mans87a] Mansey, P., Pennington, S., and Myklebust, A., "A Device-Independent Graphic System for Mechanism Synthesis", *Proceedings of the 10th Applied Mechanisms Conference*, New Orleans, December, 1987.

[Mans87b] Mansey, P., "A PHIGS Based Graphics Interface for MECSYN", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1987.

[Myer88] Myers, B., "A Taxonomy of Window Manager User Interfaces", *IEEE Software*, September 1988, pp. 65-84.

[Myer89] Myers, B., "User-Interface Tools: Introduction and Survey", *IEEE Software*, January 1989, pp. 15-23.

[Nye90] Nye, A., X Protocol Reference Manual, Vol. 0, O'Reilly and Associates, Inc., Sebastopol, California, 1990.

[Penn91] Pennington, S. L., "A Software Engineering Approach to the Integration of CAD/CAM Systems", Dissertation - Doctor of Philosophy in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1991.

[Pöps90] Pöpsel, J., and Hornung, C., "Highlight Shading: Lighting and Shading in a PHIGS+/PEX-Environment", *Computers & Graphics*, Vol. 14, No. 1, 1990, pp. 55-64.

[Quer90] Quercia, V., and O'Reilly, T., X Window System User's Guide, Vol. 3, O'Reilly and Associates, Inc., Sebastopol, California, 1990.

[Rubi91] Rubin, M., "VIEW -- A User-Tailorable Graphics Interface for Circuit Design Graphics", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 59-67.

[Sche86] Scheifler, R., Gettys, J., "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2, April 1986, pp. 79-109.

[Schu91] Schultz, K., "PEX - The Road to a Multi-Vendor 3D World", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 157-162.

[Stro91] Stroustrup, B., The C++ Programming Language, *Addison-Wesley Publishing Company*, Reading, Massachusetts, 1991.

[Sung90] Sung, H., Rogers, G., and Kubitz, W., "A Critical Evaluation of PEX", *IEEE Computer Graphics and Applications*, November, 1990, pp. 65-75.

[Tayl91] Taylor, A., "Adding Immediate Mode Graphics to the PHIGS Interface", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 135-143.

[That87] Thatch, B., "A PHIGS Based Interactive Graphical Preprocessor for Spatial Mechanism Analysis and Synthesis", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1987.

[That88] Thatch, B., and Myklebust, A., "A PHIGS-Based Graphics Input Interface for Spatial-Mechanism Design", *IEEE Computer Graphics and Applications*, March, 1988, pp. 26-37.

[Wamp88a] Wampler, S., "Development of a CAD System for Automated Conceptual Design of Supersonic Aircraft", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1988.

[Wamp88b] Wampler, S., Myklebust, A., Jayaram, S., and Gelhausen, P., "Improving Aircraft Conceptual Design - A PHIGS Interactive Graphics Interface for ACSYNT", *AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting*, Atlanta, Georgia, September 7-9, 1988.

[Wamp91a] Wampler, S., "Development of a graPHIGS Based Object-Oriented Graphics System", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 145-155.

[Wils91] Wilson, M., and Conway, A., "Enhanced Interaction Styles for User Interfaces", *IEEE Computer Graphics and Applications,* March 1991, pp. 79-90.

[Youn90] Young, D., The X Window system, Programming and Applications with Xt, OSF/Motif Edition, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1990.

Appendix A - Event Description

Description

Events are packets of information that report an occurrence of some action. In general they come from menu items, but they can originate from other sources such as windows or the interface manager.

Events are each a member of a linked list. If multiple events are generated before they are processed, then this list grows. Eventually the events are extracted from the queue and processed one by one by the application program.

Reserved Values

If an event id is equal to zero, then the event is a message from the interface manager. The type field of events generated by the interface manager is ignored. The possible reasons that events are generated from the interface manager are:

> **WINDOW_RAISE** - this event is generated each time a new window is raised. To determine the window that was raised, get the window pointer with the get_window_ptr() call.
>
> **NO_WINDOW_SELECTED** - this event is generated each time the user presses a mouse while the cursor is not over a window.

Other than an event id of zero, no other event values are reserved.

Member Functions

Constructor

```
Event()
Event( id, type, reason )
```

id - *integer.* This is the event id.
type - *integer.* This value indicates the type of object that generated the event.
reason - *integer.* This is the value that indicates what caused the event to trigger.

Information Retreiving Functions

The following calls can be used to identify where an event came from and why it was generated.

```
int     get_id()
int     get_type()
int     get_reason()
```

Once an event has been identified, the following calls can be used to get more information from the event. If one of these calls is inappropriately called, then an error message will be generated.

```
char        *get_string_ptr()
char        get_char_value()
float       get_float_value()
int         get_int_value()
```

```
Window          *get_window_ptr()
```

Information Setting Functions

If the event identification values are not set in the constructor or need to be modified after the constructor call is issued, the following calls can be used.

```
void            set_id( int _id )
void            set_type( int _type )
void            set_reason( int _reason )
```

To set specific data values within the an event, use the following calls.

```
void            set_char_ptr( char *_ptr )
void            set_float_ptr( float *_ptr )
void            set_int_ptr( int *_ptr )
void            set_string_ptr( char *_ptr )
void            set_window_ptr( Window *_window )
```

Processing Functions

During event processing, the extract_next_event() is used to remove the current event from the linked list and return a pointer to the next event in the list. Events are extracted in the order that they occurred.

```
Event   *extract_next_event();
```

To add an event to the linked list, the following call must be issued from one of the events that is already in the list.

void   add_event_to_list(Event *event)

event - *pointer to type Event*. This is the event to add to the linked list.

Sample Usage

```
#include "event.h"
#include "interface.h"
#include "interface_manager.h"

Event         *event;
int           exit = NO;

do {
   // process as normal
   event = interface_manager.process( 100 );

   while ( event != NULL )
      {

      switch ( event.get_id() )
         {
         case 0:
            if ( event.get_reason() == WINDOW_RAISE )
               printf( "A window has been raised\n" );
            else if ( event.get_reason() == NO_WINDOW_SELECTED )
               printf( "Please select a window\n" );
            break;
         case THE_EXIT_BUTTON:
            exit = YES;
            break;
         case ANYTHING_ELSE:
            // do whatever
            break;
         }

      }

   event->extract_next_event();
   } while ( exit == NO );
```

Appendix B - Interface Manager Description

Description

The interface manager is the central controller for the framework. The interface manager performs the following functions:

1. Controls the distribution of view indices
2. Raises and lowers windows
3. Channels keyboard and mouse input to windows

The interface manager performs these functions by maintaining a pointer to a linked list of windows. A pointer to the active window (the window with the highest priority) is also maintained.

The interface manager uses three input devices:

1. **Mouse Buttons** - the first mouse button is set to trigger a PHIGS choice event each time it is pressed and released.
2. **Keyboard** - the keyboard generates a PHIGS choice event each time a key is pressed.
3. **Locator** - the mouse is put into PHIGS sample mode.

These three devices are initialized each time the process() call is entered. The user is free to change their event modes at anytime, but it should be noted that these are the modes that the devices are in when the process() call is exited.

For more information on how the interface manager works, refer to the section "Class Overview".

Member Functions

Constructors

```
Interface_manager( wsid )
```

wsid - *integer.* This is the PHIGS identifier for the open workstation that the interface manager is to be associated with. This identifier will be passed on to the windows when they are added to the interface manager.

Miscellaneous Functions

The interface manager is responsible for the distribution of PHIGS view indices. If a program needs to use a view, then it must either request an index to use or tell the interface manager the index that it will be using. To avoid conflict in view usage, it is recommended that the request_next_view_index() call be used whenever a view index is needed. When a program is finished with the view, the interface manager can be told that the index is again available with the make_view_available() call.

```
void            make_view_available( view_index );
void            make_view_unavailable( view_index );
```

```
int                    request_next_view_index();
```

view_index - *integer.* PHIGS view index.

Only windows that are under the control of the interface manager are displayed on screen. Once under the control of the interface manager, the window's priority will be dynamically controlled. User input will also be appropriately redirected to the window. To remove a window from the display, the window must be removed from the control of the interface manager. This is usually accomplished with the remove_window() call. Some windows, though, require specialized functions to close and will provide their own closing functions. An example of this is a pop-up menu which uses the cancel_close() and ok_close() calls.

```
void    add_window( window, <perform_flag> )
void    remove_window( window, <perform_flag> )
```

window - *pointer to type Window.* This is the pointer to the window to be manipulated.
perform_flag - *integer.* This is an optional argument that is either PERFORM or DO_NOT_PERFORM. The value indicates if the workstation should be updated to display the changes. If the argument is not specified, then the workstation will update by default.

Windows can be moved in front of or behind all the other windows with the following two functions:

```
void    raise_window( window, <perform_flag> )
void    lower_window( window, <perform_flag> )
```

window - *pointer to type Window.* This is the pointer to the window to be manipulated.
perform_flag - *integer.* This is an optional argument that is either PERFORM or DO_NOT_PERFORM. The value indicates if the workstation should be updated to display the changes. If the argument is not specified, then the workstation will update by default.

To freeze the current window priority order, turn the raise operation flag off. This will disable the interface manager feature of window raising. It will also force all input to be directed to the current active window. If the mouse is clicked outside this window, then the input will be ignored. The raise operation flag is usually set to on and is only modified for specialized purposes such as forcing input to go to a pop-up menu.

```
void    turn_raise_operation_flag_on()
void    turn_raise_operation_flag_off()
```

Once all the necessary windows have been added to the interface manager, the system can be told to go with the process() call.

```
Event   *process( time_out )
```

time_out - *float*. This is the length of time in seconds that this call should wait for user input before returning. If no interaction takes place, then the call will just return.

Return value: The process() call returns a pointer to the Event class. If the call times out or no events are generated, then the pointer will be NULL, otherwise it will point to the first member in the linked list of events. For information on how to process events, see the section "Event Descriptions".

To get a pointer to the window that is currently on top of all the other windows, call the following function:

```
Window          *get_active_window()
```

To get a specific window pointer from the interface manager, the following call can be used:

```
Window          *get_window( name )
```

name - *pointer of type char*. This is a pointer to the string used to identify the window.

The following function provides an easier way of updating the workstation. It performs identically to a PHIGS update workstation call.

```
void update_workstation()
```

If it is necessary to get the workstation id for a PHIGS function, then it can be retrieved using the following call. It is recommended that this call only be used from within the definition of a class. If this call needs to be made from outside a class, then some class was most likely not properly implemented.

```
int      get_wsid()
```

Sample Usage

```
#include "interface.h"
#include "interface_manager.h"

void main()

{

// first open PHIGS
...
...
...

// create an interface manager
Interface_manager iman(1);

// create some windows
...
...
...

// add the windows to the interface manager
iman.add_window( &win1 );
iman.add_window( &win2 );
...
...
...

// set one of the windows to be on top
iman.raise_window( &win2 );

Event         *event;
int           exit = NO;

do {
   // process
   event = iman.process( 100 );

   while ( event != NULL )
      {

      switch ( event.get_id() )
         {
         case THE_EXIT_BUTTON:
            exit = YES;
            break;
         case ANYTHING_ELSE:
            // do whatever
            break;
         }

      }

   event->extract_next_event();
   } while ( exit == NO );

// close PHIGS
close_PHIGS();

}
```

Appendix C - Menu Item Descriptions

The following section presents a summary of the features of the menu items created. The section describes the following features for each menu item:

- Description
- Appearance
- States
- Events
- Managers
- Constructors
- Member Functions
- Sample Usage

The menu items implemented thus far are:

- Check Box
- Frame
- Label
- Number
- Push Button
- Radio Button
- Separator
- Slider

Check Box

Description

A check box is used to toggle an option on or off.

Appearance

Figure 19 shows the dimensions of a check box. The dimensions controlled by the programmer are the size and shadow thickness. By default, the remaining dimensions are set using the following relations:

label_margin_thickness = 0.25*size

highlight_thickness = 0.3

highlight_margin_thickness = 0.5

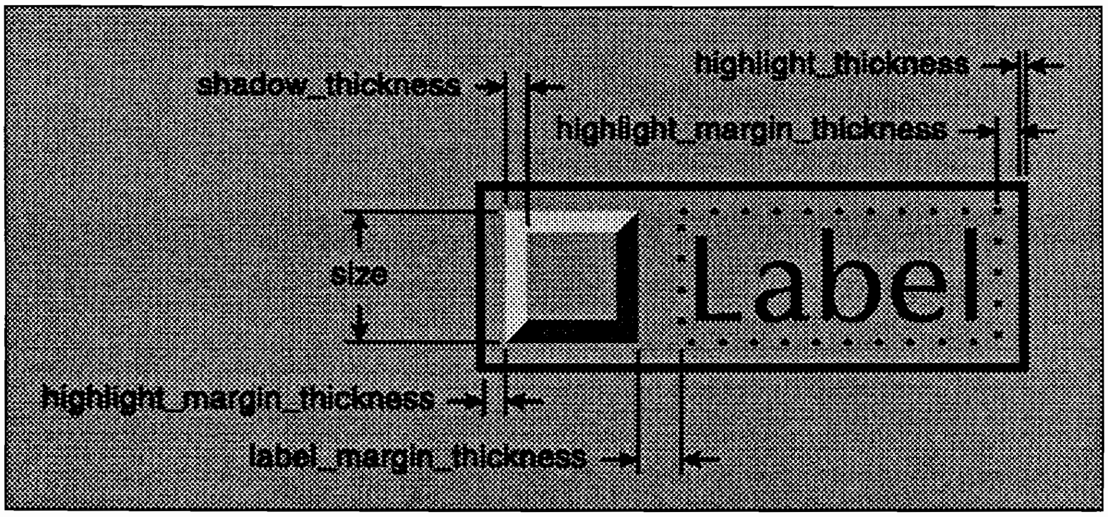


Figure 19 - Check Box Dimensions

States

A check box is always in one of the following three states:

ACTIVE - button is raised and waiting for input.

SELECTED - button is lowered and waiting for input.

INACTIVE - button is raised, and cannot accept input.

Figure 20 shows what each of the check box states looks like.

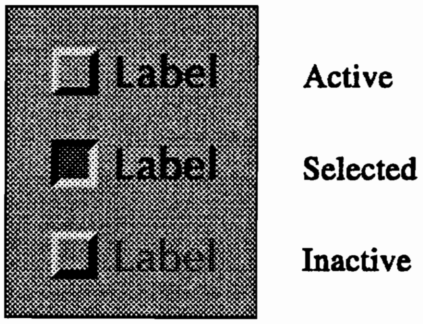


Figure 20 - Check Box States

Events

type: CHECK_BOX

reasons: TOGGLE_DOWN or TOGGLE_UP

times: DELAYED or IMMEDIATE

data: None

Manager

None.

Constructors

Standard Arguments:

STD_ARGS = structure_number, x, y, horizontal_alignment, vertical_alignment, size

Function Call Syntax:

```
Check_box( STD_ARGS, string )
Check_box( STD_ARGS, string, color )
Check_box( STD_ARGS, string, color, indices )
Check_box( STD_ARGS, font, string )
Check_box( STD_ARGS, font, string, color )
Check_box( STD_ARGS, font, string, color, indices )
Check_box( STD_ARGS, thickness, string )
Check_box( STD_ARGS, thickness, string, color )
Check_box( STD_ARGS, thickness, string, color, indices )
Check_box( STD_ARGS, thickness, font, string )
Check_box( STD_ARGS, thickness, font, string, color )
Check_box( STD_ARGS, thickness, font, string, color, indices )
Check_box( STD_ARGS, display_structure, max_width, max_height )
Check_box( STD_ARGS, display_structure, max_width, max_height, color )
Check_box( STD_ARGS, display_structure, max_width, max_height, color, indices )
Check_box( STD_ARGS, display_structure, max_width, max_height, scale )
Check_box( STD_ARGS, display_structure, max_width, max_height, scale, color )
Check_box( STD_ARGS, display_structure, max_width, max_height, scale, color,
  indices )
Check_box( STD_ARGS, thickness, display_structure, max_width, max_height )
Check_box( STD_ARGS, thickness, display_structure, max_width, max_height, color )
Check_box( STD_ARGS, thickness, display_structure, max_width, max_height, color,
  indices )
Check_box( STD_ARGS, thickness, display_structure, max_width, max_height, scale )
Check_box( STD_ARGS, thickness, display_structure, max_width, max_height, scale,
  color )
Check_box( STD_ARGS, thickness, display_structure, max_width, max_height, scale,
  color, indices )
```

Argument Descriptions:

color - *pointer of type Color.* This is the base color to be used for the menu item. The default is gray.
display_structure - *integer.* This is the PHIGS structure identifier for the structure to be displayed with the menu item (instead of a text string). The structure is assumed to be centered at (0,0).
font - *integer.* This is the PHIGS font identifier to be used for menu item. The default font identifier is 9.
horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the menu item with respect to the x value given.
indices - *array of 4 integers.* These are color table entries that can be used and modified by the menu item. If the indices are used in the constructor, then menu item will use the color table, otherwise the color are sent directly to the display device.
max_width, max_height - *floats.* These values describe the maximum dimensions of the display structure.
scale - *float.* This is a factor by which the display structure can be scaled before displaying. If the scale argument is not supplied, then the display structure will be scaled to fit with the menu item based on the max_width and max_height arguments.
size - *float.* This is the size of the check box button.
string - *pointer of type char.* This is the string to be displayed with the menu item.
structure_number - *integer.* This is the structure identifier to be used to create the menu item.
thickness - *float.* This is the menu item shadow thickness. The default is 0.4.
vertical_alignment - *integer.* This value is either TOP, CENTER, or BOTTOM and describes the alignment of the menu item with respect to the y value given.
x, y - *floats.* These are the coordinates that describe the location of the menu item.

Member Functions

State Control Functions:

```
void        select( event );
void        change_state_to_active( <perform_flag> );
void        change_state_to_selected( <perform_flag> );
void        change_state_to_inactive( <perform_flag> );
int         get_state()
```

event - *pointer to type Event.* This is one of the events in the linked list of events. The event generated by selecting this menu item will be appended to the linked list.
perform_flag - *integer.* This is an optional argument that is either PERFORM or DO_NOT_PERFORM. The value indicates if the workstation should be updated to display the change in the menu item state. If the argument is not specified, then the workstation will update by default.

Event Control Functions:

void	set_event_time(event_time)
int	get_event_time()
Event	*get_toggle_up_event()
Event	*get_toggle_down_event()
void	set_toggle_up_id(id)
void	set_toggle_down_id(id)

event_time - *integer.* This value is either DELAYED or IMMEDIATE and sets the event time for the menu item. The default is IMMEDIATE.
id - *integer.* This is the identification number for the event. If no id is specified, then it will automatically be set to the structure id used for the menu item.

Miscellaneous Functions:

None.

Sample Usage

```
#include "check_box.h"
#include "interface.h"

Check_box   check_box1( 1, 5, 5, CENTER, CENTER, 5, "Press Me" );

check_box1.set_event_time( DELAYED );
check_box1.set_toggle_up_id( 5 );
check_box1.change_state_to_selected( DO_NOT_PERFORM );
```

Frame

Description

A frame is used to visually group a set of menu items.

Appearance

Figure 21 shows the dimensions of a frame. The dimensions controlled by the programmer are the height, width, and shadow thickness.

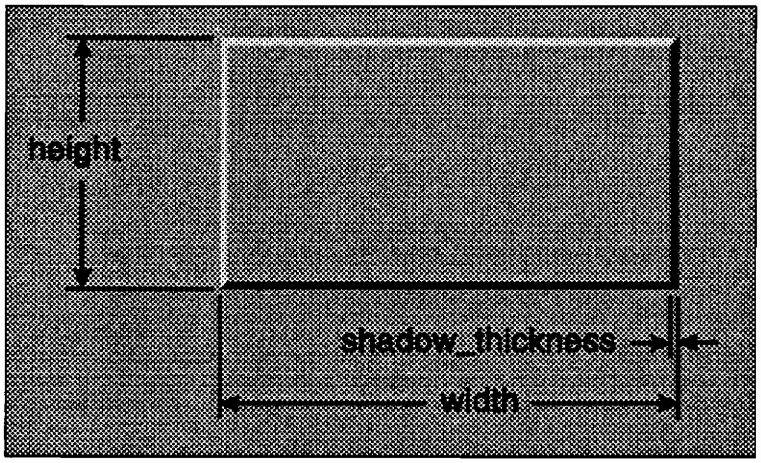


Figure 21 - Frame Dimensions

States

A frame is always in one of the following two states:

RAISED - the frame appears to be raised above the surface.

LOWERED - the frame appears to be lowered below the surface.

Events

None.

Manager

None.

Constructors

Standard Arguments:

```
STD_ARGS = structure_number, x, y, horizontal_alignment, vertical_alignment, width,
  height, state
```

Function Call Syntax:

```
Frame( STD_ARGS )
Frame( STD_ARGS, color )
Frame( STD_ARGS, color, indices )
Frame( STD_ARGS, thickness )
Frame( STD_ARGS, thickness, color )
Frame( STD_ARGS, thickness, color, indices )
```

Argument Descriptions:

color - *pointer of type Color.* This is the base color to be used for the menu item. The default is gray.
horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the menu item with respect to the x value given.
indices - *array of 4 integers.* These are color table entries that can be used and modified by the menu item.
width, height - *floats.* These values specify the size of the menu item.
state - *integer.* This value is either RAISED or LOWERED and specifies the state of the frame.
structure_number - *integer.* This is the structure identifier to be used to create the menu item.
thickness - *float.* This is the menu item shadow thickness. The default is 0.4.
vertical_alignment - *integer.* This value is either TOP, CENTER, or BOTTOM and describes the alignment of the menu item with respect to the y value given.
x, y - *floats.* These are the coordinates that describe the location of the menu item.

Member Functions

State Control Functions:

```
int     get_state()
```

Event Control Functions:

None.

Miscellaneous Functions:

None.

Sample Usage

```
#include "interface.h"
#include "frame.h"

Frame        frame( 1, 5, 5, CENTER, CENTER, 10, 12, LOWERED );
```

Label

Description

A label is used to display either a text string or a PHIGS structure.

Appearance

Figure 22 shows the dimensions of a label. The dimension controlled by the programmer is the height. The character width is dependent on the font chosen for the label.

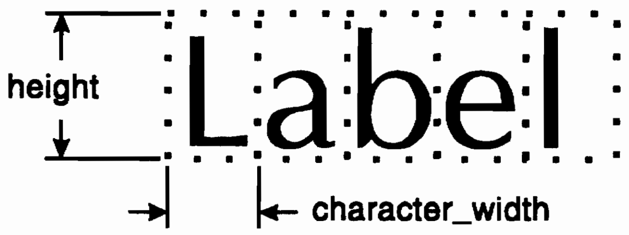


Figure 22 - Label Dimensions

States

None.

Events

None.

Manager

None.

Constructors

Standard Arguments:

STD_ARGS = structure_number, x, y, horizontal_alignment, vertical_alignment

Function Call Syntax:

```
Label( STD_ARGS, character_height, string )
Label( STD_ARGS, character_height, string, color )
Label( STD_ARGS, character_height, string, color, index )
Label( STD_ARGS, character_height, string, index )
Label( STD_ARGS, character_height, string, font, color )
Label( STD_ARGS, character_height, string, font, color, index )
Label( STD_ARGS, character_height, string, font, index )
Label( STD_ARGS, display_structure, max_width, max_height )
Label( STD_ARGS, display_structure, max_width, max_height, scale )
```

Argument Descriptions:

character_height - *float.* This is the height of the letters used in the menu item.
color - *pointer of type Color.* This is the base color to be used for the menu item. The default is gray.
display_structure - *integer.* This is the PHIGS structure identifier for the structure to be displayed with the menu item (instead of a text string). The structure is assumed to be centered at (0,0).

font - *integer*. This is the PHIGS font identifier to be used for menu item. The default font identifier is 9.
horizontal_alignment - *integer*. This value is either RIGHT, CENTER, or LEFT and describes the alignment of the menu item with respect to the x value given.
index - *integer*. If specified with the color argument, then this color table index is modified to match the color argument, otherwise the label uses the color specified by the index.
max_width, max_height - *floats*. These values describe the maximum dimensions of the display structure.
scale - *float*. This is a factor by which the display structure can be scaled before displaying.
string - *pointer of type char*. This is the string to be displayed with the menu item.
structure_number - *integer*. This is the structure identifier to be used to create the menu item.
vertical_alignment - *integer*. This value is either TOP, CENTER, or BOTTOM and describes the alignment of the menu item with respect to the y value given.
x, y - *floats*. These are the coordinates that describe the location of the menu item.

Member Functions

State Control Functions:

None.

Event Control Functions:

None.

Miscellaneous Functions:

float	get_width()
float	get_height()
int	get_format()
void	set_color(color)
void	set_color(color, color_index)
void	set_color(color_index)
void	set_color_from_reference(reference_color)

color - *pointer of type Color*. This is the base color to be used for the menu item.
color_index - *integer*. If specified with the color argument, then this color table index is modified to match the color argument, otherwise the label uses the color specified by the index.
reference_color - *pointer of type Color*. The color of the label is set to either 0 or 1 (black or white) depending on which has a higher contrast with the reference color.

Sample Usage

```
#include "interface.h"
#include "label.h"

Label       l1( 1, 30, 40, RIGHT, TOP, 2, "This is a label" );
```

Number

Description

A number can be used to display a formatted number on screen.

Appearance

Figure 23 shows the dimensions of a number. The dimension controlled by the programmer is height. The character width is dependent on the font chosen for the number.

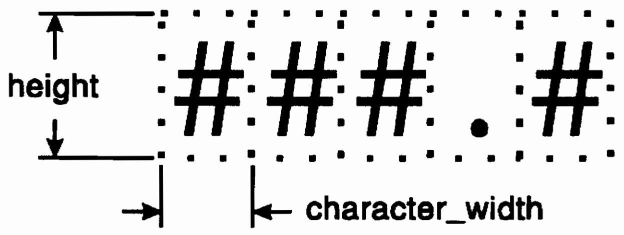


Figure 23 - Number Dimensions

States

None.

Events

None.

Manager

None.

Constructors

Standard Arguments:

STD_ARGS = structure_number, x, y, horizontal_alignment, vertical_alignment, character_height, format, value

Function Call Syntax:

```
Number( STD_ARGS );
Number( STD_ARGS, color );
Number( STD_ARGS, color, index );
Number( STD_ARGS, index );
Number( STD_ARGS, font, color );
Number( STD_ARGS, font, color, index );
Number( STD_ARGS, font, index );
```

Argument Descriptions:

character_height - ***float.*** This is the height of the letters used in the menu item.

color - *pointer of type Color.* This is the base color to be used for the menu item. The default is gray.
font - *integer.* This is the PHIGS font identifier to be used for menu item. The default font identifier is 9.
format - *float.* This value indicates the format specifier to be applied to the number. For example, 3.2 means 3 placed before the decimal and two places after the decimal.
horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the menu item with respect to the x value given.
index - *integer.* If specified with the color argument, then this color table index is modified to match the color argument, otherwise the label uses the color specified by the index.
structure_number - *integer.* This is the structure identifier to be used to create the menu item.
value - *float.* This is the value for the menu item to display.
vertical_alignment - *integer.* This value is either TOP, CENTER, or BOTTOM and describes the alignment of the menu item with respect to the y value given.
x, y - *floats.* These are the coordinates that describe the location of the menu item.

Member Functions

State Control Functions:

None.

Event Control Functions:

None.

Miscellaneous Functions:

float	get_width()
float	get_height()
void	set_color(color)
void	set_color(color, color_index)
void	set_color(color_index)
void	set_color_from_reference(reference_color)
void	set_value(value, <perform_flag>)

color - *pointer of type Color.* This is the base color to be used for the menu item.
color_index - *integer.* If specified with the color argument, then this color table index is modified to match the color argument, otherwise the label uses the color specified by the index.
reference_color - *pointer of type Color.* The color of the label is set to either 0 or 1 (black or white) depending on which has a higher contrast with the reference color.
perform_flag - *integer.* This is an optional argument that is either PERFORM or DO_NOT_PERFORM. The value indicates if the workstation should be updated to display the change in the menu item state. If the argument is not specified, then the workstation will update by default.
value - *float.* This is the value for the menu item to display.

Sample Usage

```
#include "interface.h"
#include "number.h"

Number          n1( 1, 30, 40, RIGHT, TOP, 2, 3.2, 34.598 );
```

Push Button

Description

A push button is used to initiate a sequence of events.

Appearance

Figure 24 shows the dimensions of a push button. The dimensions controlled by the programmer are the height, width and shadow thickness. By default, the remaining dimensions are set using the following relations:

highlight_thickness = 0.3

highlight_margin_thickness = 0.5

label_height = 0.5*(height - 2*shadow_thickness)

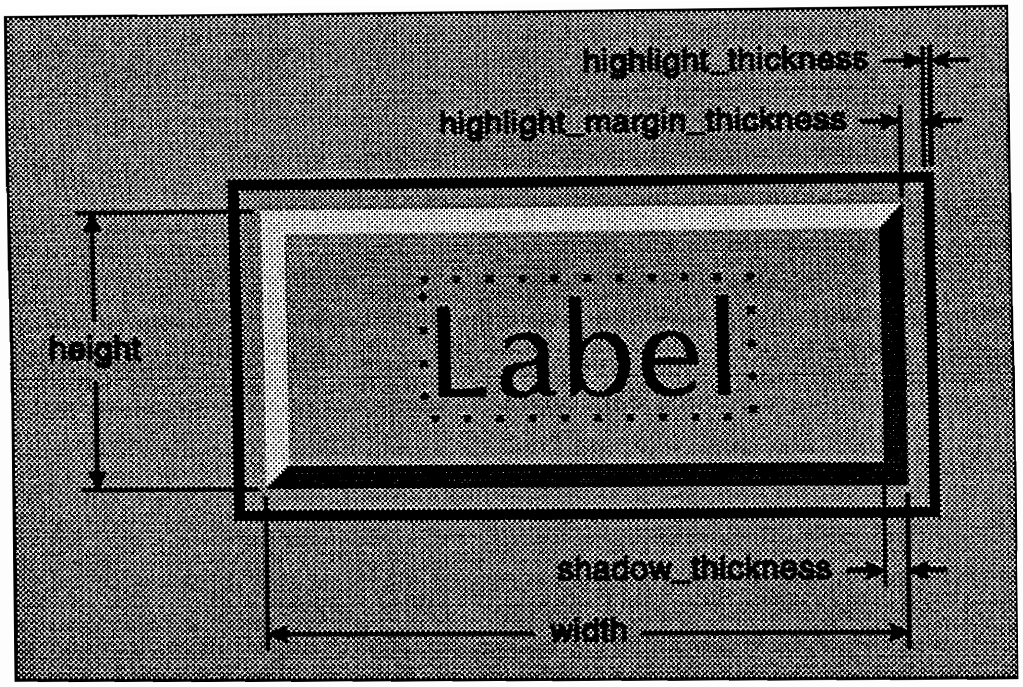


Figure 24 - Push Button Dimensions

States

A push button is always in one of the following three states:

ACTIVE - button is raised and waiting for input

SELECTED - button is lowered

INACTIVE - button is raised, and cannot accept input

Figure 25 shows what each of the push button states looks like.

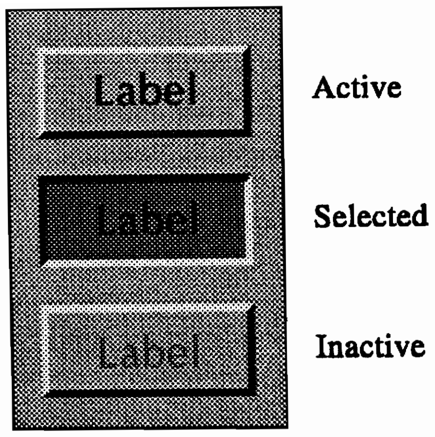


Figure 25 - Push Button States

Events

type: PUSH_BUTTON

reasons: BUTTON_PRESS

times: IMMEDIATE

data: None

Manager

None.

Constructors

Standard Arguments:

STD_ARGS = structure_number, x, y, horizontal_alignment, vertical_alignment, width, height

Function Call Syntax:

```
Push_button( STD_ARGS )
Push_button( STD_ARGS, color )
Push_button( STD_ARGS, color, indices )
Push_button( STD_ARGS, thickness )
Push_button( STD_ARGS, thickness, color )
Push_button( STD_ARGS, thickness, color, indices )
Push_button( STD_ARGS, string )
Push_button( STD_ARGS, string, color )
Push_button( STD_ARGS, string, color, indices )
Push_button( STD_ARGS, font, string )
Push_button( STD_ARGS, font, string, color )
Push_button( STD_ARGS, font, string, color, indices )
Push_button( STD_ARGS, thickness, string )
Push_button( STD_ARGS, thickness, string, color )
Push_button( STD_ARGS, thickness, string, color, indices )
Push_button( STD_ARGS, thickness, font, string )
Push_button( STD_ARGS, thickness, font, string, color )
Push_button( STD_ARGS, thickness, font, string, color, indices )
Push_button( STD_ARGS, display_structure, scale )
Push_button( STD_ARGS, display_structure, max_width, max_height )
Push_button( STD_ARGS, display_structure, scale, color )
Push_button( STD_ARGS, display_structure, max_width, max_height, color )
Push_button( STD_ARGS, display_structure, scale, color, indices )
Push_button( STD_ARGS, display_structure, max_width, max_height, color, indices )
Push_button( STD_ARGS, thickness, display_structure, scale )
Push_button( STD_ARGS, thickness, display_structure, max_width, max_height )
Push_button( STD_ARGS, thickness, display_structure, scale, color )
Push_button( STD_ARGS, thickness, display_structure, max_width, max_height, color )
Push_button( STD_ARGS, thickness, display_structure, scale, color, indices )
Push_button( STD_ARGS, thickness, display_structure, max_width, max_height, color,
  indices )
```

Argument Descriptions:

color - *pointer of type Color.* This is the base color to be used for the menu item. The default is gray.

display_structure - *integer.* This is the PHIGS structure identifier for the structure to be displayed with the menu item (instead of a text string). The structure is assumed to be centered at (0,0).
font - *integer.* This is the PHIGS font identifier to be used for menu item. The default font identifier is 9.
horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the menu item with respect to the x value given.
indices - *array of 4 integers.* These are color table entries that can be used and modified by the menu item. If the indices are used in the constructor, then menu item will use the color table, otherwise the color are sent directly to the display device.
max_width, max_height - *floats.* These values describe the maximum dimensions of the display structure.
scale - *float.* This is a factor by which the display structure can be scaled before displaying. If the scale argument is not supplied, then the display structure will be scaled to fit with the menu item based on the max_width and max_height arguments.
size - *float.* This is the size of the check box button.
string - *pointer of type char.* This is the string to be displayed with the menu item.
structure_number - *integer.* This is the structure identifier to be used to create the menu item.
thickness - *float.* This is the menu item shadow thickness. The default is 0.4.
vertical_alignment - *integer.* This value is either TOP, CENTER, or BOTTOM and describes the alignment of the menu item with respect to the y value given.
width, height - *floats.* These values describe the dimensions of the menu item.
x, y - *floats.* These are the coordinates that describe the location of the menu item.

Member Functions

State Control Functions:

```
void          select( event );
void          change_state_to_active( <perform_flag> );
void          change_state_to_selected( <perform_flag> );
void          change_state_to_inactive( <perform_flag> );
int           get_state()
```

event - *pointer to type Event.* This is one of the events in the linked list of events. The event generated by selecting this menu item will be appended to the linked list.
perform_flag - *integer.* This is an optional argument that is either PERFORM or DO_NOT_PERFORM. The value indicates if the workstation should be updated to display the change in the menu item state. If the argument is not specified, then the workstation will update by default.

Event Control Functions:

```
Event           *get_press_event()
void            set_press_event_id( id )
```

id - *integer*. This is the identification number for the event. If no id is specified, then it will automatically be set to the structure id used for the menu item.

Miscellaneous Functions:

None.

Sample Usage

```
#include "interface.h"
#include "push_button.h"

Push_button push_button( 1, 5, 5, LEFT, TOP, 15, 5, "Press Me" );
```

Radio Button

Description

A radio button is used to select an option from a group of mutually exclusive options. Therefore a radio button always exists in conjunction with at least one other radio button.

Appearance

Figure 26 shows the dimensions of a radio button. The dimensions controlled by the programmer are the size and shadow thickness. By default, the remaining dimensions are set using the following relations:

highlight_thickness = 0.3

highlight_margin_thickness = 0.5

label_height = 0.6*size

label_margin = 0.25*size

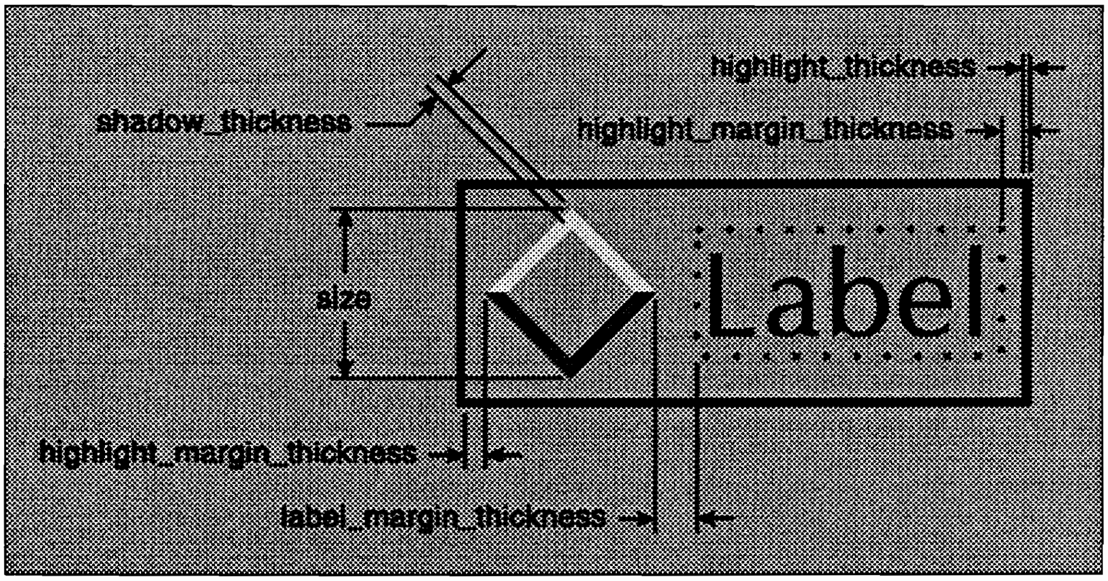


Figure 26 - Radio Button Dimensions

States

A radio button is always in one of the following three states:

ACTIVE - button is raised and waiting for input.

SELECTED - button is lowered and waiting for input.

INACTIVE - button is raised, and cannot accept input.

Figure 27 shows what each of the radio button states looks like.

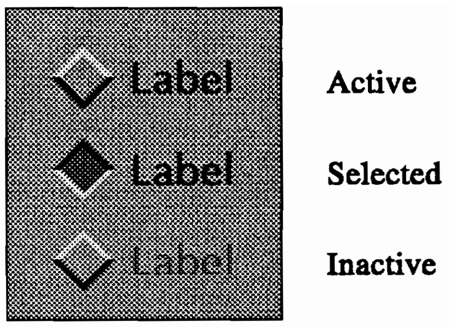


Figure 27 - Radio Button States

Events

type: RADIO_BUTTON

reasons: TOGGLE_DOWN or TOGGLE_UP

times: DELAYED or IMMEDIATE

data: None

Manager

Radio Button Manager.

Constructors

Standard Arguments:

STD_ARGS = structure_number, x, y, horizontal_alignment, vertical_alignment, size

Function Call Syntax:

```
Radio_button( STD_ARGS, string )
Radio_button( STD_ARGS, string, color )
Radio_button( STD_ARGS, string, color, indices )
Radio_button( STD_ARGS, font, string )
Radio_button( STD_ARGS, font, string, color )
Radio_button( STD_ARGS, font, string, color, indices )
Radio_button( STD_ARGS, thickness, string )
Radio_button( STD_ARGS, thickness, string, color )
Radio_button( STD_ARGS, thickness, string, color, indices )
Radio_button( STD_ARGS, thickness, font, string )
Radio_button( STD_ARGS, thickness, font, string, color )
Radio_button( STD_ARGS, thickness, font, string, color, indices )
Radio_button( STD_ARGS, display_structure, max_width, max_height )
Radio_button( STD_ARGS, display_structure, max_width, max_height, color )
Radio_button( STD_ARGS, display_structure, max_width, max_height, color, indices )
Radio_button( STD_ARGS, display_structure, max_width, max_height, scale )
Radio_button( STD_ARGS, display_structure, max_width, max_height, scale, color )
Radio_button( STD_ARGS, display_structure, max_width, max_height, scale, color,
  indices )
Radio_button( STD_ARGS, thickness, display_structure, max_width, max_height )
Radio_button( STD_ARGS, thickness, display_structure, max_width, max_height, color )
Radio_button( STD_ARGS, thickness, display_structure, max_width, max_height, color,
  indices )
Radio_button( STD_ARGS, thickness, display_structure, max_width, max_height, scale )
Radio_button( STD_ARGS, thickness, display_structure, max_width, max_height, scale,
  color )
Radio_button( STD_ARGS, thickness, display_structure, max_width, max_height, scale,
  color,  indices )
```

Argument Descriptions:

color - *pointer of type Color.* This is the base color to be used for the menu item. The default is gray.
display_structure - *integer.* This is the PHIGS structure identifier for the structure to be displayed with the menu item (instead of a text string). The structure is assumed to be centered at (0,0).
font - *integer.* This is the PHIGS font identifier to be used for menu item. The default font identifier is 9.
horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the menu item with respect to the x value given.

indices - *array of 4 integers*. These are color table entries that can be used and modified by the menu item. If the indices are used in the constructor, then menu item will use the color table, otherwise the color are sent directly to the display device.
max_width, max_height - *floats*. These values describe the maximum dimensions of the display structure.
scale - *float*. This is a factor by which the display structure can be scaled before displaying. If the scale argument is not supplied, then the display structure will be scaled to fit with the menu item based on the max_width and max_height arguments.
size - *float*. This is the size of the check box button.
string - *pointer of type char*. This is the string to be displayed with the menu item.
structure_number - *integer*. This is the structure identifier to be used to create the menu item.
thickness - *float*. This is the menu item shadow thickness. The default is 0.4.
vertical_alignment - *integer*. This value is either TOP, CENTER, or BOTTOM and describes the alignment of the menu item with respect to the y value given.
x, y - *floats*. These are the coordinates that describe the location of the menu item.

Member Functions

State Control Functions:

```
void        select( event );
void        change_state_to_active( <perform_flag> );
void        change_state_to_selected( <perform_flag> );
void        change_state_to_inactive( <perform_flag> );
int         get_state()
```

event - *pointer to type Event*. This is one of the events in the linked list of events. The event generated by selecting this menu item will be appended to the linked list.
perform_flag - *integer*. This is an optional argument that is either PERFORM or DO_NOT_PERFORM. The value indicates if the workstation should be updated to display the change in the menu item state. If the argument is not specified, then the workstation will update by default.

Event Control Functions:

```
void        set_event_time( event_time )
int         get_event_time()
Event       *get_toggle_up_event()
Event       *get_toggle_down_event()
void        set_toggle_up_id( id )
void        set_toggle_down_id( id )
```

event_time - *integer*. This value is either DELAYED or IMMEDIATE and sets the event time for the menu item. The default is IMMEDIATE.
id - *integer*. This is the identification number for the event. If no id is specified, then it will automatically be set to the structure id used for the menu item.

Miscellaneous Functions:

None.

Sample Usage

```
#include "radio_button.h"
#include "radio_button_manager.h"
#include "interface.h"

Radio_button      rb1( 1, 5, 5, LEFT, TOP, 5, "Radio Button 1" );
Radio_button      rb2( 2, 5, 10, LEFT, TOP, 5, "Radio Button 2" );

Radio_button_manager rbm;

rbm.add_managed_menu_item( &rb1 );
rbm.add_managed_menu_item( &rb2 );
```

Separator

Description

A separator is used to visually separate a group of menu items. It is usually used to split a menu into multiple sections.

Appearance

Figure 28 shows the dimensions of a separator. The dimensions controlled by the programmer are the length and shadow thickness.

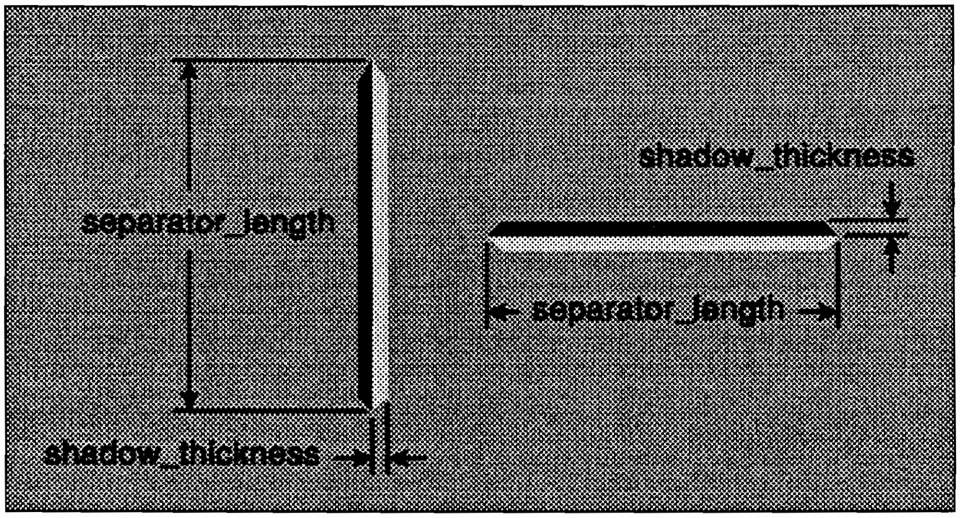


Figure 28 - Separator Dimensions

States

None.

Events

None.

Manager

None.

Constructors

Standard Arguments:

```
STD_ARGS = structure_number, x, y, format, alignment, length
```

Function Call Syntax:

```
Separator( STD_ARGS );
Separator( STD_ARGS, color );
Separator( STD_ARGS, color, indices );
Separator( STD_ARGS, thickness );
Separator( STD_ARGS, thickness, color );
Separator( STD_ARGS, thickness, color, indices );
```

Argument Descriptions:

alignment - *integer*. If the format value is HORIZONTAL, then this value is either RIGHT, CENTER or LEFT. If the format value is VERTICAL, then this value is either TOP, CENTER or BOTTOM.
color - *pointer of type Color*. This is the base color to be used for the menu item. The default is gray.
format - *integer*. This value is either HORIZONTAL or VERTICAL.
indices - *array of 2 integers*. These are color table entries that can be used and modified by the menu item. If the indices are used in the constructor, then menu item will use the color table, otherwise the color are sent directly to the display device.
length - *float*. This is the length of the separator.
structure_number - *integer*. This is the structure identifier to be used to create the menu item.
thickness - *float*. This is the menu item shadow thickness. The default is 0.4.
x, y - *floats*. These are the coordinates that describe the location of the menu item.

Member Functions

State Control Functions:

None.

Event Control Functions:

None.

Miscellaneous Functions:

None.

Sample Usage

```
#include "interface.h"
#include "separator.h"

Separator   separator( 1, 0, 5, HORIZONTAL, LEFT, 20 );
```

Slider

Description

A slider is used to enter a value from a range of values. The visual interaction of the slider allows the user to have a more accurate feel for the boundaries and range of the value being entered.

Appearance

Figure 29 shows the dimensions of a slider. The dimensions controlled by the programmer are the height, width and shadow thickness. By default, the remaining dimensions are set using the following relations:

highlight_thickness = 0.3

highlight_margin_thickness = 0.5

label_height = 0.6*height

label_margin_thickness = 0.2*height

number_margin_thickness = 0.5*height

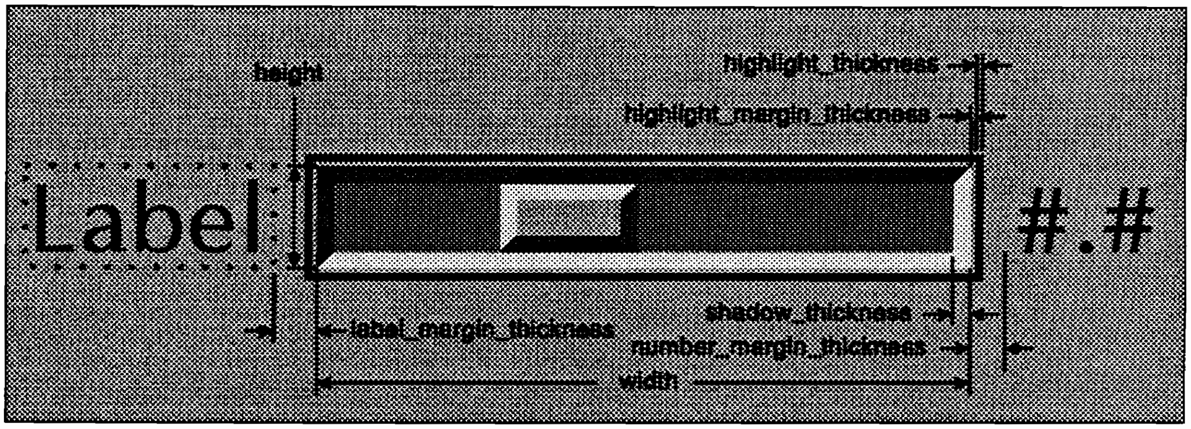


Figure 29 - Slider Dimensions

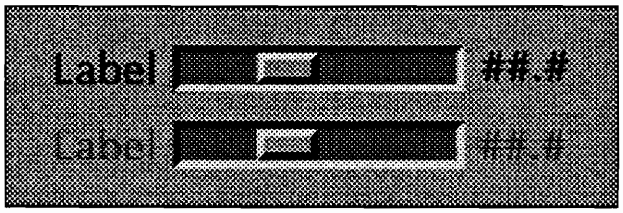
States

A slider is always in one of the following two states:

ACTIVE - ready for input

INACTIVE - not currently accepting input

Figure 30 shows what each of the slider states looks like.



Active

Inactive

Figure 30 - Slider States

Events

type: SLIDER

reasons: SLIDER_CHANGE

times: IMMEDIATE, DELAYED

data: single float value

Manager

None.

Constructors

Standard Arguments:

```
STD_ARGS = structure_number, x, y, horizontal_alignment, vertical_alignment, width,
  height, min_value, max_value
```

Function Call Syntax:

```
Slider( STD_ARGS );
Slider( STD_ARGS, base_color );
Slider( STD_ARGS, base_color, indices );
Slider( STD_ARGS, thickness );
Slider( STD_ARGS, thickness, base_color );
Slider( STD_ARGS, thickness, base_color, indices );
```

Argument Descriptions:

color - *pointer of type Color.* This is the base color to be used for the menu item. The default is gray.
horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the menu item with respect to the x value given.
indices - *array of 4 integers.* These are color table entries that can be used and modified by the menu item. If the indices are used in the constructor, then menu item will use the color table, otherwise the color are sent directly to the display device.
min_value, max_value - *floats.* These are the upper and lower bounds for the value associated with the slider.
structure_number - *integer.* This is the structure identifier to be used to create the menu item.
thickness - *float.* This is the menu item shadow thickness. The default is 0.4.
vertical_alignment - *integer.* This value is either TOP, CENTER, or BOTTOM and describes the alignment of the menu item with respect to the y value given.
width, height - *floats.* These values describe the dimensions of the menu item.
x, y - *floats.* These are the coordinates that describe the location of the menu item.

Member Functions

State Control Functions:

```
void        change_state_to_active( <perform_flag> );
void        change_state_to_inactive( <perform_flag> );
int         get_state()
```

perform_flag - *integer.* This is an optional argument that is either PERFORM or DO_NOT_PERFORM. The value indicates if the workstation should be updated to display the change in the menu item state. If the argument is not specified, then the workstation will update by default.

Event Control Functions:

```
void        set_change_event_id( id )
void        set_event_time( event_time )
int         get_event_time()
```

event_time - *integer.* This value is either DELAYED or IMMEDIATE and sets the event time for the menu item. The default is IMMEDIATE.

id - *integer.* This is the identification number for the event. If no id is specified, then it will automatically be set to the structure id used for the menu item.

Miscellaneous Functions:

void	add_label(position, string);
void	add_label(position, string, font);
void	add_label(position, display_structure, max_width, max_height);
void	add_label(position, display_structure, max_width, max_height, scale);
void	add_number(position, format);
void	add_number(position, format, font);
void	set_initial_value(value);
void	set_value(value, <perform_flag>);
void	set_step_size(step_size);

display_structure - *integer.* This is the PHIGS structure identifier for the structure to be displayed with the menu item (instead of a text string). The structure is assumed to be centered at (0,0).
font - *integer.* This is the PHIGS font identifier to be used for menu item. The default font identifier is 9.
format - *float.* This value indicates the format specifier to be applied to the number. For example, 3.2 means 3 placed before the decimal and two places after the decimal.
max_width, max_height - *floats.* These values describe the maximum dimensions of the display structure.
perform_flag - *integer.* This is an optional argument that is either PERFORM or DO_NOT_PERFORM. The value indicates if the workstation should be updated to display the change in the menu item state. If the argument is not specified, then the workstation will update by default.
position - *integer.* This value indicates where the number or label should be added to the slider. Legal values are RIGHT and LEFT.
scale - *float.* This is a factor by which the display structure can be scaled before displaying.
step_size - *float.* This is the amount the slider will move when clicked on or moved with the keyboard. The default step size is one-tenth the difference between the minimum and maximum values of the slider.
string - *pointer of type char.* This is the string to be displayed with the menu item.
value - *float.* This is the value for the menu item to display.

Sample Usage:

```
#include "interface.h"
#include "slider.h"

Slider      s1( 1, 5, 5, LEFT, TOP, 25, 5, 0, 360 );
s1.add_label( LEFT, "Angle" );
```

```
        s1.add_number( RIGHT, 3.0 );
        s1.set_initial_value( 180 );
        s1.set_step_size( 10 );
```

Appendix D - Menu Item Manager Description

Description

This section describes the menu item manager class. This class is inherited by all menu item manager classes such as radio button managers. The functions in this class are therefore accessible in those classes.

Member Functions

Constructor

The constructor does not need to be called by the programmer of a menu item manager class because it is automatically called.

Management Functions

To add a menu item to a menu item manager, use the following function:

```
void    add_managed_menu_item( menu_item )
```

menu_item - *pointer to type Menu_item.* This is the address of the menu item to be added to the menu item manager.

To retreive the first menu item in the linked list of menu items managed by the menu item manager (the managed menu item linked list), use the following function:

```
Menu_item        *get_first_managed_menu_item()
```

Return Value - A pointer to the first menu item in the managed menu item linked list.

Appendix E - Window Descriptions

The following section presents a summary of the features of the windows created. The section describes the following features for each window:

- Description
- Appearance
- Constructors
- Member Functions
- Sample Usage

The window implemented thus far are:

- Geometry Manager
- Pop-up Menu
- Simple View
- Static Menu

Common Functions

These are functions common to all windows. The functions are defined in the Window base class. Any class that is derived from the Window base class will thus have these functions as members.

The following function can be used to give the window a name. This name can later be used to get a pointer to the window from the interface manager. Window names are used for convenience purposes, so it is not necessary to use this function unless you later want to use the name.

```
void    set_name( name )
```

name - *pointer to type char.* This is a pointer to the string to be used as the window name.

The following function is used to get the name of a window. It can be used to identify an unknown window pointer.

```
char    *get_name()
```

The following function performs a string compare on the window's name and the string supplied. If the strings are the same, a zero is returned, otherwise a non-zero value is returned.

```
int     is_this_your_name( name )
```

name - *pointer to type char.* This is a pointer to the string to be compared against the window's name.

Once the window has been added to an interface manager, a pointer to the interface manager can be retrieved with the following call:

```
Interface_manager       *get_manager()
```

Geometry Manager

Description

A geometry manager is a sophisticated PHIGS view. The view contains a border that allows the window to be stretched and shrunk. The window can also be moved on screen. The button in the upper right corner toggles the window between its normal size and a programmer defined full screen. The window is modeled after a Motif window.

Appearance

Figure 31 shows what a geometry manager looks like.

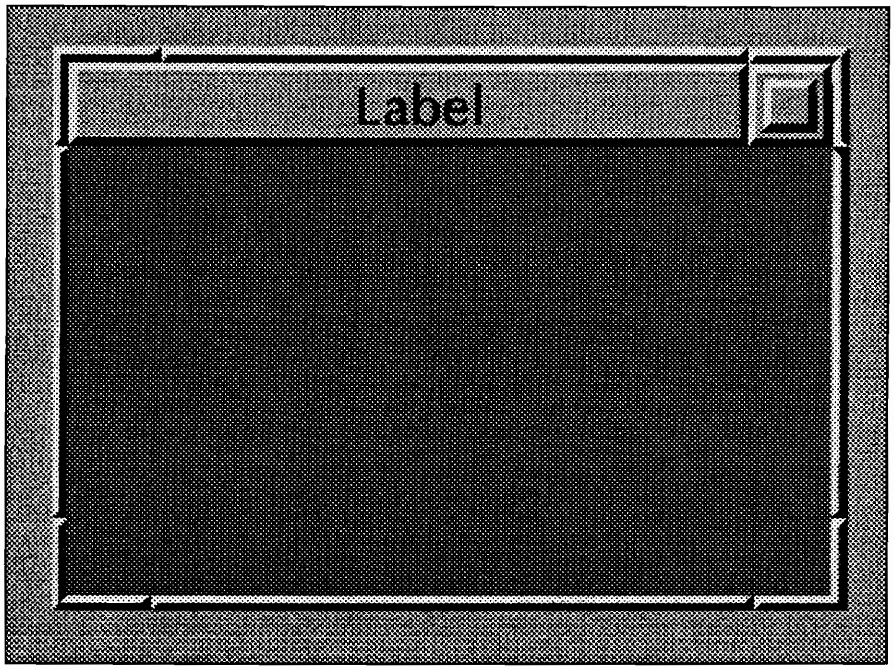


Figure 31 - Geometry Manager Dimensions

Constructors

Standard Arguments:

STD_ARGS = structure_number, utility_structure_number, x, y, horizontal_alignment, vertical_alignment, width, height, scale

Function Call Syntax:

```
Geometry_manager( STD_ARGS )
Geometry_manager( STD_ARGS, indices )
Geometry_manager( STD_ARGS, title )
Geometry_manager( STD_ARGS, title, indices )
Geometry_manager( STD_ARGS, title, font )
```

Geometry_manager(STD_ARGS, title, font, indices)
Geometry_manager(STD_ARGS, title, character_height)
Geometry_manager(STD_ARGS, title, character_height, indices)
Geometry_manager(STD_ARGS, title, character_height, font)
Geometry_manager(STD_ARGS, title, character_height, font, indices)
Geometry_manager(STD_ARGS, thickness)
Geometry_manager(STD_ARGS, thickness, indices)
Geometry_manager(STD_ARGS, thickness, title)
Geometry_manager(STD_ARGS, thickness, title, indices)
Geometry_manager(STD_ARGS, thickness, title, font)
Geometry_manager(STD_ARGS, thickness, title, font, indices)
Geometry_manager(STD_ARGS, thickness, title, character_height)
Geometry_manager(STD_ARGS, thickness, title, character_height, indices)
Geometry_manager(STD_ARGS, thickness, title, character_height, font)
Geometry_manager(STD_ARGS, thickness, title, character_height, font, indices)

Argument Descriptions:

character_height - *float.* This is the height of the letters used in the title.
color - *pointer of type Color.* This is the base color to be used for the window. The default is gray.
font - *integer.* This is the PHIGS font identifier to be used for the title. The default font identifier is 9.
horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the window with respect to the x value given.
indices - *array of 4 integers.* These are color table entries that can be used and modified by the window. If the indices are used in the constructor, then window will use the color table, otherwise the color are sent directly to the display device.
scale - *float.* This is a factor by which the NPC are mapped to the WC.
structure_number - *integer.* This is the structure identifier to be used to create the window.
thickness - *float.* This is the window shadow thickness. The default is 0.4.
title - *pointer to type char.* This is a pointer to the string to be displayed as a title.
utility_structure_number - *integer.* This is the structure identifier that will be periodically used by the geometry manager.
vertical_alignment - *integer.* This value is either TOP, CENTER, or BOTTOM and describes the alignment of the window with respect to the y value given.
width, height - *floats.* These are the dimensions of the window. The sizes are specified with respect to NPC space.
x, y - *floats.* These are the coordinates that describe the location of the window. The coordinate system is NPC space.

Member Functions

Operation Control Functions

These two functions control whether the window can be raised by the interface manager. If raising is set to off, then the interface manager will not raise the window when it is selected by the user. These functions do not affect the interface manager functions raise_window() and lower_window(). By default the raising operation is on for geometry managers.

```
void    turn_raising_off()
void    turn_raising_on()
```

This function returns YES or NO based on the state of the raise flag.

```
int     get_raise_flag()
```

When the button in the upper right corner of the geometry manager is clicked, the geometry manager will expand to full view. By default, full view is defined as the entire screen. Use this function to set a new limiting area for the geometry manager.

```
void    set_limits( float _x_max, float _x_min, float _y_max, float _y_min )
```

x_max, x_min, y_max, y_min - *floats*. These are NPC values that define the boundaries for the geometry manager.

Miscellaneous Functions

These functions apply to the 3D view managed by the geometry manager. It is important to only use these functions to control the properties of the view.

Modifying the view directly with PHIGS calls will corrupt the data in the geometry manager class and will lead to unwanted and unpredictable results.

```
float   get_far_clipping_indicator()
float   get_near_clipping_indicator()
float   get_scale()
float   get_view_index()
float   get_x_angle()
float   get_x_center()
float   get_y_angle()
float   get_y_center()
float   get_z_angle()
int     get_HLHSR_mode()
int     get_projection_type()
int     get_shielding_indicator()
void    associate_structure( int _structure_id, float _priority )
void    pan_down( float _percent)
void    pan_left( float _percent)
void    pan_right( float _percent)
void    pan_up( float _percent)
void    set_center_to( float _x_center, float _y_center)
void    set_far_plane_distance( float _distance)
void    set_n_projection_reference_point( float _n)
void    set_near_plane_distance( float _distance)
void    set_projection_reference_point( float _u, float _v, float _n )
void    set_projection_to_parallel()
void    set_projection_to_perspective()
void    set_rotation_to( float_x_angle, float _y_angle, float_z_angle )
void    set_shielding_color( Color *_color )
void    set_shielding_color( int _color_index )
void    set_u_projection_reference_point( float _u )
void    set_v_projection_reference_point( float _v )
void    set_view_plane_distance( float _distance )
void    set_x_center_to( float _x_center )
void    set_x_rotation_to( float _angle )
void    set_x_visual_rotation_delta( float _delta_angle )
void    set_y_center_to( float _y_center )
void    set_y_rotation_to( float _angle )
void    set_y_visual_rotation_delta( float _delta_angle )
void    set_z_rotation_to( float _angle )
void    set_z_visual_rotation_delta( float _delta_angle )
void    turn_far_clipping_off()
void    turn_far_clipping_on()
void    turn_HLHSR_off()
void    turn_HLHSR_on()
void    turn_near_clipping_off()
void    turn_near_clipping_on()
void    zoom( float _factor )
```

Sample Usage

```
#include "geometry_manager.h"
#include "interface.h"

Geometry_manager  gman( 1, 2, 0, 0, LEFT, TOP, 1.0, 0.7, 100 );

gman.turn_raising_off();
gman.turn_HLHSR_off();
gman.set_x_center_to( 10 );
gman.associate_structure( 100, 0.5 );
```

Pop-up Menu

Description

A pop-up menu is used to let a user select several options. The menu is "popped" on top of the other windows and does not disappear until the user is finished working with it. While the pop-up menu is active, other windows cannot be raised. To view what is in one of the windows, the pop-up menu can be moved by dragging it to a more appropriate location.

Appearance

Figure 32 shows what a pop-up menu looks like. The scale used for a pop-up menu is 100 times the NPC. This means that a menu that is that size of the screen would be 100 units wide and 100 units high.

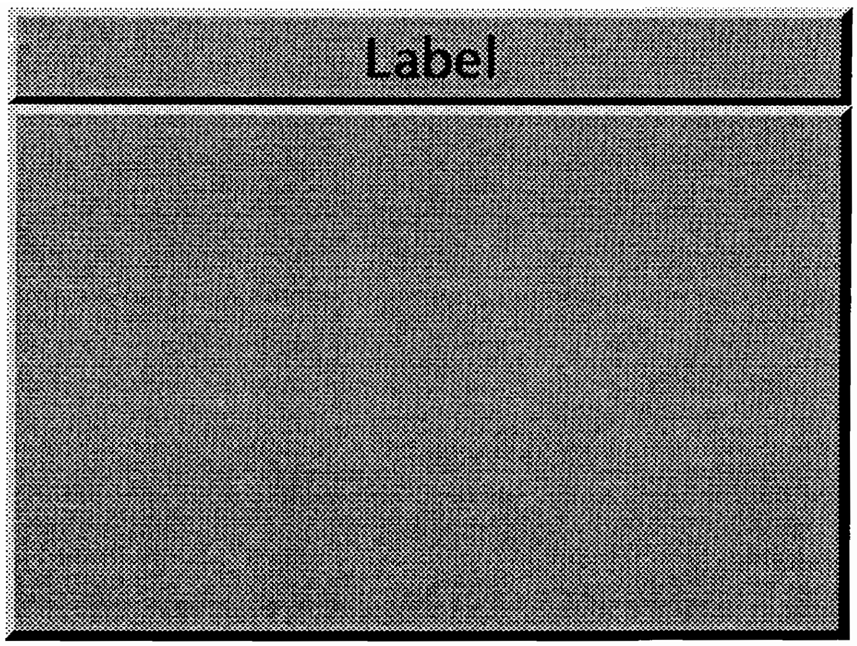


Figure 32 - Pop-up Menu Dimensions

Constructors

Standard Arguments:

STD_ARGS = structure_number, utility_structure_number, x, y, horizontal_alignment, vertical_alignment, width, height, title, character_height

Function Call Syntax:

```
Pop_up_menu( STD_ARGS )
Pop_up_menu( STD_ARGS, color )
Pop_up_menu( STD_ARGS, color, indices )
Pop_up_menu( STD_ARGS, thickness )
Pop_up_menu( STD_ARGS, thickness, color )
Pop_up_menu( STD_ARGS, thickness, color, indices )
Pop_up_menu( STD_ARGS, font )
Pop_up_menu( STD_ARGS, font, color )
Pop_up_menu( STD_ARGS, font, color, indices )
Pop_up_menu( STD_ARGS, font, thickness )
Pop_up_menu( STD_ARGS, font, thickness, color )
Pop_up_menu( STD_ARGS, font, thickness, color, indices )
```

Argument Descriptions:

character_height - *float.* This is the height of the letters used in the title.
color - *pointer of type Color.* This is the base color to be used for the window. The default is gray.
font - *integer.* This is the PHIGS font identifier to be used for the title. The default font identifier is 9.
horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the window with respect to the x value given.
indices - *array of 4 integers.* These are color table entries that can be used and modified by the window. If the indices are used in the constructor, then window will use the color table, otherwise the color are sent directly to the display device.
structure_number - *integer.* This is the structure identifier to be used to create the window.
thickness - *float.* This is the window shadow thickness. The default is 0.4.
title - *pointer to type char.* This is a pointer to the string to be displayed as a title.
vertical_alignment - *integer.* This value is either TOP, CENTER, or BOTTOM and describes the alignment of the window with respect to the y value given.
width, height - *floats.* These are the dimensions of the window. The sizes are specified with respect to NPC space.
x, y - *floats.* These are the coordinates that describe the location of the window. The coordinate system is NPC space.

Member Functions

Operation Control Functions:

To open a pop-up menu, add it to the interface manager with the interface manager function add_window(). To close it there are two options:

1. **cancel_close** - this will cause all the menu items in the menu to revert to the state that they were in when the menu was opened. No events will be generated.
2. **ok_close** - this will leave the menu items in the states they are currently in. It will also cause any menu items that need to generate delayed events to do so now.

DO NOT use the interface manager function remove_window() to close a pop-up menu. The functions ok_close() and cancel_close() automatically invoke that procedure in addition to the other operations they perform.

```
void    ok_close( event )
void    cancel_close()
```

event - *pointer to type Event*. If any delayed events need to be generated, then those events will be appended to the linked list that this member belongs to. If this member is NULL, then it will become the first member in the linked list of generated delayed events.

Miscellaneous Functions:

None.

Sample Usage

```
#include "event.h"
#include "interface.h"
#include "interface_manager.h"
#include "pop_up_menu.h"

Pop_up_menu pop_up( 1, 0.3, 0.7, LEFT, TOP, 0.5, 0.3, "Menu", 2 );
```

```
// add menu items created elsewhere
pop_up.add_menu_item( &push_button );
pop_up.add_menu_item( &slider );

// add the menu to the previously created interface manager
interface_manager.add_window( &pop_up );

Event       *event;
int         exit = NO;

do {
   // process as normal
   event = interface_manager.process( 100 );

   while ( event != NULL )
      {

      switch ( event.get_id() )
         {
         case OK_CLOSE:
            exit = YES;
            pop_up.ok_close( event );
            break;
         case CANCEL_CLOSE:
            exit = YES;
            pop_up.cancel_close();
            break;
         case ANYTHING_ELSE:
            // do what ever
            break;
         }

      }

   event->extract_next_event();
   } while ( exit == NO );
```

Simple View

Description

This window is an encapsulation of a PHIGS view. The appearance on screen is identical to a PHIGS view. The difference lies in the member functions included in the class that allow easier manipulation of the view properties. The view also has the ability to raise itself above other windows if indicated to do so by the user.

Appearance

Figure 33 shows what a simple view looks like. A simple view appears on screen just like a PHIGS view. The border color and shielding color displayed here are optional attributes.

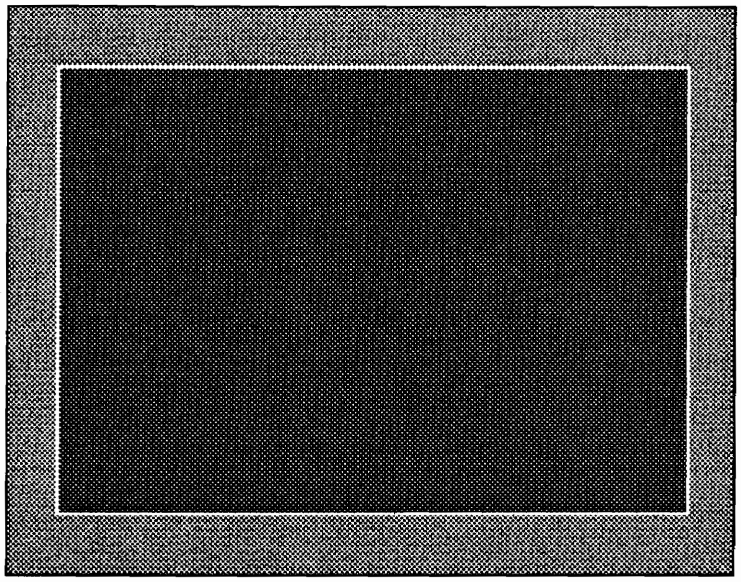


Figure 33 - Simple View Dimensions

Constructors

Standard Arguments:

STD_ARGS = x, y, horizontal_alignment, vertical_alignment, width, height, scale

Function Call Syntax:

Simple_view(STD_ARGS)

Argument Descriptions:

horizontal_alignment - *integer.* This value is either RIGHT, CENTER, or LEFT and describes the alignment of the window with respect to the x value given.

scale - *float*. This is a factor by which the NPC are mapped to the WC.
vertical_alignment - *integer*. This value is either TOP, CENTER, or BOTTOM and describes the alignment of the window with respect to the y value given.
width, height - *floats*. These are the dimensions of the window. The sizes are specified with respect to NPC space.
x, y - *floats*. These are the coordinates that describe the location of the window. The coordinate system is NPC space.

Member Functions

Operation Control Functions

These two functions control whether the window can be raised by the interface manager. If raising is set to off, then the interface manager will not raise the window when it is selected by the user. These functions do not affect the interface manager functions raise_window() and lower_window(). By default the raising operation is off for simple views. If you do not have a need to let the user raise the window, then leave the flag set to off.

```
void    turn_raising_off()
void    turn_raising_on()
```

This function returns YES or NO based on the state of the raise flag.

```
int     get_raise_flag()
```

Miscellaneous Functions:

```
float   get_active_flag()
float   get_scale()
float   get_viewport_height()
float   get_viewport_width()
float   get_viewport_x()
float   get_viewport_y()
float   get_x_angle()
float   get_x_center()
float   get_y_angle()
float   get_y_center()
float   get_z_angle()
int     get_border_indicator()
int     get_far_clipping_indicator()
int     get_HLHSR_mode()
int     get_near_clipping_indicator()
int     get_projection_type()
int     get_shielding_indicator()
int     get_view_index()
int     get_window_clipping_indicator()
void    associate_structure( int _structure_id, float _priority )
void    pan_down( float _percent )
void    pan_left( float _percent )
void    pan_right( float _percent )
void    pan_up( float _percent )
void    set_border_color( Color *_color )
void    set_border_color( int _color_index )
void    set_center_delta( float _x_delta, float _y_delta )
void    set_center_to( float _x_center, float _y_center )
void    set_far_plane_distance( float _distance )
void    set_input_higher_than( int _reference_view_index )
void    set_input_lower_than( int _reference_view_index )
void    set_n_projection_reference_point( float _n )
void    set_near_plane_distance( float _distance )
void    set_output_higher_than( int _reference_view_index )
void    set_output_lower_than( int _reference_view_index )
void    set_projection_reference_point( float _u, float _v, float _n )
void    set_projection_to_parallel()
void    set_projection_to_perspective()
void    set_rotation_delta( float _x_delta, float _y_delta, float _z_delta )
void    set_rotation_to( float _x_angle, float _y_angle, float _z_angle )
void    set_shielding_color( Color *_color )
void    set_shielding_color( int _color_index )
void    set_u_projection_reference_point( float _u )
void    set_v_projection_reference_point( float _v )
void    set_view_index( int _view_index )
void    set_view_plane_distance( float _distance )
void    set_viewport( float _x, float _y, float _width, float _height )
void    set_viewport_location( float _x, float _y )
void    set_viewport_size( float _width, float _height )
void    set_visual_rotation_delta( float _x_delta, float _y_delta, float _z_delta )
void    set_wsid( int _wsid )
void    set_x_center_delta( float _x_delta )
void    set_x_center_to( float _x_center )
void    set_x_rotation_delta( float _x_delta )
void    set_x_rotation_to( float _x_angle )
```

void	set_x_visual_rotation_delta(float _x_delta)
void	set_y_center_delta(float _y_delta)
void	set_y_center_to(float _y_center)
void	set_y_rotation_delta(float _y_delta)
void	set_y_rotation_to(float _y_angle)
void	set_y_visual_rotation_delta(float _y_delta)
void	set_z_rotation_delta(float _z_delta)
void	set_z_rotation_to(float _z_angle)
void	set_z_visual_rotation_delta(float _z_delta)
void	turn_border_off()
void	turn_border_on(Color *_color)
void	turn_border_on(int _color_index)
void	turn_border_on()
void	turn_clipping_off()
void	turn_clipping_on()
void	turn_far_clipping_off()
void	turn_far_clipping_on()
void	turn_HLHSR_off()
void	turn_HLHSR_on()
void	turn_near_clipping_off()
void	turn_near_clipping_on()
void	turn_off()
void	turn_on()
void	turn_shielding_off()
void	turn_shielding_on(Color *_color)
void	turn_shielding_on(int _color_index)
void	turn_shielding_on()
void	zoom(float _factor)

Sample Usage

```
#include "interface.h"
#include "simple_view.h"

Simple_view simple_view( 0.5, 0.5, CENTER, CENTER, 0.7, 0.7, 25 );

simple_view.turn_on();
simple_view.turn_shielding_on();
simple_view.turn_HLHSR_on();
simple_view.set_z_visual_rotation_delta( 3.141593/6 );
```

Static Menu

Description

A static menu appears on screen for the duration of a program. The menu cannot be moved on screen, and does not restrict access to other windows. The menu usually contains buttons that initiate high level operations.

Appearance

Figure 34 shows what a static menu looks like.

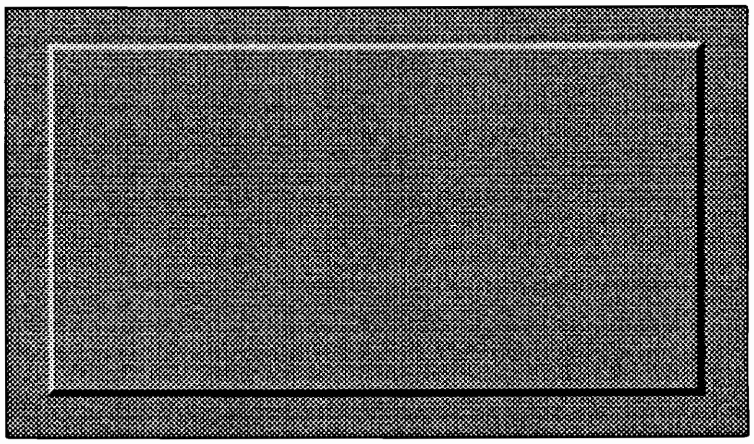


Figure 34 - Static Menu Dimensions

Constructors

Standard Arguments:

STD_ARGS = structure_number, x, y, horizontal_alignment, vertical_alignment, width, height

Function Call Syntax:

```
Static_menu( STD_ARGS )
Static_menu( STD_ARGS, color )
Static_menu( STD_ARGS, color, indices )
Static_menu( STD_ARGS, thickness )
Static_menu( STD_ARGS, thickness, color )
Static_menu( STD_ARGS, thickness, color, indices );
```

Argument Descriptions:

horizontal_alignment - *integer*. This value is either RIGHT, CENTER, or LEFT and describes the alignment of the window with respect to the x value given.
scale - *float*. This is a factor by which the NPC are mapped to the WC.
structure_number - *integer*. This is the structure identifier to be used to create the window.
vertical_alignment - *integer*. This value is either TOP, CENTER, or BOTTOM and describes the alignment of the window with respect to the y value given.
width, height - *floats*. These are the dimensions of the window. The sizes are specified with respect to NPC space.
x, y - *floats*. These are the coordinates that describe the location of the window. The coordinate system is NPC space.

Member Functions

Operation Control Functions

These two functions control whether the window can be raised by the interface manager. If raising is set to off, then the interface manager will not raise the window when it is selected by the user. These functions do not affect the interface manager functions raise_window() and lower_window(). By default the raising operation is on for static menus. If you plan on raising other windows, then leave this flag set to on. Optimal performance for a menu occurs when the menu has the highest output priority.

```
void    turn_raising_off()
void    turn_raising_on()
```

This function returns YES or NO based on the state of the raise flag.

```
int     get_raise_flag()
```

Miscellaneous Functions:

None.

Sample Usage

```
#include "event.h"
#include "interface.h"
#include "interface_manager.h"
#include "static_menu.h"

Static_menu menu( 1, 0.3, 0.7, LEFT, TOP, 0.5, 0.3 );

// add menu items created elsewhere
menu.add_menu_item( &push_button );
menu.add_menu_item( &slider );

// add the menu to the previously created interface manager
interface_manager.add_window( &menu );

Event       *event;
```

```
int          exit = NO;

do {
   // process as normal
   event = interface_manager.process( 100 );

   while ( event != NULL )
      {

      switch ( event.get_id() )
         {
         case THE_EXIT_BUTTON:
            exit = YES;
            break;
         case ANYTHING_ELSE:
            // do whatever
            break;
         }

      }

   event->extract_next_event();
   } while ( exit == NO );
```

Appendix F - Menu Manager Description

Description

This section describes the menu manager class. This class is inherited by all menu classes such as pop-up menus and static menus. The functions in this class are therefore accessible in those classes.

In general the information in this appendix is not needed unless the reader is attempting to create a window menu class. Many functions contained in this class are explained in the description of the pop-up menu and static menu class descriptions.

Member Functions

Constructor

The constructor does not need to be called by the programmer of a menu class because it is automatically called.

Management Functions

There are two ways to add menu items to a menu manager. The first method specifies each menu item individually.

```
void     add_menu_item( menu_item )
```

menu_item - *pointer to type Menu_item.* This is the address of the menu item to be added to the menu manager.

The second method specifies a menu item manager. All of the menu items managed by the menu item manager will then be added to the menu manager.

```
void     add_menu_item_manager( menu_item_manager )
```

menu_item_manager - *pointer to type Menu_item_manager.* This is the address of the menu item manager which contains the menu items to be added to the menu manager.

The manage() function should be called to tell all the menu items in the menu manager to manage themselves. The structure used by each menu item will be associated with the view index specified with a priority of 0.75.

```
void     manage( wsid, view_index )
```

wsid - *float.* This is the workstation identifier.
view_index - *integer.* This is the view index that with menu manager is using.

When a menu is closed, the following function should be used to tell each of the menu items to unmanage themselves.

```
void     unmanage()
```

Processing Functions

Once a mouse button press has been detected, the process_from_mouse() procedure should be called with the current locator values. It can be assumed that when this call returns, the mouse button will have been released.

```
void      process_from_mouse( x, y, view_index, event )
```

x, y - *floats*. These should be values retrieved from sampling the locator when the mouse button was pressed.
view_index - *integer*. This is the view index retrieved from sampling the locator when the mouse button was pressed.
event - *pointer to type Event*. This is a pointer to one of the members of the linked list of events. If a menu item in the menu manager generates an event, it will be added to this list.

If the process_from_mouse() call cannot find a menu item to process the mouse input, then the process_misc_input() virtual function will be called. This function should be defined in the class derived from the menu manager class. An example of where this function is used is in moving a pop-up menu.

```
virtual void      process_misc_input( x, y, view_index )
```

x, y - *floats*. These are the values retrieved from sampling the locator when the mouse button was pressed.
view_index - *integer*. This is the view index retrieved from sampling the locator when the mouse button was pressed.

If a keystoke is detected, then the process_from_keyboard() call can be called to see if a menu item in the menu manager can process it.

```
void      process_from_keyboard( key press, event )
```

key press - *integer*. This is the PHIGS key identifier for the key press.
event - *pointer to type Event*. This is a pointer to one of the members of the linked list of events. If a menu item in the menu manager generates an event, it will be added to this list.

Miscellaneous Functions

By default, the first menu item added to the menu manager becomes the highlighted menu item. To change the highlighted menu item, use the following call.

```
void     set_highlighted_menu_item( menu_item )
```

menu_item - *pointer to type Menu_item.* This is the address of the menu item to be made the highlighted menu item.

The get_color() function does not really belong in the menu manager class, but it greatly simplifies certain operations by being there. Ideally the menu manager class is only a manager class that has no physical properties. Often menu items need to set their color based on the color they are being displayed on. An example of this is a label which automatically determines its color based on the contrast with the color it is displayed against. Since the only connection a menu item has to the class displaying the menu item is the menu manager, it is convenient for the menu item to ask the menu manager its color. This is accomplished with the get_color() virtual function. The function should be defined for the class derived from the menu manager class - the class that has the physical property of a color.

```
virtual Color     *get_color()
```

Vita

Scott Woyak was born June 6, 1969 in Pennsylvania. He later moved to and lived in Italy, Illinois, Sweden, Minnesota, and back to Pennsylvania. At a very young age he became interested in taking things apart. This led to his interests in the fields of Engineering and Science. Scott chose Virginia Tech for his undergraduate and graduate studies because he admired the beauty of the surroundings. During his undergraduate career he met and later married Laura Mangelli. He contributes much of his graduate success to her and is very appreciative of her willingness to temporarily put aside her career goals while he pursued his.

[illegible]