

**An Object-Oriented Class Library
for the Creation of Engineering Graphs**

by

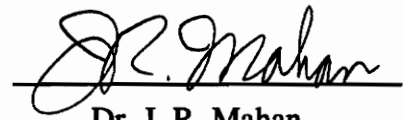
R. Steven Uhorchak

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:


Dr. Sankar Jayaram, Chairman


Dr. A. Myklebust


Dr. J. R. Mahan

May 3, 1993
Blacksburg, Virginia

C.2

LD

E655

V855

1993

U367

C.2

Abstract

Since the availability of the first PHIGS (Programmers Hierarchical Interactive Graphics System) implementation in the mid 1980's, interest in the use of PHIGS has been steadily growing among the CAD applications developer's community. Every year, more PHIGS-based CAD applications programs are being created to ensure portability and make use of the high-level support provided by PHIGS [JAYA93b].

One of the common uses of computer graphics in engineering is for the creation of graphs. Commonly used graphs, such as line graphs (two-dimensional or three-dimensional), bar charts, pie charts, surface plots, etc., can be created using PHIGS. This involves the creation of several methods to scale the data, draw the graph, display labels, display axes, display legends and several other programming tasks, which are re-created by each applications program.

This thesis describes the creation of an object-oriented class library to facilitate the creation of engineering graphs using PHIGS. This library provides the programmer with a set of tools to create commonly used graphs (line graphs, pie charts, bar charts, polar plots, and 3D plots). The class library will allow PHIGS programmers to quickly and easily create graphs for use with applications programs. This set of classes uses a PHIGS-based, Motif-like interface framework (described by Woyak and Myklebust

Abstract

[WOYA93]). User interface methods to allow the end-user of the graphing program to modify attributes of the graph (e.g. line types, legends, colors, etc.) are encapsulated within this class library and are hidden from the programmer using these classes. The library of classes, the user interface methods, and the use of this system is described in this thesis.

Acknowledgements

First, I would like to thank my parents, Robert and Jeanne Uhorchak for their love and support throughout my career. They taught me that through hard work, dedication, and by not giving up that almost all obstacles can be overcome.

To Sankar Jayaram, my advisor, I would like to give thanks for all the guidance, patience, direction, and help on my thesis as well as many other projects endeavored during my academic career at Virginia Tech.

Thanks must also be extended to Dr. Arvid Myklebust, the director of the CAD Lab and Co-director of the ACSYNT Institute for the opportunity to work on the ACSYNT project. In addition I would like to thank Dr. J. R. Mahan for being on my committee.

Since my work was supported by the ACSYNT Institute I would like to thank all of the institute members for their financial support.

Finally, I would like to thank all of the members of the CAD lab for their support and encouragement, with special thanks going to Scott Woyak for his never ending patience and understanding of all my questions.

Table of Contents

1.0 Introduction	1
2.0 Literature Review	3
3.0 Graphs in ACSYNT - A Case Study	6
3.1 Current Methods Used in ACSYNT	7
3.2 Flexibility of Graphs	12
4.0 Thesis Objectives	13
5.0 Object-Oriented Design	15
5.1 Traditional Programming Philosophy	15
5.2 Object-Oriented Philosophy	16
5.3 Object-Oriented Design	17
5.4 Object-Oriented Programming Terms (Using C++)	18
5.4.1 Class	18
5.4.2 Polymorphism (Function Overloading)	18
5.4.3 Inheritance	18
5.4.4 Encapsulation	19

5.5 Advantages of Object-Oriented Programming	20
6.0 Motif-Like Interface	23
7.0 Overview of the Design	26
7.1 Class Hierarchy	26
7.2 User Interface For Graph Modification	28
8.0 Descriptions of Classes	31
8.1 Graph Manager Class	31
8.2 Graph Class	32
8.3 Individual Graph Classes	32
8.3.1 Bar_Graph Class	34
8.3.2 Line_Graph Class	37
8.3.3 Pie_Graph Class	40
8.3.4 Polar_Graph Class	43
8.3.5 Stacked_Bar Class	45
8.3.6 XYZ_Plot Class	48
9.0 Graph Entity Classes	52
9.1 Graph Entities	52

9.1.1 Axes Class	53
9.1.2 Bar Class	58
9.1.3 Curve Class	62
9.1.4 Grid Class	65
9.1.5 Legend Class	68
9.1.6 Pie Class	71
9.1.7 Polar Axes Class	74
9.1.8 Stacked Bar Class	77
9.1.9 Text Class	80
 10.0 Other Classes	 83
10.1 Data Series Class	83
10.2 Control Entities	84
10.2.1 Deletion Class	84
10.2.2 Highlighting Class	85
10.2.3 Invisibility Class	86
 11.0 Implementation and Sample Program	 88
 12.0 Implementation of Graphs for ACSYNT	 110

13.0 Conclusion	112
14.0 References	114
Appendix A. Class Library User Guide	119
A.1 Graph Classes	119
A.1.1 Bar Graph	119
A.1.2 Line Graph	121
A.1.3 Pie Graph	123
A.1.4 Polar Graph	125
A.1.5 Stacked Bar Graph	127
A.2 Graph Entities	129
A.2.1 Axes	129
A.2.2 Bar	133
A.2.3 Curve	136
A.2.4 Grid	139
A.2.5 Legend	142
A.2.6 Pie	145
A.2.7 Polar Axes	148
A.2.8 Series	151
A.2.9 Stacked_Bar	152

A.2.10 Text 155

A.3 Control Entities 157

 A.3.1 Deletion 157

 A.3.2 Highlight 158

 A.3.3 Invisible 160

Vita 162

List of Illustrations

Figure 1. Example of Graph in ACSYNT	8
Figure 2. Present ACSYNT Graphing Routines	9
Figure 3. Procedure for Creating Graphs in ACSYNT	10
Figure 4. Class Organization of Motif-Like Interface Framework	25
Figure 5. Class Hierarchy - First Design	27
Figure 6. Class Hierarchy	29
Figure 7. Graph Entities Used For Bar Graphs	35
Figure 8. Graph Entities Used For Line Graphs	38
Figure 9. Graph Entities Used For Pie Graphs	41
Figure 10. Graph Entities Used For Polar Graphs	44
Figure 11. Graph Entities Used For Stacked Bar Graphs	47
Figure 12. Graph Entities Used For XYZ Plots	50
Figure 13. Sample Program	89
Figure 14. Line Graph Created Using Class Library	90
Figure 15. Flow of Control After User Input	92
Figure 16. Line Graph Function for Processing User Input	93
Figure 17. Axes Functions for Processing User Input	94
Figure 18. Pop-Up Menu Created by Axes Class	95
Figure 19. Example of a Bar Graph	96

Figure 20. Example of a Stacked Bar Graph	97
Figure 21. Example of a Pie Graph	98
Figure 22. Example of a Polar Graph	99
Figure 23. Example of a XYZ Plot	100
Figure 24. Pop-Up Menu Created by Bar and Stacked Bar Classes	101
Figure 25. Pop-Up Menu Created by Curve Class	102
Figure 26. Pop-Up Menu Created by Grid Class	103
Figure 27. Pop-Up Menu Created by Legend Class	104
Figure 28. Pop-Up Menu Created by Text Class	105
Figure 29. Pop-Up Menu Created by Pie Axes	106
Figure 30. Pop-Up Menu Created by Polar Axes	107
Figure 31. Example of Color Modification Menu	108
Figure 32. Example of PHIGS Primitive Attribute Modification Menu	109
Figure 33. Example of Variables Template in ACSYNT	111

1.0 Introduction

Every year thousands of CAD/CAM and engineering applications software packages are created by software engineers. In a recent survey of the use of CAD/CAM systems in a pre-selected group of 41 Fortune 500 companies, many companies reported the development of custom CAD software in-house [PENN91] [PENN92]. A significant amount of time and effort is spent by these programmers in developing the graphics interfaces to be provided to the user (e.g., user input facilities such as menus and icons, geometry rendering, data visualization, etc.). Thus, CAD applications developers are always on the lookout for better and higher-level tools which will assist them in quickly creating customized graphical user interfaces (GUI's) for their CAD systems.

The ISO standard for three-dimensional graphics, Programmers Hierarchical Interactive Graphics System (PHIGS), is one of the tools commonly used by CAD applications programmers. Although PHIGS-based programs are device-independent, PHIGS does not provide enough high-level support for the creation of custom CAD systems. In addition to graphics and GUI's, high-level tools are needed in the areas of geometric modeling, artificial intelligence, database methods, data exchange, etc. A number of researchers have been working toward providing a set of high-level tools to support CAD applications programming [JAYA90] [JAYA91] [JAYA92] [JAYA93a] [MONT91] [FLEM91] [FLEM92] [LINW93] [MYKL92a] [MYKL92b] [SCHR92]. These tools should not only

be device-independent, but also extensible and maintainable. Such reusable tools will reduce the time and money spent on implementing basic methods and techniques required by most CAD systems (e.g., GUI, graphs, etc.).

Many of these attributes can be realized by ensuring that the tools being created are object oriented. The reuse of working code from project to project was originally viewed as a promising way to address the difficulties with developing and modifying large systems.

One of the common uses of computer graphics in engineering is for the creation of graphs. Commonly used graphs, such as line graphs, bar charts, etc., can be created using PHIGS. This would involve the creation of methods to scale the data, draw the graph, display labels, display axes, display legends, and several other programming tasks which are re-created by each applications program. This thesis describes the creation of an object-oriented class library to facilitate the creation of engineering graphs using PHIGS. This library will allow PHIGS programmers to quickly and easily create graphs for use with their applications programs. The library of classes, the user interface methods and examples of the use of this system are described in this thesis.

2.0 Literature Review

Many commercial graphics software packages are available such as PHIGURE, FIGraph, DADiSP 3.0, XPLOT, GPHIGS, HOOPS, etc., but at the present time "there are still no sound object-oriented graphics systems available" [WISS90]. Some commercial systems claim to be object oriented, but are in fact not written in an object-oriented manner, they are merely object-like. PHIGURE is designed for users who need to quickly visualize data and develop graphics applications. PHIGURE is furnished with an object code library with C and FORTRAN 77 interfaces and is integrated with Motif. The graphics PHIGURE uses is GPHIGS. FIGraph is a visualization toolkit for two-dimensional and three-dimensional graphical representations using C and FORTRAN. FIGraph is based on FIGARO+. XPLOT is a program designed to quickly generate graphical output. XPLOT runs under the X Window System. It uses the OPEN LOOK graphical user interface. GPHIGS is a high-level two-dimensional and three-dimensional graphics development toolkit. GPHIGS uses C and FORTRAN 77 and is integrated with Motif. It uses the PHIGS graphics standard. HOOPS is a subroutine library callable from C, C++, and FORTRAN. HOOPS consists of routines for creating, managing, querying, and editing a graphics database. HOOPS, while being object-oriented does not use the PHIGS standard. Also, for the majority of platforms HOOPS supports, either Motif or Windows is used as the window manager.

Conventional approaches to computer graphics are failing and are causing a bottleneck in industry. Even when graphics packages provide sophisticated capabilities, a mismatch exists between what is offered by a graphics system Application Program Interface (API) and what is required by most applications programmers, because graphics systems must be managed at an extremely low level [CUNN92].

Very little is known on how to create a set of software components that can be reused in different systems with little or no change [DUNN91]. In a survey of 41 Fortune 500 companies in which 26 responded, it was found that 73 percent develop their own customized software for their needs [PENN92]. Dunn and Knight [DUNN91] summarized a case study of the problems that limit the reuse of code in an industrial setting. They concluded that a reusable library of classes written in C++ was very successful in addressing this problem. The use of object-oriented methods in computer graphics has been compared to traditional methods by Fellner [FELL91]. This study showed that object-oriented techniques significantly improve the readability of the algorithms and drastically improve productivity. The learning curve for a large class library is substantial but the investment is usually recovered quickly. Moreover, the cost of integration and testing is reduced, leading to a more effective management of the efforts of several developers. Some of the benefits of using object-oriented techniques in computer graphics have been illustrated through examples [CUNN92]. Wisskirchen states that two separate groups of researchers and developers are working in computer

graphics [WISS90]. One group concentrates on traditional graphics approaches and graphics standards (PHIGS and GKS). The other group is striving towards object-oriented systems. Although these groups are slowly coming together, object-oriented graphics systems are currently not available. Some systems which claim to be object oriented have not been created from an object-oriented design.

The development of a class library with the object-oriented paradigm will impose an engineering discipline that will allow the software to be built from reusable, interchangeable, and extensible parts. This will allow for a graphics program to be put together by just assembling the desired parts and putting them together [CUNN92]. Since all of the parts will be modular, any future applications program can use the existing parts, or make modifications or extensions to them to adapt them to a particular application with minimum effort.

3.0 Graphs in ACSYNT - A Case Study

In 1990 a group of eight U.S. aerospace companies together with several NASA and Navy centers, led by NASA Ames Research Center and Virginia Tech's CAD Laboratory, agreed, through the assistance of American Technology Initiative, to form the ACSYNT Institute [JAYA92]. This Institute is supported by a Joint Sponsored Research Agreement to continue the research and development in computer aided conceptual design of aircraft initiated by NASA Ames Research Center and Virginia Tech's CAD Laboratory. The result of this collaboration is a feature-based, parametric, computer aided aircraft conceptual design system called ACSYNT (AirCraft SYNThesis) [WAMP88a] [WAMP88b] [JAYA92]. This code is based on analysis routines begun at NASA Ames in the early 1970's. ACSYNT's CAD system is based entirely on PHIGS and includes a highly interactive graphical user interface, automatically generated surface models and shaded image displays.

The ACSYNT analysis discipline modules generate a large amount of output data for the aircraft designer. The disciplines this data covers are trajectory, aerodynamics, cycle, economics, weights, and noise. The data for these different disciplines can be displayed singularly or can be multi-disciplinary. Interactive methods have been created to allow the designer to quickly display graphs based on these data. The types of graphs currently

supported by ACSYNT includes two-dimensional line graphs, two-dimensional bar graphs, polar plots and carpet plots. An example of a graph in ACSYNT is shown in Figure 1.

3.1 Current Methods Used in ACSYNT

At the present time, the ACSYNT graphing functions are located in ten different files. The files, their function and the number of lines of code in each are listed in Figure 2. These lines of code deal only with the display of the graphs on the screen and for graph modification. They do not include the routines that generate the data for the graphs.

All of the graphs in ACSYNT are generated using similar procedures. This procedure is outlined below and illustrated in Figure 3.

1. The PHIGS structure organization is set up to accommodate the graph. This includes determining the number of structures necessary to create the graph, opening a root structure and executing all other structures from within this root structure. Then all the default attributes are inserted in all the structures.
2. The axes of the graph are then drawn. These are the axes lines themselves without labels. They are drawn at a fixed location to a fixed length. The grid for the graph is also drawn at this time. It is also drawn to a fixed set of parameters and attribute values.

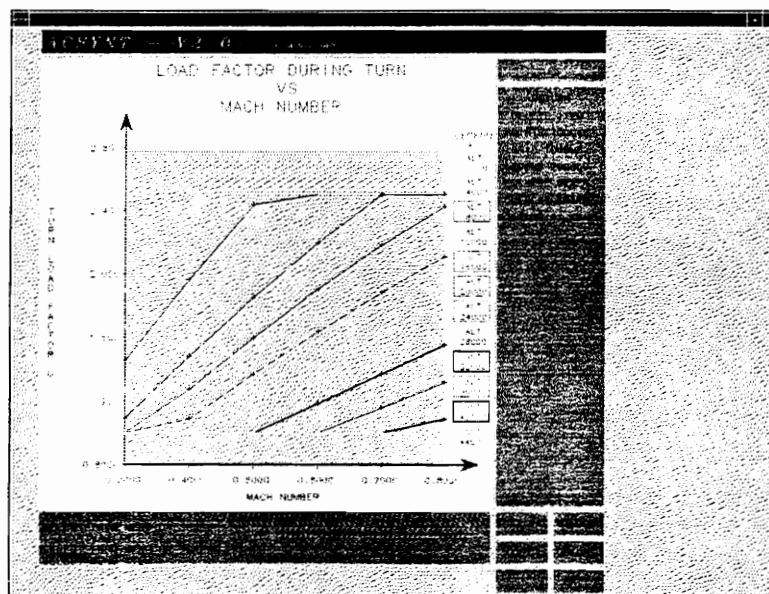


Figure 1. Example of Graph in ACSYNT

Filename	Description	# Lines
agrph1.vpi	Trajectory Graphs	4994
agrph2.vpi	Economics Graphs	4925
agrph3.vpi	Aerodynamics Graphs Cycle Graphs	3785
agrph4.vpi	VN Diagrams Boom Graphs Flexibility Routines For Graphs Overlay Routines	6931
agrph5.vpi	Envelope Graphs	2171
airgrph.vpi	Infrared (Polar) Graphs	1317
aghcp.vpi	Carpet Plots	5754
pstcar.vpi	Postscript Printing For Carpet Plots	412
middleman.c	Translator between C and FORTRAN routines	528
graphing.c	Graphing routines in C	2237
Total		33054

Figure 2. Present ACSYNT Graphing Routines

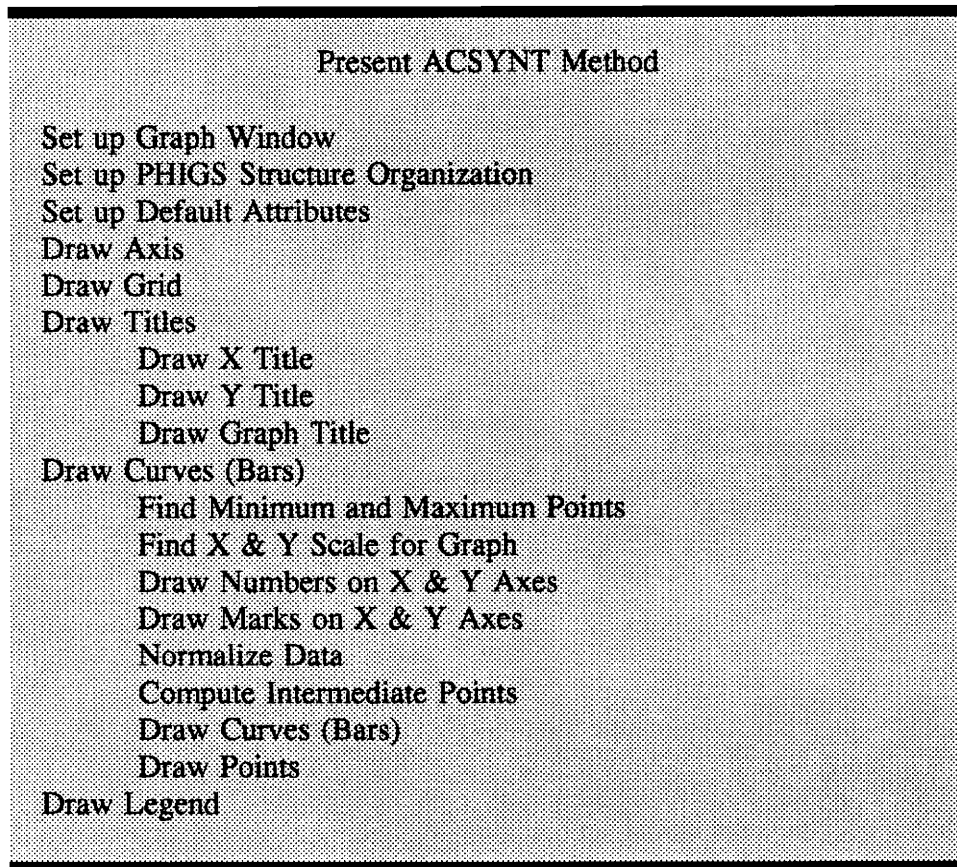


Figure 3. Procedure for Creating Graphs in ACSYNT

3. The titles of the graph, and the x and y axes labels are then drawn with fixed attributes.
4. The minimum and maximum data values of the data to be graphed are determined.
5. The proper scale to be used for the axes is then calculated. The user is not provided with methods to change this scale.
6. The marks and labels for the x and y axes are then drawn onto the axes.
7. The points for the curves or bars to be drawn are then normalized according to the calculated axes range and the curves or bars are then drawn onto the graph.
8. Finally, the legend of the graph is drawn.

All the different types of graphs for the different modules have their own independent routines to perform all of these functions. Therefore, if an attribute change is desired in all graph types in must be changed in each type of graph separately instead of in one central location. The only common functions shared by the different graph routines are those that determine the scale of the axes, normalize the data, and interpolate the data points to obtain intermediate points for drawing the curves.

The ACSYNT graphing modules are implemented using about 33,000 lines of FORTRAN and C code. If a new type of graph is to be added to ACSYNT or if a new analysis module requires graphs, code will need to be written following the procedure shown in Figure 3. This would require considerable knowledge of PHIGS, and can turn out to be a time-consuming process involving hundreds or thousands of lines of new code.

3.2 Flexibility of Graphs

The graphing functions also allow the user to apply minor modifications to improve readability and visualization of the graphs. The legend, grid, and points on the curves can be toggled on and off. Curves can be removed from the graph and the graph can be enlarged to take up the entire viewport. However, no other changes are available to the user. This is a severe limitation to a user desiring to use the graphs for a technical report or presentation. Also, the utility of the graph is reduced by the fact that the scale and range cannot be modified.

4.0 Thesis Objectives

The overall objective of the work presented in this thesis is to create a high-level tool which would allow applications programmers to quickly and easily create customized graphing functions to support computer aided design. This would save a significant amount of money and time being spent on re-creation of tools for many commonly used graphs for CAD (e.g., line graphs, surface plots, bar charts, etc.). The resulting set of tools should also meet the criteria listed below.

1. Portability and compatibility with PHIGS: PHIGS is fast gaining popularity as the graphics system chosen by CAD applications programmers. The use of PHIGS in this set of tools will ensure portability and device independence.
2. Modern look and feel: Users of graphical systems are usually familiar with the look and feel of the graphical user interfaces provided by windowing systems (MS-Windows, X-windows, Motif-based GUI's, etc.). Any graphical interface which is provided to the end user by this graphing tool should have a similar look and feel.
3. Ease of use: The toolkit should be easy to use. The CAD applications programmer should be able to create graphing methods easily and without

spending a lot of time learning the details of the tools. The tools should be simple enough for engineers to use with their programs without having to learn to use PHIGS.

4. Support CAD needs: During the design of the toolkit, the needs of CAD applications programmers should be considered. The graphs used commonly by CAD systems should be implemented such that CAD programmers should be able to easily integrate it with existing or new CAD systems.
5. Object-Oriented: In keeping with the latest trends and technology related to CAD software creation, this toolkit should be completely object oriented. This will allow programmers to include graphs at a very high level without requiring them to understand the details of the implementation of these tools. Moreover, a truly object-oriented set of tools will automatically lead to a reliable and maintainable system which can be extended or modified easily in the future.

5.0 Object-Oriented Design

5.1 *Traditional Programming Philosophy*

In general, computer languages deal with two items, data and algorithms. The data is the information to be used or calculated and the algorithms are the processes to manipulate the data.

The traditional approach is to define the procedure for a process and then use the computer to execute this procedure. Top-down design became the dominant method of design. In this method a large program is divided into several smaller tasks. If one of these smaller tasks is still too large it was further divided into even smaller tasks. This process was continued until all the tasks were of manageable size and easily programmed.

While this method allowed for the breakdown of complicated tasks and improved the clarity, reliability, and the ease of maintenance, it is easily seen how the code becomes difficult to maintain and modify as changes or additions are required, especially if they occur near the top of the program as this affects all the routines underneath them. Object-oriented programming solves this problem.

5.2 Object-Oriented Philosophy

Object-oriented programming emphasizes the data and what operations can be performed on it. The object-oriented programming approach tries to tie the language to the problem, not the problem to the language. "The idea is to design a data form that corresponds to the essential features of a problem. In C++, a *class* is a specification describing such a new data form, and an *object* is a particular data structure constructed according to that plan" [PRAT91]. The class specifies what data are associated with an object and what operations can be performed on that object.

In this way object-oriented programming works more like the actual human thought process. A personnel program, for example, would define classes to represent the personnel employee information, payroll information, etc. The class definitions for these would determine the permissible operations for each item. Then a program is designed using objects of these classes. The object in this case would be one particular employee. The method of going from classes to program design is known as bottom-up programming. This method allows the user to easily add an additional class as changes are needed or if something was overlooked. This cannot be done easily in the top-down approach.

5.3 *Object-Oriented Design*

Object-oriented design cannot be simply explained as top-down or bottom-up programming. It is best explained by the Round Trip Gestalt design. This design emphasizes the incremental and iterative development of the system through the entire system. First the system is designed as well as possible with the information available. Then the system is examined and refined again and again in an iterative manner until the system meets the design requirements.

The order of events which needs to take place for an object-oriented design is as follows:

- Identify all the classes and objects.
- Identify the significance and meaning of all the classes and objects.
- Identify the relationships among the classes and objects.
- Implement the classes and objects.

This process should be repeated until there are no new abstractions, or when already discovered classes and objects may be implemented by composing them from existing reusable components [BOOC91].

5.4 Object-Oriented Programming Terms (Using C++)

5.4.1 Class

A *class* is the C++ means of combining data representation and methods for manipulating that data into one format. Put more simply, a *class* is a classic C structure that contains in addition to variables, functions as members. At a minimum, a class is only a C structure.

5.4.2 Polymorphism (Function Overloading)

Polymorphism refers to the act of using the same name to accomplish different purposes. In C++ *polymorphism*, or function overloading, allows the user to create different functions having the same name provided they have different argument lists.

5.4.3 Inheritance

Inheritance is the concept of building a new class from classes that have already been created [ADAM92]. Using *inheritance*, code duplication can be avoided. By inheriting from a base class, a derived class assumes all the properties, both data members and member functions, of the base class in addition to the new ones it defines. Classes can be derived from numerous other classes creating a class hierarchy.

As an example, if you want to add something to an already working class you can derive a new class with all the properties of the existing working class. Then you can add code to the new class. And because the former class still exists, it can be used by other parts of your program with no side effects or extra work.

5.4.4 *Encapsulation*

"*Encapsulation* refers to the bundling together of a group of data and the functions that operate on the data" [ADAM92]. "Objects are quite frequently spoken of as encapsulations of abstractions as each object within itself contains both the knowledge of how to respond to a set of messages and the values for its internal states (as defined by the class description for an object of its class). In addition, the internals of an object (methods and state variables) are invisible to and unalterable by other objects" [FELL91].

"Because you can treat the data and the functions as an object, you do not have to concern yourself with detail" [ADAM92]. Encapsulation keeps some of the complexity of the program away from the user. This allows the user to be able to concentrate and plan on a higher level and in broader terms than was previously possible. It also reduces misuse or incorrect manipulation of data by the user.

5.5 Advantages of Object Oriented Programming

At first the use of object-oriented programming techniques will require more time and effort to learn the design philosophy and apply it correctly. But as soon as this learning period is completed, the modification of classes becomes much easier and more efficient than writing code in standard programming languages [FELL91].

Some of the many advantages of using object-oriented programming are [FELL91]:

- Given a rich set of classes, rapid prototyping is both possible and useful.
- The use of object-oriented programming makes it possible to effectively manage and integrate the efforts of many developers.

- The cost of integration and testing is small.
- The learning curve for a large class library is substantial. Nevertheless, the time is well spent and the investment is usually recovered during the next application developed.

Since object-oriented programming is an entirely new philosophy it is important to realize that software programmers and developers will not be able to simply make an immediate switch to object-oriented programming by learning a specific object-oriented language. They will require additional training on the object-oriented philosophy itself [FELL91].

"The limiting factor is that no design techniques exist that are specifically intended to take advantage of a set of reusable components" [DUNN91]. This is sure to be corrected in the near future as programmers and developers become more experienced in the new philosophy. This will further benefit those using and making the transition to object-oriented programming.

An argument against making the switch to object-oriented programming is that old software is not reusable in the object-oriented programming environment. This is not the case. Even if a particular component cannot be reused exactly as it is, the analysis

embodied in the component often provides the means with which to redesign it to fit the new application [DUNN91]. A study conducted in an industrial environment by Dunn and Knight in cooperation with Sperry Marine Incorporated showed a very high level of code reuse as well as the fact that old software can be reused [DUNN91].

Finally it is good to remember that "Good object-oriented design is still an art rather than a science. Object-oriented methodology works as an amplifier for programming skills: the good programmers get better and the weak ones fall by the wayside" [FELL91].

6.0 Motif-Like Interface

Motif is the most widely-used interface tool for workstations. However, CAD programmers have found the need to resort to other interface methods because of several reasons, some of which are:

- Motif is limited to two-dimensional graphics
- Other device-independent graphics tools (PHIGS) support three-dimensional graphics
- PHIGS-input and Motif-input models are incompatible
- Motif is not truly object oriented
- Adding a Motif interface to an existing CAD program is difficult
- Motif cannot be easily expanded by the user

Because of the reasons presented above, the author found it necessary to use a PHIGS-based interface system which is truly object oriented and provides a "Motif-Like" look and feel. One such system has been designed and implemented by Woyak and Myklebust [WOYA92, WOYA93]. Some of the tools provided by this interface system were used as the basic building blocks for the work presented in this thesis. For the sake of completeness, an overview of this class library of object-oriented interface tools is presented below.

Five major groups of classes are included in this interface framework: Windows, Interface Managers, Menu Managers, Menu Items, and Menu Item Managers. A Window is a channel of communication between the user and the applications program. The Interface Manager is the central processing object. A Menu Manager maintains a group of Menu Items which may also be controlled by a Menu Item Manager. Several different types of windows have been defined in this framework. Geometry Managers are windows that display and maintain a PHIGS view. Pop-Up Menus are windows used to display menu items and Dialogue Managers are windows used to send messages to a user. The powerful C++ features of inheritance and virtual functions allow all these windows to inherit the properties of a base "Window" class. The Interface Manager maintains a list of windows associated with it and manages these windows without regard to what "type" of window they are. Thus, any new class which inherits the Window class can automatically be managed by the Interface Manager without any changes to this interface. Figure 4 shows an example of the class organization of this framework.

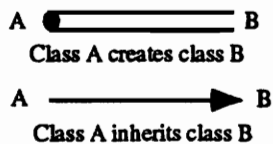
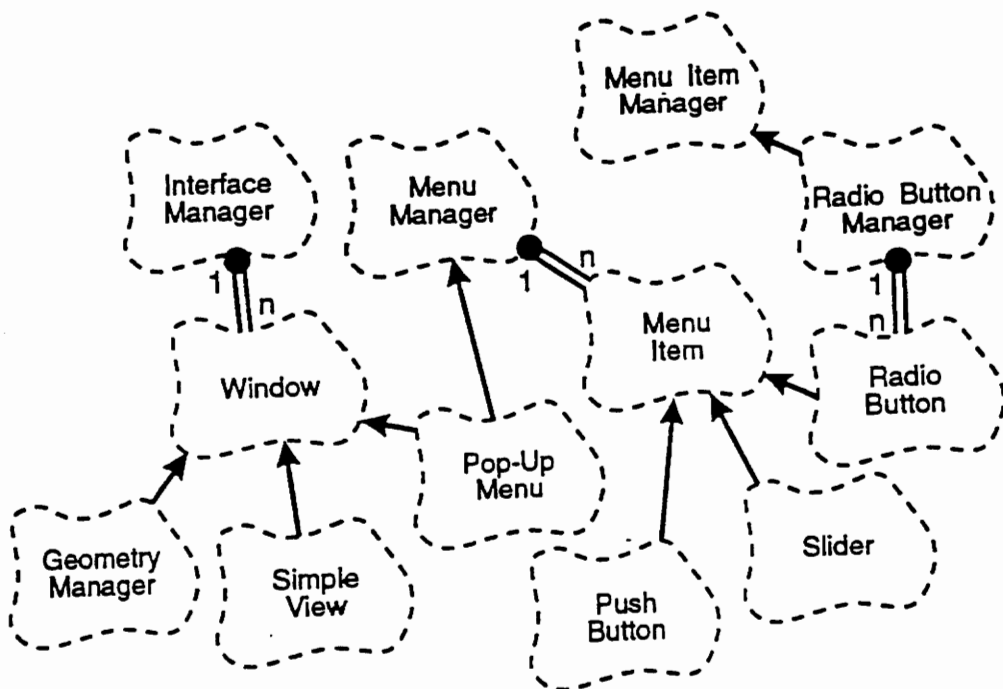


Figure 4. Class Organization of Motif-Like Interface Framework [WOYA93]

7.0 Overview of the Design

7.1 *Class Hierarchy*

The first design of the class hierarchy can be seen in Figure 5. It involves using a Geometry Manager class, and creating a set of graph entity classes (e.g., Axes, Legends, etc.). A Graph class would inherit all of these separate classes. This Graph class would then be inherited by the individual graph classes (e.g. Line, Bar, etc.). This gives the individual graph classes access to the basic entities they may need to produce the desired graph.

This structure layout was rejected for several reasons. Inheritance represents an "is a" relationship. Since "Graph" is not a kind of "Axes", "Text", etc., this setup was violating this relationship. Also, this setup is not very flexible. Since the Graph class inherits all the basic entities, the user would have to modify the entity and the graph classes and check for any other effects to the other classes if changes to any of the entities are desired or if any new entities are to be added. Also since the basic graph entities are data items, extra work would be involved in passing data from class to class to determine what should be constructed.

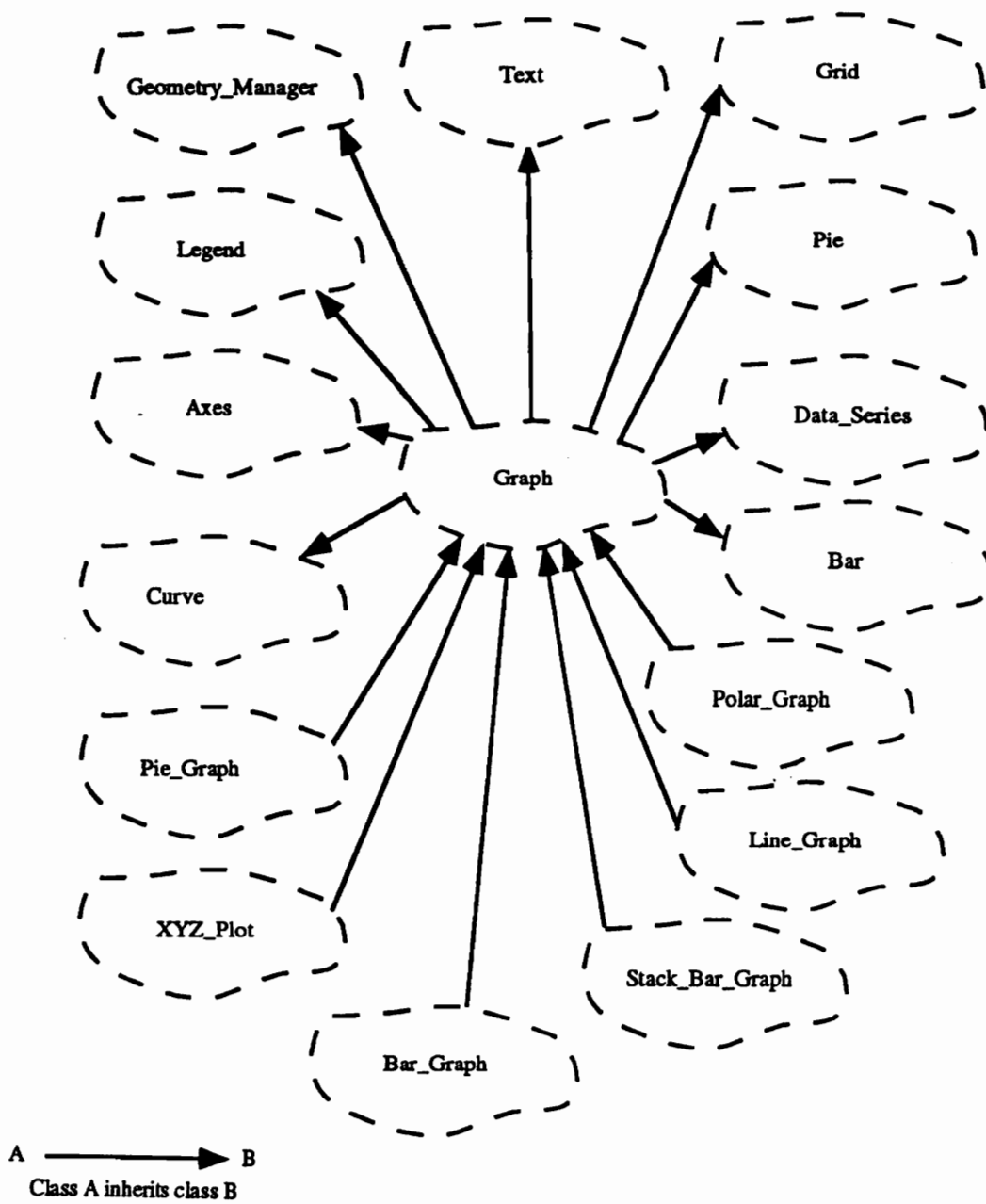


Figure 5. Class Hierarchy - First Design

The modified class hierarchy layout (Figure 6) includes the creation of a Graph Manager which inherits a Geometry Manager. The Graph Manager is inherited by a Graph class. The Graph class is then inherited by the individual graph classes. The basic graph entities are then defined in their own individual classes which can be included as instanced objects in any of the individual graph classes. This setup offered many advantages over the previous design. This setup allows for generic "common" functions to be put where all classes have access to them. It also allows greater flexibility for changing graph entities or adding new entities.

7.2 User Interface For Graph Modification

The user interface tools this system uses are part of the PHIGS-based, Motif-like interface framework described by Woyak and Myklebust [WOYA93]. The Geometry Manager controls the views into which the graphs are placed. This Geometry Manager allows the graph classes to align the viewport in an appropriate position depending on what type of graph is being displayed by sending the appropriate messages to the Geometry Manager.

The Interface Manager also aids in keeping track of PHIGS structure identifiers. By using the tools built into the library, each entity is assured of getting its own unique

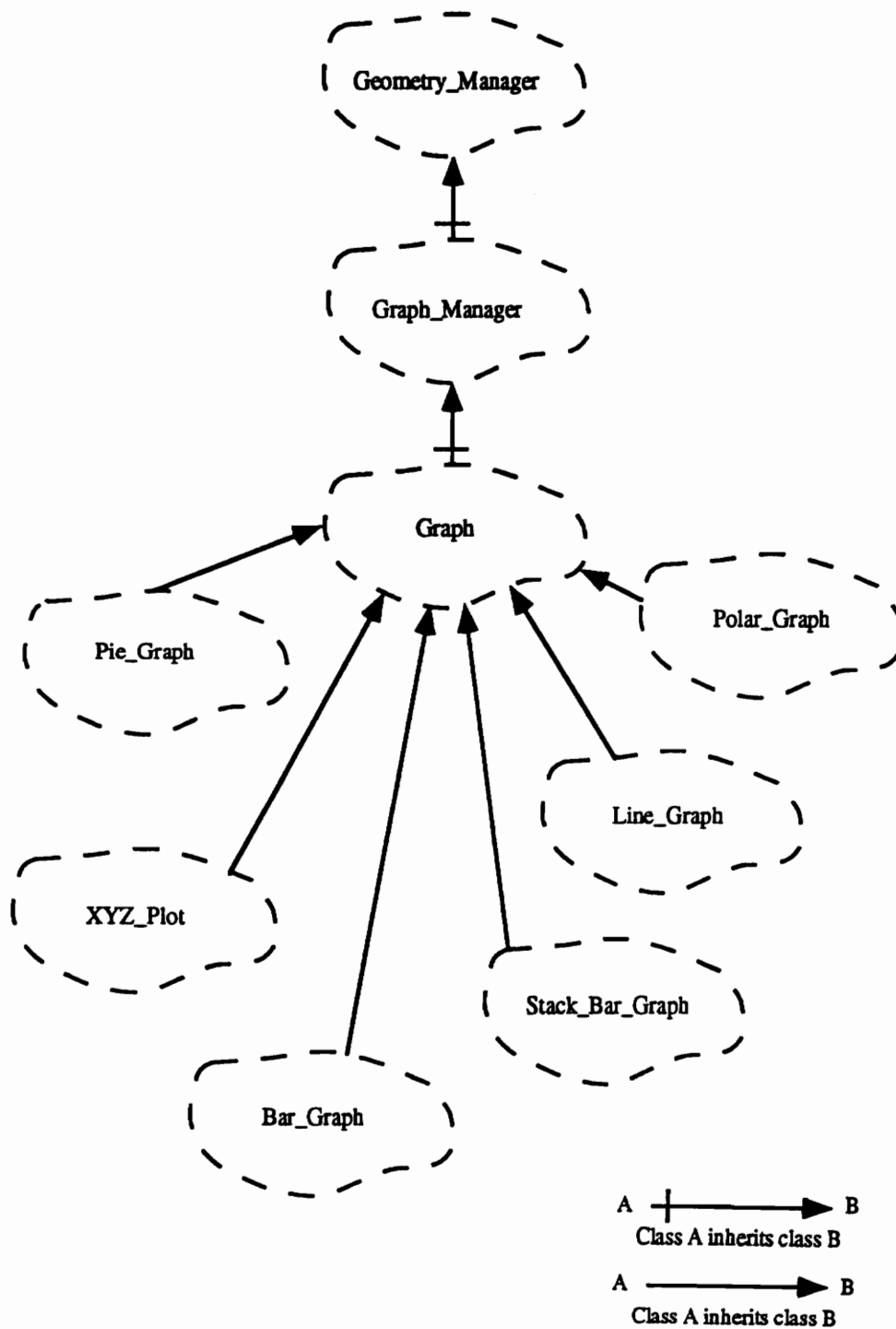


Figure 6. Class Hierarchy

structure identifier. This relieves the programmer of the burden of structure identifier management.

In addition to this all of the individual entity classes in this system make use of the various input/output devices provided for in this Motif-like interface framework, such as pop-up menus, number boxes, radio buttons, etc. The user interfaces for all of the graph entities are encapsulated within the entities themselves. The end user gains access to the interface by picking the entity with the mouse. When an entity is "picked", it uses its own internal methods to create a pop-up menu which will allow the user to make modifications to the attributes of that entity class. If any of the modifications the user makes affect any of the other entity classes, these classes will automatically modify themselves to account for the change.

8.0 Descriptions of Classes

Several classes have been defined for inclusion in this library to support two-dimensional and three-dimensional graphs. All the graphs except for the polar graphs have the ability to be displayed in two or three dimensions. All graphs are drawn in two dimensions in the XY plane, unless the user specifies otherwise.

The graphs drawn in three-dimensional are drawn with the positive Z axis coming out of the screen towards the user. In bar graphs, the cubic dimension of the bar is along the Z axes. In pie graphs, the pie is drawn in the XZ plane and the cubic height dimension is drawn along the Y axes. In line graphs, the points are drawn in three-dimensional space. The GUI framework automatically provides the end user with tools to apply three-dimensional viewing transformations to the window.

8.1 *Graph Manager Class*

The Graph Manager class inherits the Geometry Manager class provided as a part of the Motif-like interface described earlier. This class is the base class for this object-oriented toolkit from which most graph classes are derived. This is the "gateway" class between the class library being described in this thesis and the interface framework described by

Woyak and Myklebust. It inherits all the methods and data from the Geometry Manager class and is hence a "Window" class.

8.2 Graph Class

The Graph class is derived from the Graph Manager class. This class forms the base from which each individual graph class is derived. It contains methods for gaining access to graph entities displayed in a graph window.

8.3 Individual Graph Classes

Each type of graph supported by this system is a separate class. For example, the Line_Graph class allows programmers to create a line graph using PHIGS. This is done by creating an instance of the Line_Graph. Since the Line_Graph class is derived ultimately from the base "Window" class, a Line_Graph object is a "Window" object. Thus, creating a Line_Graph object automatically creates a new window with all related functions (resizing, raising, lowering, etc.) which are provided for in the Interface Manager. A number of functions are redefined by the Line-Graph class to tailor this class

to the needs of a Line_Graph. This class inherits methods from the Graph Manager for processing user input. The types (classes) of graphs defined in this class library include:

- Line_Graph
- Bar_Graph
- Stacked_Bar_Graph
- XYZ_Plot
- Pie_Graph
- Polar_Graph

Each class includes a number of public (available for use by any other function or class) functions which the programmer can use to modify the graph. The common functions are inherited from the Graph class and specific ones are inherited and redefined or defined within the class. Each class creates instances of graph entities from various graph entity classes (described later).

In the following sections a detailed description of the design of the classes is provided. This description includes the functions to be included in each class. Some of these functions are private (i.e. available for use only by the class) and some are public.

All constructors in the following classes create the entities with default attributes, but these can be changed by the programmer using the public functions.

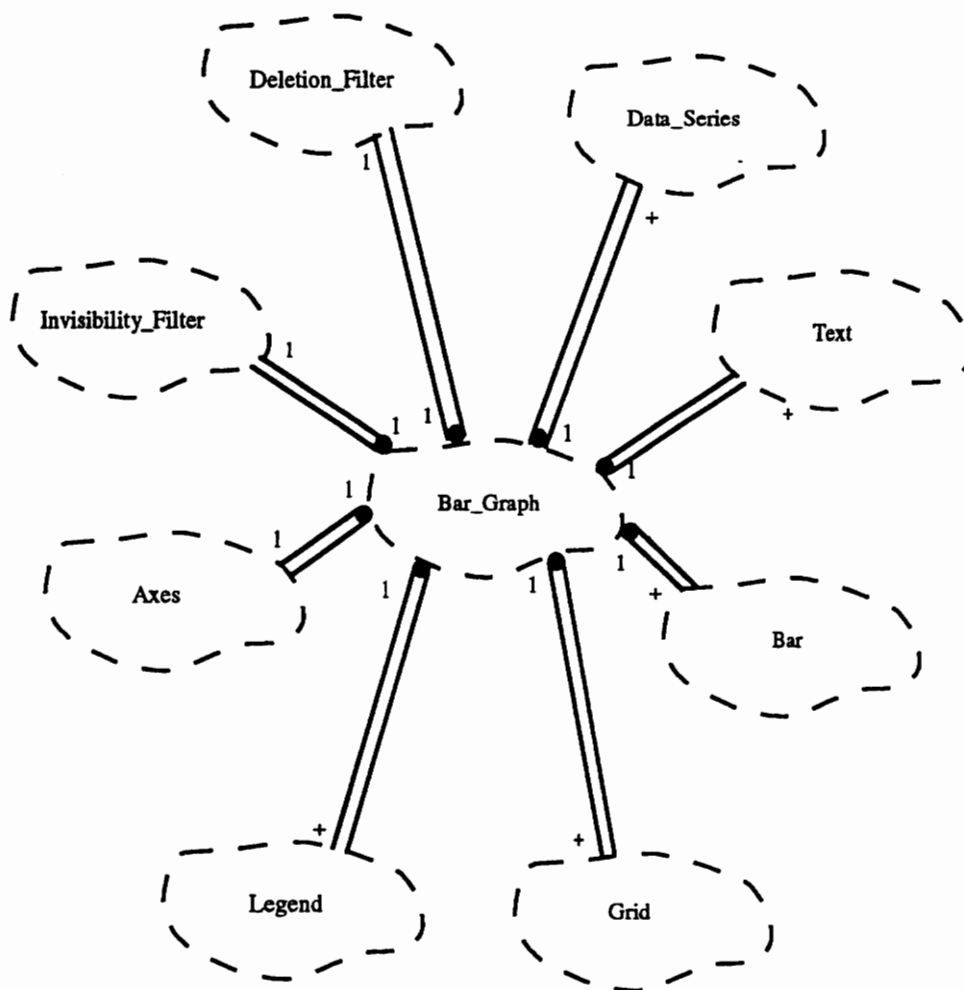
8.3.1 Bar_Graph Class

The Bar Graph class is used to generate an instance of a bar graph. Bar graphs are used for showing volume and simple time comparisons. Each bar in a bar graph represents one data value. An example of the type of data that requires a bar chart is comparing the first-unit costs of several different items. The bars can be displayed two or three dimensionally. Instances of the Axes, Grid, Data Series, Invisibility Filter, Deletion Filter, Bar, Text and Legend classes will automatically be generated and displayed by the Bar Graph class (Figure 7). A list of the functions included in the Bar Graph class is provided below.

Constructor:

Bar_Graph (two-dimensional or three-dimensional, number rows in data array, number columns in data array, data array, text array)

The constructor performs these operations:



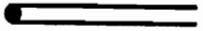
A  B
Class A creates class B

Figure 7. Graph Entities Used For Bar Graphs

- Sets any default values needed by the graph, either by user supplied input or default values.
- Copies data arrays into data series entities.
- Creates axes entity with necessary pointers and values.
- Creates grid entity with necessary pointers and values.
- Creates sets of bar and legend entities.

Public Functions:

`process_geometry_view ()`: Process input from the geometry manager and pass it on to the appropriate graph entity. This function is inherited by the interface manager and redefined by this class.

`manage ()`: Control drawing of bar graph. First the viewport is rotated for correct viewing of the graph. Then the axes and grid entities are managed. Then the bars, titles, and legend are drawn. This function is inherited from the interface manager and redefined by this class.

`draw_bars ()`: Draw the bars for the graph. Attributes for the bars are set and the bars are drawn.

`draw_titles ()`: Draw the titles for the graph. Attributes for the title and axes labels are set and the title and labels are drawn.

`draw_legend ()`: Draw the legend for the graph. Attributes for the legend are set and the legend is drawn.

`recreate_graph ()`: Recreate the graph. Tells all the graph entities to recreate themselves.

8.3.2 *Line_Graph Class*

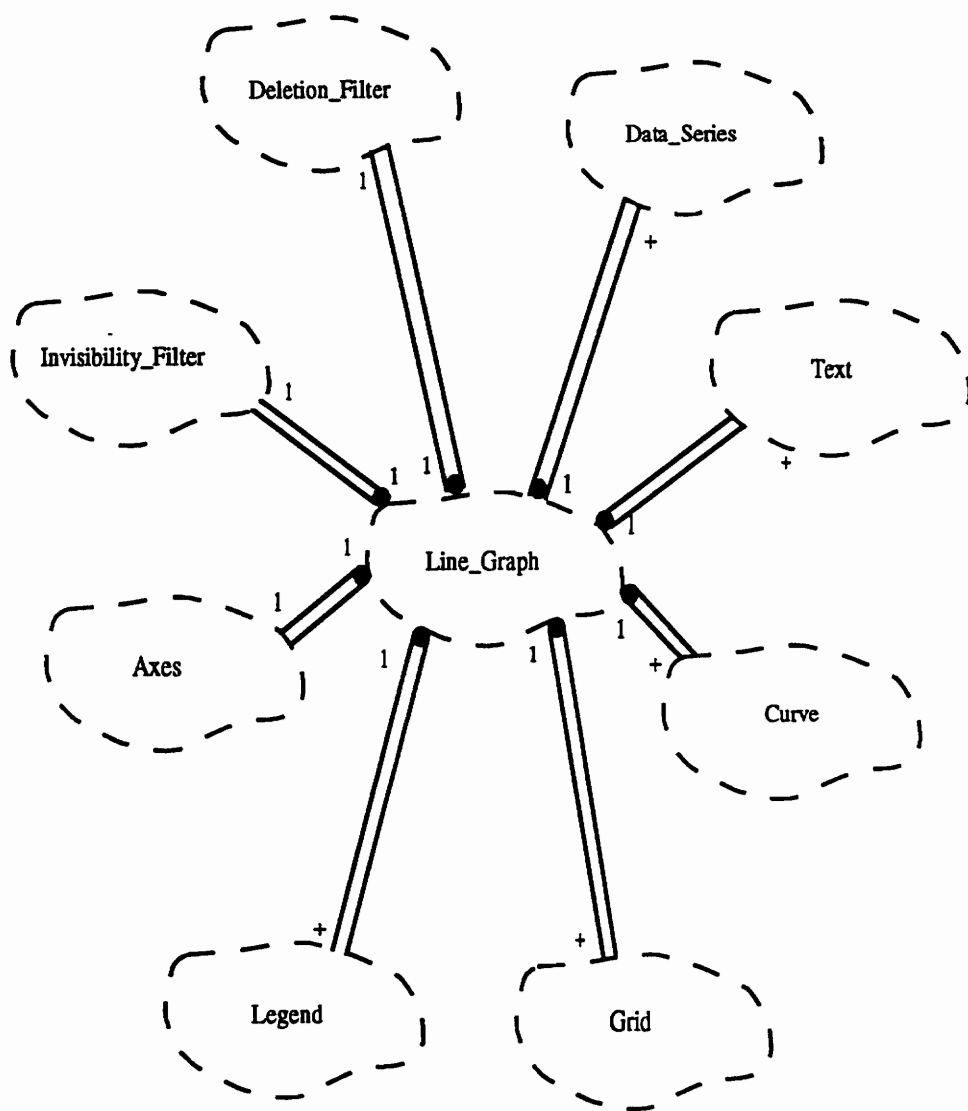
The Line Graph class generates an instance of a line graph. Line graphs show the amount of change between one data point and the next. This allows the user to easily identify data fluctuations. An example of the type of data that requires a line graph is the cashflow spent on a project per year. The graph can be displayed two or three dimensionally. Instances of the Axes, Grid, Data Series, Invisibility Filter, Deletion Filter, Curve, Text and Legend classes will automatically be generated by the Line Graph class (Figure 8). A list of the functions included in the Line Graph class is provided below.

Constructor:

`Line_Graph (number of rows in data arrays, number of columns in data arrays, x data array, y data array, text array)`

`Line_Graph (number of rows in data arrays, number of columns in data arrays, x data array, y data array, z data array, text array)`

The constructor performs these operations:




A  B
Class A creates class B

Figure 8. Graph Entities Used For Line Graphs

- Sets any default values needed by the graph, either by user supplied input or default values.
- Copies data arrays into data series entities.
- Creates axes entity with necessary pointers and values.
- Creates grid entities with necessary pointers and values.
- Creates sets of curve and legend entities.

Public Functions:

`process_geometry_view ()`: Process input from the geometry manager and pass it on to the appropriate graph entity. This function is inherited by the interface manager and redefined by this class.

`manage ()`: Control drawing of line graph. First the viewport is rotated for correct viewing of the graph. Then the axes and grid entities are managed. The curves, titles, and legend are drawn. This function is inherited by the interface manager and redefined by this class.

`draw_curves ()`: Draw the curves for the graph. Attributes for the curves are set and the curves are drawn.

`draw_titles ()`: Draw the titles for the graph. Attributes for the title and axes labels are set and the title and labels are drawn.

`draw_legend ()`: Draw the legend for the graph. Attributes for the legend are set and the legend is drawn.

recreate_graph (): Recreate the graph. Tells all the graph entities to recreate themselves.

8.3.3 *Pie_Graph Class*

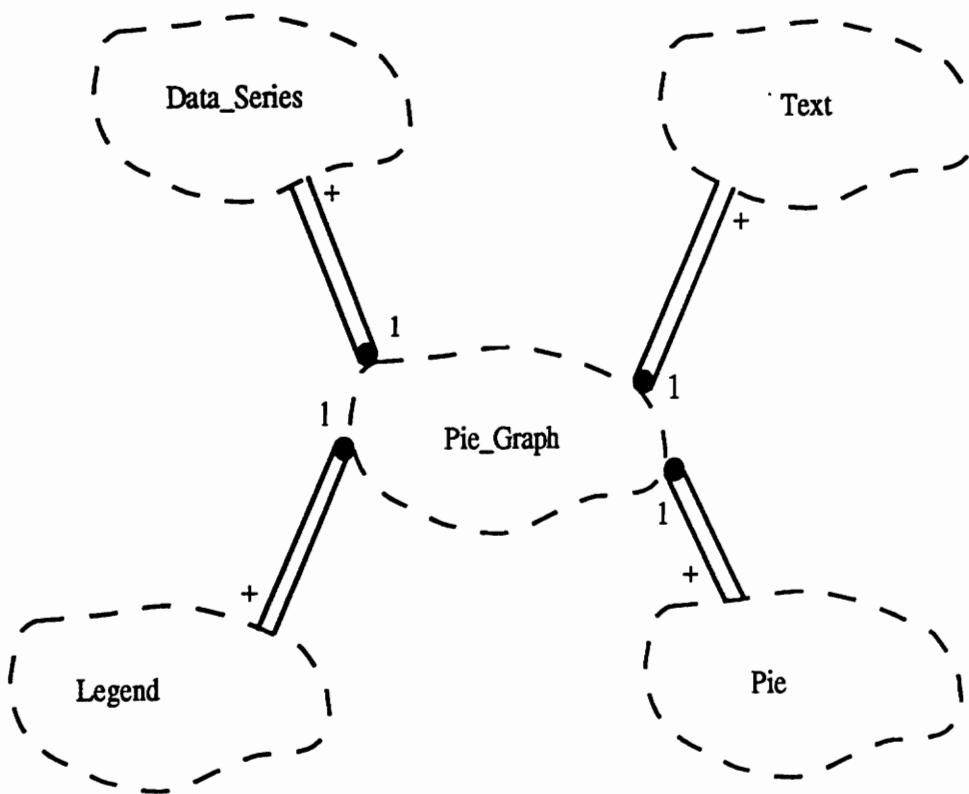
The Pie Graph class generates an instance of a pie graph. Pie graphs are used to show the percentages of the whole. An example of the type of data that requires a pie graph is the weight breakdown of an aircraft. The user can quickly determine the individual contributions of the components toward the total weight of the aircraft. The graph can be displayed two or three dimensionally. Instances of the Data Series, Pie, Text and Legend classes will automatically be generated by the Pie Graph class (Figure 9). A list of the functions included in the Pie Graph class is provided below.

Constructor:

Pie_Graph (two-dimensional or three-dimensional, number of rows in data array, number of columns in data array, data array, text array)

The constructor performs these operations:

- Sets any default values needed by the graph, either by user supplied input or default values.



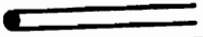
A  B
Class A creates class B

Figure 9. Graph Entities Used For Pie Graphs

- Copies data arrays into data series entities.
- Creates sets of pie and legend entities.

Public Functions:

`process_geometry_view ()`: Process input from the geometry manager and pass it on to the appropriate graph entity. This function is inherited from the interface manager and redefined by this class.

`manage ()`: Control drawing of pie graph. First the viewport is rotated for correct viewing of the graph. Then the axes and grid entities are managed. Then the pie slices, titles, and legend are drawn. This function is inherited from the interface manager and redefined by this class.

`draw_pie_slices ()`: Draw the pie slices for the graph. Attributes for the pie slices are set and pie slices are drawn.

`draw_titles ()`: Draw the titles for the graph. Attributes for the title are set and the title is drawn.

`draw_legend ()`: Draw the legend for the graph. Attributes for the legend are set and the legend is drawn.

`recreate_graph ()`: Recreate the graph. Tells all the graph entities to recreate themselves.

8.3.4 *Polar_Graph Class*

The Polar Graph class generates an instance of a polar graph. Polar graphs are used for showing the differences of magnitude and degree from point to point. The type of data that requires a polar graph is the radar or infrared signature of an aircraft. Instances of the Polar Axes, Data Series, Invisibility Filter, Deletion Filter, Curve, Text and Legend classes will automatically be generated by the Polar Graph class (Figure 10). A list of the functions included in the Polar Graph class is provided below.

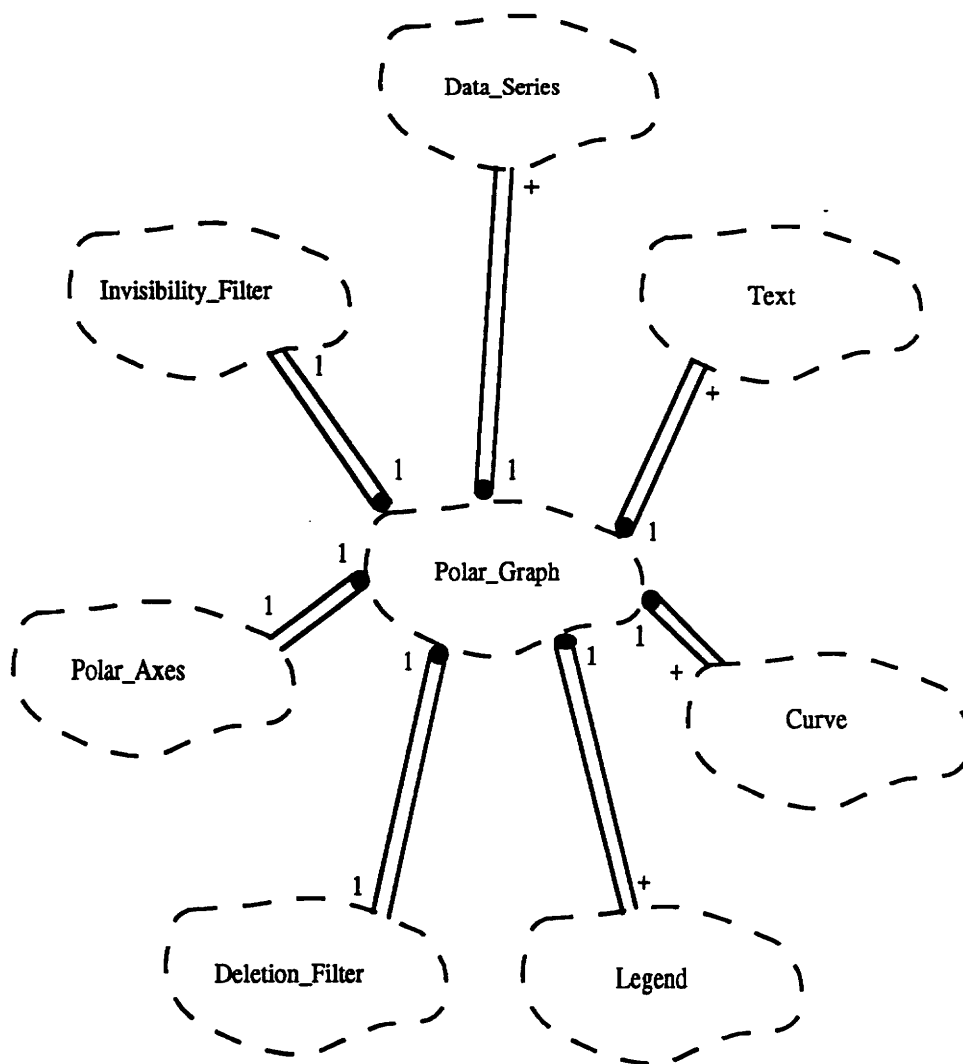
Constructor:

Polar_Graph (number of rows in data arrays, number of columns in data arrays, magnitude data array, degree data array, text array)

The constructor performs these operations:

- Sets any default values needed by the graph, either by user supplied input or default values.
- Copies data arrays into data series entities.
- Creates polar axes entity with necessary pointers and values.
- Creates sets of curve and legend entities.

Public Functions:




A  B
Class A creates class B

Figure 10. Graph Entities Used For Polar Graphs

`process_geometry_view ()`: Process input from the geometry window and pass it on to the appropriate graph entity. This function is inherited from the interface manager and redefined by this class.

`manage ()`: Control drawing of polar graph. First the viewport is rotated for correct viewing of the graph. Then the polar axes is managed. Then the curves, title, and legend are drawn. This function is inherited from the interface manager and redefined by this class.

`draw_curves ()`: Draw the curves for the graph. Attributes for the curves are set and the curves are drawn.

`draw_titles ()`: Draw the titles for the graph. Attributes for the title and axes labels are set and the title and labels are drawn.

`draw_legend ()`: Draw the legend for the graph. Attributes for the legend are set and the legend is drawn.

`recreate_graph ()`: Recreate the graph. Tells the graph entities to recreate themselves.

8.3.5 *Stacked_Bar_Graph Class*

The Stacked Bar Graph class generates an instance of a stacked bar graph. The stacked bar graph is used to show the percentages of the whole. The type of data that requires

a stacked bar graph would be the first unit cost of an aircraft. The user can quickly determine what percentage each component costs as it relates to the total price of the aircraft. The bars can be displayed two or three dimensionally. Instances of the Axes, Grid, Data Series, Invisibility Filter, Deletion Filter, Stack Bar, Text and Legend classes will automatically be generated by the Stacked Bar Graph class (Figure 11). A list of the functions included in the Stacked Bar Graph class is provided below.

Constructor:

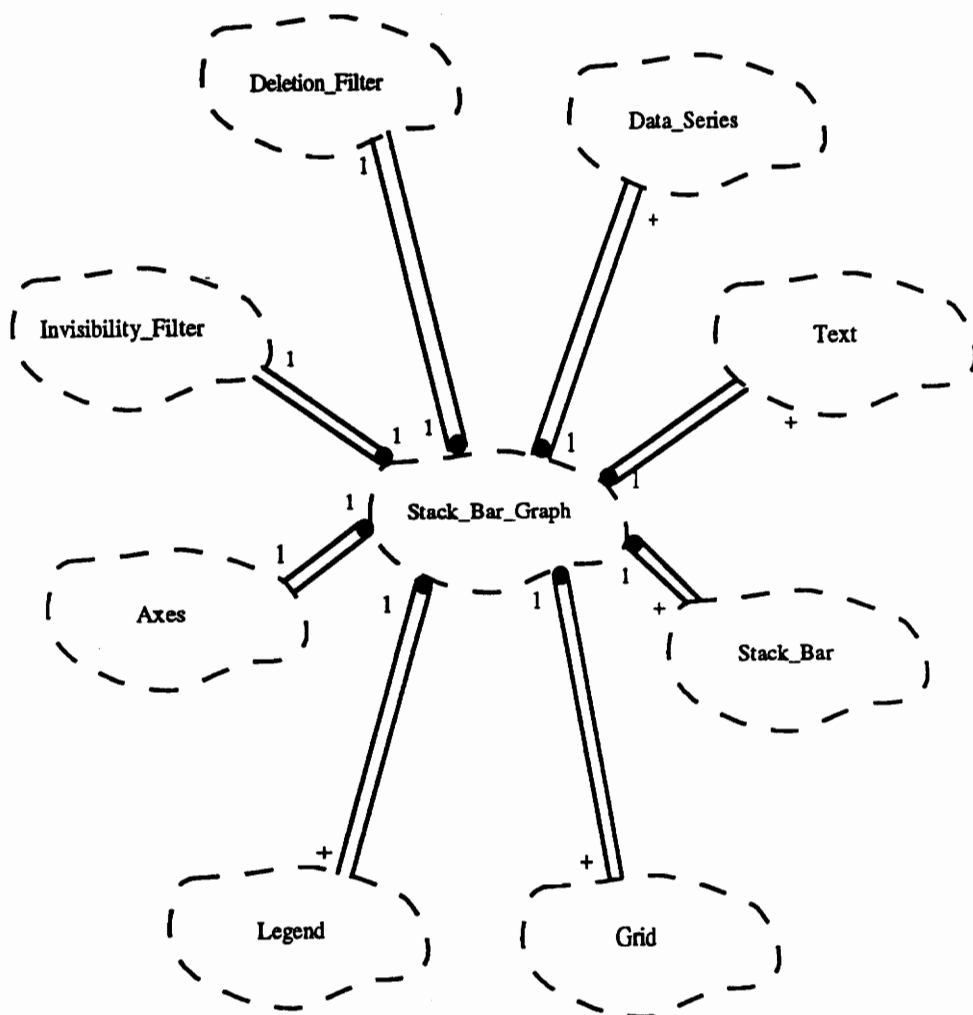
Stack_Bar_Graph (two-dimensional or three-dimensional, number rows in data array, number columns in data array, data array, text array)

The constructor performs these operations:

- Sets any default values needed by the graph, either by user supplied input or default values.
- Copies data arrays into data series entities.
- Creates axes entity with necessary pointers and values.
- Creates grid entity with necessary pointers and values.
- Creates sets of stack bar and legend entities.

Public Functions:

process_geometry_view (): Process input from the geometry window and pass it




A  B
Class A creates class B

Figure 11. Graph Entities Used For Stacked Bar Graphs

on to the appropriate graph entity. This function is inherited from the interface manager and redefined by this class.

`manage ()`: Control drawing of stack bar graph. First the viewport is rotated for correct viewing of the graph. Then the axes and grid entities are managed.

Then the stack bars, titles, and legend are drawn. This function is inherited from the interface manager and redefined by this class.

`draw_bars ()`: Draw the stack bars for the graph. Attributes for the stack bars are set and the bars are drawn.

`draw_titles ()`: Draw the titles for the graph. Attributes for the title and axes labels are set and the title and labels are drawn.

`draw_legend ()`: Draw the legend for the graph. Attributes for the legend are set and the legend is drawn.

`recreate_graph ()`: Recreate the graph. Tells all the graph entities to recreate themselves.

8.3.6 XYZ_Plot Class

The XYZ Plot Graph class generates an instance of a xyz plot. Instances of the Axes, Grid, Data Series, Invisibility Filter, Deletion Filter, Curve, Text and Legend classes will

automatically be generated by the XYZ Plot class (Figure 12). A list of the functions included in the XYZ Plot class is provided below.

Constructor:

XYZ_Plot (number of rows in data arrays, number of columns in data arrays, x data array, y data array, z data array, text array)

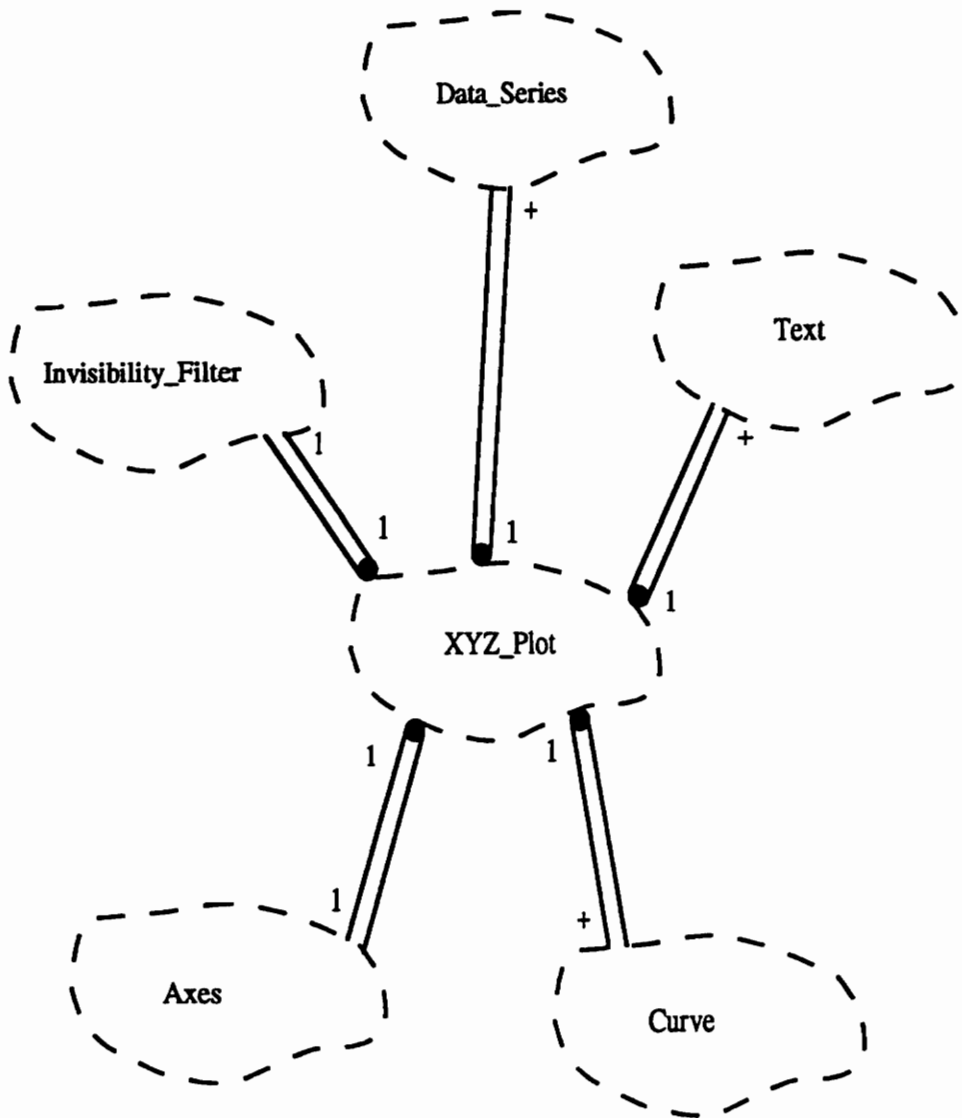
The constructor performs these operations:

- Sets any default values needed by the graph, either by user supplied input or default values.
- Copies data arrays into data series entities.
- Creates axes entity with necessary pointers and values.
- Creates grid entities with necessary pointers and values.
- Creates sets of curve and legend entities.

Public Functions:

process_geometry_view (): Process input from the geometry manager and pass it on to the appropriate graph entity. This function is inherited from the interface manager and redefined by this class.

manage (): Control drawing of xyz plot. First the viewport is rotated for viewing of the graph. Then the axes, titles, and plot are drawn. This




A  B
Class A creates class B

Figure 12. Graph Entities Used For XYZ Plots

function is inherited from the interface manager and redefined by this class.

`draw_plot ()`: Draw the plot for the graph. Attributes for the surface are set and the plot is drawn.

`draw_titles ()`: Draw the titles for the graph. Attributes for the titles are set and the titles are drawn.

`draw_legend ()`: Draw the legend for the graph. Attributes for the legend are set and the legend is drawn.

`recreate_graph ()`: Recreate the graph. Tells all the graph entities to recreate themselves.

9.0 Graph Entity Classes

A graph entity is an object created and displayed by the program. All the basic graph entity classes are defined in a similar manner. Each graph entity is derived from the Graph Manager class and inherits the "manage" function. This function will tell the entity what structure identifier to use, which view index to use, the priority to give to this view, and will pass the pointer to the Interface Manager. The entity also inherits the "unmanage" function which will allow the entity to remove itself.

9.1 *Graph Entities*

Each entity class contains its own private variables. This means that entities cannot accidentally interfere with each other. Each entity has several public functions that allow the entity itself, or other entities, to change the attributes of the entity. All these classes contain a pointer to the Graph class to which it is associated. The graph entities are:

- Axes
- Bar
- Curve
- Grid
- Legend
- Pie

- Polar Axes
- Data Series
- Stack Bar
- Text

There are three common functions among all the entity classes. They perform the same function for all the entity classes. These are:

`manage ()`: Tells the entity what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the PHIGS structure from the view and empties the structure.

`perform ()`: Creates a pop-up menu to allow user to make modifications to the entity attributes. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the entity while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that entity attribute.

9.1.1 Axes Class

The Axes class is used to draw the axis for the graph. The Axes class defaults to a two-dimensional axis in the XY plane. If a three-dimensional axis is specified, it is viewed with the Y axis vertical, and the X and Z axes pointing to the lower-right and lower-left corners of the screen, respectively. The default axes color is red. The axis is scaled automatically according to the data points to be graphed unless the user specifies the starting and ending values for each axis. The axis has six labeled division marks along each axis. The Axes class contains pointers to the Graph, and Invisibility Filter classes. This will allow the axes class to communicate with these classes.

When the Axes constructor is called the axis is assigned default attributes. If the starting and ending values for each axis are not specified by the user, an appropriate scale for each axis will be calculated. The axis is then drawn. Attributes for the axis can be set by the programmer before the axis is drawn by using the functions provided below. The program user can also change the attributes by picking the axis. This will provide the user with a pop-up menu that will allow the axis attributes to be changed without having to modify the code. A list of functions in the Axes class is given below:

Constructors:

Axes ()

Axes (2D or 3D, X-axes-start-pt, Y-axes-start-pt, Z-axes-start-pt, xlength, ylength, zlength)

Axes (2D or 3D, X-axes-start-pt, Y-axes-start-pt, Z-axes-start-pt, xlength, ylength,

zlength, color)

Axes (2D or 3D, X-axes-start-pt, Y-axes-start-pt, Z-axes-start-pt, xlength, ylength, zlength, linewidth)

Axes (2D or 3D, X-axes-start-pt, Y-axes-start-pt, Z-axes-start-pt, xlength, ylength, zlength, linewidth, color)

Axes (2D or 3D, X-axes-start-pt, Y-axes-start-pt, Z-axes-start-pt, xlength, ylength, zlength, number-divisions-x, number-divisions-y, number-divisions-z)

Axes (2D or 3D, X-axes-start-pt, Y-axes-start-pt, Z-axes-start-pt, xlength, ylength, zlength, number-divisions-x, number-divisions-y, number-divisions-z, color)

Axes (2D or 3D, X-axes-start-pt, Y-axes-start-pt, Z-axes-start-pt, xlength, ylength, zlength, number-divisions-x, number-divisions-y, number-divisions-z, linewidth)

Axes (2D or 3D, X-axes-start-pt, Y-axes-start-pt, Z-axes-start-pt, xlength, ylength, zlength, number-divisions-x, number-divisions-y, number-divisions-z, color)

Each constructor takes advantage of the polymorphism (function overloading) capability of object-oriented code. Each constructor shown has a different set of arguments allowing the user to choose which options to set. If an option is not input by the user, the constructor will assign it a default value. For example, if the user chooses the third constructor in the list the color is set. With the fourth constructor the color will be given a default value and the user is now setting the linewidth. All the constructors for the rest of the entity classes work similarly and will not be described in detail.

Destructor:

`~Axes ()`: This function checks if the axis is presently managed. If it is, it will unmanage the axis, disassociate the PHIGS structure from the view and empty the structure.

Inquiry Functions:

Inquiry functions are necessary to allow the other classes to obtain values of data members of this class.

`get_x_axes_start_pt ()`: Returns x axis starting position.

`get_y_axes_start_pt ()`: Returns y axis starting position.

`get_z_axes_start_pt ()`: Returns z axis starting position.

`get_x_axes_length ()`: Returns x axis length.

`get_y_axes_length ()`: Returns y axis length.

`get_z_axes_length ()`: Returns z axis length.

`get_x_axes_hi_boundary ()`: Returns high boundary value of x axis.

`get_y_axes_hi_boundary ()`: Returns high boundary value of y axis.

`get_z_axes_hi_boundary ()`: Returns high boundary value of z axis.

`get_x_axes_low_boundary ()`: Returns low boundary value of x axis.

`get_y_axes_low_boundary ()`: Returns low boundary value of y axis.

`get_z_axes_low_boundary ()`: Returns low boundary value of z axis.

Public Functions:

`initialize_axes ()`: Initializes the axis with the appropriate attributes, either by assigning defaults or giving user desired attributes.

`create_structure ()`: Draws the axis. This function displays the axis on the screen.

`set_color ()`: Sets the color for the axis.

`set_linewidth ()`: Sets the linewidth for the axis.

`set_x_text_height ()`: Sets the text height for x axis labels.

`set_y_text_height ()`: Sets the text height for y axis labels.

`set_z_text_height ()`: Sets the text height for z axis labels.

`set_x_text_color ()`: Sets the text color for x axis labels.

`set_y_text_color ()`: Sets the text color for y axis labels.

`set_z_text_color ()`: Sets the text color for z axis labels.

`set_type ()`: Specifies whether 2D or 3D display.

`set_pointer_to_data_series ()`: Sets pointer to the data Series class.

`set_pointer_to_invisibility_filter ()`: Sets pointer to the Invisibility Filter class.

`set_pointer_to_graph ()`: Sets pointer to the Graph class.

`scale_axes ()`: Calculates the scale to be used for each axis. First, the span between the maximum and minimum data values is determined. Then the logarithm of this span is calculated. Then the high and low boundaries are multiplied by this logarithm until their values exceed the maximum and minimum values respectively.

`manual_scale_axes ()`: High and low boundary values are specified by user. The

scale is determined by the dividing data range span by number of marks for each axis.

`x_axes_marks ()`: Draws division marks for x axis.

`y_axes_marks ()`: Draws division marks for y axis.

`z_axes_marks ()`: Draws division marks for z axis.

`recreate_structure ()`: Recreates the axis.

`perform ()`: Creates a pop-up menu to allow user to make modifications to the attributes of the axis. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the axis while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that axis attribute.

Inherited Virtual Functions

`manage ()`: Tells the axes what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the structure from the view and empties the structure.

9.1.2 Bar Class

The bar class represents a single bar in a bar graph. The bar location and width in both the X and Z dimensions is determined by the number of bars to be drawn along each axis. The bar height is determined by inquiring the limits of the axes from the Axes class. The bar data is then normalized according to the axes values. The bars are drawn with a solid interior unless the user specifies otherwise. The color of the bar steps up the color table index until the last entry of the color table is reached. The color index then wraps around to the first entry. The Bar class contains pointers to the Axes, Graph, Series, and Deletion Filter classes. This will allow the bar class to communicate with these classes.

The bar is assigned default attributes when the Bar class constructor is called. Unless the user has specified the position for the bar, its location is calculated according the position in the data array the bar represents. The bar is then drawn. Attributes for the bar can be set by the programmer before the bar is drawn by using the functions provided below. The user of the program can also change the attributes by picking the bar. This will provide the user with a pop-up menu that will allow him to change the bar attributes without having to modify the code. A list of functions in the Bar class is given below:

Constructors:

Bar ()

Bar (2D or 3D, data value, bar location, bar width)

Bar (2D or 3D, data value, bar location, bar width, color)

Bar (2D or 3D, data value, bar location, bar width, color, interior pattern)

See Axes Class for description of constructors.

Destructor:

`~Bar ()`: This function checks if the bar is presently managed. If it is, it will unmanage the bar, disassociate the PHIGS structure from the view and empty the structure.

Inquiry Functions:

`get_color_index ()`: Returns color index for the bar.

`get_interior_style ()`: Returns interior style for the bar.

`get_interior_pattern ()`: Returns interior pattern for the bar.

Public Functions:

`initialize_bar ()`: Initializes the bar with the appropriate attributes, either by assigning defaults or giving user input attributes.

`set_color ()`: Sets the color of bar.

`set_bar_location ()`: Sets bar location.

`set_interior_style ()`: Sets interior style of bar.

`set_interior_pattern ()`: Sets interior pattern of bar.

`set_total_number_of_bars ()`: Sets the total number of bars to be graphed. This number is used in determining location of bars when it is not specified by the

user.

set_row_count (): Sets the number of rows in data series presently being graphed.

set_column_count (): Sets the number of columns in data series presently being graphed.

set_total_number_sets (): Sets the number of rows in data series being graphed.

set_bar_type (): 2D or 3D display.

set_pointer_to_data_series (): Sets pointer to Data Series class.

set_pointer_to_axes (): Sets pointer to Axes class.

set_pointer_to_deletion_filter (): Sets pointer to Deletion Filter class.

set_pointer_to_graph (): Sets pointer to Graph class.

create_structure (): Draws the bar. This displays the bar on the screen.

recreate_structure (): Recreates the bar.

perform (): Creates a pop-up menu to allow user to delete or make modifications to the attributes of the bar. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the bar while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that attribute.

Inherited Virtual Functions

manage (): Tells the bar what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

unmanage (): Disassociates the structure from the view and empties the structure.

9.1.3 *Curve Class*

The Curve class is used to draw curves in the graphs. The curves default to point to point lines unless the user requests spline curves. The data points for the curves are normalized by inquiring the axes limits from the Axes class. The data values are then normalized accordingly. The color of the curves step up the color table index until the last entry of the color table is reached. The color index then wraps around to the first entry. The Curve class also contains pointers to the Axes, Graph, Series and Deletion Filter classes.

When the Curve class constructor is called the curve is assigned default or user input attributes. Then markers and curves are drawn. Attributes for the curve can be set by the programmer before the curve is drawn by using the functions provided below. The user of the program can also change the attributes by picking the curve. This will provide the user with a pop-up menu that will allow the curve and polymarker attributes to be changed without having to modify the code. A list of functions in the Curve class is given below:

Constructors:

Curve ()

Curve (row count, number column, xdata, ydata, zdata)

Curve (row count, number column, xdata, ydata, zdata, color)

Curve (row count, number column, xdata, ydata, zdata, linewidth)

Curve (row count, number column, xdata, ydata, zdata, linewidth, color)

Curve (row count, number column, xdata, ydata, zdata, linewidth, point-to-point
or spline, color)

See Axes Class for description of constructors.

Destructor:

~Curve (): This function checks if the curve is presently managed. If it is, it will unmanage the curve, disassociate the PHIGS structure from the view and empty the structure.

Inquiry Functions:

get_line_color (): Returns line color.

get_linetype (): Returns linetype.

get_marker_type (): Returns polymarker type.

get_marker_scale (): Returns marker scale.

Public Functions:

`initialize_curve ()`: Initializes the curve with appropriate attributes, either by assigning defaults or giving user input attributes.

`set_color ()`: Sets the color of the curve.

`set_linewidth ()`: Sets the linewidth.

`set_linetype ()`: Sets the linetype.

`set_marker_scale_factor ()`: Sets the polymarker scale factor.

`set_curve_type ()`: Point_to_point or spline curve.

`set_graph_type ()`: 2D or 3D display.

`set_pointer_to_data_series ()`: Set pointer to Data Series class.

`set_pointer_to_axes ()`: Set pointer to Axes class.

`set_pointer_to_polar_axes ()`: Set pointer to Polar Axes class.

`set_pointer_to_deletion_filter ()`: Set pointer to Deletion Filter class.

`set_pointer_to_graph ()`: Set pointer to Graph class.

`set_row_count ()`: Set number of rows in data series presently being graphed.

`set_number_column ()`: Set number of columns in data series being graphed.

`create_structure ()`: Draw the curve. This displays the curve on the screen.

`recreate_structure ()`: Recreate the curve.

`perform ()`: Creates a pop-up menu to allow user to delete or make modifications to the attributes of the curve. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the curve while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that attribute.

Inherited Virtual Functions

`manage ()`: Tells the curve what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the structure from the view and empties the structure.

9.1.4 Grid Class

The grid class controls the drawing of the grids on the graph. If the graph is two dimensional only one grid is drawn in the XY plane. If the graph is three dimensional, three separate grids are drawn. One each in the XY, XZ, and the YZ planes. Each grid inquires from the Axes class the starting and ending locations of the axes lines and draws the grid accordingly. The default color for the grids is gray. The grids have twenty-five divisions along each axes by default. The Grid classes also have pointers to the Invisibility Filter class.

Attributes for the grid can be set by the programmer before the grid is drawn by using the functions provided below. The user of the program can also change the attributes by picking the grid. This will provide the user with a pop-up menu that will allow him to change the grid attributes without having to modify the code. A list of functions in the

Grid class is given below.

Constructors:

Grid ()

Grid (orientation, x-start-pt, y-start-pt, z-start-pt, x-length, y-length, z-length)

Grid (orientation, x-start-pt, y-start-pt, z-start-pt, x-length, y-length, z-length, number-x-division, number-y-divisions)

Grid (orientation, x-start-pt, y-start-pt, z-start-pt, x-length, y-length, z-length, number-x-division, number-y-divisions, color)

Grid (orientation, x-start-pt, y-start-pt, z-start-pt, x-length, y-length, z-length, number-x-division, number-y-divisions, linewidth)

Grid (orientation, x-start-pt, y-start-pt, z-start-pt, x-length, y-length, z-length, number-x-division, number-y-divisions, linewidth, color)

See Axes Class for description of constructors.

Destructor:

~Grid (): This function checks if the grid is presently managed. If it is, it will unmanage the grid, disassociate the PHIGS structure from the view and empty the structure.

Public Functions:

`initialize-grid ()`: Initializes the grid with appropriate attributes, either by assigning defaults or giving user input attributes.

`create_structure ()`: Draws the grid. This displays the grid on the screen.

`set_color ()`: Sets the color of the grid.

`set_linewidth ()`: Sets the linewidth.

`set_vertical_linetype ()`: Sets linetype for the vertical grid lines.

`set_horizontal_linetype ()`: Sets linewidth for the horizontal grid lines.

`set_manual_grid_set ()`: Flag if grid has been changed by user.

`set_pointer_to_axes ()`: Sets pointer to Axes class.

`set_pointer_to_invisibility_filter ()`: Sets pointer to Invisibility Filter class.

`recreate_structure ()`: Recreates the grid.

`perform ()`: Creates a pop-up menu to allow user to make modifications to the attributes of the grid. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the grid while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that attribute.

Inherited Virtual Functions

`manage ()`: Tells the grid what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the structure from the view and empties the structure.

9.1.5 Legend Class

The legend class controls the drawing of the legends for the graph. If the graph type is a bar, stack bar, or pie graph the legend draws a box of the same color and interior style as the bar or pie slice on the graph. If the graph type is a line graph the legend draws a line of the same color, style, and marker type as in the graph. Both legend types then have the legend text displayed underneath the box or line. With the bar graph, each row of bars has a corresponding column of legends. With the line graph, the legends are displayed in a column format. The Legend class contains a pointer to the Deletion Filter class.

Attributes for the legend can be set by the programmer before the legend is drawn by using the functions provided below. The user of the program can also change the attributes by picking the legend. This will provide the user with a pop-up menu that will allow him to change the legend attributes without having to modify the code. A list of functions in the Legend class is given below.

Constructors:

Legend ():

Legend (graph type, location, legend-width, legend-height, text-string)

See Axes Class for description of constructors.

Destructor:

`~Legend ()`: This function checks if the legend is presently managed. If it is, it will unmanage the legend, disassociate the PHIGS structure from the view and empty the structure.

Public Functions:

`initialize_legend ()`: Initializes the legend with appropriate attributes, either by assigning defaults or giving user input attributes.

`set_line_color ()`: Sets line color.

`set_linewidth ()`: Sets linewidth.

`set_linetype ()`: Sets linetype.

`set_marker_scale_factor ()`: Sets polymarker scale factor.

`set_marker_type ()`: Sets polymarker type.

`set_interior_color ()`: Sets interior color.

`set_interior_style ()`: Sets interior style.

`set_interior_pattern ()`: Sets interior pattern.

`set_text_height ()`: Sets text height.

`set_text_color ()`: Sets text color.

`set_text_font ()`: Sets text font.

`set_legend_width ()`: Sets legend width.

`set_legend_height ()`: Sets legend height.

`set_legend_location ()`: Sets legend location.

`set_legend_text ()`: Set text string for legend.

`set_graph_type ()`: Set what type of graph is being drawn.

`create_structure ()`: Creates legend. This displays the legend on the screen.

`recreate_structure ()`: Recreates the legend.

`perform ()`: Creates a pop-up menu to allow user to make modifications to the attributes of the legend. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the legend while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that attribute.

Inherited Virtual Functions

`manage ()`: Tells the legend what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the structure from the view and empties the structure.

9.1.6 Pie Class

The pie class controls the drawing of the pie slices within a pie graph. The default for a pie slice is a two-dimensional slice in the XY plane. If the graph is three dimensional, the pie will be circular in the XZ plane and have height going up the Y axis. The data value for the pie slice will be normalized by summing the entire row of the data values and then calculating the percentage of each individual value. The color of the pie slices will step up the color table index until the last entry of the color table is reached. The color index then wraps around to the first entry. The Pie class has a pointer to the Data Series and Graph classes.

Attributes for the pie can be set by the programmer before the pie is drawn by using the functions provided below. The user of the program can also change the attributes by picking the pie. This will provide the user with a pop-up menu that will allow him to change the pie attributes without having to modify the code. A list of functions in the Pie class is given below.

Constructors:

Pie ()

Pie (2D or 3D, starting angle, data value)

Pie (2D or 3D, starting angle, data value, height)

Pie (2D or 3D, starting angle, data value, height, radius)

See Axes Class for description of constructors.

Destructor:

`~Pie ()`: This function checks if the pie is presently managed. If it is, it will unmanage the pie, disassociate the PHIGS structure from the view and empty the structure.

Inquiry Functions:

`get_color_index ()`: Returns color index being used.

`get_interior_style ()`: Returns interior style.

`get_interior_pattern ()`: Returns interior pattern.

Public Functions:

`initialize_pie ()`: Initializes the pie slice with appropriate attributes, either by assigning defaults or giving user input attributes.

`set_color ()`: Sets the color.

`set_origin ()`: Sets the location for the origin (center) of pie graph.

`set_height ()`: Sets the height of the pie slice.

`set_radius ()`: Sets the radius of the pie slice.

`set_column_count ()`: Tells which column value of data Series is presently being drawn.

`set_starting_angle ()`: Sets the angle from which pie slice is to be started from.

`set_interior_style ()`: Sets interior style.

`set_interior_pattern ()`: Sets interior pattern.

`set_pie_type ()`: 2D or 3D display.

`set_pointer_to_data_series ()`: Sets pointer to data Series class.

`set_pointer_to_graph ()`: Sets pointer to Graph class.

`get_number_degree ()`: Determine number of degrees pie slice should represent.

`create_structure ()`: Draw the pie slice. This displays the pie slice on the screen.

`recreate_structure ()`: Recreate the pie slice.

`perform ()`: Creates a pop-up menu to allow user to make modifications to the attributes of the pie slice. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the pie slice while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that attribute.

Inherited Virtual Functions

`manage ()`: Tells the pie slice what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the structure from the view and empties the structure.

9.1.7 Polar Axes Class

The polar axes class controls the drawing of a polar axes for the graph. The polar axes are two dimensional axes drawn in the XY plane. The default color for the axes is red. The axes have four circular divisions. The class automatically scales the axes based on the data value array to be graphed. The Polar-Axes class has a pointer to the Invisibility Filter and Graph classes.

Attributes for the polar axes can be set by the programmer before the polar axes is drawn by using the functions provided below. The user of the program can also change the attributes by picking the polar axes. This will provide the user with a pop-up menu that will allow him to change the polar axes attributes without having to modify the code. A list of functions in the Polar-Axes class is given below.

Constructors:

Polar_Axes ()

Polar_Axes (origin, radius)

Polar_Axes (origin, radius, color)

Polar_Axes (origin, radius, linewidth)

Polar_Axes (origin, radius, linewidth, color)

See Axes Class for description of constructors.

Destructor:

`~Polar_Axes ()`: This function checks if the polar-axes is presently managed. If it is, it will unmanage the polar-axes, disassociate the PHIGS structure from the view and empty the structure.

Inquiry Functions:

`get_x_origin_point ()`: Returns x value of graph origin.

`get_y_origin_point ()`: Returns y value of graph origin.

`get_z_origin_point ()`: Returns z value of graph origin.

`get_graph_radius ()`: Returns radius of graph.

`get_hi_magnitude ()`: Returns high magnitude of polar_axes.

`get_low_magnitude ()`: Returns low magnitude of polar_axes.

Public Functions:

`initialize_axes ()`: Initializes the polar-axes with appropriate attributes, either by assigning defaults or giving user input attributes.

`set_color ()`: Sets the color of the axes.

`set_linewidth ()`: Sets the linewidth.

`set_number_circles ()`: Sets the number of circular divisions.

`set_x_text_height ()`: Sets the text height of axes labels.

`set_x_text_color ()`: Sets axes text color.

`set_graph_radius ()`: Set radius of polar-axes.

`set_pointer_to_data_series ()`: Set pointer to data Series class.

`set_pointer_to_invisibility_filter ()`: Set pointer to Invisibility Filter class.

`set_pointer_to_graph_class ()`: Set pointer to Graph class.

`scale_axes ()`: Determine appropriate scale for polar-axes. Uses the same method as Axes class described earlier.

`manual_scale_axes ()`: Set scale for polar-axes. Uses same method as Axes class described earlier.

`x_axes_marks ()`: Draw polar-axes labels.

`create_structure()`: Draw the polar-axes. This displays the polar axes on the screen.

`recreate_structure ()`: Recreate the polar-axes.

`perform ()`: Creates a pop-up menu to allow user to make modifications to the attributes of the pie slice. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the polar axes while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that attribute.

Inherited Virtual Functions

`manage ()`: Tells the pie slice what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the structure from the view and empties the structure.

9.1.8 *Stacked Bar*

The Stacked Bar class controls the drawing of each stack bar in a stacked bar graph. The bar location and width in both the X and Z dimensions are determined by the number of bars to be drawn along each axes. The bar height is determined by inquiring the limits of the axes from the Axes class. The bar data is then be normalized to the axes values. The bars are drawn with a solid interior unless the user specifies otherwise. The color of the bar steps up the color table index until the last entry of the color table is reached. The color index then wraps around to the first entry. The Stacked Bar class contains pointers to the Graph and Deletion Filter classes.

When the Stacked Bar class constructor is called the bar is assigned default attributes. Unless the user has input the position for the bar, its location is calculated according the position in the data array the bar represents. The bar is then drawn. Attributes for the stack bar can be set by the programmer before the stack bar is drawn by using the functions provided below. The user of the program can also change the attributes by picking the stack bar. This will provide the user with a pop-up menu that will allow him to change the stack bar attributes without having to modify the code. A list of functions in the Stacked Bar class is given below:

Constructors:

Stack_Bar ()

Stack_Bar (2D or 3D, data value, bar location, bar width)

Stack_Bar (2D or 3D, data value, bar location, bar width, color)

Stack_Bar (2D or 3D, data value, bar location, bar width, color, interior pattern)

See Axes Class for description of constructors.

Destructor:

~Stack_Bar (): This function checks if the stack bar is presently managed. If it is, it will unmanage the stack bar, disassociate the PHIGS structure from the view and empty the structure.

Inquiry Functions:

get_color_index (): Returns color index for the bar.

get_interior_style (): Returns interior style for the bar.

get_interior_pattern (): Returns interior patter for the bar.

Public Functions:

initialize_bar (): Initializes the stack bar with the appropriate attributes, either by assigning defaults or giving user input attributes.

set_color (): Sets the color of bar.

`set_bar_location ()`: Sets bars location.

`set_interior_style ()`: Sets interior style of bar.

`set_interior_pattern ()`: Sets interior pattern of bar.

`set_total_number_of_bars ()`: Sets the total number of bars to be graphed. This number is used in determining location of bars when it is not specified by the user.

`set_row_count ()`: Sets the number of row in data series presently being graphed.

`set_column_count ()`: Sets the number of column in data series presently being graphed.

`set_total_number_sets ()`: Sets the number of rows in data series being graphed.

`set_Ylast ()`: Set the starting position for the stack bar.

`set_bar_type ()`: 2D or 3D display.

`set_pointer_to_data_series ()`: Sets pointer to data Series class.

`set_pointer_to_axes ()`: Sets pointer to Axes class.

`set_pointer_to_deletion_filter ()`: Sets pointer to Deletion Filter class.

`set_pointer_to_graph ()`: Sets pointer to Graph class.

`get_bar_height ()`: Return the height of the bar being drawn.

`create_structure ()`: Draws the bar. This displays the stack bar on the screen.

`recreate_structure ()`: Recreates the bar.

`perform ()`: Creates a pop-up menu to allow user to delete or make modifications to the attributes of the stack bar. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the stack bar while

still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that attribute.

Inherited Virtual Functions

`manage ()`: Tells the stack bar what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the structure from the view and empties the structure.

9.1.9 Text

The Text class controls the text strings that are present in the graph. These include the titles and the legend text strings. The Text class allows the user control over the text string, text font, text height, text color, and the location of the text. The text is annotation text so that it is always legible to the user.

Attributes for the text can be set by the programmer before the text is drawn by using the functions provided below. The user of the program can also change the attributes by picking the text. This will provide the user with a pop-up menu that will allow him to change the text attributes without having to modify the code. A list of functions in the

text class is given below:

Constructors:

Text ()

Text (x-location, y-location, z-location, text string)

See Axes Class for description of constructors.

Destructor:

~Text (): This function checks if the text is presently managed. If it is, it will unmanage the structure, disassociate the structure from the view and empty the structure.

Public Functions:

initialize_text (): Initializes the text with the appropriate attributes, either by assigning defaults or giving user input attributes.

set_text_color (): Sets the text color.

set_text_height (): Sets the text height.

set_text_font (): Sets the text font.

set_pointer_to_invisibility_filter (): Sets pointer to Invisibility Filter class.

create_structure (): Draws the text structure. This displays the text on the screen.

recreate_structure (): Recreates the text structure.

perform (): Creates a pop-up menu to allow user to make modifications to the

attributes of the text. Using the pop-up menu gives the user the means of accessing the functions to change the attributes of the text while still being in the program. When the user changes one of the attributes, the appropriate function call is made by the program to reset that attribute.

Inherited Virtual Functions

`manage ()`: Tells the text what structure identifier to use, which view index to use, the priority to give the view, and passes the pointer to the Interface Manager.

`unmanage ()`: Disassociates the structure from the view and empties the structure.

10.0 Other Classes

10.1 Data Series Class

The Data Series class provides the means to access and manipulate the data array sent in to be graphed. This class provides the mathematical functions required by the graph classes to supply the data values and normalize them correctly according to the type of graph being produced. It also contains the functions which determine the intermediate points for the curve class. A list of functions in the Data Series class is given below.

Public Functions:

`set_data ()`: Set data array to be graphed in Data Series class.

`get_maximum_data_value ()`: Determine maximum data value.

`get_minimum_data_value ()`: Determine minimum data value.

`get_number_rows ()`: Return number of rows in data array.

`get_number_columns ()`: Return number of columns in data array.

`get_normalized_data_value ()`: Return normalized data value.

`get_maximum_row_sum ()`: Determine maximum value of each row summed.

`get_value ()`: Return value of data array position.

`curve_fit ()`: Set up variables needed for curve-fitting algorithm.

spl3d (): Compute chord lengths.

spline (): Use blending functions and compute points for curve.

10.2 Control Entities

Filters in PHIGS are a means of applying attributes to only certain structure elements, as opposed to all elements in a structure. Control classes were necessary to control the PHIGS filters. The PHIGS filters totally reset the filter after every call made to them. The filters do not append a name to or remove it from the filter. This creates a lot of overhead for the programmer in constantly maintaining what should or should not be included in the filter at all points in the program. The creation of the control classes alleviates this task for the programmer. By using the filters one item can be appended or removed from the filter, without having to be concerned with resetting all other filter values.

10.2.1 Deletion Class

The Deletion Class is not a PHIGS filter. This class is created for a filter at the structure level. The Deletion class controls the deletion filter. Functions provided by this class

allow the user to add a structure to the filter and to see if a structure is present in the filter. When a graph is recreated the deletion filter will be checked. If a particular entities identifier is in the deletion filter this means the user does not want this entity displayed. Therefore it will not be sent the command to recreate itself.

Constructor

Deletion ()

Public Functions:

add_to_deletion_filter (): Add structure id to deletion filter.

check_deletion_array (): Check to see if a structure is present in deletion filter.

check_deletion_array_size (): Check to see how many structures are present in deletion filter.

10.2.2 Highlighting Class

The Highlighting class controls the highlighting filter used by PHIGS. Functions provided by this class allow programmers to add PHIGS class name sets to, or delete the name sets from, the inclusion and exclusion filters. The user can also inquire the current state of this filter.

Constructors:

Highlight ()

Highlight (workstation identifier, inclusion array length, inclusion array, exclusion array length, exclusion array)

Public Functions:

initialize_highlighting_filter (): Initialize the variables for the highlighting filter.

add_to_highlighting_filter (): Add PHIGS class name set to highlighting filter.

exclude_from_highlighting_filter (): Remove PHIGS class name set from highlighting filter.

inquire_highlighting_filter_state (): Inquire the present state of the highlighting filter.

10.2.3 Invisibility Class

The Invisibility class controls the invisibility filter used by PHIGS. Functions provided by this class allow programmers to add PHIGS class name sets to, or delete the name sets from, the inclusion and exclusion filters. The user can also inquire the current state of this filter.

Constructors:

Invisible ()

Invisible (workstation identifier, inclusion array length, inclusion array, exclusion array length, exclusion array)

Public Functions:

initialize_invisibility_filter (): Initialize the variables for the invisibility filter.

add_to_invisibility_filter (): Add PHIGS class name set to invisibility filter.

exclude_from_invisibility_filter (): Remove PHIGS class name set from invisibility filter.

inquire_invisibility_filter_state (): Inquire the present state of the invisibility filter.

11.0 Implementation and Sample Program

The above described classes were implemented on IBM RS/6000s. The operating system for these machines is UNIX. The programming language chosen for this implementation was C++. It provided all of the features essential to an object-oriented language (encapsulation, inheritance, and polymorphism). The graphics language chosen was PHIGS. The Graphics User Interface (GUI) chosen was developed by Woyak, and was explained earlier.

Creating graphs using the class library is a very simple and easy task. The programmer does not need to deal with PHIGS functions or structures. Instead, creating a few instances from the class library will suffice to display a graph on the screen and provide the end user with interactive methods to modify the graph for better visualization (e.g., scaling, changing colors, removing entities, etc.). Figure 13 illustrates a typical program which creates a two-dimensional line graph. First an Interface Manager is created to control the windows. The graph is created by creating one instance from the Line_Graph class. The Line_Graph constructor processes the data, normalizes the values and creates all the instances from the graph entity classes. This Line_Graph object is then added to the Interface Manager which creates a window and displays the line graph as shown in Figure 14. The function "process()" in the Interface Manager waits for input from the user. The

```

Sample main program
void main()
{
    // create interface manager
    Interface_Manager  interface_manager(1);

    // set size of data array being sent
    int    number_row = 10;
    int    number_column = 10;

    // Define data for Line_Graph - xdata, ydata
    float  xdata = {   };
    float  ydata = {   };

    // create the graph window
    Line_Graph  win1(number_row, number_column, *xdata,
*ydata);

    // add graph window to interface manager
    interface_manager.add_window(&win1, PERFORM);

    Event  *event
    int    exit = NO;

    do {
        event = interface_manager.process(30.0);

        // Flush additional events
        while (event != NULL)
        {
            event = event->extract_next_event();
        }

        } while (event->get_id != EXIT);
}

```

Figure 13. Sample Program

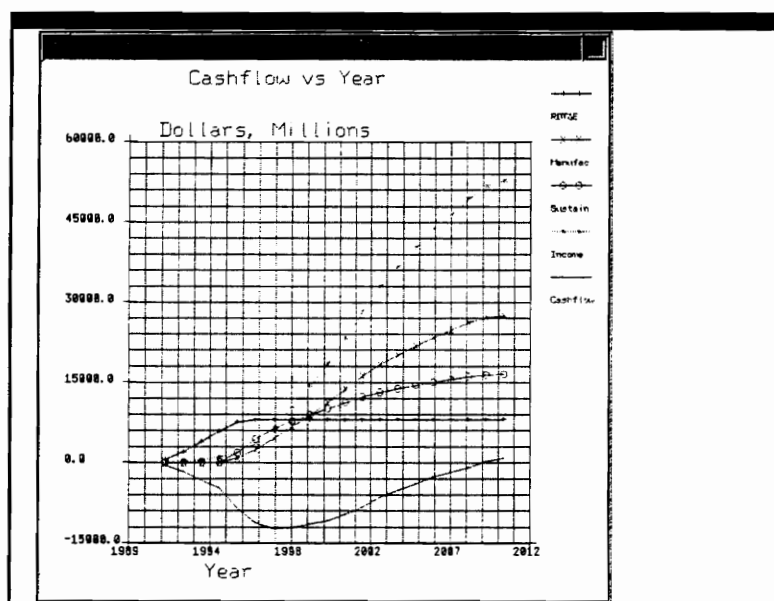


Figure 14. Line Graph Created Using Class Library

flow of control at this point is illustrated in Figure 15. If the input (from mouse) is on the window border, the Interface Manager passes control to the Geometry Manager class for processing the input. If the input is from the geometry window, the input is passed on to the Graph class. The Graph class checks the input to determine which graph entity was selected and passes control to the appropriate entity. This process is illustrated in Figure 16. Each entity class includes a function "perform()" for processing user input. Figure 17 shows an example of the "perform" function for the Axes class. This process automatically brings up appropriate pop-up menus which allow the user to modify attributes of the axes. Figure 18 shows an example of a pop-up menu displayed by the Axes class for modifying the axes parameters. Figures 19 through 23 show examples of all other types of graphs created by this class library. Figures 24 through 30 show examples of pop-up menus available for modifications by the user.

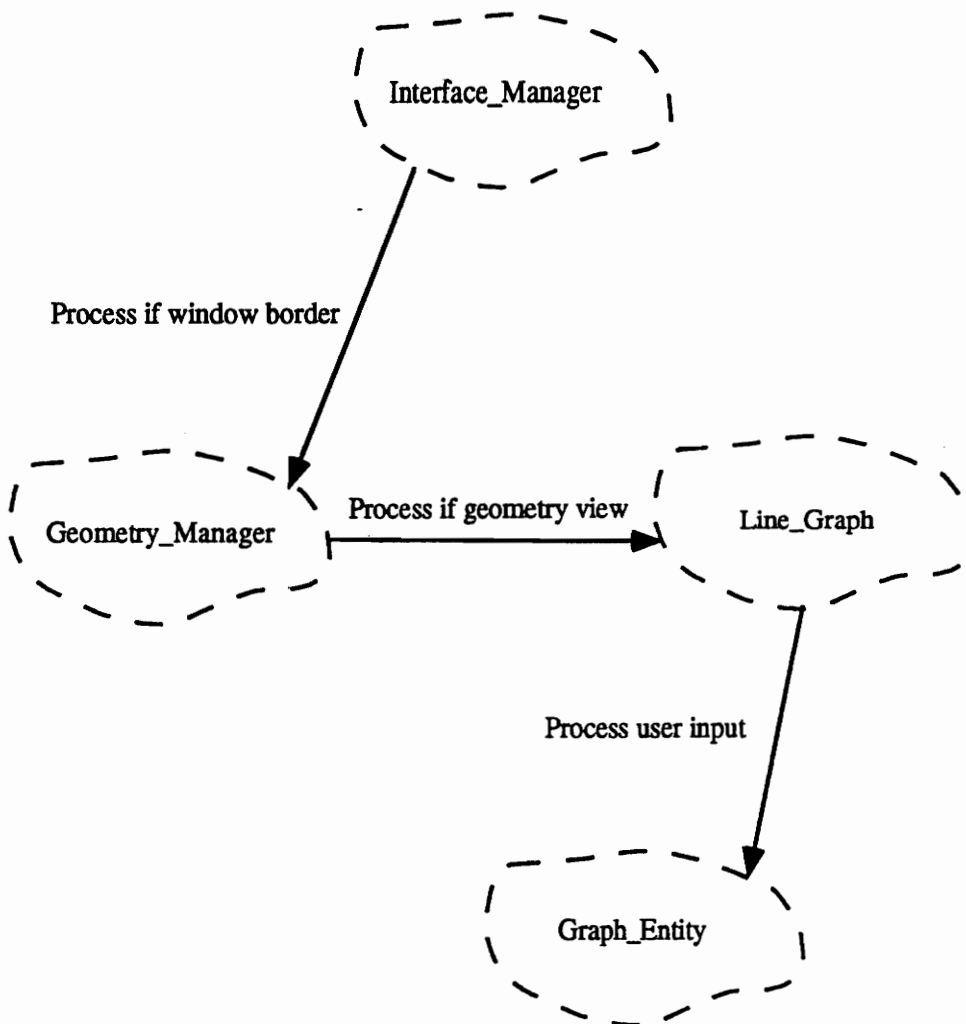


Figure 15. Flow of Control After User Input


```
void Line_Graph :: process_geometry_view( , , , )
```

```
wait for mouse button release event
```

```
if axes was picked call axes.perform
```

```
else if grid was picked call grid.perform
```

```
else if graph title was picked call title.perform
```

```
etc
```

Figure 16. Line Graph Function for Processing User Input

Axes.perform

- create pop up menu
- add attributes to menu items
- add menu items to menu
- add menu to manager
- process input
- change required entities
- update graph

Figure 17. Axes Functions for Processing User Input

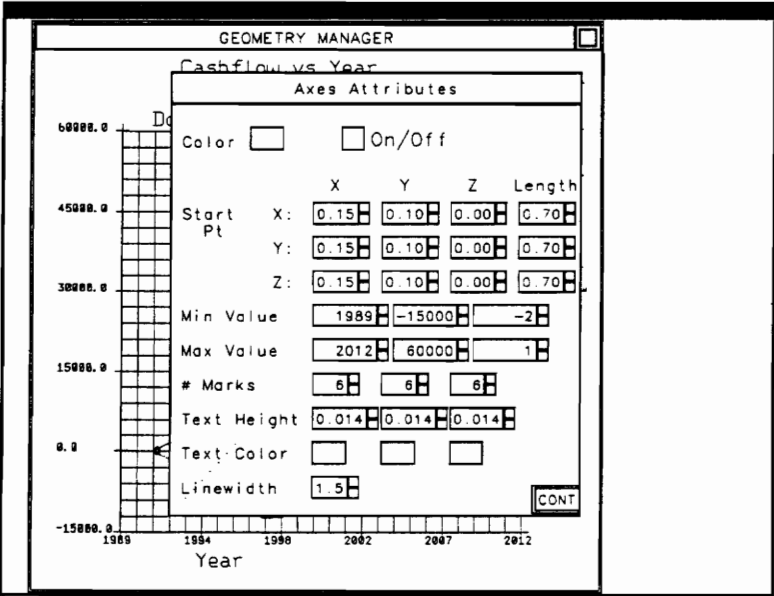


Figure 18. Pop Up Menu Created by Axes Class

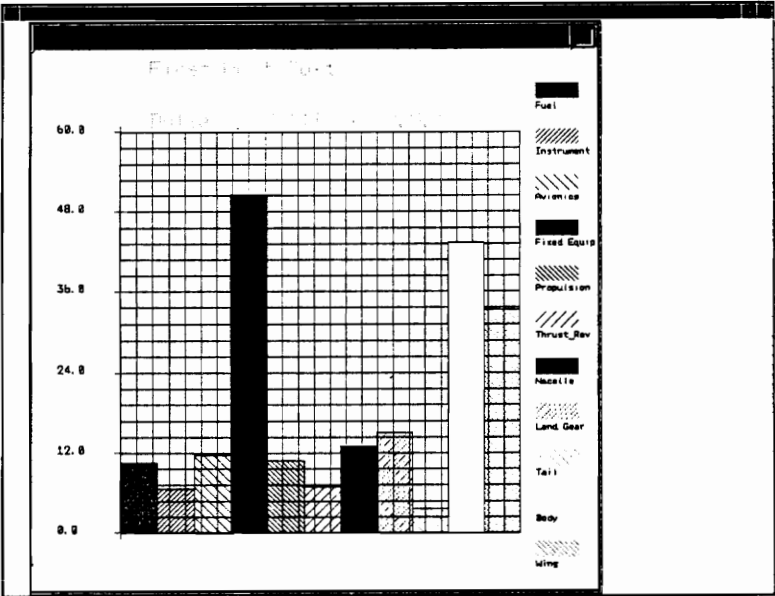


Figure 19. Example of a Bar Graph

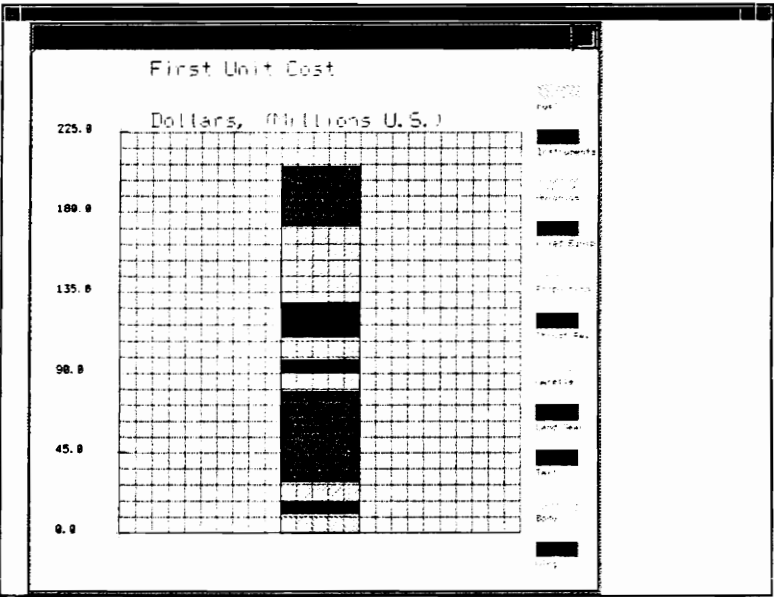


Figure 20. Example of a Stacked Bar Graph

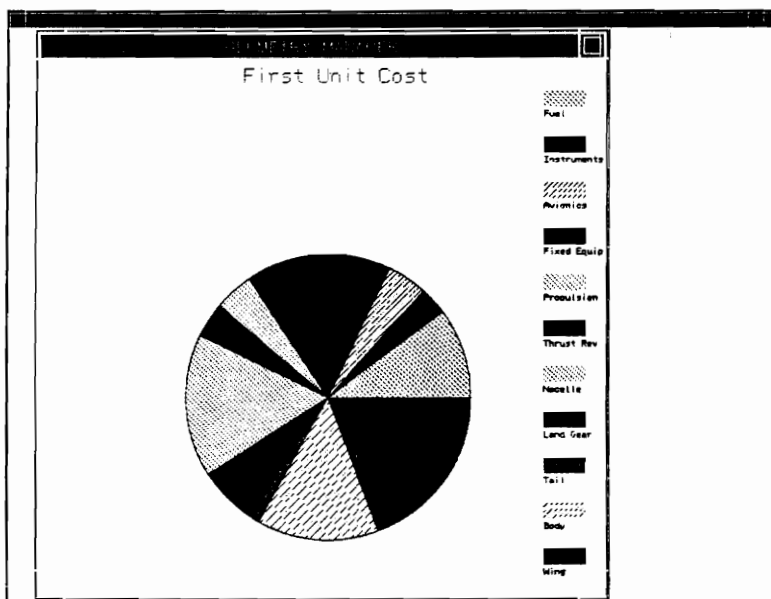


Figure 21. Example of a Pie Graph

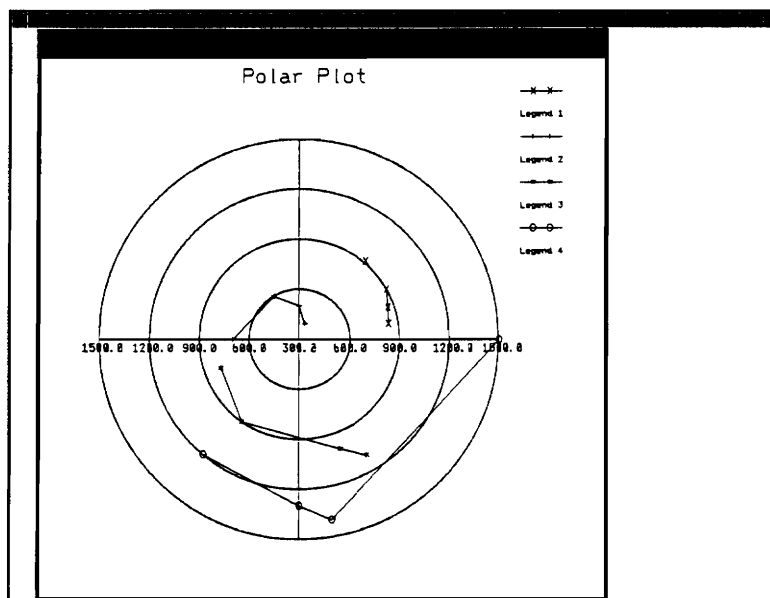


Figure 22. Example of a Polar Graph

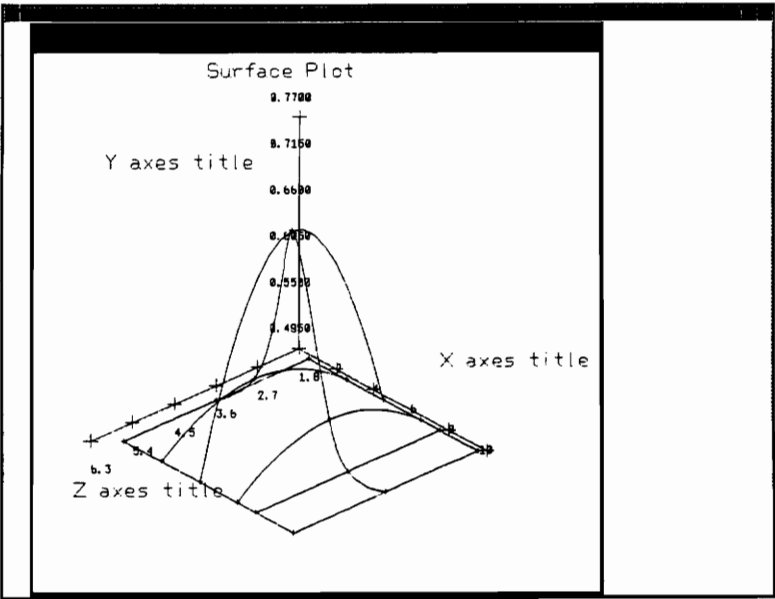


Figure 23. Example of XYZ Plot

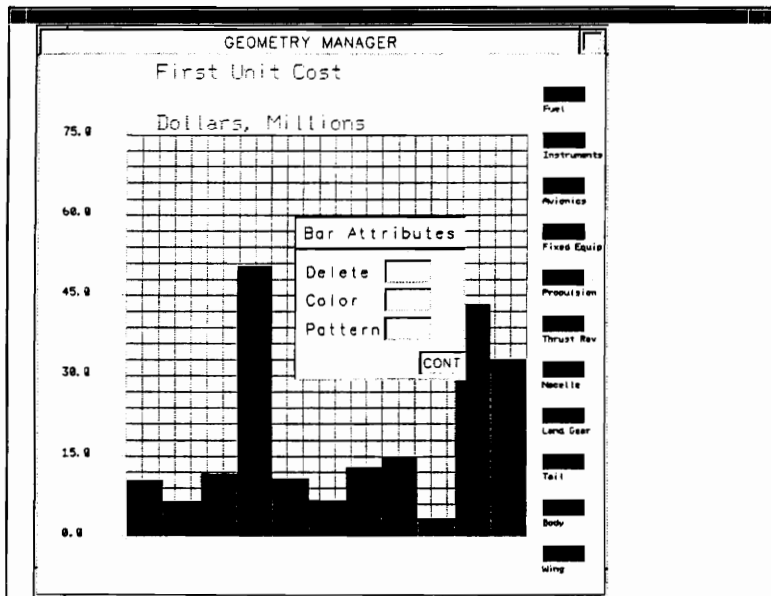


Figure 24. Example of Pop Up Menu Created by Bar and Stacked Bar Classes

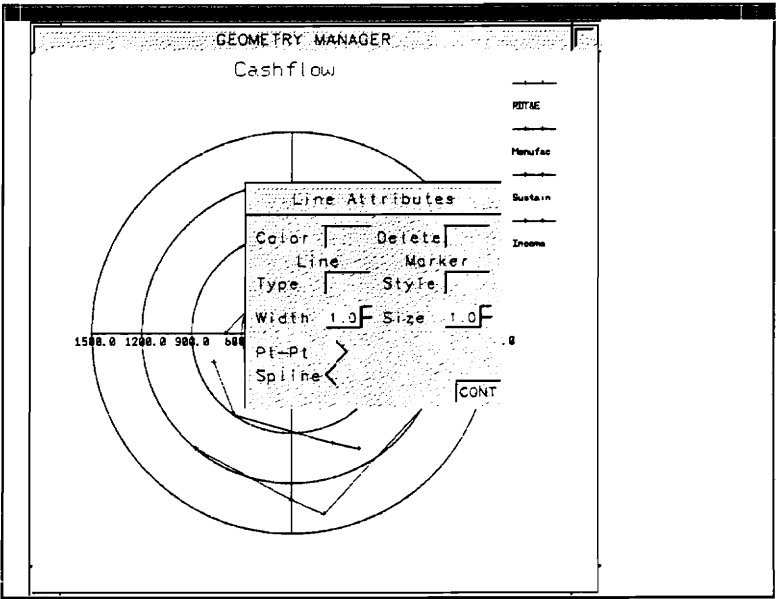


Figure 25. Example of Pop Up Menu Created by Curve Class

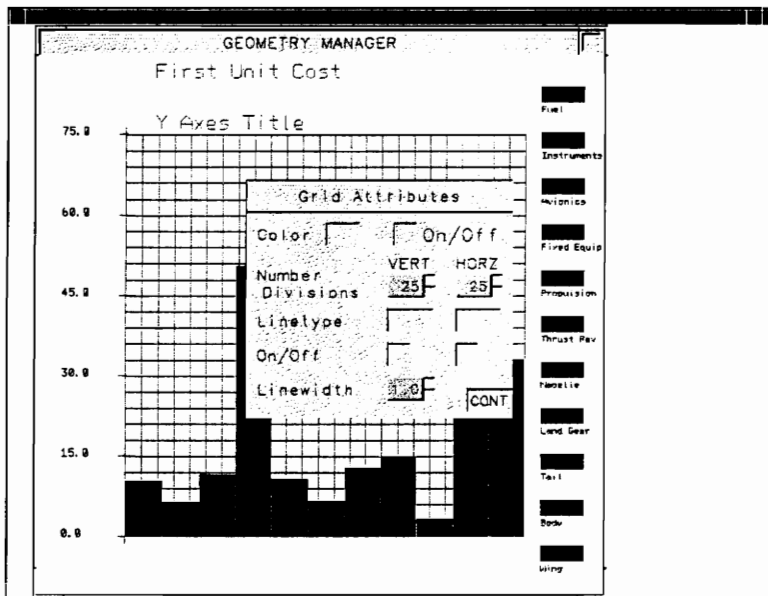


Figure 26. Example of Pop Up Menu Created by Grid Class

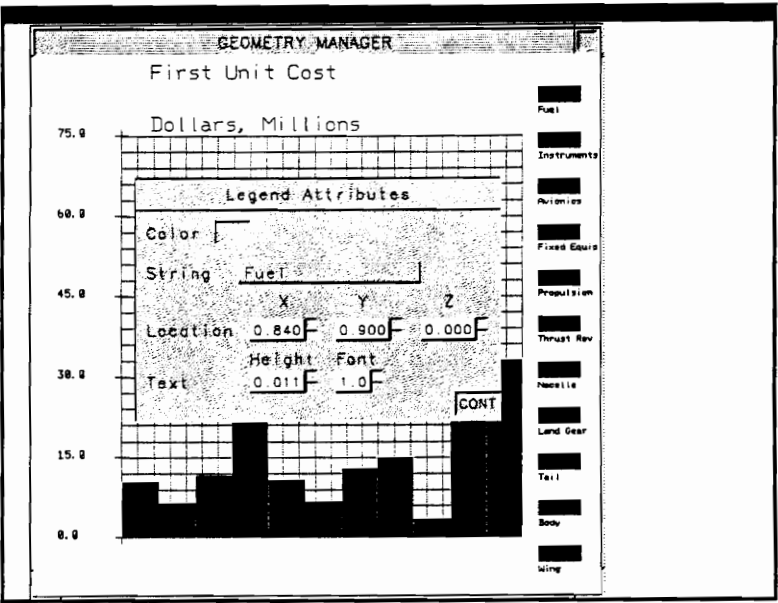


Figure 27. Example of Pop Up Menu Created by Legend Class

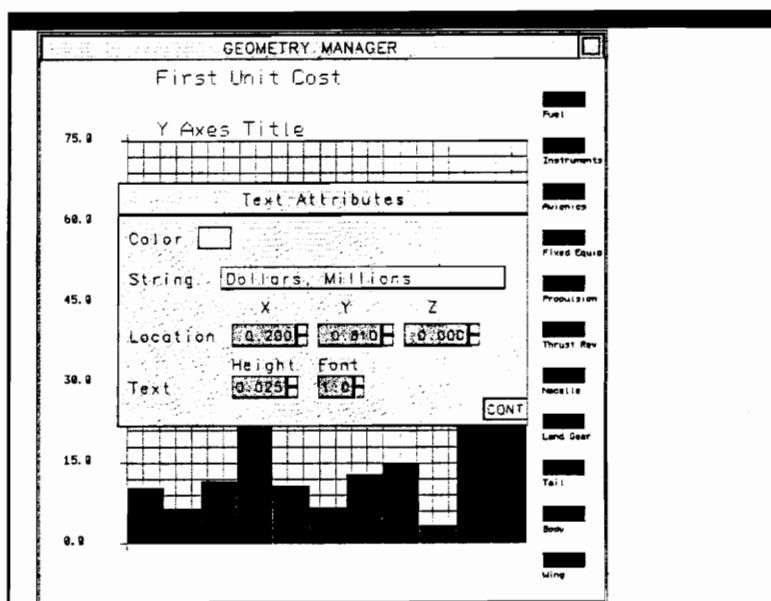


Figure 28. Example of Pop Up Menu Created by Text Class

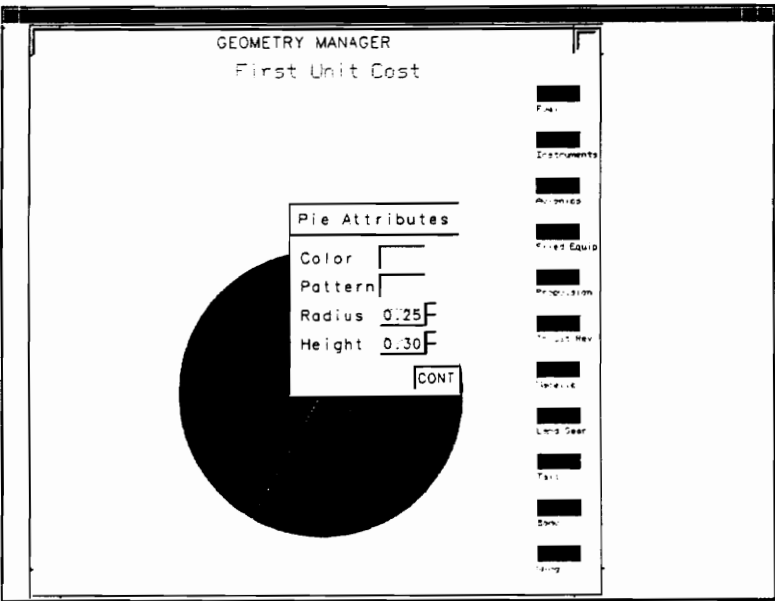


Figure 29. Example of Pop Up Menu Created by Pie Class

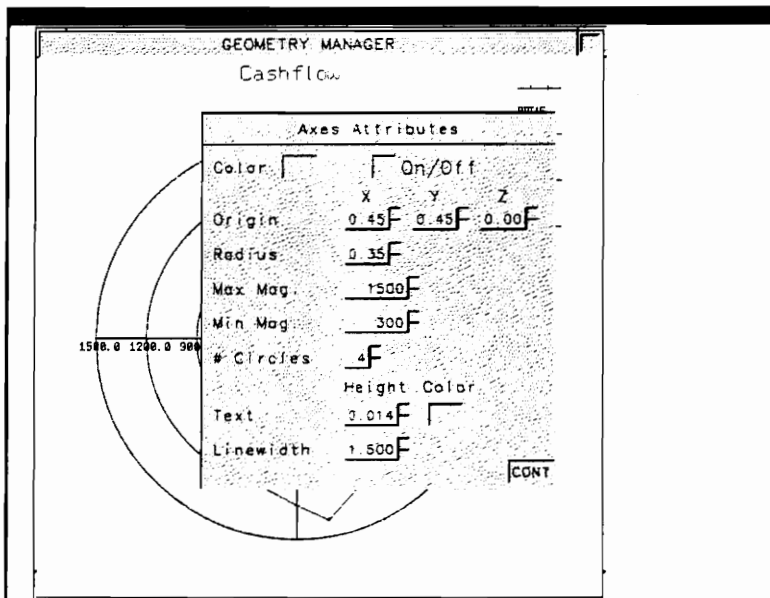


Figure 30. Example of Pop Up Menu Created by Polar Axes Class

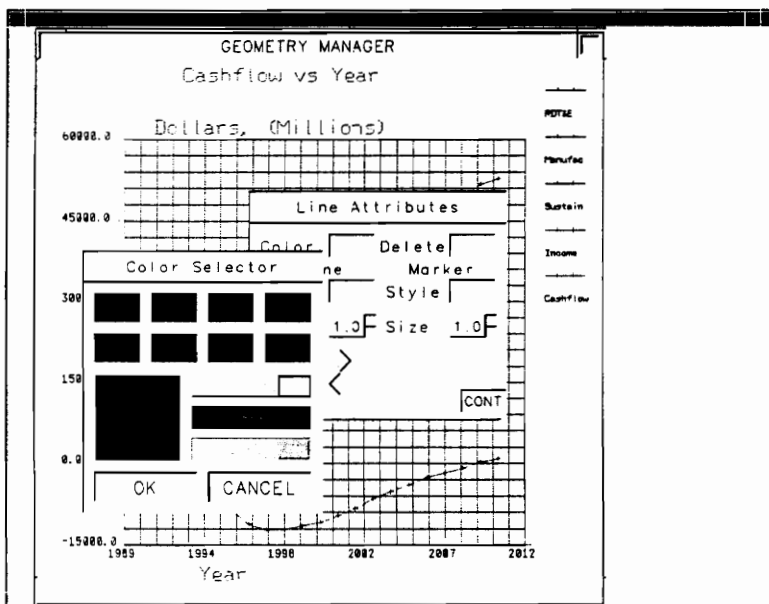


Figure 31. Example of Color Modification Menu

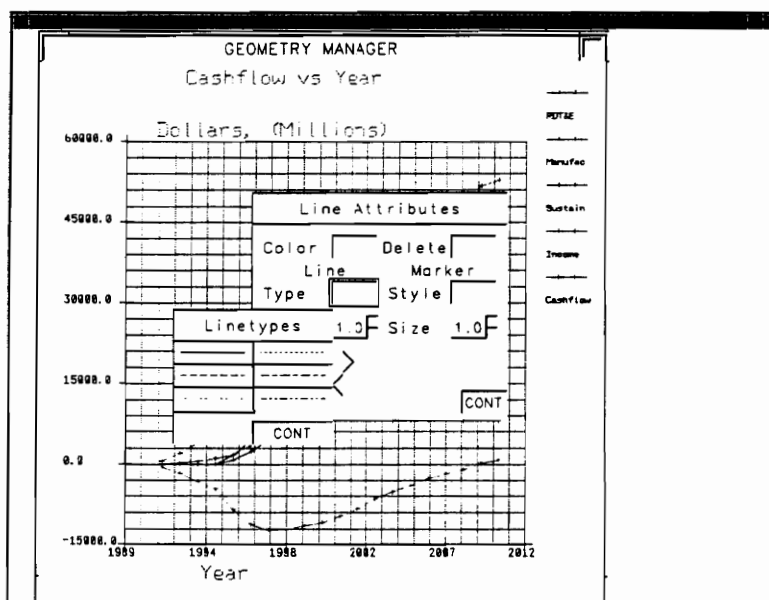


Figure 32. Example of PHIGS Primitive Attribute Modification Menu

12.0 Implementation of Graphs for ACSYNT

Within ACSYNT the variables to be graphed are chosen from a template (Figure 33) by the user. Once the user has entered the variables to be displayed, the correct arrays are read in and the data points are sent into the graphing module. The data arrays are sent into the proper graphs constructor. A new PHIGS workstation (window) is then opened for the graph. This workstation (window) takes control and displays the graph. All user interface methods (for graph modifications) are made available to the user. Once the user is done looking at the graph and exits, control is returned to the main ACSYNT window.

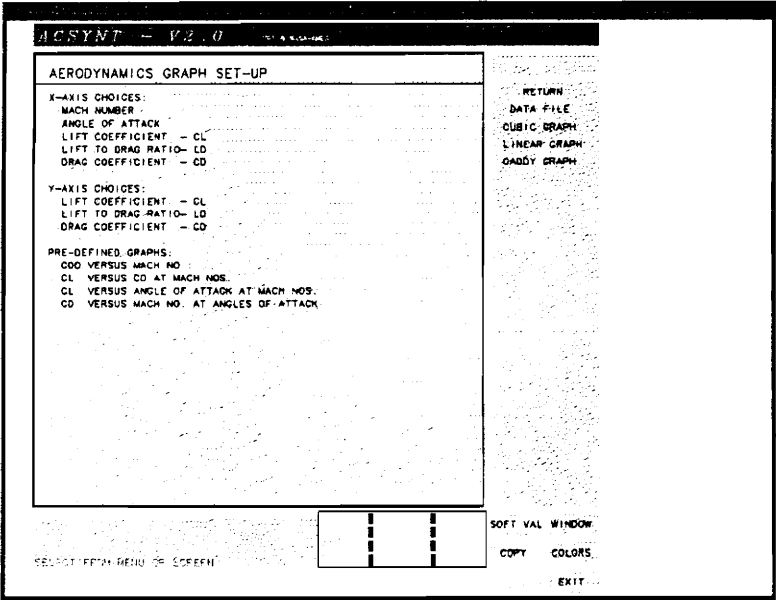


Figure 33. Example of Variables Template in ACSYNT

13.0 Conclusion

The design and creation of a set of high-level tools to facilitate the inclusion of engineering graphs in CAD applications programs has been described. These tools are provided through an object-oriented class library. This system uses PHIGS and a PHIGS-based, Motif-like interface framework for graphics and GUI support. An engineer or applications programmer with little or no knowledge of PHIGS can easily add graphing functions to his program using this set of classes. Examples of the use of this system has been described along with sample code listings. The use of object-oriented methods makes this system very flexible and extendable. Other graph entities and types of graphs can be added very easily to this class library in the future.

This class library was used to create a new set of graphing functions. Implementing these new graphs took about five months and 13,000 lines of code. This new code replaced 33,000 lines of existing code and provided much better functionality and flexibility to the end user.

Although PHIGS provides applications programmers with methods to create device-independent code, more high-level tools are needed to support the needs of engineering

and CAD applications developers. More object-oriented class libraries (such as the one described in this paper) need to be created in the future to satisfy these needs.

14.0 References

[ADAM92] Adams, L., Supercharged C++ Graphics, Blue Ridge Summit: Windcrest Books, 1992.

[BOOC91] Booch, G., Object-Oriented Design with Applications, The Benjamin/Cummings Publishing Company, Inc., 1991.

[CUNN92] Cunningham, S., et al, editor, Computer Graphics Using Object-Oriented Programming, New York: John Wiley & Sons, Inc., 1992.

[DUNN91] Dunn, M. F., and Knight, J. D., "Software Reuse in an Industrial Setting: A Case Study", 13th International Conference on Software Engineering, pp. 329-338, 1991.

[FELL91] Fellner, D. W., "Object-Oriented Programming - Does it Help in Computer Graphics?", New Results and New Trends in Computer Science, 555, pp. 132-151, June 1991.

[FLEM91] Fleming, S. and Myklebust, A., "Utilizing the graPHIGS API for CAD Applications", Proceedings of the Second International graPHIGS User's Group Conference and Workshop, Blacksburg, Virginia, October 20-23, 1991, pp. 3-10.

[FLEM92] Fleming, S. and Myklebust, A., "The Enhancement of PHIGS+ B-spline Functionality for Geometric Modeling", presented at the Fourth IFIP WG5.2 Workshop on Geometric Modeling in Computer Aided Design, Rensselaerville, New York, September 27-October 1, 1992.

[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, Dubrovnik, Yugoslavia, August 28-31, 1990.

[JAYA91] Jayaram, S. and Myklebust, A., "The Significance of Standards in CAD Education", Proceedings of University Programs in Computer-Aided Engineering, Design and Manufacturing (UPCAEDM), Ninth Annual Conference, Provo, Utah, May 16-18, 1991, pp. 144-147.

[JAYA92] Jayaram, S., Myklebust, A., and Gelhausen, P., "ACSYNT - A Standards-Based System for Parametric Computer Aided Conceptual Design of Aircraft", presented

at the AIAA Aerospace Design Conference, Irvine, California, February 3-6, 1992. AIAA paper no. AIAA-92-1198.

[JAYA93a] Jayaram, S., and Myklebust, A., "Device-Independent Programming Environments for CAD/CAM Software Creation", *Computer Aided Design*, vol. 25, no. 2, February, 1993.

[JAYA93b] Jayaram, S., and Myklebust, A., "Evaluating PHIGS for CAD Applications - A Case Study", 1st Annual PHIGS User's Group Conference, March 21-24, 1993, Orlando, Florida.

[LINW93] Lin, W., and Myklebust, A., "A Constraint Driven Solid Modeling Open Environment", accepted for presentation at the Second ACM/IEEE Symposium on Solid Modeling and Applications, Montreal, Canada, May 19-21, 1993.

[MONT91] Montgomery, D. E., Keil, M. J., and Myklebust, A., "A PHIGS+ Model Rendering System for Simulation of Spatial Mechanisms", *Proceedings of the ASME International Computers in Engineering Conference*, Santa Clara, California, August 18-22, 1991, pp. 225-233.

[MYKL92a] Myklebust, A., Lin, W., Woyak, S., Feustel, C. D., Fleming, S., Grieshaber, M., "A Research Report to IBM Corporation", February 28, 1992.

[MYKL92b] Myklebust, A., Lin, W. H., Fleming, S., Feustel, C. D., Woyak, S., Jacobson, A., LeGal, T., "A Research Report to IBM Corporation", July 15, 1992.

[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.

[PENN92] Pennington, S. and Myklebust, A., "A CASE Approach to the Integration of CAD/CAM Systems", presented at and published in the Proceedings of the Fifth International Workshop on Software Engineering and its Applications, Toulouse, France, December 7-11, 1992.

[PRAT91] Prata, S., C++ Primer Plus, Mill Valley: Waite Group Press, 1991.

[SCHR92] Schrock, E. V., Jayaram, S., and Myklebust, A., "A PHIGS-Bases Spreadsheet for Conceptual Design", presented at and published in the Proceedings of the ASEE 5th International Conference on Engineering Computer Graphics and Descriptive Geometry, Melbourne, Australia, August 17-21, 1992.

[WAMP88a] Wampler, S. G., "Development of a CAD System for Automated Conceptual Design of Supersonic Aircraft", M.S. thesis, Mechanical Engineering Department, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, May 1988.

[WAMP88b] Wampler, S. G., Myklebust, A., Jayaram, S., and Gelhausen, P., "Improving Aircraft Conceptual Design - A PHIGS Interactive Graphics User Interface for ACSYNT", AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference, Atlanta, Georgia, September 7-9, 1988, paper no. AIAA-88-4481.

[WISS90] Wisskirchen, P., Object-Oriented Graphics, New York: Springer-Verlag, 1990.

[WOYA92] Woyak, S. "A Motif-Like Object-Oriented Interface Framework Using PHIGS", Masters Thesis, Mechanical Engineering Department, VPI&SU, Blacksburg, Virginia, September 1992.

[WOYA93] Woyak, S., and Myklebust, A., "A Motif-Like Object-Oriented Interface Framework Using PHIGS", 1st Annual PHIGS User's Group Conference, March 21-24, 1993, Orlando, Florida.

Appendix A. Class Library User Guide

A.1 Graph Classes

A.1.1 Bar Graph

Description

A bar graph is the class which controls the creation of a bar graph.

Appearance

The graph will appear as a normal bar graph. It will have an axes, titles, legend, and be displayed in two dimensions.

Constructors

Standard Arguments:

None

Function Call Syntax:

`Bar_Graph(type, number_row, number_column, yarray, text_array)`

Argument Descriptions:

number_column - *integer*. Number of columns in the data array.

number_row - *integer*. Number of rows in the data array.

type - *integer*. Switch for whether graph is two dimensional or three dimensional display. Default is two.

yarray - *float*. Data values to be graphed.

text_array - *character*. Text strings of titles and legends.

Private Data Members:

axes - *pointer of type Axes*. Pointer to the axes.

bar_array - *array of pointers of type Bar*. Pointer to the bars of the bar graph.

deletion - *pointer of type Deletion*. Pointer to the deletion filter.

grid - *pointer of type Grid*. Pointer to grid class which contains xy grid.

invisible - *pointer of type Invisible*. Pointer to the class that controls the invisibility filter.

legend_array - *array of pointers of type Legend*. Pointer to the legends of the bar graph.

main_title - *pointer of type Text*. Pointer to text class which contains graph title.

max_color_index - *integer*. Number of colors in the color table.

xdata - *pointer of type Series*. Pointer to series class which contains x data values.

ydata - *pointer of type Series*. Pointer to series class which contains y data values.

y_title - *pointer of type Text*. Pointer to text class which contains y axes title.

yzgrid - *pointer of type Grid*. Pointer to grid class which contains yz grid.

zdata - *pointer of type Series*. Pointer to series class which contains z data values.

Member Functions

Public Functions:

void	drawBars() . Sets attributes for bars and displays them.
void	drawLegend() . Sets attributes for legend and displays it.
void	drawTitles() . Sets attributes for graph title and axes labels and displays them.
void	manage() . Controls graphing of bar graph.
void	processGeometryView(choice, x, y, view_index, depth, pick_path, event) . Determine which graph entity has been chosen.
void	recreateGraph() . Tell graph entities to recreate themselves.

A.1.2 Line Graph

Description

Line graph is the class which controls the creation of a line graph.

Appearance

The line graph will appear as a normal line graph. It will have an axes, titles, legend, and be displayed two dimensionally.

Constructors

Standard Arguments:

None

Function Call Syntax:

```
Line_Graph(number_row, number_column, xarray, yarray, text_array)
Line_Graph(number_row, number_column, xarray, yarray, zarray,
text_array)
```

Argument Descriptions:

number_column - *integer*. Number of columns in the data array. (Number of points per line)

number_row - *integer*. Number of rows in the data array. (Number of lines)

xarray - *float*. X data values to be graphed.

yarray - *float*. Y data values to be graphed.

zarray - *float*. Z data values to be graphed.

text_array - *character*. Text strings of titles and legends.

Private Data Members:

axes - *pointer of type Axes*. Pointer to the axes.

curve_array - *array of pointers of type Curve*. Pointer to the curves of the line graph.

curve_type - *integer*. Flag for whether curve to be point to point or spline curve.

deletion - *pointer of type Deletion*. Pointer to deletion filter.

grid - *pointer of type Grid*. Pointer to grid class which contains xy grid.

invisible - *pointer of type Invisible*. Pointer to the class that controls the invisibility filter.

legend_array - *array of pointers of type Legend*. Pointer to the legends of the bar graph.

main_title - *pointer of type Text*. Pointer to text class which contains graph title.

max_color_index - *integer*. Number of colors in the color table.

type - *integer*. Switch for whether graph is two dimensional or three dimensional display. Default is two.

xdata - *pointer of type Series*. Pointer to series class which contains x data values.

x_title - *pointer of type Text*. Pointer to text class which contains x axes title.

xzgrid - *pointer of type Grid*. Pointer to grid class which contains xz grid.

ydata - *pointer of type Series*. Pointer to series class which contains y data values.

y_title - *pointer of type Text*. Pointer to text class which contains y axes title.

yzgrid - *pointer of type Grid*. Pointer to grid class which contains yz grid.

zdata - *pointer of type Series*. Pointer to series class which contains z data values.

z_title - *pointer of type Text*. Pointer to text class which contains z axes title.

Member Functions

Public Functions:

void	draw_curves() . Set attributes for curves and display them.
void	draw_legend() . Set attributes for legend and display it.
void	draw_titles() . Set attributes for graph title and axes labels and display them.
void	manage() . Control graphing of line graph.
void	process_geometry_view(choice, x, y, view_index, depth, pick_path, event) . Determine which graph entity has been chosen.
void	recreate_graph() . Tell graph entities to recreate themselves.

A.1.3 Pie Graph

Description

Pie graph is the class which controls the creation of a pie graph.

Appearance

The pie graph will appear as a normal pie graph. It will have titles, a legend, and be displayed two dimensionally.

Constructors

Standard Arguments:

None

Function Call Syntax:

`Pie_Graph(type, number_row, number_column, yarray, text_array)`

Argument Descriptions:

number_column - *integer*. Number of columns in the data array.

number_row - *integer*. Number of rows in the data array.

type - *integer*. Switch for whether graph is two dimensional or three dimensional display. Default is two.

yarray - *float*. Y data values to be graphed.

text_array - *character*. Text strings for titles and legends.

Private Data Members:

legend_array - *array of pointers of type Legend*. Pointer to the legends of the bar graph.

main_title - *pointer of type Text*. Pointer to text class which contains graph title.

max_color_index - *integer*. Number of colors in the color table.

pie_array - *array of pointers of type Pie*. Pointer to the slices of the pie graph.

ydata - *pointer of type Series*. Pointer to series class which contains y data values.

Member Functions

Public Functions:

void	draw_legend(). Set legend attributes and display it.
void	draw_pie_slices(). Set pie slice attributes and display them.
void	draw_title(). Set attributes for graph title and display it.
void	manage(). Control drawing of pie graph.
void	process_geometry_view(choice, x, y, view_index, depth, pick_path, event). Determine which graph entity has been chosen.
void	recreate_graph(). Tell graph entities to recreate themselves.

A.1.4 Polar Graph

Description

Polar graph is the class which controls the creation of a polar graph.

Appearance

The polar graph will appear as a normal polar graph. It will have a title and a legend.

Constructors

Standard Arguments:

None

Function Call Syntax:

```
Polar_Graph(number_row, number_column, magnitude_array, degree_array,  
            text_array)
```

Argument Descriptions:

degree_array - *float*. Array containing degree values to be graphed.
number_column - *integer*. Number of columns in the data array.
number_row - *integer*. Number of rows in the data array.
magnitude_array - *float*. Array containing magnitudes to be graphed.
text_array - *character*. Text strings for titles and legends.

Private Data Members:

axes - *pointers of type Polar Axes*. Pointer to the polar axes class.
curve_array - *array of pointers of type Curve*. Pointer to the curves of the polar graph.
curve_type - *integer*. Flag for whether curves to be displayed point to point or with spline curve.
degree - *pointer of type Series*. Pointer to series class which contains degree data values.

invisible - *pointer of type Invisible*. Pointer to the class which controls the invisibility filter.

legend_array - *array of pointers of type Legend*. Pointer to legend class for the legends of the graph.

main_title - *pointer of type Text*. Pointer to text class which contains graph title.

max_color_index - *integer*. Number of colors in the color table.

magnitude - *pointer of type Series*. Pointer to series class which contains magnitude data values.

Member Functions

Public Functions:

void	draw_curves(). Set attributes for curves and display them.
void	draw_legend(). Set attributes for legend and display it.
void	draw_titles(). Set attributes for graph title and axes labels and display them.
void	manage(). Control drawing of polar graph.
void	process_geometry_view(choice, x, y, view_index, depth, pick_path, event). Determine which graph entity has been chosen.
void	recreate_graph(). Tell graph entities to recreate themselves.

A.1.5 Stack Bar Graph

Description

A stack bar graph is the class which controls the creation of a stack bar graph.

Appearance

The graph will appear as a normal stack bar graph. It will have an axes, titles, legend, and be displayed two dimensionally.

Constructors

Standard Arguments:

None

Function Call Syntax:

`Stack_Bar_Graph(type, number_row, number_column, yarray, text_array)`

Argument Descriptions:

number_column - *integer*. Number of columns in the data array.

number_row - *integer*. Number of rows in the data array.

type - *integer*. Switch for whether graph is two dimensional or three dimensional display. Default is two.

yarray - *float*. Data values to be graphed.

text_array - *character*. Text strings for titles and legends.

Private Data Members:

axes - *pointer of type Axes*. Pointer to the axes.

bar_array - *array of pointers of type Stack Bar*. Pointer to the bars of the bar graph.

deletion - *pointer of type Deletion*. Pointer to deletion filter.

grid - *pointer of type Grid*. Pointer to grid class which contains xy grid.

invisible - *pointer of type Invisible*. Pointer to the class that controls the invisibility filter.

legend_array - *array of pointers of type Legend*. Pointer to the legends of the bar graph.

main_title - *pointer of type Text*. Pointer to text class which contains graph title.

max_color_index - *integer*. Number of colors in the color table.

sum - *float*. Sum of data row for normalization of bars.

xdata - *pointer of type Series*. Pointer to series class which contains x data values.

ydata - *pointer of type Series*. Pointer to series class which contains y data values.

y_title - *pointer of type Text*. Pointer to text class which contains y axes title.

yzgrid - *pointer of type Grid*. Pointer to grid class which contains yz grid.

zdata - *pointer of type Series*. Pointer to series class which contains z data values.

Member Functions

Public Functions:

void	drawBars() . Set attributes for bars and draw them.
void	drawLegend() . Set attributes for legend and draw it.
void	drawTitles() . Set attributes for graph title and axes labels and draw them.
void	manage() . Control drawing of stack bar graph.
void	process_geometry_view(choice, x, y, view_index, depth, pick_path, event) . Determine which graph entity has been chosen.
void	recreate_graph() . Tell graph entities to recreate themselves.

A.2 Graph Entities

A.2.1 Axes

Description

An axes is the axes of the graph.

Appearance

Figure 1 shows the appearance of an axes. The attributes which can be controlled by the programmer are the lengths of the individual axes, color, linethickness, location, number of increments along each axes, and the legend height, color, and font.

Constructors

Standard Arguments:

AXES_STD_ARGS = x_start_pt, y_start_pt, z_start_pt, xlength, ylength, zlength

Function Call Syntax:

```
Axes()  
Axes(AXES_STD_ARGS)  
Axes(AXES_STD_ARGS, color_index)  
Axes(AXES_STD_ARGS, linewidth)  
Axes(AXES_STD_ARGS, linewidth, color_index)  
Axes(AXES_STD_ARGS, number_of_marks_x, number_of_marks_y,  
      number_of_marks_z)  
Axes(AXES_STD_ARGS, number_of_marks_x, number_of_marks_y,  
      number_of_marks_z, color_index)  
Axes(AXES_STD_ARGS, number_of_marks_x, number_of_marks_y,  
      number_of_marks_z, linewidth)  
Axes(AXES_STD_ARGS, number_of_marks_x, number_of_marks_y,  
      number_of_marks_z, linewidth, color_index)
```

Argument Descriptions:

color_index - *integer*. Index number of the color table entry you wish to use. Default is red.

linewidth - *float*. Scale factor by which the normal linethickness will be multiplied. Default is one.

number_of_marks_x - *integer*. Number of scale marks on the x axes. Default is six.

number_of_marks_y - *integer*. Number of scale marks on the y axes. Default is six.

number_of_marks_z - *integer*. Number of scale marks on the z axes. Default is six.

xlength - *float*. Length of the x axes. Default is 0.7.

x_start_pt[] - *float*. Array which holds the starting location for the x axes line.

ylength - *float*. Length of the y axes. Default is 0.7.

y_start_pt[] - *float*. Array which holds the starting location for the y axes line.

zlength - *float*. Length of the z axes. Default is 0.7.

z_start_pt[] - *float*. Array which holds the starting location for the z axes line.

Private Data Members:

axes_visible[] - *integer*. Array for holding name of set which contains axes visibility number.

cb_1 - *integer*. Switch for storing check box position.

class_name[] - *integer*. Array for holding entries for inclusion into name sets.

color_format - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.

graph - *pointer of type Graph*. Pointer to the graph.

hi_boundary_x - *float*. Maximum value of the x axes.

hi_boundary_y - *float*. Maximum value of the y axes.

hi_boundary_z - *float*. Maximum value of the z axes.

invisible - *pointer of type Invisible*. Pointer to the class that controls the invisibility filter.

label_id[] - *integer*. Array of label identifiers to be used by axes class for locating axes attributes during structure traversal.

low_boundary_x - *float*. Minimum value of the x axes.

low_boundary_y - *float*. Minimum value of the y axes.

low_boundary_z - *float*. Minimum value of the z axes.

managed - *integer*. Switch for whether a structure has already been created.

manager - *pointer of type Interface_Manager*. Pointer to interface manager.

polymark_scale_fact - *float*. Value by which the normal polymarker scale factor will be multiplied. Default is one.

structure - *PHIGS_Structure_ID*. Structure identifier into which axes will be drawn.

type - *integer*. Switch for whether graph is two dimensional or three dimensional display. Default is two.

view_index - *integer*. View index.

wsid - *integer*. Workstation identifier.

x_axes_manual - *integer*. Switch for whether calculated axes scale has been overwritten by the user.

xdata - *pointer of type Series*. Pointer to series class which contains x data values.

xstep - *float*. Value of the increment between the scale marks along the x axes.

x_text_color_index - *integer*. Value of the color table index number to be used for the x axes increment labels. Default is white.

x_text_height - *float*. Character height for the x axes increment labels. Default is 0.014.

y_axes_manual - *integer*. Switch for whether calculated axes scale has been overwritten by the user.

ydata - *pointer of type Series*. Pointer to series class which contains y data values.

ystep - *float*. Value of the increment between the scale marks along the y axes.

y_text_color_index - *integer*. Value of the color table index number to be used for the y axes increment labels. Default is white.

y_text_height - *float*. Character height for the y axes increment labels. Default is 0.014.

z_axes_manual - *integer*. Switch for whether calculated axes scale has been overwritten by the user.

zdata - *pointer of type Series*. Pointer to series class which contains z data values.

zstep - *float*. Value of the increment between the scale marks along the z axes.

z_text_color_index - *integer*. Value of the color table index number to be used for the z axes increment labels. Default is white.

z_text_height - *float*. Character height for the z axes increment labels. Default is 0.014.

Member Functions

Public Functions:

void	set_color(color_index)
void	set_linewidth(linewidth)
void	set_pointer_to_data_series(xdata, ydata, zdata)
void	set_pointer_to_graph(graph)
void	set_pointer_to_invisibility_filter(invisible)
void	set_structure_id(id)
void	set_type(type)
void	set_x_text_color(x_text_color_index)
void	set_x_text_height(x_text_height)
void	set_y_text_color(y_text_color_index)
void	set_y_text_height(y_text_height)
void	set_z_text_color(z_text_color_index)
void	set_z_text_height(z_text_height)

Inquiry Functions:

int	get_structure_id()
float	get_x_axes_hi_boundary()
float	get_x_axes_low_boundary()
float	get_x_axes_length()
float	get_x_axes_start_pt()
float	get_y_axes_hi_boundary()
float	get_y_axes_low_boundary()
float	get_y_axes_length()
float	get_y_axes_start_pt()
float	get_z_axes_hi_boundary()
float	get_z_axes_low_boundary()
float	get_z_axes_length()
float	get_z_axes_start_pt()

A.2.2 Bar

Description

A bar is a bar on a bar graph.

Appearance

Figure 2 shows the appearance of a bar. The attributes which can be controlled by the programmer are the color, width, interior pattern, and location.

Constructors

Standard Arguments:

BAR_STD_ARGS = type, yvalue, bar_location, bar_width

Function Call Syntax:

```
Bar()  
Bar(BAR_STD_ARGS)  
Bar(BAR_STD_ARGS, color_index)  
Bar(BAR_STD_ARGS, color_index, interior_pattern)
```

Argument Descriptions:

bar_location[] - *float*. Location of bar on the graph.

bar_width[] - *float*. Width of bar.

color_index - *integer*. Index number of the color table entry you wish to use. Default is red.

interior_pattern - *integer*. Type of pattern to be used in the fill area of the bar.

type - *integer*. Switch for whether graph is two dimensional or three dimensional display. Default is two.

yvalue - *float*. Normalized height value for the bar.

Private Data Members:

axes - *pointer of type Axes*. Pointer to the axes.

class_name[] - *integer*. Array for holding entries for inclusion into name sets.

color_format - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.

column_count - *integer*. Counter for the which bar is presently being drawn.

deletion - *pointer of type Deletion*. Pointer to the deletion filter.

graph - *pointer of type Graph*. Pointer to the graph.

interior_style - *integer*. Type of style to be used in the fill area of the bar. Default is solid.

label_id[] - *integer*. Array of label identifiers to be used by bar class for locating bar attributes during structure traversal.

managed - *integer*. Switch for whether a structure has already been created.

manager - *pointer of type Interface_Manager*. Pointer to interface manager.

number_column - *integer*. Number of columns in the data series being graphed. (Number of bars per row)

number_row - *integer*. Number of rows in the data series being graphed. (Number of sets to be graphed)

row_count - *integer*. Counter for the which row is presently being drawn.

structure - *PHIGS_Structure_ID*. Structure identifier into which axes will be drawn.

view_index - *integer*. View index.

wsid - *integer*. Workstation identifier.

xdata - *pointer of type Series*. Pointer to series class which contains x data values.

y_bar_manual - *integer*. Switch for whether y value was manually sent in by the user or if data is being read from array and normalized.

ydata - *pointer of type Series*. Pointer to series class which contains y data values.

z_bar_width - *float*. Width of bar along the z axes.

zdata - *pointer of type Series*. Pointer to series class which contains z data values.

Member Functions

Public Functions:

void	set_bar_location(bar_location)
void	set_bar_type(type)
void	set_color(color_index)
void	set_column_count(column_count)
void	set_interior_pattern(interior_pattern)
void	set_interior_style(interior_style)
void	set_pointer_to_axes(axes)

void	set_pointer_to_data_series(xdata, ydata, zdata)
void	set_pointer_to_deletion_filter(deletion)
void	set_pointer_to_graph(graph)
void	set_total_number_bars(number_column)
void	set_total_number_sets(number_row)
void	set_row_count(row_count)

Inquiry Functions:

int	get_color_index()
int	get_interior_pattern()
int	get_interior_style()
int	get_structure_id()

A.2.3 Curve

Description

A curve is a curve on the graph.

Appearance

Figure 3 shows the appearance of a curve. The attributes which can be controlled by the programmer are the color, linewidth, linetype, marker type, marker scale factor, and type of curve.

Constructors

Standard Arguments:

`CURVE_STD_ARGS = row_count, number_column, xdata, ydata, zdata`

Function Call Syntax:

```
Curve()  
Curve(CURVE_STD_ARGS)  
Curve(CURVE_STD_ARGS, color_index)  
Curve(CURVE_STD_ARGS, linewidth)  
Curve(CURVE_STD_ARGS, linewidth, color_index)  
Curve(AXES_STD_ARGS, linewidth, curve_type, color_index)
```

Argument Descriptions:

color_index - *integer*. Index number of the color table entry you wish to use. Default is red.

curve_type - *integer*. Switch for whether curve to be point to point or spline. Default is point to point.

deletion - *pointer of type Deletion*. Pointer to deletion filter.

linewidth - *float*. Scale factor by which the normal linethickness will be multiplied. Default is one.

number_column - *integer*. Number of columns in the data series being graphed. (Number of points per line)

row_count - *integer*. Number of rows in the data series being graphed. (Number of lines)

xdata - *pointer of type Series*. Pointer to series class which contains x data values.
ydata - *pointer of type Series*. Pointer to series class which contains y data values.
zdata - *pointer of type Series*. Pointer to series class which contains z data values.

Private Data Members:

axes - *pointer of type Axes*. Pointer to axes class.
class_name[] - *integer*. Array for holding entries for inclusion into name sets.
color_format - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.
graph - *pointer of type Graph*. Pointer to the graph.
label_id[] - *integer*. Array of label identifiers to be used by curve class for locating curve attributes during structure traversal.
linetype - *integer*. Switch for which linetype should be used. Default is solid.
line_visible[] - *integer*. Array for holding name of set which contains curve visibility number.
managed - *integer*. Switch for whether a structure has already been created.
manager - *pointer of type Interface_Manager*. Pointer to interface manager.
marker_type - *integer*. Switch for which marker type should be used.
marker_visible[] - *integer*. Array for holding name of set which contains marker visibility number.
polar_axes - *pointer of type Polar_Axes*. Pointer to Polar_Axes class.
polymark_scale_fact - *float*. Value by which the normal polymarker scale factor will be multiplied.
structure - *PHIGS_Structure_ID*. Structure identifier into which axes will be drawn.
view_index - *integer*. View index.
wsid - *integer*. Workstation identifier.

Member Functions

Public Functions:

void	set_color(color_index)
void	set_curve_type(curve_type)
void	set_graph_type(type)
void	set_linetype(linetype)
void	set_linewidth(linewidth)

void	set_marker_scale_factor(polymark_scale_fact)
void	set_marker_type(marker_type)
void	set_number_column(number_column)
void	set_pointer_to_axes(axes)
void	set_pointer_to_data_series(xdata, ydata, zdata)
void	set_pointer_to_deletion_filter(deletion)
void	set_pointer_to_graph(graph)
void	set_pointer_to_polar_axes(polar_axes)
void	set_row_count(row_count)

Inquiry Functions:

int	get_line_color()
int	get_linetype()
int	get_marker_type()
float	get_marker_scale()
int	get_structure_id()

A.2.4 Grid

Description

A grid is a grid on the graph.

Appearance

Figure 4 shows the appearance of a grid. The attributes which can be controlled by the programmer are the color, linethickness, vertical linetype, horizontal linetype, and the density of the grid.

Constructors

Standard Arguments:

GRID_STD_ARGS = orientation, x_start_pt, y_start_pt, z_start_pt, xlength, ylength, zlength

Function Call Syntax:

```
Grid()  
Grid(GRID_STD_ARGS)  
Grid(GRID_STD_ARGS, number_x_axes_divisions, number_y_axes_divisions )  
Grid(GRID_STD_ARGS, number_x_axes_divisions, number_y_axes_divisions,  
    color_index )  
Grid(GRID_STD_ARGS, number_x_axes_divisions, number_y_axes_divisions,  
    linewidth )  
Grid(GRID_STD_ARGS, number_x_axes_divisions, number_y_axes_divisions,  
    linewidth, color_index )
```

Argument Descriptions:

color_index - *integer*. Index number of the color table entry you wish to use. Default is gray.

linewidth - *float*. Scale factor by which the normal linethickness will be multiplied. Default is one.

number_x_axes_divisions - *integer*. Number of divisions along the x axes. Default is twenty-five.

number_y_axes_divisions - *integer*. Number of divisions along the y axes. Default is twenty-five.

orientation - *integer*. Switch for designating which plane the grid is to be drawn into. Default is XY.

xlength - *float*. Length of the x axes.

x_start_pt[] - *float*. Array which holds the starting location for the x axes line.

ylength - *float*. Length of the y axes.

y_start_pt[] - *float*. Array which holds the starting location for the y axes line.

zlength - *float*. Length of the z axes.

z_start_pt[] - *float*. Array which holds the starting location for the z axes line.

Private Data Members:

axes - *pointer of type Axes*. Pointer to axes class.

cb_1 - *integer*. Switch for storing check box position.

cb_2 - *integer*. Switch for storing check box position.

cb_3 - *integer*. Switch for storing check box position.

class_name[] - *integer*. Array for holding entries for inclusion into name sets.

color_format - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.

horizontal_linetype - *integer*. Type of line to be used for horizontal grid lines. Default is solid.

invisible - *pointer of type Invisible*. Pointer to the class that controls the invisibility filter.

label_id[] - *integer*. Array of label identifiers to be used by grid class for locating grid attributes during structure traversal.

managed - *integer*. Switch for whether a structure has already been created.

manager - *pointer of type Interface_Manager*. Pointer to interface manager.

vertical_linetype - *integer*. Type of line to be used for vertical grid lines. Default is solid.

view_index - *integer*. View index.

wsid - *integer*. Workstation identifier.

x_grid_structure - *PHIGS_Structure_ID*. Structure identifier into which x grid lines will be drawn.

xy_x_grid_visible[] - *integer*. Array for holding name of set which contains grid line visibility number.

xy_y_grid_visible[] - *integer*. Array for holding name of set which contains grid line visibility number.

xz_x_grid_visible[] - *integer*. Array for holding name of set which contains grid line visibility number.

xz_y_grid_visible[] - *integer*. Array for holding name of set which contains grid line visibility number.

y_grid_structure - *PHIGS_Structure_ID*. Structure identifier into which y grid lines will be drawn.

yz_x_grid_visible[] - *integer*. Array for holding name of set which contains grid line visibility number.

yz_y_grid_visible[] - *integer*. Array for holding name of set which contains grid line visibility number.

Member Functions

Public Functions:

void	set_color(color_index)
void	set_horizontal_linetype(horizontal_linetype)
void	set_linewidth(linewidth)
void	set_pointer_to_axes(axes)
void	set_pointer_to_invisibility_filter(invisible)
void	set_structure_id(id)
void	set_vertical_linetype(vertical_linetype)

Inquiry Functions:

int	get_x_grid_structure_id()
int	get_y_grid_structure_id()

A.2.5 Legend

Description

A legend is the legend box on the graph.

Appearance

Figure 4 shows the appearance of a legend. The attributes which can be controlled by the programmer are the color, linethickness, linetype, marker size, marker type, interior style, interior pattern, text color, text font, legend height, and legend width.

Constructors

Standard Arguments:

LEGEND_STD_ARGS = type, location, legend_width, legend_height,
text_string

Function Call Syntax:

Legend()
Legend(LEGEND_STD_ARGS)

Argument Descriptions:

legend_height - *float*. Height of legend area.
legend_width - *float*. Width of the legend area.
location - *float*. Location of the legend area.
text_string - *character*. Legend text string.
type - *integer*. Switch for whether legend is for a curve or a bar.

Private Data Members:

class_name[] - *integer*. Array for holding entries for inclusion into name sets.
color_format - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.
color_index - *integer*. Index number of the color table entry you wish to use. Default is red.

deletion - *pointer of type Deletion*. Pointer to deletion filter.
interior_pattern - *integer*. Interior pattern for fill area of bar legend.
interior_style - *integer*. Interior style pattern for fill area of bar legend. Default is solid.
linewidth - *float*. Scale factor by which the normal linethickness will be multiplied. Default is one.
linetype - *integer*. Type of line to be drawn. Default is solid.
legend_visible[] - *integer*. Array for holding name of set which contains legend visibility number.
label_id[] - *integer*. Array of label identifiers to be used by legend class for locating legend attributes during structure traversal.
managed - *integer*. Switch for whether a structure has already been created.
manager - *pointer of type Interface_Manager*. Pointer to interface manager.
marker_type - *integer*. Type of marker to be drawn.
polymark_scale_fact - *float*. Value by which the normal polymarker scale factor will be multiplied.
string_location - *float*. Location of legend text string.
structure - *PHIGS_Structure_ID*. Structure identifier into which axes will be drawn.
view_index - *integer*. View index.
wsid - *integer*. Workstation identifier.
text_color - *integer*. Value of the color table index number to be used for the legend text color. Default is white.
text_font - *integer*. Character font for the legend text. Default is one.
text_height - *float*. Character height for the legend text. Default is 0.011.

Member Functions

Public Functions:

void	set_graph_type(type)
void	set_interior_color(color_index)
void	set_interior_pattern(interior_pattern)
void	set_interior_style(interior_style)
void	set_legend_height(legend_height)
void	set_legend_location(location)
void	set_legend_text(text_string)
void	set_legend_width(legend_width)
void	set_line_color(color_index)
void	set_linetype(linetype)
void	set_linewidth(linewidth)

void	set_marker_scale_factor(polymark_scale_fact)
void	set_marker_type(marker_type)
void	set_pointer_to_deletion_filter(deletion)
void	set_text_color(text_color)
void	set_text_font(text_font)
void	set_text_height(text_height)

Inquiry Functions:

int	get_structure_id()
-----	--------------------

A.2.6 Pie

Description

A pie is a pie slice in a pie graph.

Appearance

Figure 5 shows the appearance of a pie. The attributes which can be controlled by the programmer are the color, height, radius, interior pattern, interior style and number of degrees slice represents.

Constructors

Standard Arguments:

PIE_STD_ARGS = type, starting_angle, yvalue

Function Call Syntax:

Pie()
Pie(PIE_STD_ARGS)
Pie(PIE_STD_ARGS, height)
Pie(PIE_STD_ARGS, height, radius)

Argument Descriptions:

height - *float*. Height of pie slice in three dimensional display. Default is 0.3.
radius - *float*. Radius of pie slice. Default is 0.25.
starting_angle - *integer*. Angle from which pie slice will be started. Default is zero.
type - *integer*. Switch for whether graph is two dimensional or three dimensional display. Default is two.
yvalue - *float*. Number of degrees slice represents.

Private Data Members:

class_name[] - *integer*. Array for holding entries for inclusion into name sets.

color_format - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.

color_index - *integer*. Index number of the color table entry you wish to use. Default is red.

column_count - *integer*. Counter for the which bar is presently being drawn.

graph - *pointer of type Graph*. Pointer to the graph.

interior_pattern - *integer*. Type of pattern to be used in the fill area of the bar.

interior_style - *integer*. Type of style to be used in the fill area of the bar. Default is solid.

label_id[] - *integer*. Array of label identifiers to be used by pie class for locating pie attributes during structure traversal.

managed - *integer*. Switch for whether a structure has already been created.

manager - *pointer of type Interface_Manager*. Pointer to interface manager.

number_degree - *integer*. Number of degrees pie slice will represent.

origin - *float*. Location of the pie graph origin. Default is 0.4, 0.45, 0.5.

slice_manual - *integer*. Switch for whether number of degrees needs to be calculated.

structure - *PHIGS_Structure_ID*. Structure identifier into which axes will be drawn.

view_index - *integer*. View index.

wsid - *integer*. Workstation identifier.

ydata - *pointer of type Series*. Pointer to series class which contains y data values.

Member Functions

Public Functions:

void	set_color(color_index)
void	set_column_count(column_count)
void	set_height(height)
void	set_interior_pattern(interior_pattern)
void	set_interior_style(interior_style)
void	set_pie_type(type)
void	set_pointer_to_data_series(ydata)
void	set_pointer_to_graph(graph)
void	set_origin(origin)
void	set_radius(radius)
void	set_starting_angle(starting_angle)

Inquiry Functions:

int	get_color_index()
int	get_interior_pattern()
int	get_interior_style()
int	get_number_degree()
int	get_structure_id()

A.2.7 Polar_Axes

Description

A polar axes is the axes for a polar graph.

Appearance

Figure 6 shows the appearance of a polar axes. The attributes which can be controlled by the programmer are the radius of the axes, color, linethickness, location, and number of increments.

Constructors

Standard Arguments:

POLAR_STD_ARGS = origin, radius

Function Call Syntax:

```
Polar_Axes()  
Polar_Axes(POLAR_STD_ARGS)  
Polar_Axes(POLAR_STD_ARGS, color_index)  
Polar_Axes(POLAR_STD_ARGS, linewidth)  
Polar_Axes(POLAR_STD_ARGS, linewidth, color_index)
```

Argument Descriptions:

color_index - *integer*. Index number of the color table entry you wish to use. Default is red.

linewidth - *float*. Scale factor by which the normal linethickness will be multiplied. Default is one.

origin - *float*. Origin of the polar axes. Default is 0.45, 0.45, 0.0.

graph_radius - *float*. Radius of the polar axes. Default is 0.35.

Private Data Members:

axes_visible[] - *integer*. Array for holding name of set which contains axes visibility number.

cb_1 - *integer*. Switch for storing check box position.

class_name[] - *integer*. Array for holding entries for inclusion into name sets.

color_format - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.

degree - *pointer of type Series*. Pointer to array containing the degrees.

graph - *pointer of type Graph*. Pointer to the graph.

hi_magnitude - *float*. Maximum value of the magnitudes.

invisible - *pointer of type Invisible*. Pointer to the class that controls the invisibility filter.

label_id[] - *integer*. Array of label identifiers to be used by polar axes class for locating polar axes attributes during structure traversal.

low_magnitude - *float*. Minimum value of the magnitudes.

magnitude - *pointer of type Series*. Pointer to array containing the magnitudes.

managed - *integer*. Switch for whether a structure has already been created.

manager - *pointer of type Interface_Manager*. Pointer to interface manager.

number_circles - *float*. Number of circles on the axes. Default is four.

structure - *PHIGS_Structure_ID*. Structure identifier into which axes will be drawn.

view_index - *integer*. View index.

wsid - *integer*. Workstation identifier.

x_axes_manual - *integer*. Switch for whether calculated axes scale has been overwritten by the user.

xstep - *float*. Value of the increment between the increment marks along the x axes.

x_text_color_index - *integer*. Value of the color table index number to be used for the x axes increment labels. Default is white.

x_text_height - *float*. Character height for the x axes increment labels. Default is 0.014.

Member Functions

Public Functions:

void	set_color(color_index)
void	set_graph_radius(graph_radius)
void	set_linewidth(linewidth)
void	set_number_circles(number_circles)
void	set_pointer_to_data_series(magnitude, degree)
void	set_pointer_to_graph(graph)
void	set_pointer_to_invisibility_filter(invisibility)
void	set_x_text_color(x_text_color)

void	set_x_text_height(x_text_height)
------	----------------------------------

Inquiry Functions:

float	get_graph_radius()
float	get_hi_magnitude()
float	get_low_magnitude()
int	get_structure_id()
float	get_x_origin_pt()
float	get_y_origin_pt()
float	get_z_origin_pt()

A.2.8 Series

Description

Class for storing and making calculations with data values.

Private Data Members:

array - *float*. Array for storing data values.

number_row - *integer*. Number of rows.

number_column - *integer*. Number of columns.

Member Functions

Public Functions:

void set_data(number_row, number_column, data)

Inquiry Functions:

float	get_maximum_data_value()
float	get_maximum_row_sum()
float	get_minimum_data_value()
float	get_normalized_data_value(high_boundary, low_boundary, length, row, column)
int	get_normalized_pie_data_value(total_sum, row, column)
int	get_number_columns()
int	get_number_rows()
float	get_value(row, column)

A.2.9 *Stack_Bar*

Description

A stack bar is a stack bar on a stack bar graph.

Appearance

Figure 7 shows the appearance of a stack bar. The attributes which can be controlled by the programmer are the color, width, interior style, interior pattern and location.

Constructors

Standard Arguments:

STACK_BAR_STD_ARGS = type, yvalue, bar_location, bar_width

Function Call Syntax:

```
Stack_Bar()  
Stack_Bar(Stack_Bar_STD_ARGS)  
Stack_Bar(Stack_Bar_STD_ARGS, color_index)  
Stack_Bar(Stack_Bar_STD_ARGS, color_index, interior_pattern)
```

Argument Descriptions:

bar_location[] - *float*. Location of bar on the graph.

bar_width[] - *float*. Width of bar.

color_index - *integer*. Index number of the color table entry you wish to use. Default is red.

interior_pattern - *integer*. Type of pattern to be used in the fill area of the bar.

type - *integer*. Switch for whether graph is two dimensional or three dimensional display.

yvalue - *float*. Y data value bar represents.

Private Data Members:

axes - *pointer of type Axes*. Pointer to the axes.

class_name[] - *integer*. Array for holding entries for inclusion into name sets.
color_format - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.
column_count - *integer*. Counter for the which bar is presently being drawn.
counter - *integer*.
deletion - *pointer of type Deletion*. Pointer to deletion filter.
graph - *pointer of type Graph*. Pointer to the graph.
interior_style - *integer*. Type of style to be used in the fill area of the bar. Default is solid.
label_id[] - *integer*. Array of label identifiers to be used by stack bar class for locating stack bar attributes during structure traversal.
managed - *integer*. Switch for whether a structure has already been created.
manager - *pointer of type Interface_Manager*. Pointer to interface manager.
number_column - *integer*. Number of columns in the data series being graphed. (Number of bars per stack)
number_row - *integer*. Number of rows in the data series being graphed. (Number of stacks)
row_count - *integer*. Counter for the which row is presently being drawn.
structure - *PHIGS_Structure_ID*. Structure identifier into which axes will be drawn.
view_index - *integer*. View index.
wsid - *integer*. Workstation identifier.
xdata - *pointer of type Series*. Pointer to series class which contains x data values.
Y - *float*. Normalized y data value.
y_bar_manual - *integer*. Switch for whether y value was manually sent in by the user.
ydata - *pointer of type Series*. Pointer to series class which contains y data values.
Ylast - *float*. Records position to start next bar.
z_bar_width - *float*. Width of bar along the z axes.
zdata - *pointer of type Series*. Pointer to series class which contains z data values.

Member Functions

Public Functions:

void	set_bar_location(bar_location)
void	set_bar_type(type)
void	set_color(color_index)

void	set_column_count(column_count)
void	set_interior_pattern(interior_pattern)
void	set_interior_style(interior_style)
void	set_pointer_to_axes(axes)
void	set_pointer_to_data_series(xdata, ydata, zdata)
void	set_pointer_to_deletion_filter(deletion)
void	set_pointer_to_graph(graph)
void	set_row_count(row_count)
void	set_total_numberBars(number_column)
void	set_total_number_sets(number_row)
void	set_Ylast(Ylast)

Inquiry Functions:

float	get_bar_height()
int	get_color_index()
int	get_interior_pattern()
int	get_interior_style()
int	get_structure_id()

A.2.10 Text

Description

Text is the text located on the graph.

Appearance

Figure 8 shows the appearance of text. The attributes which can be controlled by the programmer are the text, text color, text height, and location.

Constructors

Standard Arguments:

TEXT_STD_ARGS = x_location, y_location, z_location, text_string

Function Call Syntax:

```
Text()  
Text(TEXT_STD_ARGS)
```

Argument Descriptions:

- text_string[]** - *character*. Text string.
- x_location** - *float*. Array which holds the x position of the text string.
- y_location** - *float*. Array which holds the y position of the text string.
- z_location** - *float*. Array which holds the z position of the text string.

Private Data Members:

- class_name[]** - *integer*. Array for holding entries for inclusion into name sets.
- color_format** - *integer*. Specifies whether using color table entries(indexed) or direct color values. Default is indexed.
- invisible** - *pointer of type Invisible*. Pointer to the class that controls the invisibility filter.
- label_id[]** - *integer*. Array of label identifiers to be used by text class for locating text attributes during structure traversal.
- location[]** - *float*. Location of text string.
- managed** - *integer*. Switch for whether a structure has already been created.

manager - *pointer of type Interface_Manager*. Pointer to interface manager.
structure - *PHIGS_Structure_ID*. Structure identifier into which axes will be drawn.
text_color - *integer*. Value of the color table index number to be used for the text. Default is red.
text_font - *integer*. Text font to be used. Default is one.
text_height - *float*. Character height for the text.
text_visible[] - *integer*. Array for holding name of set which contains text visibility number.
view_index - *integer*. View index.
wsid - *integer*. Workstation identifier.

Member Funtions

Public Functions:

void	set_pointer_to_invisibility_filter(invisible)
void	set_text_color(text_color)
void	set_text_font(text_font)
void	set_text_height(text_height)

Inquiry Functions:

int	get_structure_id()
-----	--------------------

A.3 Control Functions

A.3.1 Deletion

Description

Class for control of the deletion filter.

Constructors

Function Call Syntax:

Deletion()

Private Data Members:

incl - *integer*. Array containing structure identifiers to be included in the deletion filter.

inclen - *integer*. Length of the inclusion array(incl).

Member Functions

Public Functions:

void add_to_deletion_filter(structure)

Inquiry Functions:

int check_deletion_array(size)

void check_deletion_array(structure)

A.3.2 Highlight

Description

Class for control of the highlighting filter.

Constructors

Standard Arguments:

HIGHLIGHT_STD_ARGS = wsid, inclen, incl, exclen, excl

Function Call Syntax:

Highlight()
Highlight(HIGHLIGHT_STD_ARGS)
Highlight(wsid)

Argument Descriptions:

excl - *integer*. Array containing structure identifiers to be excluded from the highlighting filter.
exclen - *integer*. Length of the exclusion array(excl).
incl - *integer*. Array containing structure identifiers to be included in the highlighting filter.
inclen - *integer*. Length of the inclusion array(incl).
wsid - *integer*. Workstation identifier.

Private Data Members:

errind - *integer*. Error indicator from request.
errind = 0: successful inquiry of highlighting filter state completed.
errind = 1: inquire inclusion filter length less than inclusion array length.
errind = 2: inquire exclusion filter length less than exclusion array length.
errind = 3: inquire inclusion and inquire exclusion filter lengths less than inclusion and exclusion array lengths.
exlen - *integer*. Length of the exclusion filter.
inlen - *integer*. Length of the inclusion filter.

Member Functions

Public Functions:

void	add_to_highlighting_filter(wsid, ainclen, aincl)
void	exclude_from_highlighting_filter(wsid, aexclen, aexcl)

Inquiry Functions:

void	inquire_highlighting_filter_state(inq_inlen, inq_exlen, errind, inq_inclen, inq_incl, inq_exclen, inq_excl)
------	--

A.3.3 Invisible

Description

Class for control of the highlighting filter.

Constructors

Standard Arguments:

INVISIBLE_STD_ARGS = wsid, inclen, incl, exclen, excl

Function Call Syntax:

Invisible()
Invisible(INVISIBLE_STD_ARGS)
Invisible(wsid)

Argument Descriptions:

excl - *integer*. Array containing structure identifiers to be excluded from the invisibility filter.
exclen - *integer*. Length of the exclusion array(excl).
incl - *integer*. Array containing structure identifiers to be included in the invisibility filter.
inclen - *integer*. Length of the inclusion array(incl).
wsid - *integer*. Workstation identifier.

Private Data Members:

errind - *integer*. Error indicator from request.
errind = 0: successful inquiry of invisibility filter state completed.
errind = 1: inquire inclusion filter length less than inclusion array length.
errind = 2: inquire exclusion filter length less than exclusion array length.
errind = 3: inquire inclusion and inquire exclusion filter lengths less than inclusion and exclusion array lengths.
exlen - *integer*. Length of the exclusion filter.
inlen - *integer*. Length of the inclusion filter.

Member Functions

Public Functions:

void	add_to_invisibility_filter(wsid, ainclen, aincl)
void	exclude_from_invisibility_filter(wsid, aexclen, aexcl)

Inquiry Functions:

void	inquire_invisibility_filter_state(inq_inlen, inq_exlen, errind, inq_inclen, inq_incl, inq_exclen, inq_excl)
------	--

Vita

R. Steven Uhorchak was born on January 30, 1968 and lived in the small town of Canton, New York until he was eleven. His family then moved to Cary, North Carolina, a suburb of Raleigh, and are still living their today. With a strong interest in automobiles, engines, mechanical systems, and computers the author enrolled in the University of North Carolina at Charlotte's Mechanical Engineering program and completed a Bachelor of Science in May 1991. He then completed the requirements for a Master of Science in Mechanical Engineering at Virginia Polytechnic Institute and State University. The author now hopes to begin his engineering career with a consulting engineering or construction firm.

A handwritten signature in black ink that reads "R. Steven Uhorchak". The signature is written in a cursive, flowing style with a large, stylized 'h' at the end.