An Object Oriented Curve and Surface Framework

by

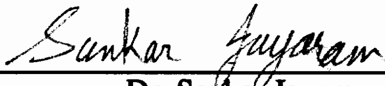
Alan L. Jacobson

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. Arvid Myklebust



Dr. Sankar Jayaram

Dr. Charles Reinholtz

May 4, 1993
Blacksburg, Virginia

c.2

LD
5655
V855
1993
J336
c.2

Abstract

Each year, thousands of custom CAD/CAM/CAE applications are being created to meet the needs of industry, education and research. To facilitate this process, programmers should have available to them high-level tools which are device-independent. These enablers will allow for the rapid creation of highly portable software.

From this need, a curve and surface framework is created. This framework allows the engineer to create, manipulate, and analyze geometric entities. The design allows new curve and surface implementations to be fully integrated into the framework without requiring modifications to other entities within the framework. The framework utilizes the International Standards Organization (ISO) and American National Standard Institute (ANSI) 3-D graphics standard, PHIGS, and is thus device independent. Furthermore, the use of the IGES data exchange standard enables the transfer of data through files with an ISO/ANSI/ASME accepted format.

Five curve classes and two surface classes are implemented to demonstrate the potential of the framework. The curve classes include a Line, Polyline, Arc, Cubic Bézier, and Non-Uniform Cubic B-Spline Curve classes, and the surface classes include the Non-Uniform Cubic B-Spline Surface and the Planar Surface classes.

The result of the design is a highly extensible and maintainable framework for geometric modeling which reduces the work required by programmers in the creation and integration of geometric entities in complex software.

Acknowledgments

Throughout my education, there has never been a more constant guiding force than my family. I thank both my mother and father for their continual understanding and support.

Perhaps the most influential person in my engineering curriculum was my first professor in mechanical engineering, Dr. Robert Jerard. Through the classwork, research, and friendship that developed during my undergraduate years at the University of New Hampshire, he taught me an immeasurable amount.

And now, with my current research, I have joined a new research team at the Virginia Tech CAD Laboratory. I would like to thank Dr. Arvid Myklebust who provided me with a stimulating environment and many interesting projects.

The financial support for my research came from an IBM grant. I would like to thank IBM for providing me the resources to continue my education, and equipment to perform the research on.

Throughout the development of this project, I was able to collect valuable input from each of the members of my advisory committee. I would like to thank Dr. Myklebust, Dr. Jayaram, and Dr. Reinholtz for their guidance.

Finally I would like to thank all of the members of the CAD Laboratory who helped me both in my research, and in taking a break from it all. I would especially like to thank

Brett, Andy, Scott, and Shahab, each of your friendships provided tremendous support, I will remember all of you dearly. I wish you all good luck in completing your academic pursuits, and in continuing your professional development.

Table of Contents

Introduction 1

Background 3

Problem Definition 8

Goals
- Ease of use 10
- Use of graphics standards 11
- Close analogy to physical reality 11
- Extensibility 12

Literature Survey
- Curves 13
- Surfaces 15
- Object-Oriented Programming 16
- Frameworks 17

Program Development 19

Object-Oriented Programming
- Inheritance 24
- Encapsulation 24
- Polymorphism 25

Class Overview
- Points and Vectors
  - Point Class 28
  - Vector Class 30
- Curves
  - Curve Base Class 36
  - Curve Class Hierarchy 39
  - Derived Classes 44
  - Arc 47
    - Construction 47
    - Set Methods 48
    - Curve Inquiry 49
    - Virtual Get Methods 49
    - PHIGS Methods 50
  - Cubic Bézier 51
    - Construction 51
    - Virtual Set Methods 52
    - Virtual Get Methods 52

PHIGS Methods....53
Utility Methods....54

Non-Uniform B-Spline Curve....55
Construction....58
Internal Methods....59
Set Methods....60
Virtual Set Methods....61
Virtual Get Methods....61
Curve Inquiry....62
Analysis Methods....62
PHIGS Methods....63
Utility Methods....64
Operators....64

Line....65
Construction....65
Virtual Set Methods....65
Virtual Get Methods....66
Set Methods....66
PHIGS Methods....66
Analysis Methods....67

Polyline....68
Construction....68
Internal Methods....69
Virtual Set Methods....69
Virtual Get Methods....70
PHIGS Methods....70

Abstract Curve Classes....72

Surfaces
Surface Class Hierarchy....72
Surface Base Class....73
Derived Classes....74

Non-Uniform B-Spline Surface....76
Multiple Cross-Section Sweep....78
Single Cross-Section Swept Along Guide Curve....81
Surface of Revolution....83
Control Point Specification....85
Constructors....85
Internal Methods....86
Virtual Set Methods....87
Virtual Get Methods....88
PHIGS Methods....88
Analysis Methods....89

Planar Surface 90
Constructors 90
Internal Methods.......................................92
Virtual Set Methods 92
Virtual Get Methods...................................93
PHIGS Methods..94

Material Properties 95
Primitives..100

Tools Used 102

Sample Usage
Example 1 103
Example 2 106
Example 3 110
Example 4 115

Results...119

Conclusions...127

References..130

Appendix A: Curve Class
Curve Class...137
Arc Class 138
Cubic Bézier Class..140
Non-Uniform B-Spline (NUBS) Curve Class.........................143
Polyline Class...149
Line Class 152

Appendix B: Surface Class
Surface Class 156
Non-Uniform B-Spline (NUBS) Surface Class........................157
Planar Surface Class 162

Appendix C: Auxiliary Classes
Point Class...168
Vector Class..170
Material Property Class...172
Material Library Class 173
Direct Color Class...174
Direct Color Library Class 175

Vita...176

List of Illustrations

Figure 1. Highly Extensible Curve And Surface Framework 9
Figure 2. First Generation Of Framework Consisting of C Language Routines 20
Figure 3. Current Design Of Object Oriented Curve And Surface Framework 21
Figure 4. 3-Dimensional Rotation About An Arbitrary Axis 33
Figure 5. Utilizing The Curve Base Class In The Creation Of General Algorithms 38
Figure 6. Representation Of Derived Curves Using The Base Class 40
Figure 7. Curve and Surface Class Organization 41
Figure 8. Development of Curve Class Hierarchy 43
Figure 9. Construction Of Simple Primitives Using Cross-Sectional Sweeps 80
Figure 10. Multiple Sweeps Of A Single Cross-Section 82
Figure 11. Surfaces Of Revolution 84
Figure 12. Control Of Surface Properties 96
Figure 13. Construction Methods For Creating Simple Curves 105
Figure 14. Construction Methods For Creating Simple Surfaces 109
Figure 15. Construction Method For Creating A Surface Of Revolution 112
Figure 16. Construction Method For Creating A Swept Surface 116
Figure 17. NUBS Surface Representation Of Solid Model Utilizing The Solid Modeling Framework And The Curve And Surface Framework 121
Figure 18. Simple Curves Generated From The Framework 122
Figure 19. Simple Surface Generated From The Framework 123
Figure 20. Surface Of Revolution Generated From The Framework 124
Figure 21. Swept Surface Generated From The Framework 125
Figure 22. Free-Form Surface Generated From The Framework 126
Figure 23. Creating A Custom Engineering Design Modeler Using Object Oriented Enablers 129

Introduction

Computers have become increasingly powerful, as both the hardware and software that is being designed become more advanced. One of the most noticeable changes in computers is the use of graphics. It has been said that "A picture is worth a thousand words". However, until recently graphical methods were popular in computer software, due to both hardware and software limitations. Currently, the development of tools in areas such as Graphical User Interfaces (GUI's), Computer Aided Design, Manufacturing, and Engineering (CAD/CAM/CAE), scientific visualization, and many other graphically intensive areas are being made at an increasing rate.

As more programmers try to apply the new graphical technology to their problems, they face the challenge of learning new skills such as geometric modeling and Computer Aided Geometric Design (CAGD). But the need for the new programmers to become experts in geometric modeling may be reduced by providing a tool to solve their modeling needs.

The development of graphical standards, such as the graphical language PHIGS, have gone a long way towards facilitating the programmer's ability to use the graphical capabilities of the software, but higher level tools are still necessary [Jaya89] [Jaya90].

A curve and surface framework is created to address this need. This framework will allow the engineer to utilize the computer to create the graphical entities desired. Furthermore, the framework is designed to utilize the International Standards Organization (ISO) and American National Standard Institute (ANSI) 3-D graphics standard, PHIGS for graphics, and the International data exchange standard IGES. Thus, a high level tool is established

which will support a machine independent environment for the development of curve and surface applications.

Background

There are many ways of designing curves and surfaces in computer applications. A historical look at the development of curves and surfaces, as well as the more recent application of this technology using the computer, provides insight into how curves and surfaces are defined, and the problems many designers encounter.

People have studied and used curves and surfaces in great detail, as early as history was first recorded, and actually, in the recording of this early history itself. During the Greek and Roman empires, great achievements were made in the field of geometry. During this period the one-dimensional point, two-dimensional primitives such as lines, polygons, arcs, and circles, as well as three-dimensional primitives such as cubes, spheres, and pyramids were identified and developed. These basic primitives were used extensively in scientific tasks such as architecture, astronomy, and physics, as well as in many other fields such as philosophy and religion. Even today, these basic primitives are used extensively in engineering and science as well as a wide array of non-scientific fields.

The methods by which basic primitives are defined have evolved with time in many different ways. Definitions for primitives, such as the line, took on new meanings. Instead of defining a line segment with the geometric definition: all the points contained by placing a straight-edge between the two points, the line could be defined by an algebraic equation. With the development of parametric geometry, and parametric equations, the methodology varied once again. Currently there are a plethora of methods that can be used to define a line segment.

As the number of methods to perform a task increased, the flexibility and variability in the design process also increased. Unfortunately, with this increase in flexibility came an increase in the complexity of the model. This complexity was acknowledged very early in the birth of the CAD field. Along with the utilization of computers came an increase in the number of methodologies used. Each method had various advantages and disadvantages, and as old definitions were applied to CAD models, the need for new representations became necessary.

The roots of geometric modeling grew tremendously with the first computer graphics systems developed for CAD. Ivan Sutherland's Sketchpad system (1963) was one of the earliest pioneers in the field [Suth63].

It has been seen in numerous instances that the capabilities of CAD/CAM/CAE software are often directly related to the underlying curve and surface definitions. This is seen clearly in both the automobile and aerospace industries where a great amount of research has been devoted to geometric modeling. The development of new mathematical models for curves and surfaces and new computational methods is often a prerequisite to improving software.

Major research efforts in the area of Computer Aided Geometric Design (CAGD) began throughout industry, with leaders such as P. Bézier at Rénault, P. de Castlejau at Rénault, J.C. Ferguson at Boeing, and Coons at MIT. Many other companies such as Douglas, Lockheed, and McDonnell also made notable contributions. The CAD modeling of the era emphasized the appearance of geometric shapes, with the notable exception of finite element models [Mortenson, 1985].

At about the same time, initial work in the area of CAM began with Numerical Control (NC) machines. The NC process used a "subtractive model" to represent the machining of materials. In order to control the cutter, new ways of extracting shape data became necessary.

Already, the methods used to create geometric computer models varied greatly. Many companies used digitized data to create curves and surfaces, others used equations, and still others used keyboard input. Along with these different methods of entering data were various definitions of curves that could solve the problems of industry.

A major theme throughout this evolution in CAD was the need for smooth surfaces. From this desire for smooth surfaces came the use of new curve definitions such as splines, B-splines, and conics. Each of these methods solved different problems encountered in the design process.

Further variation occurred between fields such as CAM and CAE due to the attempt to solve different problems. This great diversity in curve definitions made the integration of various software and methodologies difficult.

With research being performed on different applications at a large number of companies, varying solutions were found. Furthermore, since many of the results were kept proprietary, at times it seemed that the field was diverging into several application specific areas that could never be integrated.

As problems in the field of CAD became more complex, the need for advances in computer languages was quickly realized. D.T. Ross developed an advanced compiler language

(1967) at MIT for graphics programming [Ross67]. Similarly, teams of researchers in industry began to develop their own graphical languages and drivers in order to display their research. Unfortunately, there was no standard to these graphical languages, and again, a variety of methods were used to perform similar tasks. This led to each piece of software being hardware dependent.

With rapid advances in computer hardware came a great difficulty in updating research projects to utilize the full potential of new machines. Furthermore, results were not easily shared between research groups, and different programs could not be easily integrated due to language incompatibilities.

Many of the problems may be attributed to the rapid growth of the CAD and CAGD fields during the early stages of its development. This growth was not an anomaly, but part of a tremendous growth in computer software, as hardware was created which had the power to solve new problems.

The solution to the problems caused by this large growth was answered in part by standardization. One of the first two-dimensional graphical standards was the CORE system in 1978. This standard, although accepted by ANSI, was later rejected by ISO. In 1985 the Graphical Kernel System (GKS) became the international (ISO) and American (ANSI) standard for two-dimensional graphics. The addition of the more advanced Programmer's Hierarchical Interactive Graphics Standard (PHIGS) in 1987, provided a three-dimensional graphics standard, and has since become an international standard for both two-dimensional and three-dimensional graphics. With the acceptance of these standards, computer graphics applications could be written in such a way as to provide

hardware independence and allow the integration of CAD/CAM/CAE applications more readily [Jaya90][Jaya93].

Currently more advanced languages are being developed and may become the new graphical standards. Examples of these languages include PHIGS+ and Object-Oriented PHIGS [Wamp91].

Along with these developments in graphical languages, data exchange standards were created to facilitate integration of software. Currently, the IGES standard has been used effectively to provide a stable neutral file platform. However, development of a new data exchange protocol, the Product Data Exchange using STEP, is underway which may serve to replace or supplement the IGES standard [Rank92].

In addition to both graphical language and data transfer standardization, a de facto standardization has occurred with programming languages. Early in the development of the CAGD field, the use of FORTRAN prevailed due to its availability and speed for arithmetic computations. As Unix became widely used, many programmers migrated to C due to its ability to efficiently and easily handle the many complexities which arose in large, complex programs. Currently, the development of object-oriented languages are again changing the de facto standard for graphical programming to a new fourth generation language C++.

Undoubtedly the short history of the field will be pushed further, and into new directions by the combined needs of industry, research, and academia. The development of an object-oriented curve and surface framework is viewed as a logical progression towards a successful future in the CAGD field.

Problem Derivation/Definition

Each year thousands of custom CAD applications are being written to satisfy the needs of industry, research, and education. The programmers of these applications are currently required to have a strong foundation in areas such as: geometric modeling, computational geometry, graphical languages, file transfer protocols, and many others. Furthermore, as these fields develop more advanced techniques, significant changes to the custom application may become necessary.

The need for high level tools for graphics is easily recognized. The development of an object oriented framework for the creation of curves and surfaces will provide a platform for programmers and developers to base their modeling needs. The framework should reduce the programmers need for expertise in geometric modeling, as well as provide tools to reduce the amount of coding necessary. The framework should also create more maintainable code that is easily extensible. Finally the developed framework should be device and language independent, allowing for portability across dissimilar systems.

Based on current standards in computer graphics, the framework should support programming languages such as FORTRAN, C, and the hybrid object-oriented language C++, which is supported by a large number of computer systems. In addition, the framework should utilize the international three-dimensional graphical language standard, PHIGS. Finally, the framework should facilitate the transfer of information through the international file transfer standard IGES. Figure 1 illustrates how the framework may provide a stable platform as changes occur in the evolution of computer languags, graphics standards, and file transfer protocols.

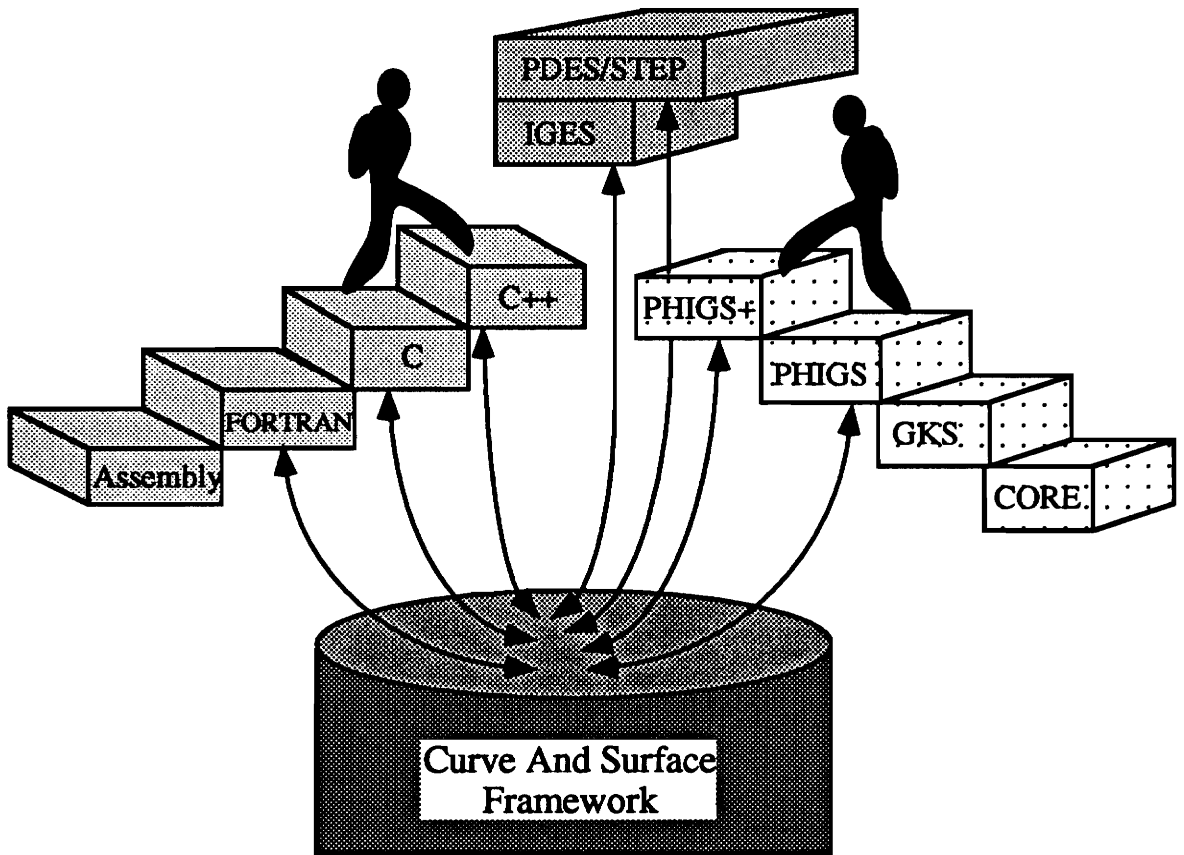


Figure 1 **Highly Extensible Curve and Surface Framework**

Goals

The ultimate goal of this thesis is the creation of an integrated environment which supports the creation of geometric entities. The environment should allow new entities to be fully integrated without requiring modification to existing code.

The creation of a curve and surface framework must meet high standards in several areas. The primary concerns are as follows: ease of use, use of graphics standards, close analogy to physical reality, and extensibility. Specifics on each of these areas are discussed in the following sections.

Ease of use

Programmers designing CAD systems are currently faced with the challenge of writing a large quantity of code to handle curve and surface geometry. The problems associated with this are two-fold. The first problem is that the programmer must have a high level of understanding of geometric modeling and the graphics language which is being used in order to get the desired results. This expertise is difficult to obtain, and requires that a continual effort be made by the programmer as new techniques are developed. This problem is coupled with the amount of code which must be written in order to achieve the desired results.

It is the goal of this research to eliminate much of the expertise and to reduce the amount of code necessary to create curves and surfaces. Finally, the framework must operate in a

way that is logical to users. Extensive documentation is never a substitute for clear, logical code. Methods for creating curves and surfaces should follow what a designer would expect.

Use of graphics standards

The move towards standards has facilitated many achievements in the field of CAD. In order to provide an environment which programmers on a variety of platforms will be able to use, the use of PHIGS will be vital. Furthermore, the ability to use IGES formats with the framework will provide an ability for the framework to communicate with all programs adhering to the ISO standard for data communication. This will allow for the integration of software quickly and easily.

By providing these abilities in the framework it will be easier to update programs as standards change. Instead of modifying all of the programs which use the framework, the framework itself may be modified.

Close analogy to physical reality

Mathematical models of curves and surfaces are frequently used to represent physical objects. The use of the framework should follow the physical reality of these objects. For example, surfaces should have properties such as color, specularity, and translucency. These characteristics should be part of the surface, and should be easily modified by the user.

Extensibility

It is obvious that the framework cannot be designed to take into consideration every type of curve and surface that a user might want to incorporate. Thus, it is important that the user be able to add to the framework. These additions must not compromise the integrity of the original framework, and should be easily incorporated without requiring an extensive knowledge of the existing code.

This ability to add new features, and expand the framework to handle new curves and surfaces will make the framework extensible. Thus, it will be enhanced more readily and will keep pace with the progress in the field.

Literature Survey

The developments of geometric modeling, and the application of this field to the computer in Computer Aided Geometric Design (CAGD) fill the literature. Although these fields are growing rapidly, in many directions, there are several fundamental principles on which both are developed.

This section will analyze the fundamental literature central to the development of curves, surfaces, object-oriented programming, and the design of frameworks.

Curves

The development of curves and surfaces accelerated rapidly with the introduction of the computer to the area of mechanical design. With the advent of this new technology, new methodologies for using curves were extensively developed. A very comprehensive survey of these techniques is given by Boehm et. al. [Boeh84].

The Bézier curve is a fundamental curve from which many curves and surfaces are derived. The Bézier curve has several properties which make it particular useful in CAD applications, these include: affine invariance, convex hull property, endpoint interpolation, and shape control [Fari90]. The shape of the Bézier curve may be intuitively controlled through the manipulation of the interior control points. Furthermore, the free-form curve

will be guaranteed to lie within the control polygon described by the control points. This allows the engineer a reliable method for controlling the shape of the curve.

One drawback of the Bézier curve is its lack of connective properties. Yamaguchi demonstrates how special arrangements of the control points may be used to properly connect sections of Bezier curves [Yama88].

The B-spline curve combines the features of the Bézier curve with excellent connective properties. The number of control vertices is independent of the order of the B-spline curve, thus allowing for great flexibility in design. The B-spline curve may be made to exactly match a Bézier curve, thus demonstrating that the Bézier curve is a special case of the B-spline [Yama88].

Two additional forms of the B-spline allow increased flexibility in design. These forms include the Non-Uniform B-spline (NUBS) and the Non-Uniform Rational B-spline (NURBS). Each form allows increased flexibility in shape control; however, with this increased control comes an increase in the complexity of the curve definition.

Several good references on these curve forms are: Farin [Fari90], Yamaguchi [Yama88], Bars87] Mortenson [Mort85], Beach [Beac91], and Faux [Faux79].

Surfaces

The development of surfaces closely parallels the development of curves. The progression from a curve, which is one-dimensional parametrically, to a surface which is two-dimensional in parametric space, brings with it extra difficulty.

Perhaps the simplest surface definition is the ruled surface. The ruled surface is defined by connecting two curves with straight line segments from corresponding points on each curve. A large number of engineering surfaces may be described using the ruled surface, which is desirable due to its simple mathematical form and because ruled surfacs are easily manufactured.

In the late 1960's and early 1970's a more general technique was developed by Coons and Forrest [Coon67] [Forr72]. This technique allows four boundary curves to be used to define a surface patch. Each of the four curves are blended to form a surface.

Next, the definition of patches through points was developed. This may actually seem to be a step backwards from the previous use of curves to define a surface; however, the point defintion allows a great degree of shape control. Examples of such surface definitions are seen in the bicubic Hermite, and Ferguson surfaces [Mort85].

The Bézier and B-spline surfaces follow directly from the respective curve definitions. Again, the connective properties of the Bézier definition make the B-spline surface the more commonly used surface in engineering applications.

Object-Oriented Programming

The use of object-oriented design and object-oriented programming (OOP) in the development of new software has grown dramatically in the last decade. The success of this new paradigm is reflected in its ability to break down complex problems into simpler parts through the use of a structured hierarchy. Further simplification is created for the use through the hiding mechanism of provided by encapsulation.

Although structured programming languages allow for an object-oriented design, full implementation of object-oriented methods is most easily performed through the use of an object-oriented language. Several features, such as inheritance and polymorphism, are not available in structured programming languages.

Stroustrup and Booch describe the fundamental methods in object-oriented design [Stro88] [Booc88]. Stroustrop demonstrates the use of C++ in implementing the object-oriented design within an application [Stro91].

The use of object-oriented programming in computer graphics has been discussed in the literature for several years. Knolle points out the need for object-oriented graphics to replace PHIGS [Knol90]. Wampler has developed a prototype object-oriented graphics language based on graPHIGS [Wamp91].

Frameworks and Toolkits

The development of toolkits, in the form of extensive libraries containing both subroutines and data structures, has grown rapidly with the use of structured programming. These libraries provided the user with the tools necessary to perform extensive work with little understanding of the underlying methodology. However, in order to enhance these libraries, the user was forced to develop independent routines which could not be fully integrated into the library. Furthermore, object-oriented techniques such as polymorphism, inheritance, and encapsulation were not available.

Even with these limitations, many toolkits have been developed which provide excellent functionality. One notable example in the geometric modeling field is Flemings' Advanced B-Spline Toolkit. This toolkit enhances the functionality of the B-spline provided by PHIGS [Flem91][Flem92a][Flem92b]. It has been stated that the future of the CAD field depends on the development of these toolkits [Jaya89][Jaya90].

The development of object-oriented frameworks has proven successful through many recent projects. The design and implementation of an object-oriented graphical user interface framework by Woyak [Woya92][Woya93] and a solid modeling framework by Lin [Lin92] [Lin93] show great promise for the future design of frameworks. Further success is shown in the integration of these frameworks as shown in the development of a fully parametric, feature-based, solid modeler [Mykl93].

The development of a curve and surface framework may be approached from several directions depending on the intended use of the final product. Johnson and McReynold

develop a curve class for the purpose of rendering curves on a personal computer [John92]. This differs greatly from the development performed by Klein and Slusallek who develop a curve and surface framework and apply differential geometry techniques [Klei92]. Using this framework, the user defines a parametric curve and a method for evaluating a point on the surface. The methodolgies on which the framework is based provide extensive analysis capabilities using the differential geometry equations. Each of the surfaces in the framework is based on a curve developed from the curve class. Thus a strong interrelationship is developed between the curves and surfaces.

Program Development

The need for high level tools for curves and surfaces was originally realized during the creation of a solid modeler. Using the Solid Modeling Framework developed by Lin [Lin92] [Lin93], the B-Spline Toolkit created by Fleming [Flem92] [Flem93], and the Graphical User Interface Framework developed by Woyak [Woya92] [Woya93], a solid modeling program was developed. In the process of creating the solid modeler, it was realized that a great deal of time and expertise was devoted to the creation and manipulation of curves and surfaces.

As the solid modeling program developed, so to did the first generation of the curve and surface framework. In its first draft, illustrated in Figure 2, the environment simply contained C routines for creating Non-Uniform B-Spline (NUBS) surfaces. This later evolved into a C++ NUBS surface class which was used to display the boundary surfaces of solid models created within the Solid Modeling Framework and utilized Fleming's B-spline toolkit.

Since that original version, the framework has undergone many changes, and has evolved into a much broader, and more usable tool. The framework currently includes both curve and surface classes with many auxiliary classes, as shown in Figure 3. The design allows the user to rapidly integrate the framework through a flexible interface.

The design cylce has followed the common iterative process of software engineering. Requirements are made, analysis and design follow, and then implementation and testing

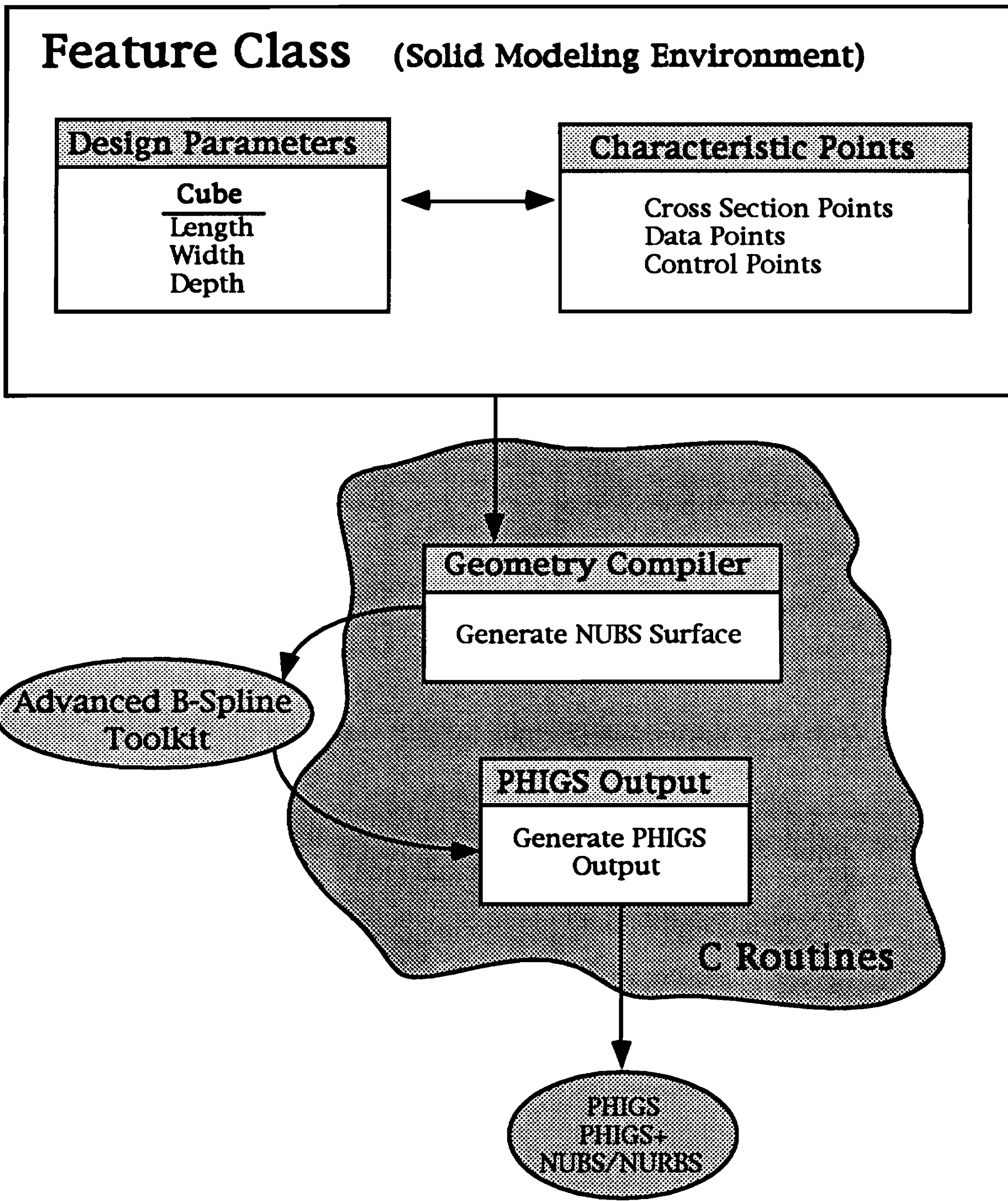


Figure 2 First Generation Of The Framework Consisting Of C Language Routines

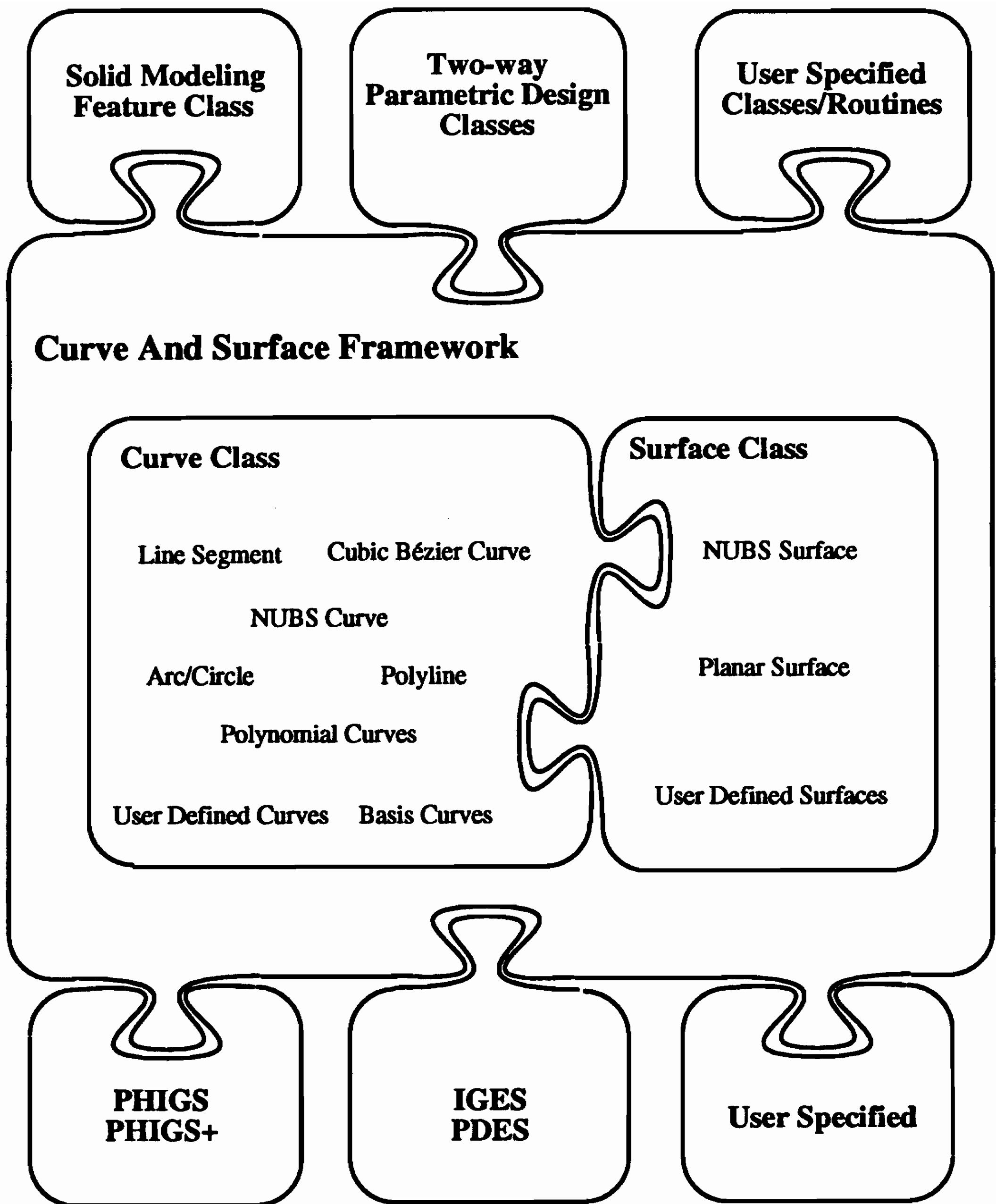


Figure 3 **Current Design Of Object-Oriented Curve And Surface Framework**

are pursued. After testing, redesign and analysis are often necessary, and at times the requirements are changed to allow a more feasible or more fertile result to ensue.

It is widely known that software design cycle is an ongoing process, and for the framework to be truly successful, it is recognized that continual research and development will be necessary.

Object-Oriented Programming

Since its early beginnings, software design has been governed by its methodology. At each critical point in the evolution of programming, new approaches were developed to aid the programmer in handling increasingly complex problems.

The evolution of programming languages began with machines which used patching of wires to control their processes. This method was obviously only applicable to very simple problems. Machine and Assembly languages were next developed to allow somewhat longer programs to be written. And then the first high-level languages were written (FORTRAN and Cobol).

High-level languages allowed for large programs (several thousand lines) to be written, but did not require a rigid structure. Later, the development of the first structured languages were completed. This allowed the concept of control structures, subroutines, recursion, and local variables to manage the complexities of large programs. It has been suggested that the average programmer can create and maintain programs that are up to 50,000 lines long using this structured approach [Schi92].

Although structured programming has yielded excellent results, it fails when applied to large programs and problems that are highly complex. The object-oriented approach was developed to allow more complex programs to be written and more easily maintained.

Object-oriented languages share three common defining traits: encapsulation, polymorphism, and inheritance [Wegn86] [Meye87].

Inheritance

Perhaps the most powerful feature of object-oriented programming is the concept of inheritance. Inheritance is the ability of an object to acquire the properties of another object. The inherited object may then add new features, or redefine some of the inherited properties to create a new type.

This ability allows for the concept of hierarchy to be applied to programs. Hierarchical classification is what allows most information to be made manageable. For example, a human is a type of animal, and a man is a type of human. In each case, the child class inherits from its parent class certain qualities, and adds to those qualities certain specifying features.

Encapsulation

Encapsulation is the method by which code, and the data it manipulates, is kept safe from outside interference. In objet oriented languages, methods and data may be bound together in "black boxes" called objects. Methods and data may be made public, and thus accessible to members outside of the object, or private, and only accessible by members inside the object. Thus public members act as the interface to the outside, and then activate methods

which are private to the object. The public members are thus used to provide a controlled interface to the private members.

Polymorphism

Polymorphism (from the Greek, meaning "many forms") is the quality in object-oriented programming which allows a name to be used to represent several related tasks. The purpose of this may be seen by observing an example of a language which does not support polymorphism. In C, as in many languages, there are several functions used to get the absolute value of a number which depend on the data type being used: abs(), labs(), and fabs(), for integers, long integers, and floats respectively. The need for three functions makes the simple task of taking an absolute value more complicated.

In an object-oriented language, the process of finding the absolute value may be named with one method abs(), which then invokes the appropriate method depending on the data type specified by the incoming argument.

The concept of polymorphism is the idea of "one interface, multiple methods." This allows the use of a generic interface to be used for a general class of actions.

Virtually all languages allow some degree of polymorphism. This may be seen by the arithmetic operator. The compiler automatically recognizes the types on either side, and proceeds accordingly. This form of polymorphism is called operator overloading, and is also available in object-oriented languages.

Using operator overloading, the arithmetic operator may be defined to take on additional arguments consisting of any object for which it is defined. This allows the user to perform complex arithmetic operations without the apparent use of a subroutine (although hidden from the user, that is exactly what is being done).

Polymorphism is further developed with the use of virtual functions. A virtual function is one which is defined in the base class, and is inherited to all of the derived classes. This allows the user to interact with many inherited objects in the same way. Although each object may have different methods for achieving the virtual function, the interface remains constant. The use of virtual functions is not only important for usability, but also for extensibility and maintainability. By utilizing virtual functions, as new objects are added to the environment, other objects will still be able to interact with the new object without requiring new coding.

Class Overview

The creation of classes is a central part of object-oriented design. Unlike structured programming, where the processes are defined in modules termed subroutines or functions, the object-oriented approach starts from the design of classes. These classes will allow the programmer to model real world complexities, and will allow problems to be broken down into smaller pieces.

The framework is based on the development of five fundamental classes:

- Points
- Vectors
- Curves
- Surfaces
- Material Properties

These classes are used to create a hierarchical structure using two types of relationships: "is a type of" and "has a". The "is a type of" relationship implies inheritance, whereas the "has a" relationship implies that an object contains or controls another object. This method of arranging objects hierarchically is a key method to the object oriented paradigm.

The following sections provide an outline to the development of each of the five fundamental classes used in the framework. More detail of coding and usage is given in later sections.

Points and Vectors

Points and vectors are key components of CAGD. The Point and Vector classes are primarily used as tools for other classes, and are reused frequently throughout the framework.

A point is defined by its Cartesian coordinates (x,y,z). The components may then be manipulated using methods familiar to geometric modeling.

Vectors are again defined by three components (x,y,z) and provide for many of the operations used. Although the data structure is similar between points and vectors, the methods acting on each and the results they produce are different. Both the point and vector classes allow for vector math to be performed and use operator overloading.

Both classes are available to the programmer as tools for creating curves and surfaces, or they may be used outside of the curve and surface environment.

Point Class

The point class consists of the data and methods necessary for the creation and manipulation of three-dimensional, cartesian coordinates. The constructors for the point class are as follows:

```
Point( x_value, y_value, z_value)
Point()
```

The latter constructor may be used in order to allocate the memory for the point, and values may then be set later using the provided set methods.

Vector arithmetic is provided with the addition and subtraction operators. Adding a vector to a point yields another point. Subtracting two points however, yields a vector. Finally an equals operator allows new points to be set to other points.

```
operator=( Point new_point)
operator+( Point added_vector)
operator-( Point subtracted_point)
```

Set and get operations are also provided for each of the data elements in standard format get_element() or set_element(). These include:

```
get_x()
get_y
get_z()
set_x( Point new_x)
set_y( Point new_y)
set_z( Point new_z)
set_xyz( Point new_x, Point new_y, Point new_z)
```

A print utility is provided to allow quick output of vectors. This operation will output the three components in the form (x_value, y_value, z_value). No line feeds are placed into this print statement so that it may be more easily incorporated into other output routines.

```
print()
```

Vector Class

The vector class provides for many standard vector operations to be performed. Vectors are represented by three data elements (x,y,z). The constructors for the vector class allow for the full specification of the vector or just the allocation of the necessary memory.

```
Vector( Point x_value, Point y_value, Point z_value)
Vector()
```

Operations for setting and getting each are similar to those shown for the Point class. The element to be changed, or returned is preceded by the key word get or set.

```
set_x( Point new_x)
set_y( Point new_y)
set_z( Point new_z)
set_xyz( Point new_x, Point new_y, Point new_z)
get_x()
get_y()
get_z()
```

The vector operations provided for the vector class are more extensive than those provided for the Point class. These include the vector arithmetic dot and cross operations, as well as the ability to calculate the magnitude of a vector, normalize a vector, or rotate a vector about an arbitrary axis. The dot and cross operations are also included outside of the Vector class. This allows the syntax for the method to be more intuitive to the user.

```
get_magnitude()
normalize()
rotate( Vector normal_vector, float amount)
cross( Vector vector_1, Vector vector_2)
dot( Vector vector_1, Vector vector_2)
```

```
operator=( Vector new_vector)
operator+( Vector added_vector)
operator-( Vector subtracted_vector)
operator*( float scalar)
operator/( float scalar)
```

With the exception of the rotate function, each of these methods contain straightforward vector mathematics. The rotate function however is more complex.

The vector rotation function, illustrated in Figure 4, is used in several classes and is of great importance to the framework. In order to subdivide an arc section, it is possible to determine a start vector, and a desired angle of rotation. Since the normal of an arc is known, it is then possible to determine the rotated vector using the vector rotation algorithm. To create surfaces of revolution, it is also necessary to use the vector rotation algorithm to find additional cross sections about the axis of revolution.

To define a three-dimensional rotation about an arbitrary axis, matrix algebra is used [Beac91]. Given the vector (x,y,z), the rotated vector (x',y',z') is solved by the multiplication of a transformation matrix A. To solve for the transformation matrix, the problem is broken down into three simpler rotations, one about each axis.

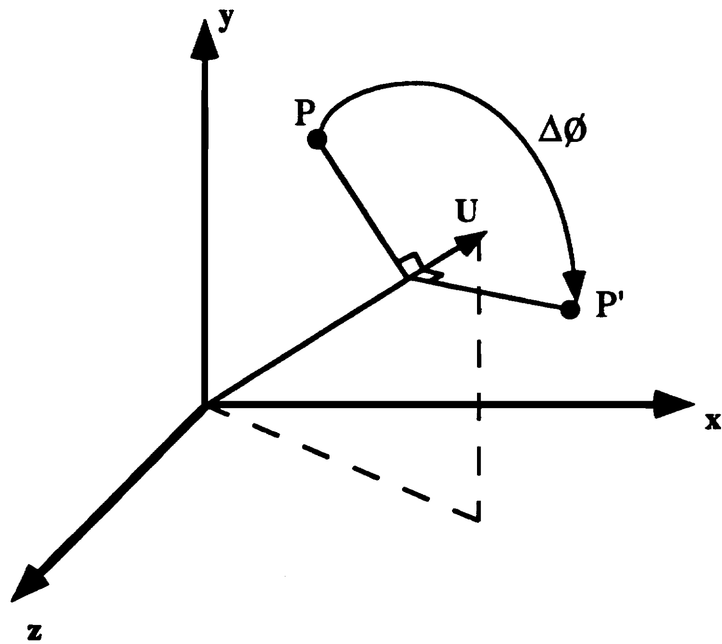


Figure 4 **3-Dimensional Rotation About An Arbitrary Axis**

A transformation matrix **B** is found that rotates the normal vector **U** coincident to the z-axis. This allows the rotation to be performed using the familiar transformation matrix:

$$A = B^T \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} B \qquad \text{Equation 1}$$

To determine **B**, three linearly independent vectors, whose transformations are known, are found first. First, **U** transforms into $(0\ \ 0\ \ 1)^T$. Second, the axis of rotation for this transformation is the vector $\mathbf{U} \times (0\ \ 0\ \ 1)^T$ or $(u_2\ \ -u_1\ \ 0)^T$. Finally, it may be shown that the z-axis transforms into a vector that is the reflection of **U** through the plane containing the z-axis and $\mathbf{U} \times (0\ \ 0\ \ 1)^T$. Thus the vector $(0\ \ 0\ \ 1)^T$ transforms into $(-u_1\ \ -u_2\ \ u_3)^T$. This yields the following three equations:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = B \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \qquad \begin{bmatrix} u_2 \\ -u_1 \\ 0 \end{bmatrix} = B \begin{bmatrix} u_2 \\ -u_1 \\ 0 \end{bmatrix} \qquad \begin{bmatrix} -u_1 \\ -u_2 \\ u_3 \end{bmatrix} = B \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \qquad \text{Equation 2}$$

These equations may be combined into

$$\begin{bmatrix} 0 & u_2 & -u_1 \\ 0 & -u_1 & -u_2 \\ 1 & 0 & u_3 \end{bmatrix} = B \begin{bmatrix} u_1 & u_2 & 0 \\ u_2 & -u_1 & 0 \\ u_3 & 0 & 1 \end{bmatrix} \qquad \text{Equation 3}$$

Solving for B in terms of U, and substituting the results into equation 2, after several lengthy manipulations, the general transformation matrix is defined as,

$$A = \begin{bmatrix} u_1^2 + \left(u_2^2 + u_3^2\right)\cos\theta & u_1 u_2(1-\cos\theta) - u_3 \sin\theta & u_1 u_3(1-\cos\theta) + u_2 \sin\theta \\ u_1 u_2(1-\cos\theta) - u_3 \sin\theta & u_2^2 + \left(u_1^2 + u_3^2\right)\cos\theta & u_2 u_3(1-\cos\theta) + u_1 \sin\theta \\ u_1 u_3(1-\cos\theta) - u_2 \sin\theta & u_2 u_3(1-\cos\theta) - u_1 \sin\theta & u_3^2 + \left(u_2^2 + u_1^2\right)\cos\theta \end{bmatrix}$$

Equation 4

A print utility is provided to allow quick output of vectors. This operation will output the three components in the form (x_value, y_value, z_value). No line feeds are placed into this print statement so that it may be more easily incorporated into other output routines.

print()

Curves

Previously, a historical look at the development of curves and surfaces was presented. This history provides some insight to the problems faced in creating tools for the CAD programmer. One of the greatest problems in creating a curve class is caused by the tremendous diversity in curve types and methods used on these varying types.

Although many designers rely on simple curve primitives such as line segments and arcs, others require highly advanced definitions such as the Non-Uniform B-Spline (NUBS). Furthermore, designers familiar with the techniques used to create one type of curve, may desire to apply these same techniques to new types. For example, a programmer who wishes to represent a model using NUBS, but is familiar with primitives such as line segments and arcs, may wish to define the model using data points which define the desired shape. An example of this is seen by the NUBS representation of an arc. The designer may define the arc by specifying the center point and two endpoints; however, the NUBS is defined with control points and knots. It is thus necessary to provide a method which makes the use of the control point and knot values transparent to this user.

The user may simply wish to convert from one type of definition to another. For example, the user may have defined a shape using a cubic Bézier curve. However, in order to use the curve in other CAD software, it may be necessary to have a NUBS definition of the curve. This requires that the NUBS be used to approximate the cubic Bézier curve. This type of methodology is also provided by the framework.

Graphical display of the resulting curves is allowed through the PHIGS graphical language standard. This provides the user with machine-independent code for rapid integration over a variety of platforms. The addition of methods for output to alternate graphical libraries may be added in the future.

Routines for analysis and manipulation of various curves are given for several classes including the calculation of first and second derivatives, conversion from control space to Cartesian space, editing of data and control points on the curve, as well as many others. These routines enable the user to use the framework not only to construct and display graphical items, but to actually use the graphics in meaningful ways.

In order to exchange the curve information generated within the framework to other computer software, the IGES data exchange specification may be used. Although the framework may be integrated directly using function calls, additional functionality exists for the creation of IGES data files containing the curves in the framework. If however, a curve is added to the framework and no IGES format is defined by the user, the entity may either be approximated by another curve type or skipped in the IGES file.

Curve Base Class

The power of the Curve and Surface Framework rests in the creation of the Curve and Surface base classes. These abstract base classes provide the mechanism for curves and sufaces to communicate with each other. This communication allows one type of curve to approximate another type of curve. Or, as will be seen in the surface class, it allows

several different types of curves to be used in the contruction of a surface. Furthermore, the base class allows new curves and sufaces added to the framework to be recognized immediately by the other curves and surfaces within the framework without modification of existing code.

The creation of a Curve base class is a difficult task due to the great diversity in curve types. The abstract base class contains data and methods which all curves must contain. Each specific instance of a curve type must then define how each specific method will be performed, and utilize the data required. Figure 5 illustrates this mechanism. Two derived curve types are shown, the Line and the Arc. When a method such as get_points() is called by a line, a different method is activated than when get_points is called by an Arc.

In order to provide flexibility to the user, no data is specified in the Curve abstract base class. This allows the user to define and store curves using data such as, equations, data points, or control points. However, certain methods are required in order to allow for the approximation of various curve types by other curve types. These methods are primarily chosen so that curves may be approximated using the NUBS. The decision to utilize the NUBS curve is due to its wide use and consequent support, and its ability to accurately approximate a wide range of curves. Furthermore, this architecture supports the approximation of curves with polyline curves, and may be implemented for various other types of curves.

```
virtual    void       set_number_of_points(int number_of_points)
virual     int        get_number_of_points()
virtual    Point*     get_points()
virtual    int*       get_continuity()
virtual    Vector*    get_tangents()
```

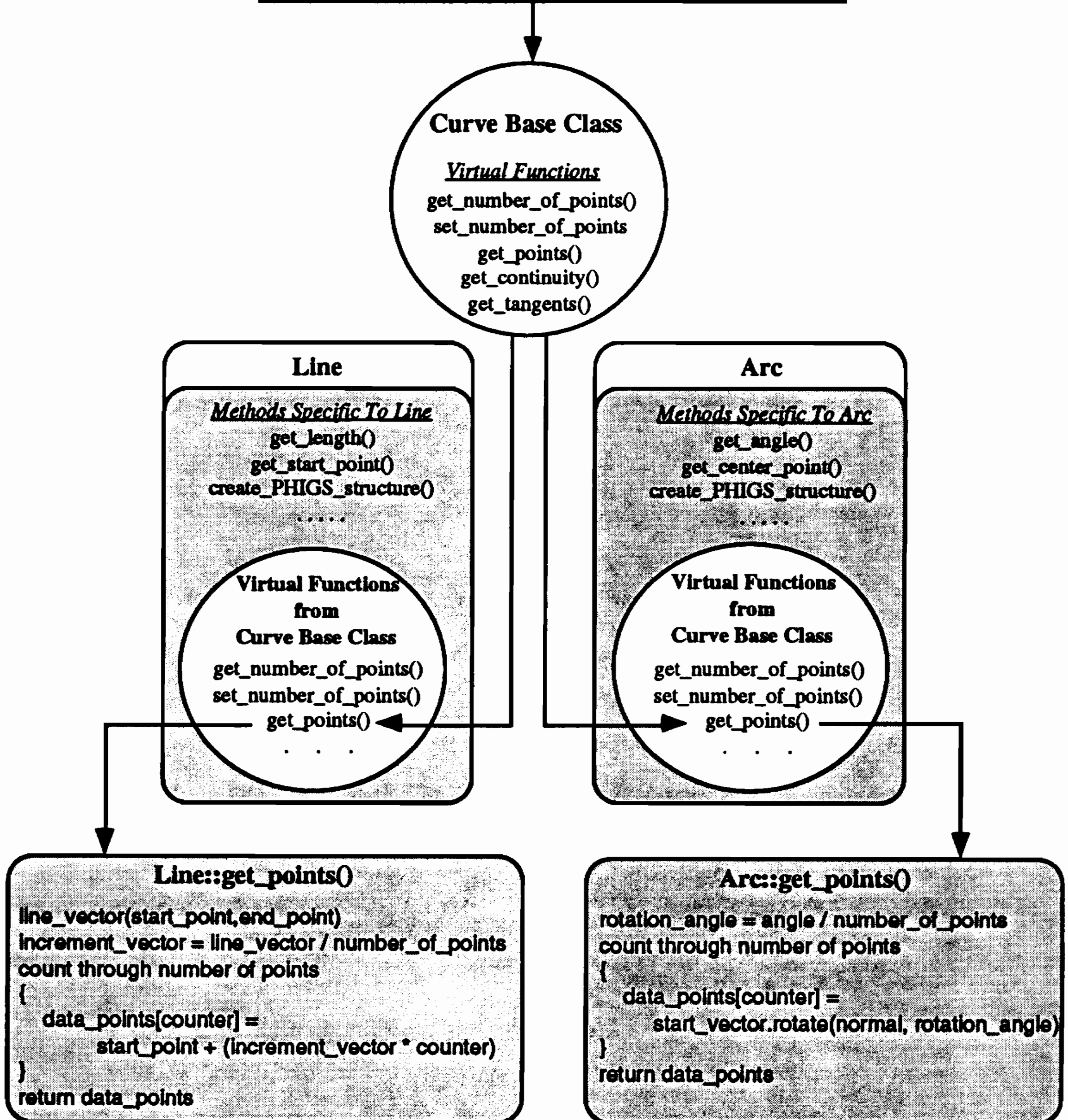


Figure 5 Utilizing The Curve Base Class In The Creation Of General Algorithms

The decision to not add graphics or file exchange routines to the list of virtual functions in the abstract base class allows the user to create curve definitions without having to write graphical routines. Yet, if the curve satisfies the functional requirements of the Curve base class, other curves, such as the NUBS, may be used to approximate the original curve. An example of how on curve may be used to represent other curves is shown if Figure 6. Both the Polyline and NUBS curve are shown with examples of how arrays of other curves connected head to tail would be approximated with a single curve. The created NUBS and Polyline curves may then be displayed, analyzed, and placed into an IGES file using the methods already constructed for the NUBS and Polyline classes.

Curve Class Hierarchy

Due to the diversity in curve types, several different curve classes are created. A hierarchy is developed to aid the programmer in the creation of new curve types, and to increase software reuse.

Figure 7 demonstrates both the developed hierarchy. The solid lines represents an "is a" relationship; thus, all of the classes shown are types of curves, and inherit from the Curve class.

The hierarchy contains both abstract base classes and derived classes. All of the classes are ultimately derived from the abstract base Curve class. Derived classes contain the complete curve definition along with all of the methods. These derived curves are fully developed classes and may be used to construct objects (instances) by the user. A derived class must

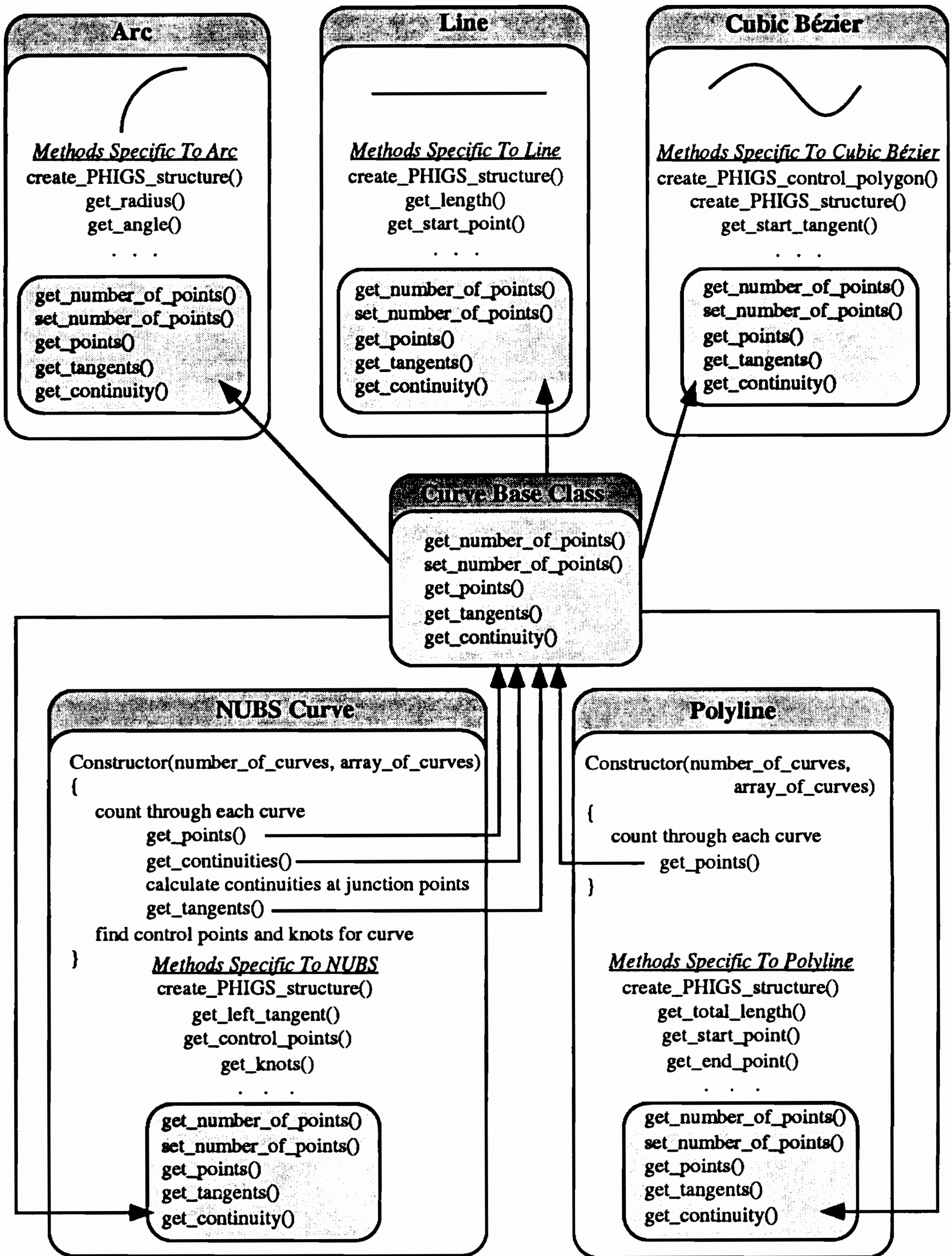


Figure 6 **Representation Of Derived Curves Using The Base Curve Class**

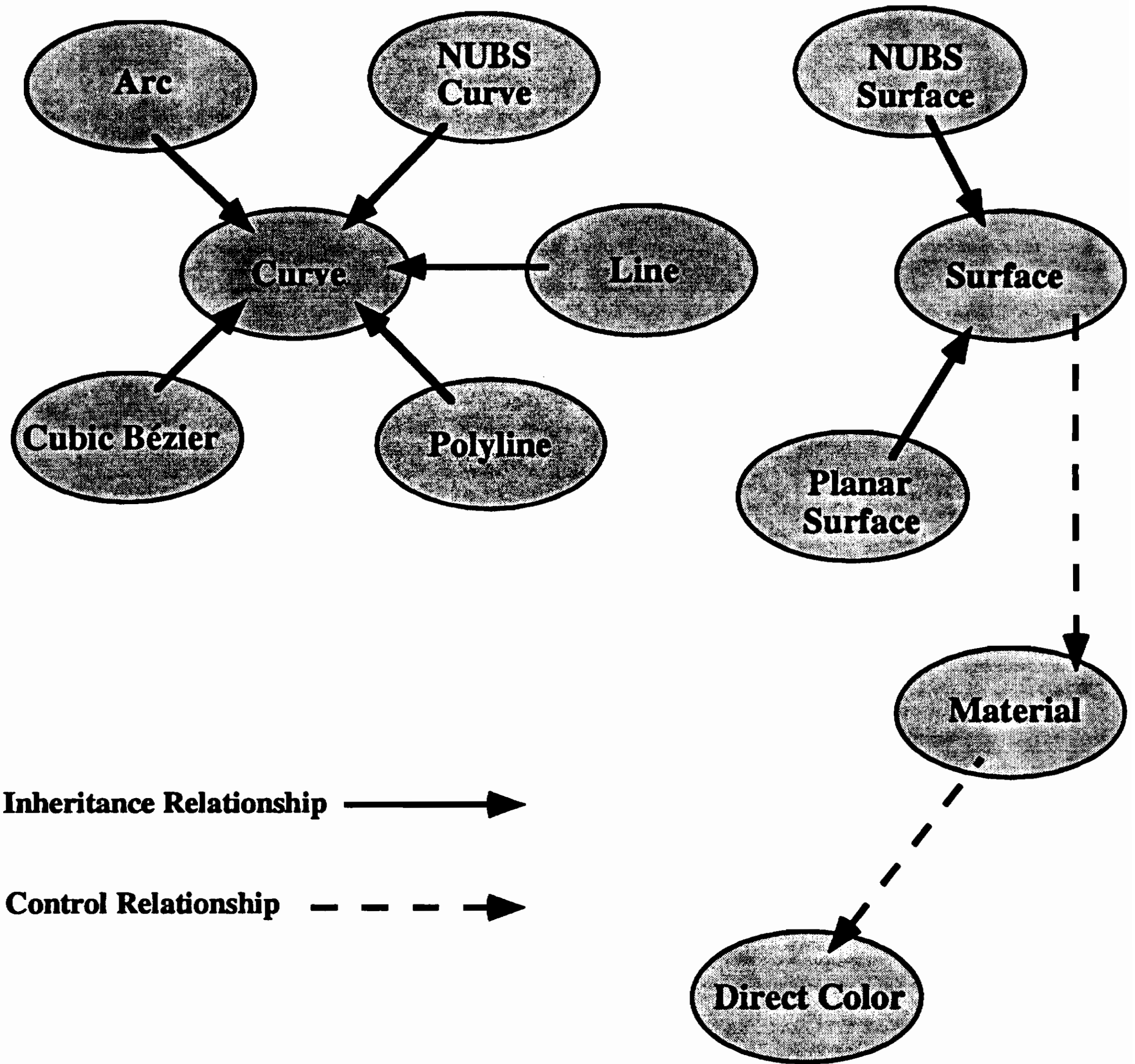


Figure 7 Curve and Surface Class Organization

contain each of the virtual functions defined in the base Curve class unless these functions are defined in an abstract base class which it inherits from.

Since many functions and data structures are similar, or identical for several curve types, additional abstract classes may be developed. These abstract base classes are not invoked for the creation of objects, but are used to develop new classes. These classes contain either data structures methods, or both, which are used for a group of derived curves or another abstract base class.

Two examples of abstract base classes that may be developed are the Basis curve class and the Bézier curve class. These abstract classes are used to aid in the derivation of a variety of curves. Due to the number of curves which may be contained under each of these classes, and the similarity in the methods these curves use, an abstract base class may be created. The construction of a multi-level hierarchy may then ensue as shown in Figure 8.

The Basis curve class allows basis functions to be stored within a structure inside the abstract base class. This function is then applied to the control points which are again stored in the base class, to yield data points along the curve.

The Algebraic base class enables the user to utilize polynomial algebraic equations in the definition of curves. Computation of a general equation given a set of points through fitting and interpolation algorithms may be incorporated as constructors for this class.

When curves of these types are implemented, all of the functions of the base class are inherited. These functions may be rewritten to provide new, or more optimal, functionality, or they may be left unchanged. If the user desires to rewrite all of the

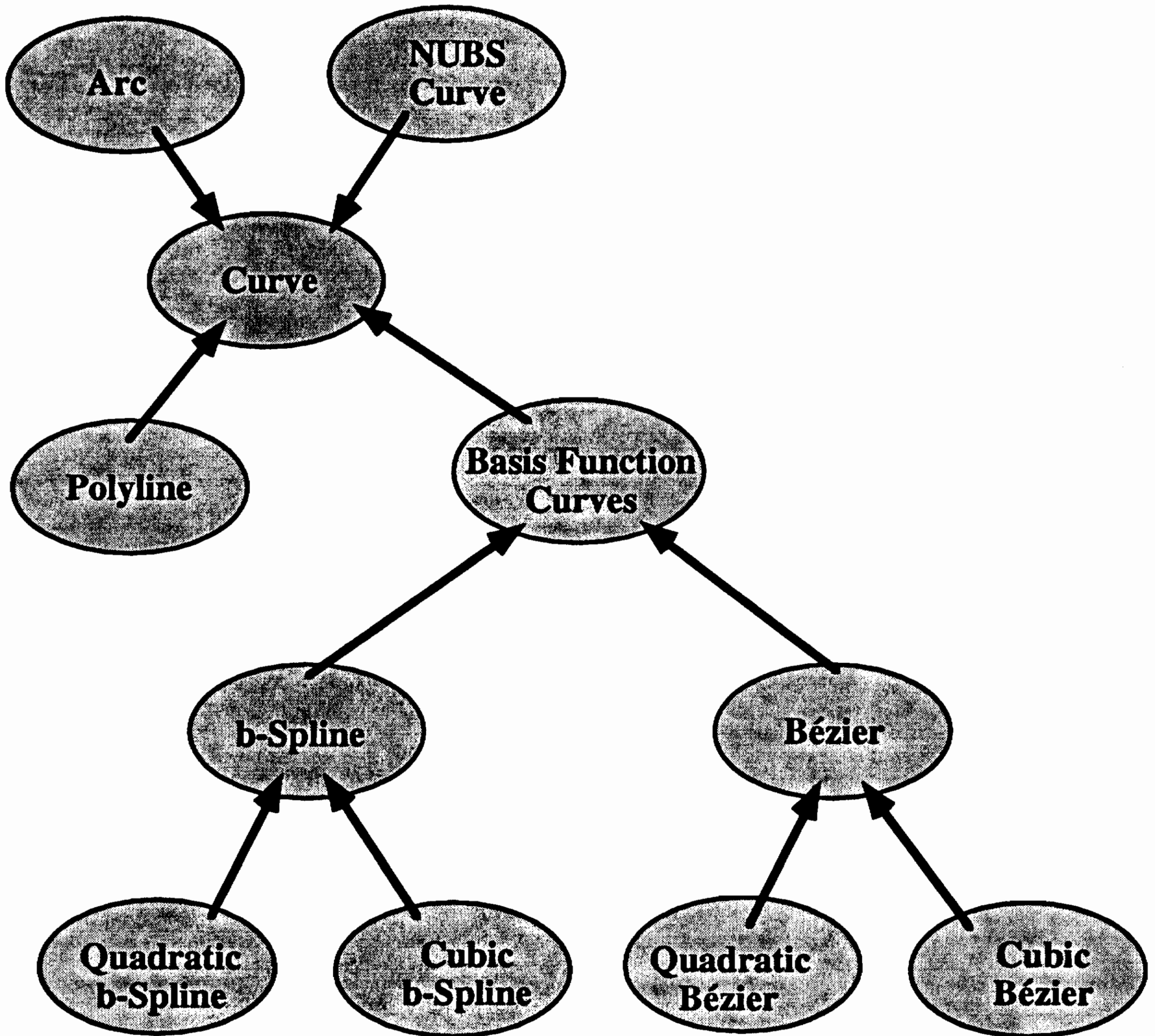


Figure 8 Development of Curve Class Hierarchy

functions, the class may be added directly to the Curve class without using the inheritance of the abstract classes. An example of this is seen with the Cubic Bézier curve inheriting directly from the Curve class, as opposed to the Basis class; however, it could be positioned in either place (as long as it is not placed in both areas).

Derived Classes

The data structure and methods provided for each of the derived curve types is varied depending on how the curve is normally used. For example, methods for editing the control points of a NUBS are provided, but this function has little meaning for an arc. The data structure of the two curve types are thus logically different. The NUBS curve would obviously contain control points and knots, whereas an arc would not.

Several derived classes have been defined using the Curve base class. These include:

- Arc
- Cubic Bézier
- Non-Uniform B-Spline (NUBS)
- Line Segment
- Polyline

Both the Polyline and NUBS classes allow for curves of any type to be approximated. In order to maintain a high degree of extensibility, the approximations use only the virtual functions which are provided by the Curve base class. The following figure shows the basics behind this approximation. An array of curves may be passed into the NUBS curve

constructor. These input curves must be connected in a head to tail manner for the resulting curve to be accurate. The NUBS constructor then querries each of the individual curves for information utilizing the virtual functions defined within the Curve base class. These

values are then processed and the resulting NUBS approximation is created. Similar methodology is used for the Polyline curve.

It should be noted that many constructors exist for each of the derived classes. These constructors vary depending on the way the derived curve is defined and the manner in which a user might normally wish to construct a curve of that type.

The ability to change the number of data points which define a particular derived curve enables the accuracy of approximations to be increased. For example, when representing a NUBS using a Polyline, the number of points on the NUBS may be increased to provide a better linear approximation.

An algorithm for this approximation technique is shown below:

```
count through each input curve
{
        get_points()
}

get accuracy of approximation

while( accuracy of approximation less than requested accuracy)
{
        increase number of points on input curves
        count through each input curve
        {
                get_points()
        }
        get accuracy of approximation
}
```

Currently the ability to check the accuracy of an approximation has not been implemented into the framework. However, the base class has been designed to accept this extension with the inclusion of a method for setting the number of points on a curve.
It should be noted that many curve definitions are not readily subdivided to n number of points. For example, if a Polyline is defined by n points, the addition of a single point (n+1 points) will not significantly effect most approximations of the Polyline. Furthermore, the decision of where to place the additional point creates additional problems. Thus, for many curves the functionality must be utilized carefully. In future versions, if error computations are to be used, a better method may include a function to increase the number of points, or decrease the number of points. These methods may not always increment the number of points by one, but will add an 'intelligent' number of points which will increase the accuracy of the next approximation.

In the following section the data structure and methods for several of the derived curve types are shown. Further descriptions of the classes are contained in Appendix A.

Arc

An arc may be constructed in a variety of ways, which is provided for through constructor overloading; however, a single definition for storing the arc within the class makes the creation of additional methods more economical. Thus, an arc is defined internally as follows:

```
Point  start_point
Point  center_point
Point  end_point,
float  angle_of_rotation
Vector normal
```

Rotation is determined using a clockwise rotation about the normal. Although the arc is over defined using this number of arguments in the definition, they are used to make the derivations of several functions faster and easier. Each method is based on the definition of a circular arc, however, different forms of this definition are utilized in different methods. For example, to find a given number of points on the suface, the rotate_vector() function, developed in the vector class is utilized. Using this method, the vector created by the center point and the start point is rotated about the normal. This allows for an equal angular subdivision of the arc. Other techniques may be developed utilizing varying parameterizations in the future.

Construction

The construction of an arc may take on a number of forms The first constructor is the most general form. Using the first constuctor, an arc is created in a clockwise direction about the

specified normal. Only the direction of the normal is used (the magnitude of the normal may be any value). The second constructor does not allow full control of the arc, since the direction that the arc will be created in may not be specified. Changing the order of the start and end points will reverse the direction that the arc is drawn. Using the general constructor, a circle may be created using the Arc class. With the less flexible methodology a circle may be made using two or more arcs which are connected in the correct manner.

```
Arc(Point start_point, Point center_point, Vector normal, float angle)
Arc(Point start_point, Point center_point, Point end_point)
```

Set Methods

The following set methods should be used cautiously. Although any of the operations will be performed without error, if the final product (start, center, and end points) does not yield a valid arc, errors will result when future methods are used.

The set normal operation may be used with either constructor. If the constructor which does not specify a normal is used, a normal is calculated for the given arc. This provides a useful way for reversing the direction of the arc.

```
set_start_point(start_point)
set_center_point(center_point)
set_end_point(end_point)
set_normal(normal)
```

In order to change the number of points on the arc which are used as data points during conversion to other curve forms (such as Polyline and NUBS), the set number of points call is used. The set_number_of_points method may be used in a manual way by the user, or in a coded automatic type of operation, such as an iterative approximation technique.

The routine simply changes the private variable (number_of_points) to the new number. This value is used in approximating the curve, and is necessary by the virtual function get_number_of_points() declared in the base class.

set_number_of_points(number_of_points)

Curve Inquiry

Each of the following functions returns the value corresponding value within the Arc class. None of the methods require computations.

Point	(start_point)	get_start_point()
Point	(center_point)	get_center_point()
Point	(end_point)	get_end_point()
Vector	(normal)	get_normal()

The get_angle() function returns the angle created by the arc. This angle is returned in radians and is given between (0-2π).

angle	get_angle()

Virtual Get Methods

The following functions are defined in order to satisfy the virtual functions of the base Curve class. The continuity of all of the data points specified is set to 2. By defining the continuity of all of the interior points to be C^2 continuous, the tangents do not have to be specified, thus the get_tangents() method simply returns a NULL pointer. The get_number_of_points() call returns the value set in the set_number_of_points() call, or

uses the default value contained in the constructor used. Finally, the get points function returns the desired number of data points. In order to calculate intermediate points along the arc, the vector rotation method is called. This method rotates the start vector a given angle about a normal, where in this case the normal is the normal of the arc.

```
int*        get_continuity()
int         get_number_of_points()
Point*      get_points()
Vector*     get_tangents()
```

PHIGS Methods

The following functions use the graPHIGS library to create screen geometry. The create_PHIGS_structure call uses a NUBS approximation of the arc with the specified number of points and the graPHIGS Non-Uniform B-Spline (NUBS) representation. The point representations use the graPHIGS polymarker. The calls which do not utilize the object oriented phigs structure require that the structure that the entity is being placed is open before the create function is called.

```
create_PHIGS_structure()
create_PHIGS_start_point()
create_PHIGS_center_point()
create_PHIGS_end_point()
create_PHIGS_structure(PHIGS_Structure* phigs_structure*)
create_PHIGS_start_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_center_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_end_point(PHIGS_Structure* phigs_structure*)
```

Cubic Bézier

The cubic Bézier curve is defined through the use of control points. In a single cubic Bézier section, four control points define the curve. The first and last control points define the start and end points, while the interior control points are related to the start and end tangents. In the case of the cubic Bézier curve, the inner control points are one third the distance along the initial and final tangent vectors.

The cubic Bézier is stored within the class with the following data:

Point	start_point
Vector	start_tangent
Point	end_point
Vector	end_tangent

The internal methods are then based on the standard definition of the cubic Bézier curve:

$$\mathbf{P}(t) = (1-t)^3\, \mathbf{P}_0 \;+\; 3(1-t)^2\, t\mathbf{P}_1 \;+\; 3(1-t^2)\, \mathbf{P}_2 \;+\; t^3\, \mathbf{P}_3 \qquad \text{Equation 5}$$

where

P_1	= start_point
P_2	= start_point + (start_tangent/3)
P_3	= end_point + (end_tangent/3)
P_4	= end point
t	= parametric value along curve (0,1)

Construction

The construction of the cubic Bézier curve is traditionally performed through the specification of the control points which form its control polygon. Although other methods exist, this is the only on currently provided. The first and last points correspond to the start

and end of the curve, and lie on the curve at the beginning and ending points. The inner control points relate to the tangent (3 x tangent vector) in both magnitude and direction. These points are specified also.

```
Cubic_Bezier(Point start_point, Vector start_tangent, Vector end_tangent, Point
        end_point)

Cubic_Bezier(Point control_point_1, Point control_point_2, Point control_point_3,
        Point control_point_4)
```

Virtual Set Methods

The set_number_of_points() function must be provided due to its inclusion in the base class. The use of a subdivision technique to create a bezier curve with the given number of control points may be implemented within this method.

```
set_number_of_points( number_of_points)
```

Virtual Get Methods

The following virtual functions return the values contained within the class data structure. The curve may be approximated by the NUBS curve using the start points, and start tangent conditions.

```
int*        get_continuity()
int         get_number_of_points()
Point*      get_points()
Vector*     get_tangents()
```

PHIGS Methods

Because PHIGS does not include a method for representing the cubic Bézier curve, in order to display the curve it must be placed into a form that PHIGS will allow. For accurate representations, the NUBS curve class is used. This should provide and exact representation of the cubic Bézier curve. In addition, several other display functions are also provided. When points are being displayed the polymarker is used. The display of the control hull utilizes the polyline, and the tangents use the line segment primitive.

```
create_PHIGS_structure()
create_PHIGS_control_polygon()
create_PHIGS_start_point()
create_PHIGS_start_tangent_point()
create_PHIGS_end_tangent_point()
create_PHIGS_end_point()
create_PHIGS_start_tangent()
create_PHIGS_end_tangent()
create_PHIGS_structure(phigs_structure*)
create_PHIGS_control_polygon(phigs_structure*)
create_PHIGS_start_point(phigs_structure*)
create_PHIGS_start_tangent_point(phigs_structure*)
create_PHIGS_end_tangent_point(phigs_structure*)
create_PHIGS_end_point(phigs_structure*)
create_PHIGS_start_tangent(phigs_structure*)
create_PHIGS_end_tangent(phigs_structure*)
```

In order to display the curve, the use of the NUBS Curve class is employed. The approximation of the Bézier curve by the NUBS Curve however, is exact.

The implementation of the cubic Bézier curve without deriving it from the Basis Curve class is an example of how a curve class may be derived without using the inheritance provided. By defining each of the functions in a manner which is optimal for the cubic

Bézier curve, overhead is lowered by eliminating the need for more general, and less efficient methods.

Utility Methods

The print function allows all of the control points to be displayed. The points are printed in normal Cartesian form (x,y,z) from first to last control point.

```
print()
```

Non-Uniform B-Spline Curve

The Non-Uniform B-Spline (NUBS) curve is defined using control points and knots. These values may be entered directly, or obtained through the use of Fleming's B-Spline Toolkit [FLEM92a] [FLEM92b]. A degree three NUBS curve may be constructed through the definition of data points, continuity conditions, and tangent vectors. If this latter method is used, the input information is kept so that editing may be made to either the data points or the control points. Thus the data structure for the NUBS curve is as follows:

```
int         parameterization
int         number_of_data_points
Point*      data_points
int*        continuities
Vector*     tangents

Point*      control_points
float*      knots
```

If the curve is created using control points and knots, the remaining data may not be obtained, however, if the data points, continuities, and tangents are given in the constructor, either form may be used. The parameterization variable designates the type of parameterizationto be used (centripetal, chord length, or uniform) and is described in detail later. The continuity array contains an integer value representing the continuity desired at the corresponding data point. Hence the fifth continuity value corresponds to the continiuty condition desired at the fifth data point. If C^0 continuity is specified, two tangents must also be stored, if C^1 continuity is specified, one tangents must be stored, and if C^0 continuity is specified, no tangents are necessary. These tangents are placed into the array of tangents, and are stored as Vectors. If the control points or knots are needed, the

method get_parametric_values() will create the suface definition using Flemings's Advanced Modeling Toolkit [FLEM92a].

Ultimately, the basic definition of the NUBS curve is in terms of the control points and knots. In the early 1970's Carl Deboor derived the following recursive relationship for the NUBS basis functions.

$$B_i^k(t) = \frac{t - t_i}{t_{i+k-1} - t_i} B_i^{k-1}(t) + \frac{t_{i+k} - t}{t_{i+k} - t_{i+1}} B_i^{k-1}(t) \qquad \text{Equation 6}$$

$$B_i^1(t) = \begin{Bmatrix} 1, t_i \le t \le t_{i+1} \\ 0, otherwise \end{Bmatrix} \qquad \text{Equation 7}$$

The NUBS curve is then defined using these basis functions with the corresponding control points and knots.

$$C(t) = \sum_{i=0}^{n} C_i B_i^k(t) \qquad n \ge k\text{-}1 \qquad \text{Equation 8}$$

$C(t)$ = NUBS Curve
C_i = Control point
B_i^k = i^th B-spline basis function of order k

Methods for approximating other curves of the Curve class with the NUBS curve are included in this class. The NUBS class uses only functions which are contained in the Curve base class to assure the successful integration of new curve types. Using this methodology, NUBS curves may be constructed by sending an array of various curves which are connected in a head to tail manner to the NUBS constructor. Each curve has associated with it a definition of continuities at each data point. These continuities are used

to construct a NUBS curve which matches in continuity at each of the points. Rules for solving the continuity at junction points where two curves of different continuities meet (for example a line joined with an arc) are given later in this section. Using this method, the resulting NUBS curve may be C^0, C^1, or C^2 continuous, depending on the continuity conditions of the input curves.

As this approximation is performed, the original form of each of the curves which are being approximated is kept. This allows the designer to use the original definitions of the curves for future analysis if desired, and the NUBS curve for other applications, such as display. An additional constructor which allows the input curves to be linked with the NUBS curve may be created which would allow the NUBS to be updated as the input curves are modified; however, this would require a new data structure for the class, and may be best implemented through the development of a second NUBS curve class.

Three forms of parameterizations are available to the designer, uniform, chord length, and centripetal. Uniform parameterization is the simplest type, and defines the knot spacing to be identical for each interval. For many applications this method is too simple, and ignores the geometry of the data points. Chord length parameterization includes a measure of geometry. The knot spacing is determined by the spacing of the data points from the following equation.

$$\frac{t_i - t_{i-1}}{t_{i+1} - t_i} = \frac{\|d_i - d_{i+1}\|}{\|d_{i+1} - d_i\|} \qquad \text{Equation 9}$$

t_i = i^{th} domain knot
d_i = i^{th} data point

Centripetal knot spacing is derived from a physical analogy to centripetal acceleration. This method requires that the knot spacing be proportional to the square root of the distance between the data points.

$$\frac{t_i - t_{i-1}}{t_{i+1} - t_i} = \sqrt{\frac{\|d_i - d_{i+1}\|}{\|d_{i+1} - d_i\|}} \qquad \text{Equation 10}$$

These three methods are perhaps the most common methods mentioned in the literature, However, other parameterization methods exist. The implementation of various methods may be made to provide different shape characteristics for the curve.

Construction

The construction of NUBS may take on many forms. For this reason, a number of methods are provided for their construction. The methods which allow for data points to be specified use Fleming's Advanced Modeling B-Spline Toolkit. This inverts the data into the appropriate control points, based on the specified parameterization. The input of control points directly allows for an exact interpretation to be made without external calls. A final method provided allows for the input of an array of curves to be approximated with a single NUBS curve. This method uses the virtual functions provided in the base Curve class and Fleming's toolkit to create a NUBS approximation of the given data.

```
NUBS_Curve(Point* array_of_data_points, Vector* array_of_tangents, int
        paramatization_method, int number_of_points, int
        number_of_tangents, int* array_of_continuities)

NUBS_Curve(int number_of_curves, Curve* array_of_curves)

NUBS_Curve(Point* array_of_control_points, float* array_of_knots, int
        number_of_control_points, int number_of_knots)
```

Internal Methods

The allocate_space method uses the number of curves and the array of input curves to allocate the necessary space for the total number of data points, tangents, and continuities. This method should not be invoked unless these input curves have been constructed.

```
allocate_space(int number_of_curves, Curve* array_of_input_curves)
```

Each of the following set functions require that the required space has been allocated for each variable before being called (allocate_space() method). At each point, the continuity conditions may be determined by checking the continuity specified to each side of the point.

```
set_continuities(int number_of_curves, Curve* array_of_input_curves)
set_data_points(int number_of_curves, Curve* array_of_input_curves)
set_number_of_points(int number_of_curves, Curve* array_of_input_curves)
```

The total continuity condition at a point is based on Table 1. This total continuity must be determined in order to allocate the correct amount of space for the tangents in the u-direction (along each given input curve).

When a curve is specified without parametric values (data point curve definitions as opposed to control point definitions) the following function is used to calculate each of the parametric values (control points and knots).

```
float*          get_parametric_values()
```

Table 1 **Determination Of Continuity At A Point By Analyzing Continuities Of Curves Being Joined**

Continuity From Left	Continuity From Right	Total Continuity
0	0	0
1	0	0
2	0	0
0	1	0
1	1	0
2	1	1
0	2	0
1	2	1
2	2	2

Set Methods

The set parameterization currently accepts an integer (1,2, or 3) which represents the type of paramatization being used. The main use for setting the paramatization is for the use of the Advanced Modeling B-Spline Toolkit [Flem91]. When this toolkit is invoked, these three parameterization may be used to yield different results as follows:

Table 2 **Setting The Method Of Parameterization**

Value	Type of Parameterization
1	Uniform
2	Chord Length
3	Centripetal

The parameterization value is a private variable in the class and may be set during construction, or using the set_parameterization call. If the set_parameterization call is used, and the value of the parameterization variable has changed, the curve must be inverted again, and control points solved. Thus, if the type of parameterization is known, it should be specified in the constructor. If no parameterization is chosen, and one is needed for inversion, the Chord Length (2) method is used as a default.

```
set_parameterization( int parameterization_type)
```

Virtual Set Methods

The set_number_of_points() call must be defined due to the fact that the class is a type of Curve. However, methods for subdividing the curve into any given number of sections have not been implemented at this time. For this reason, the set_number_of_points() method is a blank function, which has no effect.

```
set_number_of_points( int number_of_points)
```

Virtual Get Operations

Each of the following get functions are required by the Curve base class. Each of the functions returns the values stored in the private data area of the class. Because each of these functions may be called at any time after construction, all of the values must be solved for during the construction process. Currently, the use of the control point constructor, where the control points and knots are given by the user may cause a problem if the curve is approximated by the NUBS_Curve. Since no data points are given, the get_points()

function will not return the appropriate values. This problem may be solved with the addition of virtual functions to the base Curve class, or by solving for the corresponding control points on the curve.

```
int*        get_continuity()
int         get_number_of_points()
Point*      get_points()
Vector*     get_tangents()
int         get_number_of_tangents()
int         get_paramatization()
```

Curve Inquiry

In several analysis routines, different values on the curve are needed. These range from control points values, data point values, to knot values. The get_parametric_values() function, an internal method, must be called if the curve was not defined using parametric values, in order to use the following functions:

```
int         get_number_of_control_points()
int         get_number_of_knots()
float*      get_control_points()
float*      get_knots()
Point*      get_data_points()
```

The get_data_points() method is only effective when the constructor used to create the curve uses data points as opposed to control points.

Analysis Methods

Due to the wide use of the NUBS curve, several analysis tools provide the user with a variety of functionality. These curvature analysis functions may aid the engineer in

determining a variety of engineering quantities and qualities. The following methods use algebraic equations to solve for the respective values. For each, the derivatives of the blending functions are used at the corresponding control point values. Left and right simply implies the direction in which the tangent is taken, from the left implies that the point is approached from the lower control value towards the higher control point value.

```
Vector*     get_left_tangent(Point data_point)
Vector*     get_right_tangent(Point data_point)
Vector*     get_left_normal(Point data_point)
Vector*     get_right_normal(Point data_point)
```

"Geometric Modeling" by Mortenson is used as a reference for each of these routines [Mort85]. The corresponding page numbers are referred to in the coding, and variable naming has followed Mortenson's convention.

PHIGS Methods

Several PHIGS functions may be invoked on the constructed curve for visualization. The create structure method uses the NUBS functionality to create a representation. Currently the composite fill area is not working properly, and thus it has been disabled, however, this method will produced a filled area for planar closed curves. The tangents and accelerations utilize a PHIGS line segment and the corresponding anlaysis functions. Both the left and right values are displayed. As with other classes, if the object oriented phigs structure is not used, the structure open state must be present.

```
create_PHIGS_structure()
create_PHIGS_control_polygon()
create_PHIGS_tangents()
create_PHIGS_fill_area()
create_PHIGS_structure(PHIGS_Structure* phigs_structure*)
create_PHIGS_control_polygon(PHIGS_Structure* phigs_structure*)
```

```
create_PHIGS_fill_area()
create_PHIGS_structure(PHIGS_Structure* phigs_structure*)
create_PHIGS_control_polygon(PHIGS_Structure* phigs_structure*)
create_PHIGS_tangents(PHIGS_Structure* phigs_structure*)
create_PHIGS_accelerations(PHIGS_Structure* phigs_structure*)
create_PHIGS_fill_area(PHIGS_Structure* phigs_structure*)
```

Utility Methods

The print function allows all of the control points to be displayed. The points are printed in normal Cartesian form (x,y,z) from first to last control point.

```
print()
```

Operators

The equal operator has been provided for ease in copying NUBS to new structures. All of the elements in the classes data structure are copied to the new structure (by value).

```
operator=(NUBS_Curve copy_of_NUBS)
```

Line

The Line class is perhaps the simplest curve class. The Line class defines a line segment by the definition of a start and end point.

Construction

Two methods are given for the construction of a line segment. The most commonly used, two point definition, and the point and vector form.

```
Line( start_point, end_point)
Line( start_point, vector)
```

Virtual Set Methods

The set number of points operation divides the line segment into equal segments. The number of points may be set to values greater than one. However, the usefulness of increasing the number of points on a line segment for accuracy in approximation may not be present, this method may be utilized to match the number of points of one cross section with another. For example, if sweeping from a line to a polyline with four sections, the line may be divided into for equal sections to yield the correct number of points. If this spacing is not correct, the Polyline may be used to give the correct spacing.

```
set_number_of_points()
```

Virtual Get Methods

Each of the following get functions returns a value stored in the class data structure with the exception of the get_tangents operation. This operations returns a zero magnitude tangent. As seen in the Composite Line class, the use of the zero tangent vector is necessary until new paramatization techniques are employed in the translation of groups of curves.

```
int*        get_continuity()
int         get_number_of_points()
Point*      get_points()
Vector*     get_tangents()
```

Set Methods

To aid the editing of a line segment, each of the following functions is included. Neither method will generate an error, and both will result in a line segment being drawn. In the case of a zero magnitued line, PHIGS does not generate an error, and neither does the Line class.

```
set_start_point( new_point)
set_end_point( end_point)
```

PHIGS Methods

Each of the following PHIGS methods utilizes PHIGS polyline and polymarker methods.

```
create_PHIGS_structure()
create_PHIGS_start_point()
create_PHIGS_end_point()
create_PHIGS_structure(phigs_structure*)
create_PHIGS_start_point(phigs_structure*)
create_PHIGS_end_point(phigs_structure*)
```

Analysis Methods

Frequently, the distance between a point and a line is necessary. This functionality is included in the Line class. The get_distance() method finds the shortest (perpendicular) distance between a point and the given line. The function returns both the distance and the direction in a Vector value. This method is well utilized in the creation of surfaces of revolution. In these functions, the distance from a profile curve point, to the axis of rotation (a Line) is found. This distance is then used as a vector to be rotated as additional cross sections are determined. Using this methodology, the line segment is treated as a line. If the shortest distance does not fall on the existing line segment, no error is generated, instead the value of the vector which is found is returned. If the magnitude of the line segment is zero, an error will be generated (division by zero), unable to determine direction of line segment.

```
float     get_distance( Point point)
```

Polyline

The antiquated polyline is a fundamental entity in both geometric modeling and graphics. The ability to approximate complex curves using a series of linear segments has proven valuable in numerous applications. Currently, even the complex curve shapes supported directly through graphical libraries such as graPHIGS, use linear approximations to represent the surfaces graphically.

Because of this ability to approximate other curves, the Polyline includes a constructor which accepts any curve derived from the Curve base class and attempts to represent the curve with linear segments. Currently the methods do not include an ability to set an error tolerance for the approximated surface, but this type of method may be incorporated in the future.

The data which is used internally to define the Polyline simply contains an array of ordered points, and an integer value storing the total number of points.

```
int     number_of_points
Point*  points
```

From this, points along the curve may be found using the basic equation of a line.

Construction

The polyline is a set of line segments connected in a head to tail manner. The class has the ability to take other curves which are members of the Curve class, and represent them with

a polyline. This methodology uses the number of points set in the base curve, and currently has no iterative methodology employed for adjusting the number of points to acheive an error tolerant approximation. The second method of construction shown simply places line segments through the ordered array of points in a 'connect the dots' manner.

```
Polyline( int number_of_curves, Curve* array_of_curves)
Polyline( int number_of_points, Curve* array_of_points)
```

Internal Methods

The set_continuity() function is given to automatically set the continuity of each of the internal points to C^0, and the end points to C^1. The end points will remain C^1 continuos until the adjoining sections (if any are applied) are known. If these are also Polylines, the two C^1 ends, joined together will form a C^0 point (see NUBS Curve Class Continuity Table 1). This method uses the number of points stored in the class data structure to set each of the continuities.

```
set_continuity()
```

Virtual Set Methods

The set number of points operation is present, however, this value is automatically set during construction to be one greater than the number of line segments. To create more, or less points along the curve will require the implementation of an additional methodology. Currently, no use for this has been seen.

set_number_of_points(number_of_points)

Virtual Get Methods

The following functions are supplied for satisfaction of the base class. The get_tangents() call returns an array of zeros, which yields the correct results. Methods may be incorporated to calculate the correct magnitude of the parametric tangent. Unfortunately, without new paramatization techniques, this may hinder the ability to mesh with other curves when the composite cuve is combined with other curve types in the creation of a new curve (such as a NUBS curve). When the curves are reparamatized, the tangents will change, unless each curve has a unit paramatization of some sort.

get_continuity()
get_number_of_points()
get_points()
get_tangents()

PHIGS Methods

Several PHIGS operations are provided to assist in the visualization of the completed curve. The create_PHIGS_structure() uses the PHIGS polyline call. To create points, the three dimensional polymarker is employed.

create_PHIGS_structure()
create_PHIGS_start_point()
create_PHIGS_end_point()
create_PHIGS_points()
create_PHIGS_structure(phigs_structure*)
create_PHIGS_start_point(phigs_structure*)
create_PHIGS_end_point(phigs_structure*)
create_PHIGS_points(phigs_structure*)

Abstract Curve Classes

The development of abstract base classes enables the user to create a variety of derived curve types quickly and easily. Instead of developing all curves directly from the Curve class, additional abstract base classes may be used. The development of general techniques, such as the use of numerical differentiation, displaying structures through PHIGS, and several others may be implemented in a general way in the abstract base class and inherited to derived classes. Derived classes may then redefine these methods using optimized techniques, or may use the abstract base class definitions directly.

Two examples of possible abstract base classes are the Basis and Bézier curve classes. Curves which utilize basis functions may be manipulated and analyzed using general methods which apply to large groups of basis function curves. These methods are included in the abstract base class. As an example, many methods would be repeated if several different Bézier curves were developed of varying degrees. Thus general methods which could be used for any order Bézier may be placed within an abstract class. In this case some of the methods may be placed within the Basis abstract class (those methods which may be utilized by all Basis curves) and some are placed into the Bézier abstract base class. Then each derived Bézier curve would be able to inherit the general methods contained in either the Bézier class or the Basis class. If more optimized techiniques are desired, these functions could then be redefined in the derived classes.

Surfaces

Surfaces may be defined, created, and manipulated using a variety of methods. As discussed earlier, the methods used to create an object are not always directly related to how a surface is defined. This discrepancy often requires that processing be done to convert the initial input used to create an object into a form which is compatible with the surface definition.

The four methods used most frequently in engineering applications to define surfaces include: the use of an equation, the use of data points, the use of control points, and the transformation of a curve. At this time, the first method has not been incorporated; however, the implementation of such methods could be made within the framework. The implementation of the latter methods have all been incorporated.

Each surface may contain material properties. These properties are described in the Material Class, and allow the surface to be displayed in a realistic manner rapidly.

Surface Class Hierarchy

As with the curve class, a hierarchy is developed to aid the programmer in the creation of new surface types, and to increase software reuse. Figure 6 demonstrates both the developed class organization.

Both the Planar and NUBS Surface classes allow the construction of surfaces through the input of curves derived from the base Curve class. These methods include: sweeping a curve, revolving a curve, or the definition of a surface which passes through a series of isoparametric curves. By defining the methods in such a way as to only use the virtual functions specified in the Curve class, as new curves are added no changes will be necessary in the surface classes.

Surface Base Class

Similar to the Curve abstract base class, the Surface base class allows for derived surfaces to contain an independent data structure. Furthermore methods may be redefined or added to the derived classes. By using the base class, methods may be written in a general way for all surfaces, allowing new surfaces to be added to the framework without extensive changes being necessary to all of the classes.

Each derived surface class must contain methods for all of the virtual functions defined in the Surface class. These methods include:

```
virtual     int          get_number_of_curves
virtual     Curve**      get_curves()
virtual     int*         get_continuities_v()
virtual     Vector*      get_tangents_v()

virtual     Material*    get_material()
```

This creates a surface by storing the rows of data corresponding to an isoparametric line in an isoparametric curve. Examples of this are seen in the specific instances of the Planar and NUBS Surface classes. Additionally, the information along the isoparametric curves in the other parameter are stored utilizing the two additional methods get_continuities_v() and get_tangents_v(). By requiring these virtual functions, the methods for approximating each curve type with other curve types may be reused for the surface definitions, thus a planar surface may be approximated with a NUBS surface as well as the reverse approximation of the NUBS surface with a planar surface.

Derived Classes

The developed surface classes include several common methods for creating surfaces. These methods include the creation of surfaces of revolution by the definition of a profile curve and an axis of revolution. A swept surface may be created by defining the required number of cross sectional curves. The swept surface may be used to create ruled surfaces, or to create C^2 continuous NUBS surfaces depending on the methods used in construction.

When defining profile curves, or cross sectional curves, any curve that is derived from the Curve class may be used. In addition, as new curves are added to the Curve class, no changes will be necessary in the surface class.

Future additions to the Surface class may be made in a way similar to those made in the Curve class. Groups of surfaces containing similar properties may use multiple inheritance as shown with the Polynomial and Basis Curve classes.

Two classes are created to demonstrate the potential of the Surface class:

- Non-Uniform B-Spline (NUBS) Surface
- Planar Surface

Complete functional descriptions of these classes are presented in Appendix B.

Non-Uniform B-Spline (NUBS) Surface

The NUBS surface is a highly flexible surface which is able to accurately represent most engineering objects. Although the basic definition of the NUBS surface is in terms of control points and knots, many designers desire to construct the surface using different definitions.

The NUBS surface is a tensor product surface where the basis functions are products of two separate univariate (curve) bases. The definition of the NUBS surface is an extension of the NUBS curve definition.

$$S(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{m} C_{ij} B_i^k(u) B_j^l(v) \qquad \text{Equation 11}$$

$S(u,v)$ = NUBS surface of two variables
C_{ij} = control points
B_i^k = i^{th} basis function of order k as a function of u
B_j^l = j^{th} basis function of order l as a function of v

Similar to the curve, the data may be stored in two basic forms, the free form utilizing control points and knots, and the data point form. The data which must be stored to fully define the surface may take on two different forms: a free form using the control points and knots, and another utilizing the data points, continuity conditions, and tangents. Each form has advantages and disadvantages, so a solution in which both are available to the user is attempted.

Using the free form definition of the surface, each control point must be stored in an (mxn) rectangular array of coordinates. Each row of control points is stored as a NUBS Curve. Associated with each control point is a knot value in both parametric directions; however, along each isoparametric curve, the knots in of one parameter must stay fixed. The use of the NUBS Curve class to store part of the information which defines the surface proves to be both economical and practical.

Another method for storing the surface is to store the data points on the surface in an (mxn) rectangular array of coordinates, with the corresponding continuity conditions in the two parameter directions. Depending on the continuity conditions, as shown in the Curve class, tangent values may also need to be stored. Using the inversion techniques developed by Fleming [FLEM91] a free-form representation may be realized.

Becuase the data structure consists of isoparametric curves containing data which matches that contained in the NUBS curve class, the surface data is stored utilizing the class. Each isoparametric curve in the u-direction is stored as a NUBS curve. This also allows either the free form surface definition of data point definition to be utilized. The only additional information then required within the suface class is an array of knots in the v-direction for the free form definition, or an array of continuities in the v-direction and the tangents which may result depending on the continuity values for the data point form. Thus the NUBS Surface class data structure contains the following:

```
int             number_of_curves
NUBS_Curve**    isoparametric_nubs_curves (one-dimensional array of NUBS Curves)
int*            continuities_v
Vector*         tangents_v

float*          knot_v
```

Methods are provided for the creation of NUBS surfaces using a variety of methods. Construction techniques include the surface of revolution. Using this method a profile curve is defined using any curve derived from the Curve class. Next, an axis of revolution is defined. With this information, the constructor creates a NUBS surface which approximates the desired surface of revolution. During this process the control points and knot values are found, and stored for future computations.

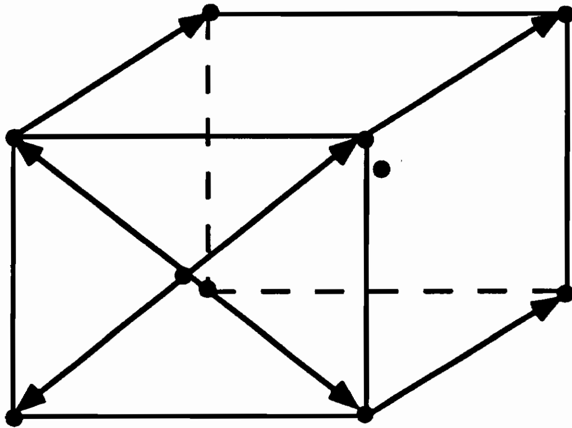
A similar method is the swept surface constructor. Using this routine, the user enters any number of cross sectional curves, along with the definitions of the continuities and tangents in the sweep direction. This information is again processed to find the control points and knot values, and thus the definition of the NUBS surface. Using this constructor automatically forces the paramater distance between points on each isoparametric curve to match the values defined on the initial curve. This guarantees a valid surface.

Additionally, surfaces may be constructed using the free form surface definition through the input of the necessary control points and knot values.

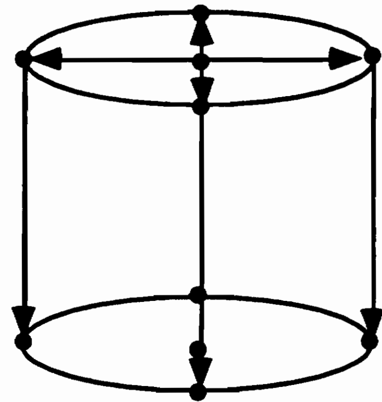
Multiple Cross-Section Sweep

By defining cross-sections of an object (two or more), a surface is created by sweeping from corresponding points on each section. Although each cross-section may be different in shape, and in the types of curves used, the number of data points on each cross-section must remain constant. Furthermore, the data points will be mapped to each other in a direct correspondance, thus the parametric values of the data points must be the same on each curve.

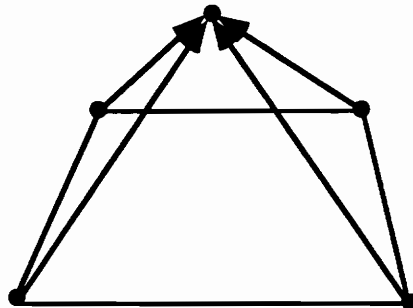
Using the multiple cross-section definition many engineering shapes may be constructed. Perhaps all basic primitives are most easily constructed using this method. Figure 9 illustrates the construction of several primitives using this sweep method.



Cube



Cylinder



Pyramid

Each cross-section contains five points (one point is defined twice to close the polyline). In order to close the end faces, five coincident points are defined at the center of each of the end faces.

Figure 9 Construction of Simple Primitives Using Cross-Section Sweeps

Single Cross-Section Swept Along Guide Curve

By defining a single cross-section, and a guide curve, several methods may be used to create a surface. The first method, which has been fully implemented translates the cross section along the guide curve. At each section along the guide curve where data points have been given specifying the guide curve, no rotation of the cross-section will exist. The second method uses the definition of the osculating plane to determine the appropriate rotation of the cross-section at each of the data points. This method may be used to create more flexible 'free-form like' surfaces. Figure 10 illustrates several primitives constructed using these methods.

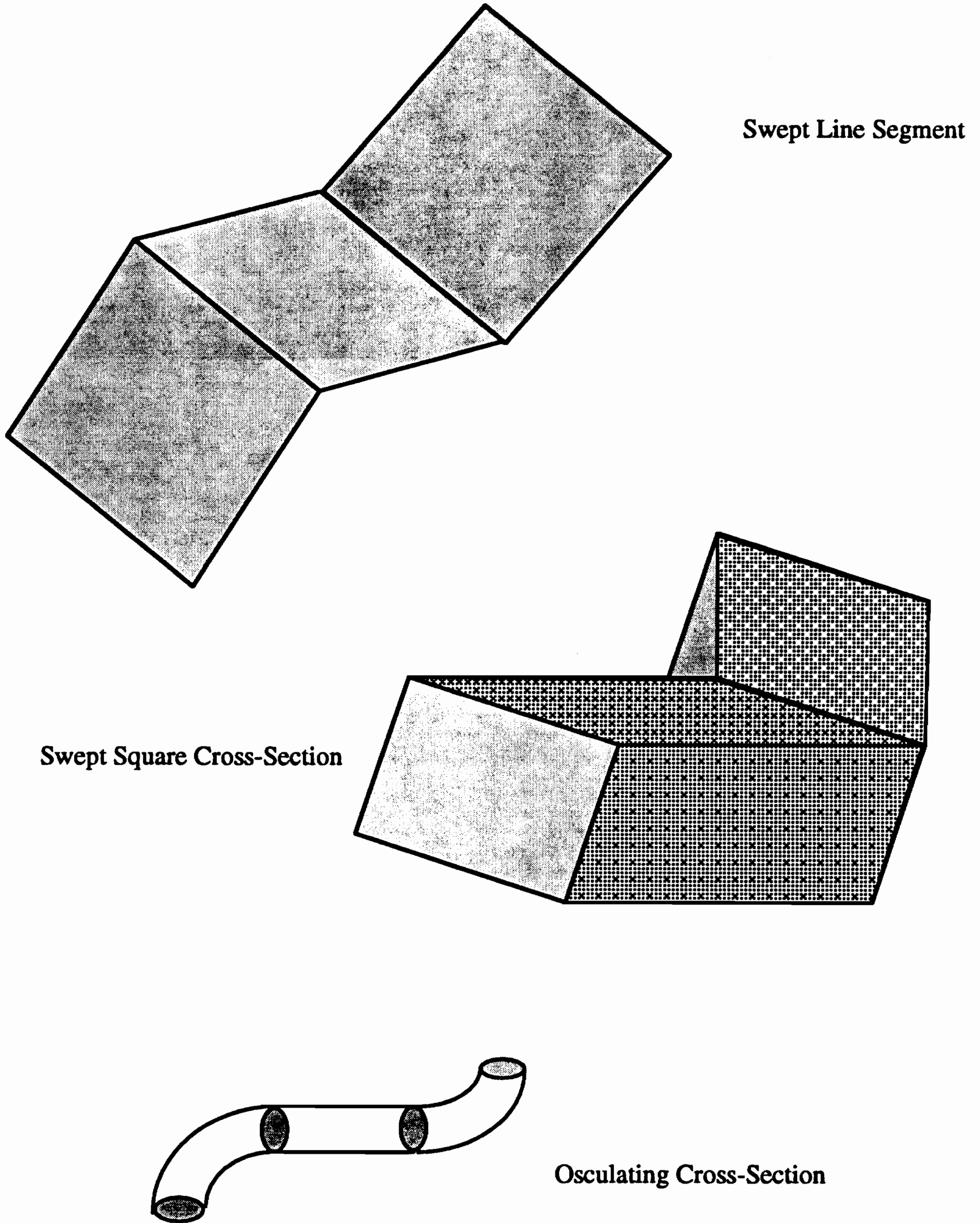


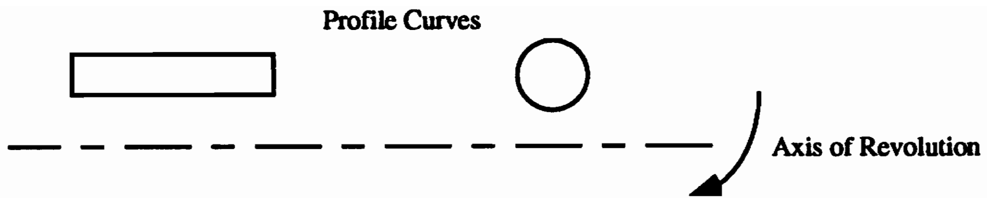
Figure 10 Multiple Sweeps Of A Single Cross-Section

Surface of Revolution

The use of the surface of revolution has been widely used in the CAD field. Using a profile curve and an axis of rotation (line segment), a surface is generated which has circular cross-sections. To create the surface using a sweep methodology, the characteristic points and tangents of the profile curve are rotated about the axis of rotation a given number of times. By rotating both the characteristic points and given tangents on each cross-section, the integrity of the original profile curve definition is kept. The number of cross-sections to accurately create a circular cross-section depends entirely on the type of surface which is being generated. Using a planar surface, an increase in the number of cross-sections is directly related to the accuracy of the final surface. A NUBS surface may use very few cross-sections (4-8) to approximate the surface very accurately [FLEM92a]. If more advanced surfaces such as the NURBS surface are incorporated, exact surface representation may be created.

During construction, the axis of revolution is not restricted to the plane of the profile curve, nor is the profile curve restricted to be planar. Furthermore, the axis of revolution, although given as a line segment, is used as if it were a line. Thus the magnitude (as long as it is not equal to zero) is not important.

Figure 11 shows the construction of a wine glass using the surface of revolution technique.



The surface of revolution may be used to create a variety of surfaces, including the wall of a cylinder or a torroid as shown above, or the wine glass shown below.

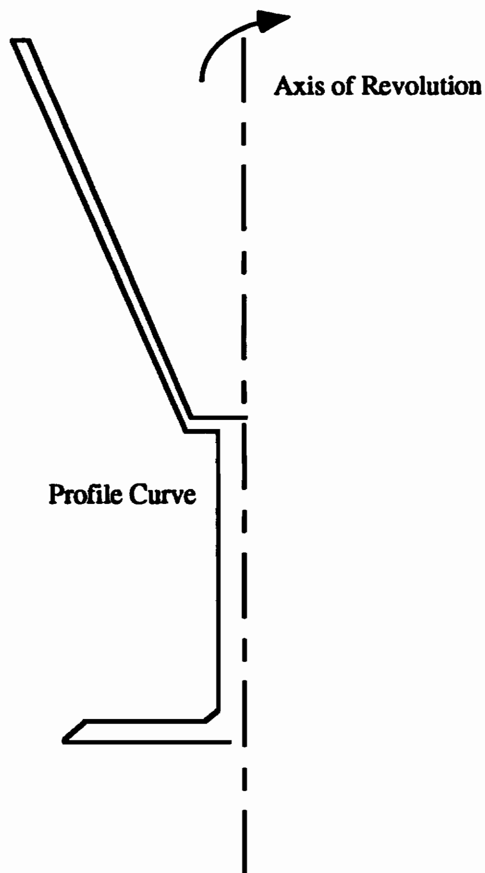


Figure 11 Surfaces of Revolution

Control Point Specification

Surfaces may be created directly by specifying the control points and knot values. Using this methodology requires an understanding of parametric curves, as well as the specific type of surface being employed. Although it is easier for an inexperienced user to make an error using this methodology, many experienced users may find this method to be useful. This method may be used to create single patches, or more complex, multi-patch surfaces. Again, the ordering of the points is critical. Most surface definitions require an equal number of points be made along each isoparametric curve (four sided patches).

Constructors

The NUBS Surface may be created using one of several methods. These methods include sweep operations, surface of revolution, and control point definitions. When curves are being input to the constructor, they are accepted as an array of Curve pointers. Thus, each curve in the array must be a member of the Curve class. Because the constructor contains pointers to the curves, any of the Curve class virtual functions may be accessed. This allows additional curves to be added to the base class without effecting the surface classes.

```
NUBS_Surface( int number_of_cross_sections, Curve** array_of_curves, int*
                        continuities)
```

Using this constructor, two or more cross sectional curves are specified. These curves are then connected using a NUBS Surface. If the continuity variable is set to zero, then sweep is obtained by creating four sided patches which use linear segments between each cross

section. If the continuity is set to equal two, then the resulting surface is rounded, and the continuity at each cross section is maintained as C^2 continuous.

Sweeping along a guide curve yields similar results to those shown above; however, using this constructor, the curve shape between sections is more easily controlled and the continuities at a cross section may be set to C^1.

```
NUBS_Surface( Curve** cross_sectional_curves, Curve** guide_curve)
```

Using the surface of revolution construction, a surface may be created by specifying a profile curve and an axis of revolution. The profile curve is created similar to cross sectional curves (as an array of Curve pointers). The profile curve is defined using the Line class.

```
NUBS_Surface( Curve** profile_curve_array, Line* axis_of_revolution)

NUBS_Surface( Curve** profile_curve_array, Line* axis_of_revolution,
                    int number_of_cross_sections)
```

Internal Methods

Depending on the method being used to construct the surface, a number of curves must be created. For example, if the surface of revolution is being used, each of the transformed profile curves must be calculated. With the swept cross section along a guide curve, again the transformed curves must be solved. Internal methods for these operations enable the user to create new surfaces readily. Depending on the input arguements, different methods are applied.

```
create_NUBS_curves( Curve** array_of_curves)
create_NUBS_curves( Curve** profile_curve, Line* axis_of_revolution)
create_NUBS_curves( Curve** cross_section, Curve** guide_curve)
```

Additional functions are provided for allocating the correct amount of memory, setting the tangents along the v-direction (cross sectional curves are put into the u-direction), and the continuities in the v-direction. Again, operator overloading is used.

```
set_v_tangents()
set_v_tangents( Curve** guide_curve)
set_v_continuity( int* array_of_continuities)
set_v_continuity( Curve** guide_curve)
```

Virtual Set Methods

Two set functions are defined as virtual functions in the Surface base class. These functions must be satisfied in each of the derived classes. The set_material() function sets a pointer to the material properties of the surface. If no material properties are set, a default property is used. This material property is currenlty being used to create aesthetically pleasing displays; however, engineering material constants could be added to increase the functionality of the material properties.

```
set_material( Material* material)
```

The set_color() operation also is used for display purposes. This function sets a pointer to a direct color value (RGB).

```
set_color( Direct_Color color)
```

Virtual Get Methods

Similar to the Curve class, the Surface class incorporates several virtual functions so that the interpretation of surfaces may be made in a general way. This allows additional curves and surfaces to be added to the framework without effecting current classes.

```
Material*        get_material_property()
Direct_Color*    get_color()
Curve*           get_curves()
int*             get_continuity()
Vector*          get_tangents()
```

The get_continuity() and get_tangents() functions return values in the v-direction only. The values for the u-components of each of these calls is contained in the curves. By keeping a pointer to each curve used to create the surface, the u values are not needed.

PHIGS Methods

When creating a surface using PHIGS, several variables control the appearance of the displayed results. The Material Property class controls many of the properties associated wth lighting and shading. This is done to simplify the programmers task. The surface may be constructed using the material properties contained in the Surface class, or without those properties. However, if the material properties are not used, the user will need to insert surface properties into the PHIGS structure to view the object. This latter methodology enables the user to create surfaces without using the surface properties contained within the Material class. If the user wishes to not use the phigs_structure class provided, the initial function is provided which does not use this class.

```
create_PHIGS_structure()
```

```
create_PHIGS_control_net()
create_PHIGS_u_tangents()
create_PHIGS_v_tangents()
create_PHIGS_u_acceleration()
create_PHIGS_v_acceleration()
create_PHIGS_structure(PHIGS_Structure* phigs_structure*)
create_PHIGS_control_net(PHIGS_Structure* phigs_structure*)
create_PHIGS_u_tangents(PHIGS_Structure* phigs_structure*)
create_PHIGS_v_tangents(PHIGS_Structure* phigs_structure*)
create_PHIGS_u_acceleration(PHIGS_Structure* phigs_structure*)
create_PHIGS_v_acceleration(PHIGS_Structure* phigs_structure*)
create_PHIGS_structure_without_surface_properties(PHIGS_Structure*
                                                   phigs_structure)
create_PHIGS_structure_with_surface_properties(PHIGS_Structure*
                                                phigs_structure)

set_PHIGS_surface_properties(PHIGS_Structure* phigs_structure)
```

Analysis Methods

The ability to analyze a surface is possible through the use of the NUBS curve class. Each curve may be analyzed using by getting the specific curve, or the entire surface may be analyzed using one of the following functions:

```
Vector*     get_tangents_u_direction()
Vector*     get_tangents_v_direction()
Vector*     get_acceleration_u_direction()
Vector*     get_acceleration_v_direction
```

Planar Surface

The Planar Surface class is a simple surface class which allows for primitive structures to be created and displayed. Using the ability of the Polyline class to approximate complex curves with linear sections, each isoparametric curve is approximated using the linear curves. Currently the data points used to create the surface are guaranteed to be on the constructed planar approximation; however, as stated earlier, an iterative methodology for approximating the surface to a tolerance has not been incorporated at this time.

The surface is stored as data points along each isoparametric line. Thus, each isoparametric line is stored as a Polyline curve. Again, each isoparametric line must have the same number of points on each cross section which is assured through the use of the provided constructors.

Creation of the planar surface may be made using methods similar to those presented for the NUBS Surface. These methods include the surface of revolution, swept surfaces, approximation of input curves, patch definitions, and data point input.

Construction

The Planar Surface class includes constructors for swept surfaces and surfaces of revolution. The swept surface construction may include the specification of multiple cross-sections, or a single cross section and a guide curve. Each of the curves used in the constructors must be derived from the Curve base class. The Polyine class is used to approximate the input curves with linear segments. Currently iterative methods are not

being used to increase the accuracy of the intial approximation; however, the ability to set the number of points along each of the input curves is a virtual function in the Curve base class, and thus could be added in the future.

```
Planar_Surface( int number_of_cross_sections, Curve** array_of_curves, int*
                array_of_continuities)
```

Using this constructor, two or more cross-sectional curves are specified. These curves are then connected using a Planar Surface.

Sweeping along a guide curve yields similar results to those shown above, since the surface is created by using linear segments between sections, the guide curve is first approximated using the Polyline class.

```
Planar_Surface( Curve** cross_sectional_curves, Curve** guide_curve)
```

Using the surface of revolution construction, a surface may be created by specifying a profile curve and an axis of revolution. The profile curve is created similar to cross-sectional curves (as an array of Curve pointers). The profile curve is defined using the Line class.

```
Planar_Surface( Curve** profile_curve_array, Line* axis_of_revolution)
Planar_Surface( Curve** profile_curve_array, Line* axis_of_revolution,
                int number_of_cross_sections)
```

Internal Methods

Depending on the method being used to construct the surface, a number of curves must be created. For example, if the surface of revolution is being used, each of the transformed profile curves must be calculated. With the swept cross-section along a guide curve, again the transformed curves must be solved. Internal methods for these operations enable the user to create new surfaces readily. Depending on the input arguements, different methods are applied.

```
create_linear_curves( Curve** array_of_curves)
create_linear_curves( Curve** array_of_curves, Curve* guide_curve)
create_linear_curves( Curve** profile_curve, Line* axis_of_revolution)
```

Additional functions are provided for allocating the correct amount of memory, setting the tangents along the v-direction (cross sectional curves are put into the u-direction), and the continuities in the v-direction. Again, operator overloading is used.

```
set_v_tangents()
set_v_tangents( Curve** guide_curve)
set_v_continuity( int* array_of_continuities)
set_v_continuity( Curve** guide_curve)
```

Virtual Set Methods

Two set functions are defined as virtual functions in the Surface base class. These functions must be satisfied in each of the derived classes. The set_material() function sets a pointer to the material properties of the surface. If no material properties are set, a default property is used. This material property is currenlty being used to create aesthetically

pleasing displays; however, engineering material constants could be added to increase the functionality of the material properties.

set_material(Material* material)

The set_color() operation also is used for display purposes. This function sets a pointer to a direct color value (RGB).

set_color(Direct_Color* color)

Virtual Get Methods

Similar to the Curve class, the Surface class incorporates several virtual functions so that the interpretation of surfaces may be made in a general way. This allows additional curves and surfaces to be added to the framework without effecting current classes.

```
Material*        get_material_property()
Direct_Color*    get_color()
Curve*           get_curves()
int*             get_continuity()
Vector*          get_tangents()
```

The get_continuity() and get_tangents() functions return values in the v-direction only. The values for the u-components of each of these calls is contained in the curves. By keeping a pointer to each curve used to create the surface, the u values are not needed.

PHIGS Methods

Display of the surface is enabled using PHIGS functions. Due to the fact that the surface is planar in nature, filled polygons may be used to represent the final shaded image.
Each of the planar surfaces use three sided patches to guarantee that each facet is planar, although many surfaces could use planar four sided patches this would not guarantee that the resulting surface was planar. If the patches are relatively large, this may not yield an aesthetically appealing image due to the large triangular facets created. This problem may be solved by increasing the number of cross sections, or points along each curve.

When creating a surface using PHIGS, several variables control the appearance of the displayed results. The Material Property class controls many of the properties associated wth lighting and shading. This is done to simplify the programmers task. The surface may be constructed using the material properties contained in the Surface class, or without those properties. However, if the material properties are not used, the user will need to insert surface properties into the PHIGS structure to view the object. This latter methodology enables the user to create surfaces without using the surface properties contained within the Material class. If the user wishes to not use the phigs_structure class provided, the initial function is provided which does not use this class.

```
create_PHIGS_structure()
create_PHIGS_structure_without_surface_properties( PHIGS_Structure*
            phigs_structure)
create_PHIGS_structure_with_surface_properties( PHIGS_Structure*
            phigs_structure)

set_PHIGS_surface_properties( PHIGS_Structure* phigs_structure)
```

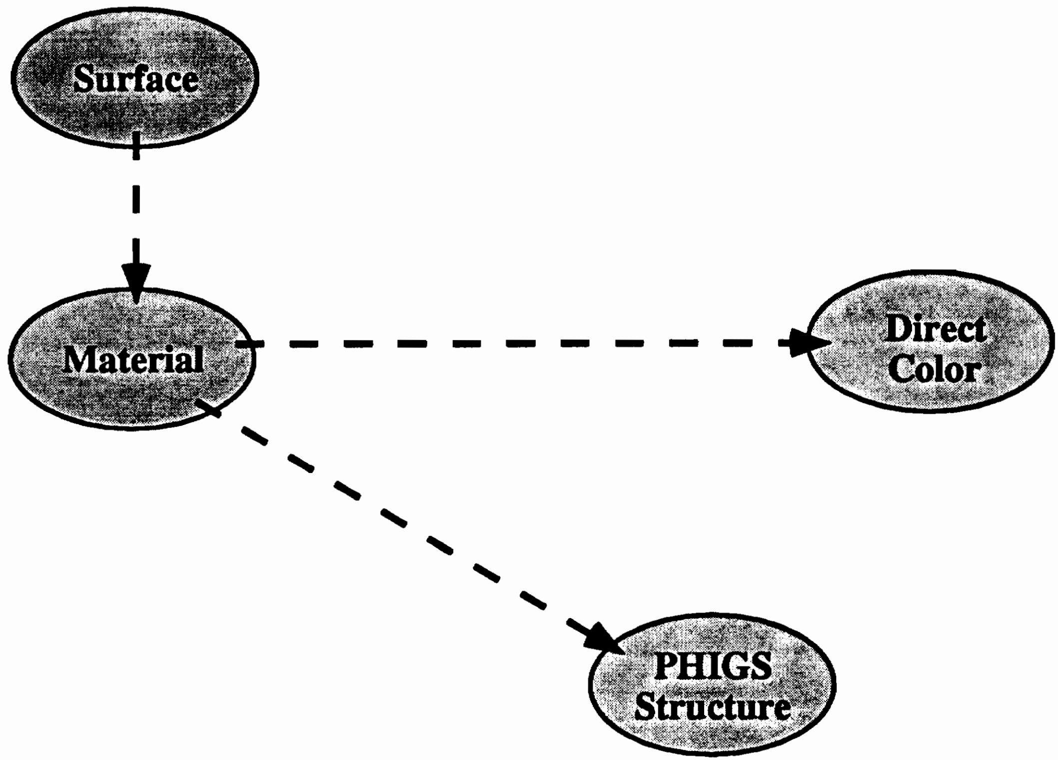
Material Properties

Designers who are creating surfaces for objects will invariably wish to display these surfaces. In order to provide a realistic model of the surface, several attributes must be set which depend on the type of material the surface represents. For displaying purposes, properties such as ambient reflection, diffuse reflection, specularity, and color are defined. Although some applications do not model physical surfaces and are purely mathematical or statistical in nature, the use of material properties may still be used to facilitate the rapid production of aesthetically pleasing displays.

To aid the user in setting material properties, a library is constructed which contains commonly used materials with their respective surface properties. The user may also create additional libraries to contain materials which are not contained in the provided library.

Many operations are provided to the user through the Material Property class. The lighting and color properties of an object may be queried or modified quickly and easily. Complete descriptions of the Material Property and Material Library classes are given in Appendix C.

The use of the Material Property class enables the user to display models in such a way that the output is realistic and accurate in relation to the material they are attempting to model. Furthermore, this is possible without extensive knowledge of items such as the diffuse lighting exponent of a material; however, these values may be set independently also.



Control Relationship

Figure 12 Control of Surface Properties

Additional surface properties may be added to the Material Property class in the future to handle properties related to CAD and CAM such as roughness, strength, and processing specifications.

The data contained in the Material class include:

```
char*            name
Direct_Color*    surface_color
Direct_Color*    specular_color
float            ambient_reflection_coefficient
float            diffuse_reflection_coefficient
float            specular_coefficient
float            specular_exponent
float            transparency_coefficient
```

In order to construct a material, each of the properties is set. Future modification may allow the specification of any number of values, with the rest being set to default values.

Each of the data fields may be modified using set methods, and the current values may be found using the get methods. The class may be used to set the values into a PHIGS structure by utilizing the set_PHIGS_property() function. This function requires that a pointer to a PHIGS structure be passed into the routine so that the structure may be opened, and the current values for each of the material properties data elements is updated. These properties may be removed using the remove_PHIGS_properties() function during a structure open state.

```
set_PHIGS_properties( PHIGS_Structure* phigs_structure)
remove_PHIGS_properties()
```

The PHIGS structure used in this class is an object oriented version of the PHIGS structure.

The color properties of a surface are created using the Direct Color class and Direct Color Libraries. These classes are created in a manner similar to that of the Material Property and Material Library classes. Colors are defined using their RGB values, and may then be used as direct colors in PHIGS. Although indexed colors are not used with the current implementation, it is possible to add this to the environment. The use of direct color keeps the user from the potential conflicts that may arise when using color indices. Furthermore, the user is not limited by the number of indices allowed on a particular workstation.

Using the Direct Color class, any number (within memor limitations) of colors may be defined. These colors may then be referred to using the objects name. Thus, the user has available an ability to create an endless array of colors for future use. The data elements of the color class inclue the red, green, and blue values which define the color, and the character string name which describes the color.

```
char*      name
float      red
float      green
float      blue
```

Each of the red, green, and blue values should range between zero and one. The class allows for the modification of each of the data fields using the standard set operations, and obtaining current values through the use of the get functions. Three additional methods are of significant merit. The equal operator sets a new color equal to a previous color, in all respects except for the name of the color. The change_intensity method allows the user to brighten or darken a color without changing the shade. And finally, the get_PHIGS_form

returns a three dimensional arrary of floats (pointer) which contains the RGB value of the color.

```
operator=( new_direct_color)
change_intensity( scalar)
get_PHIGS_form()
```

Complete descriptions of the Direct Color and Direct Color Library classes are given in Appendix C.

Primitives

Although the primitive class is not a fundamental class in the framework, it demonstrates some of the future directions in which the framework may be expanded. Users creating models often think of shapes at a higher level than points, vectors, curves and surfaces. Instead of these simple geometric entities, more complex primitives are used. These may include curve primitives such as:

- square
- triangle
- circle
- ellipse

And surface primitives such as:

- square patch
- triangular patch
- circular patch
- elliptical patch
- cube
- cylinder
- conic

Each of these primitives utilizes the appropriate curve and surface classes to create the more complex geometry. Multiple constructors allow the user a variety of methods for creating each shape, as well as methods for displaying the shapes through PHIGS. The use of

primitives allows for additional information to be encapsulated in the geometry. This characteristic data takes on great importance in the design and production complex systems.

More complex primitives may be created by the programmer to fit custom applications. For example, for the development of an architecture application, primitives such as walls, doors, windows, staircases, etc. may be of particular importance. The development of object libraries for various applications may enhance the development process.

Tools Used

The development and testing of the framework utilized IBM RS/6000's operating under UNIX, AIX version 3.2. This hardware provided the necessary power for the high speed arithmetic calculations needed in graphical representation and manipulation.

All of the developed code is based on standard C++ as referenced by the Annotated C++ Reference Manual [Elli90] and Bjarne Stroustrup in The C++ Programming Language [Stro91]. C++ is an extension of the C programming language which allows for the needs of object oriented programming. C++ is rapidly becoming the dominant object oriented language and the de facto standard for complex application programs.

PHIGS, the american and international standard for 3-D graphics, was chosen as the graphical language for the framework. IBM's version of PHIGS, graPHIGS provided the base for the developed display routines. In some instances, the use of graPHIGS extensions (PHIGS+) were utilized for advanced functionality.

The use of graphical languages is frequently debated between developers. Fleming points out some of the strengths and weaknesses of using graPHIGS for CAD programs [Flem91]. It is important to note that the framework is designed to allow for the addition of new graphical languages to the base environment.

Sample Usage

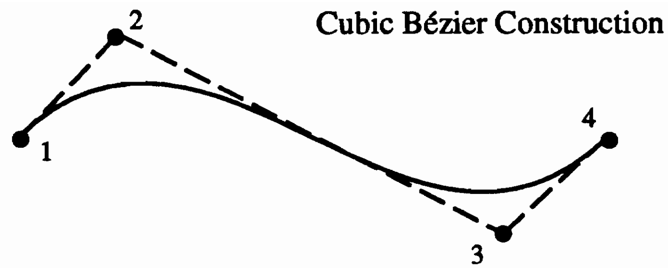
The creation and visualization of curves and surfaces is greatly facilitated using the Curve and Surface Framework. This section contains several sample programs which use the framework to create graphical representations of a variety of objects using several different methods provided by the framework. The curves and surfaces produced by each of the sample programs are presented in the Results section.

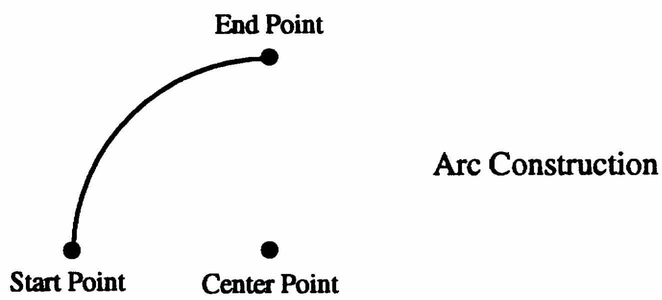
Example 1

The creation of curves using the framework is straightforward. The following example demonstrates the creation and display of three curves. For each curve (cubic Bézier, arc, and polyline) a figure showing the resulting display is also shown.

The cubic Bézier is constructed through the input of four control points. Graphical representation is possible through the use of a NUBS representation as discussed in the Class Overview section. Similar methods are used for representing the arc primitive graphically.

```
#include Curve_and_Surface_Framework.h

main()
{
	// Create A Cubic Bezier Curve
	Point	bezier_point_1(	0.0,  0.0, 0.0);
	Point	bezier_point_2(	0.5,  0.5, 0.0);
	Point	bezier_point_3(	1.0, -0.5, 0.0);
	Point	bezier_point_4(	1.5,  0.0, 0.0);

	Cubic_Bezier	bezier(bezier_point_1, bezier_point_2, bezier_point_3,
	bezier_point_4);


	// Create An Arc
	Point	arc_start_point(	0.0, 0.5, 0.0);
	Point	arc_center_point(	0.0, 0.0, 0.0);
	Point	arc_end_point(	0.5, 0.0, 0.0);

	Arc	arc(arc_start_point, arc_center_point, arc_end_point);


	// Create A Polyline
	Point	point_1(	0.0, 0.5, 0.0);
	Point	point_2(	0.5, 0.9, 0.0);
	Point	point_3(	0.7, 0.5, 0.0);
	Point	point_4(	1.4, 0.5, 0.0);
	Point	point_5(	1.4, 0.0, 0.0);
	Point	point_6(	0.0, 0.0, 0.0);

	Point	polyline_points[6];
	polyline_points[0] = point_1;
	polyline_points[1] = point_2;
	polyline_points[2] = point_3;
	polyline_points[3] = point_4;
	polyline_points[4] = point_5;
	polyline_points[5] = point_6;

	int	number_of_points = 6;
	Polyline	polyline(number_of_points, polyline_points);


	// Display Curves
	bezier.create_PHIGS_structure()
	bezier.create_PHIGS_polygon()

	arc.create_PHIGS_structure
	arc.create_PHIGS_center_point()

	polyline.create_PHIGS_structure()

}
```





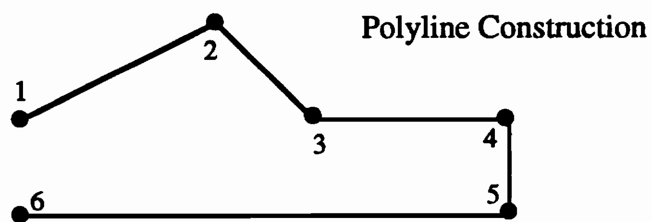


Figure 13 Construction Methods For Simple Curves

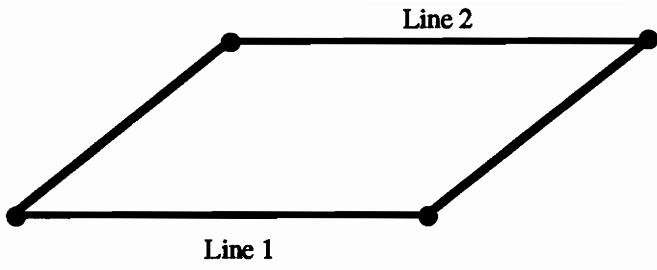
Example 2

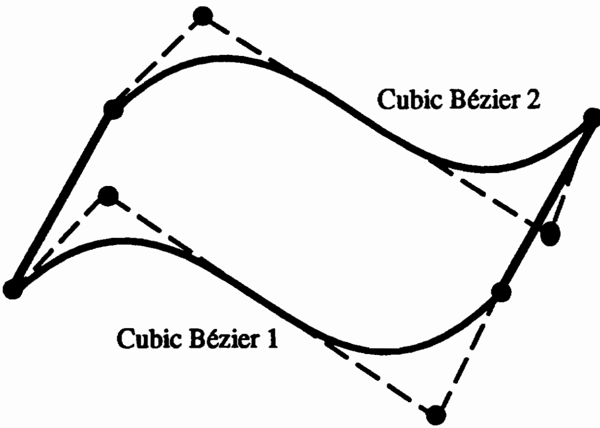
There are many methods which may be used to create a surface. In this example several suface patches will be created and displayed. First, a planar surface is constructed by sweeping a line segment along a linear curve. Next, a ruled surface is created by defining a Bézier curve, and sweeping it along a linear curve. In creating the NUBS surface, the tangents which are defined by the cubic Bézier curve are used. Thus four points, and eight tangents are given (four tangents from Bézier curve and four tangents from linear sweep). Finally, a free-form surface is defined through the definition of four points and eight tangents to form a C^2 continuous surface in both parametric directions.. The tangents combined with the continuity conditions at each point are used to create the resulting NUBS surface through the inversion process provided by Fleming's Advanced Modeling Toolkit.

Each surface may be assigned suface properties in order to create an aesthetically pleasing display.

```
#include Curve_And_Surface_Framework.h

main()
{
    // Create Arc 1
    Point       arc_1_start_point(       0.0, 2.0, 0.0);
    Point       arc_1_center_point(      0.0, 1.5, 0.0);
    Point       arc_1_end_point(        -0.5, 1.5, 0.0);

    Arc         arc_1(arc_1_start_point, arc_1_center_point, arc_1_end_point);


    // Create Arc 2(
    Point       arc_2_start_point       = arc_1_end_point;
    Point       arc_2_center_point(     -1.5, 1.5, 0.0);
    Point       arc_2_end_point(        -1.0, 1.0, 0.0);

    Arc         arc_2(arc_2_start_point, arc_2_center_point, arc_2_end_point);


    // Create A Cubic Bezier Curve
    Point       bezier_point_1 =        arc_2_end_point;
    Point       bezier_point_2(         -1.5,  0.6, 0.0);
    Point       bezier_point_3(          0.1,  0.2, 0.0);
    Point       bezier_point_4(         -1.2, -0.2, 0.0);

    Cubic_Bezier        cubic_bezier(bezier_point_1, bezier_point_2, bezier_point_3,
    bezier_point_4);


    // Create A Polyline
    Point       line_a_start =  bezier_point_4;
    Point       line_a_end(     -1.2, -0.4, 0.0);
    Point       line_b_end(     0.1, -0.4, 0.0);

    Point       polyline_points[3];
    polyline_points[0] = line_a_start;
    polyline_points[1] = line_a_end;
    polyline_points[2] = line_b_end;

    int  number_of_points = 3;
    Polyline    polyline(number_of_points, polyline_points);


    // Approximate All Curves With A Single NUBS Curve
    Curve*      curves[4];
    curve[0]    = &arc_1
    curve[1]    = &arc_2
    curve[2]    = &bezier;
    curve[3]    = &polyline;

    int  number_of_curves = 4;
    NUBS_Curve          nubs_curve(number_of_curves, curve);
```

```
// Approximate All Curves With A Single Polyline Curve
Polyline        polyline_approximation(number_of_curves, curve);


// Create Axis of Revolution
Point           axis_start_point(       0.0, -2.0, 0.0);
Point           axis_end_point(         0.0,  2.0, 0.0);

Line            axis_of_revolution(axis_start_point, axis_end_point);


// Create B-spline Surface Of Revolution
NUBS_Surface        nubs_surface(nubs_curve, axis_of_revolution);


// Create Planar Surface Of Revolution
Planar_Surface      planar_surface(polyline_approximation, axis_of_revolution);


// Set Surface Properties
Material_Library    material_library;           // Activates material library

nubs_surface.set_material(concrete);
planar_surface.set_material(steel);


// Create PHIGS Structure
nubs_surface.create_PHIGS_structure_with_surface_properties(phigs_structure);
planar_surface.create_PHIGS_structure_with_surface_properties(phigs_structure);

}
```





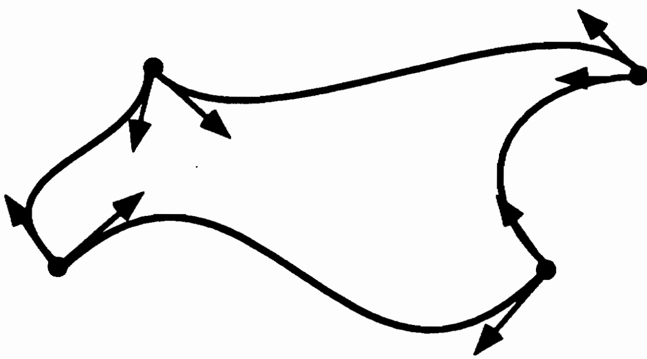


Figure 14 Construction Methods For Simple Surfaces

Example 3

The previous example demonstrated the ability to create a free-form surface. However, of more common use to the engineer is the creation of closed surfaces. These surfaces are used in the modeling of most engineering objects. Using the surface of revolution constructor and a closed curve, the resulting surface is guaranteed to be closed. Furthermore, if both the start and end points of the profile curve lie along the axis of rotation, again, the surface is guaranteed to be closed.

The following code shows the creation of a PHIGS Non-Uniform B-Spline (NUBS) surface model of an urn. The urn is made by creating a profile curve and sweeping it about an axis of rotation. Surface properties are set using the Material Library contained within the framework. A NUBS surface is created and displayed with the set material properties. In addition, a planar approximation of this surface may also be created with the addition of only a single line of code. Although this planar surface is only an approximation of the desired surface, this type of approximation may be necessary or desired in many applications. Graphical representation of this planar surface consists of triangular patches connecting the polyline approximations of the profile curve as it is rotated about the axis of revolution.

The first step in creating a suface of revolution is defining a profile curve. The curve may be made of any derived curve from the Curve base class. Each curve is joined in a head to tail manner, in order to form a continuous profile curve. Once the profile curve is

completed, the axis of revolution is defined using the Line class. Only the direction of the line is used,thus its position and magnitude are arbitrary.

The continuity of the NUBS surface of revolution in one parameter direction (along the profile curve) is dependent upon the curves used to create the profile, and in the second parameter direction, the surface is C^2 continuous. If the Planar surface is used to create the same surface the resulting continuity will be C^0 in both parameter directions.

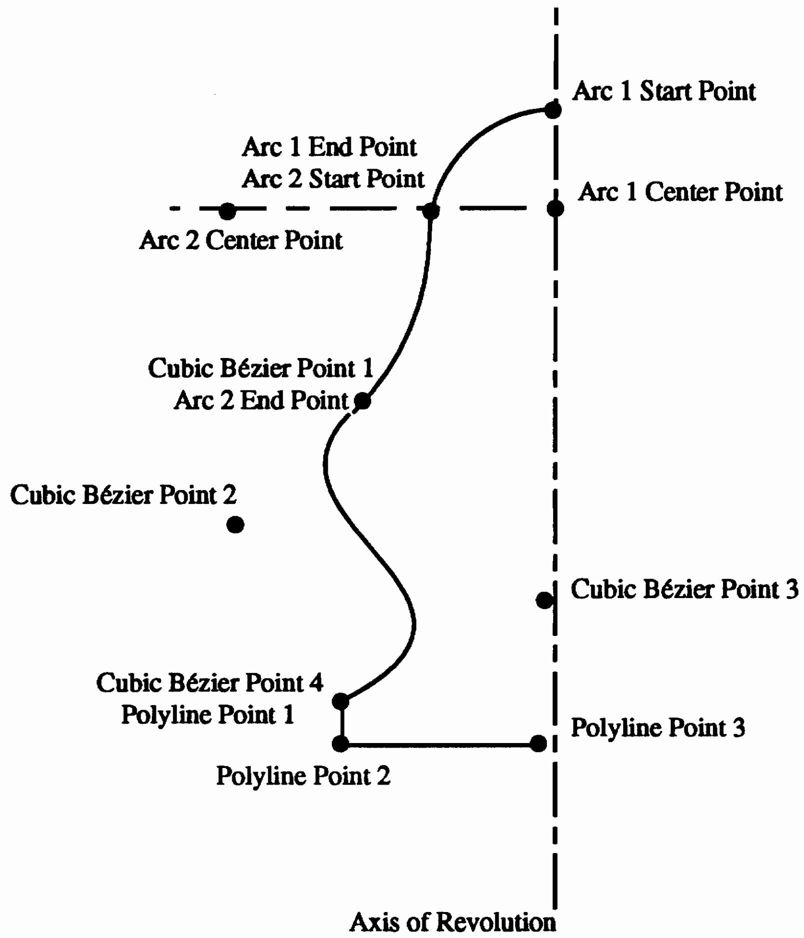


Figure 15 Construction Method For Surface of Revolution

```
#include Curve_And_Surface_Framework.h

main()
{
    // Create Arc 1
    Point       arc_1_start_point(      0.0, 2.0, 0.0);
    Point       arc_1_center_point(     0.0, 1.5, 0.0);
    Point       arc_1_end_point(       -0.5, 1.5, 0.0);

    Arc         arc_1(arc_1_start_point, arc_1_center_point, arc_1_end_point);


    // Create Arc 2(
    Point       arc_2_start_point       = arc_1_end_point;
    Point       arc_2_center_point(    -1.5, 1.5, 0.0);
    Point       arc_2_end_point(       -1.0, 1.0, 0.0);

    Arc         arc_2(arc_2_start_point, arc_2_center_point, arc_2_end_point);


    // Create A Cubic Bezier Curve
    Point       bezier_point_1 =        arc_2_end_point;
    Point       bezier_point_2(         -1.5,  0.6, 0.0);
    Point       bezier_point_3(          0.1,  0.2, 0.0);
    Point       bezier_point_4(         -1.2, -0.2, 0.0);

    Cubic_Bezier        bezier(bezier_point_1, bezier_point_2, bezier_point_3,
    bezier_point_4);


    // Create A Polyline
    Point       line_a_start =  bezier_point_4;
    Point       line_a_end(     -1.2, -0.4, 0.0);
    Point       line_b_end(     0.1, -0.4, 0.0);

    Point       polyline_points[3];
    polyline_points[0] = line_a_start;
    polyline_points[1] = line_a_end;
    polyline_points[2] = line_b_end;

    int  number_of_points = 3;
    Polyline    polyline(number_of_points, polyline_points);


    // Approximate All Curves Wtih A Single NUBS Curve
    Curve*      curves[4];
    curve[0]    = &arc_1
    curve[1]    = &arc_2
    curve[2]    = &bezier;
    curve[3]    = &polyline;

    int  number_of_curves = 4;
    NUBS_Curve          nubs_curve(number_of_curves, curve);
```

```
// Approximate All Curves Wtih A Single Polyline Curve
Polyline       polyline_approximation(number_of_curves, curve);

// Create Axis of Revolution
Point          axis_start_point(        0.0, -2.0, 0.0);
Point          axis_end_point(          0.0,  2.0, 0.0);

Line           axis_of_revolution(axis_start_point, axis_end_point);

// Create NUBS Surface Of Revolution
NUBS_Surface        nubs_surface(nubs_curve, axis_of_revolution);

// Create Planar Surface Of Revolution
Planar_Surface      planar_surface(polyline_approximation, axis_of_revolution);

// Set Surface Properties
Material_Library    material_library;          // Activates material library

nubs_surface.set_material(concrete);
planar_surface.set_material(steel);

// Create PHIGS Structure
nubs_surface.create_PHIGS_structure_with_surface_properties(phigs_structure);
planar_surface.create_PHIGS_structure_with_surface_properties(phigs_structure);

}
```

Example 4

The following code shows the creation of a PHIGS NUBS surface model of a cube with a fully enclosed slot removed. To create the model, five cross-sectional curves must be defined (two cross-sections are identical). The framework will create a surface by sweeping from one cross-section to the next. Thus, the ordering of the points is of critical importance, since the first point on the first cross section will be swept to the first point given on the next cross-section, and so forth.

In order to create a closed curve for each cross-section, the first point will be repeated as an end point. Likewise, the cross-section used at the beginning will be used again to close the surface.

Material properties may be set, and the object displayed using PHIGS.

Figure 16 Construction Method For Swept Surface

```
#include Curve_And_Surface_Framework.h

main()
{
    // Create Cross Section A
    Point           polyline_point_A[5];
    polyline_point_A[0]          = new Point(-1.0, 1.0, 0.0);
    polyline_point_A[1]          = new Point( 1.0,  1.0, 0.0);
    polyline_point_A[2]          = new Point( 1.0, -1.0, 0.0);
    polyline_point_A[3]          = new Point(-1.0, -1.0, 0.0);
    polyline_A[4]                = polyline_point_A[0];

    int  number_of_points = 5;
    Polyline        polyline_A(number_of_points, composite_point_A);


    // Create Cross Section B
    Point           polyline_point_B[5];
    polyline_point_B[0]          = new Point(-3.0, 3.0, 0.0);
    polyline_point_B[1]          = new Point( 3.0,  3.0, 0.0);
    polyline_point_B[2]          = new Point( 3.0, -3.0, 0.0);
    polyline_point_B[3]          = new Point(-3.0, -3.0, 0.0);
    polyline_B[4]                = polyline_point_B[0];

    Polyline        polyline_B(number_of_points, polyline_point_B);


    // Create Cross Section C
    Point           polyline_point_C[5];
    polyline_point_C[0]          = new Point(-3.0, 3.0, 4.0);
    polyline_point_C[1]          = new Point( 3.0,  3.0, 4.0);
    polyline_point_C[2]          = new Point( 3.0, -3.0, 4.0);
    polyline_point_C[3]          = new Point(-3.0, -3.0, 4.0);
    polyline_C[4]                = polyline_point_C[0];

    Polyline        polyline_C(number_of_points, polyline_point_C);


    // Create Cross Section D
    Point           polyline_point_D[5];
    polyline_point_D[0]          = new Point(-1.0, 1.0, 4.0);
    polyline_point_D[1]          = new Point( 1.0,  1.0, 4.0);
    polyline_point_D[2]          = new Point( 1.0, -1.0, 4.0);
    polyline_point_D[3]          = new Point(-1.0, -1.0, 4.0);
    polyline_D[4]                = polyline_point_D[0];

    Polyline        polyline_D(number_of_points, polyline_point_D);


    // Sweep From Cross Section A-D Then Back To A
    Curve*          curves[4];
    curve[0]        = &polyline_A
    curve[1]        = &polyline_B
    curve[2]        = &polyline_C;
```

```
    curve[3]    = &polyline_D;
    curve[4]    = &polyline_A;


    // Set Continuities For Ruled Surface
    int continuities[5];
    continuities[0] = 0;
    continuities[1] = 0;
    continuities[2] = 0;
    continuities[3] = 0;
    continuities[4] = 0;

    int number_of_curves = 5;
    NUBS_Surface          nubs_surface(number_of_curves, curve, continuities);


    // Set Surface Properties
    Material_Library    material_library;       // Activates material library

    nubs_surface.set_material(concrete);


    // Create PHIGS Structure
    nubs_surface.create_PHIGS_structure_with_surface_properties(phigs_structure);
}
```

Results

The development and implementation of the five fundamental classes presented in the "Class Overview" sections has been accomplished. This includes the following classes:

- Point class
- Vector class
- Curve class
- Surface class
- Material Property class

These base classes form the foundation of the curve and surface framework. To demonstrate the potential of the framework, and to allow it to be used as a useful tool without further development, several additional classes were created. These include the following curves:

- Arc class
- Basis Curve class
- Cubic Bézier class
- Line Segment class
- Non-Uniform B-Spline (NURBS) Curve class
- Polyline class
- Polynomial Curve class

In addition, the following surface classes were successfully implemented:

- Non-Uniform Rational B-Spline (NURBS) Surface class
- Planar Surface class

The framework is integrated into a fully parametric, feature-based, solid modeling application enabler created for the IBM corporation [Mykl93]. The framework fully provides all of the graphical needs for representing models produced in the solid modeling environment developed by Lin [Lin93].

Further use of the framework includes the creation of graphical models by students of the Virginia Tech CAD Laboratory. These students rapidly created numerous models using the framework. This provided a test of the framework's usability. Students were able to understand and use the framework and complete simple drawings in minutes.

The following pages show several pictures taken from demonstration applications that utilize the framework. The first picture shows the initial solid modeling demonstration program which contained the original version of the framework. This early version contained a NUBS surface class which allowed the display of a variety of solid models.

As the framework advanced, the usability of the environment increased substantially. The following figures represent the results of the sample programs. These programs were developed independently of the solid modeler, and were all created in under ten minutes by inexperienced users. Note that the coding programmers generated may be found in the Sample Usage section.

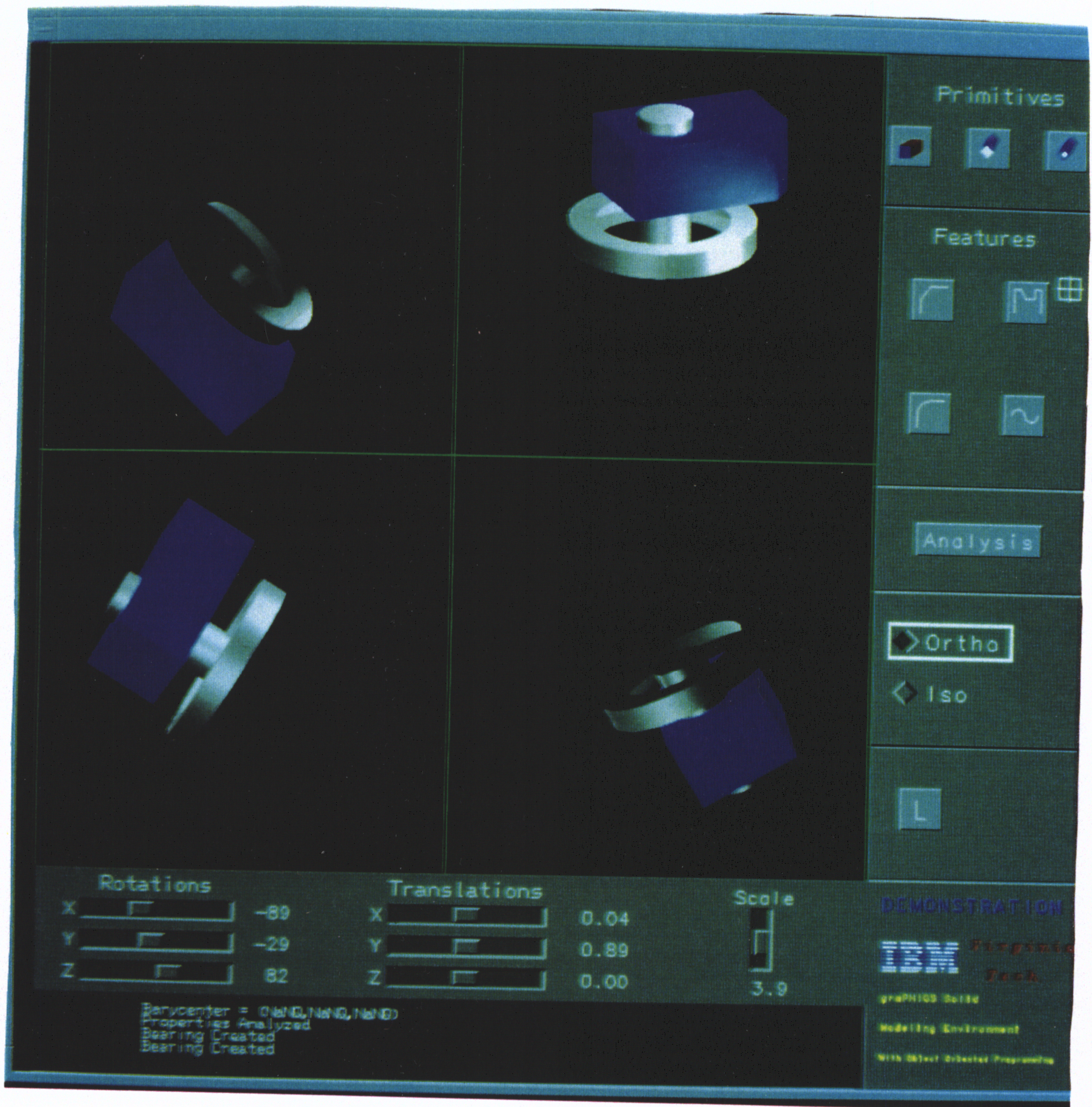


Figure 17 **NUBS Surface Representation Of Solid Model Utilizing The Solid Modeling Framework And The Curve And Surface Framework**

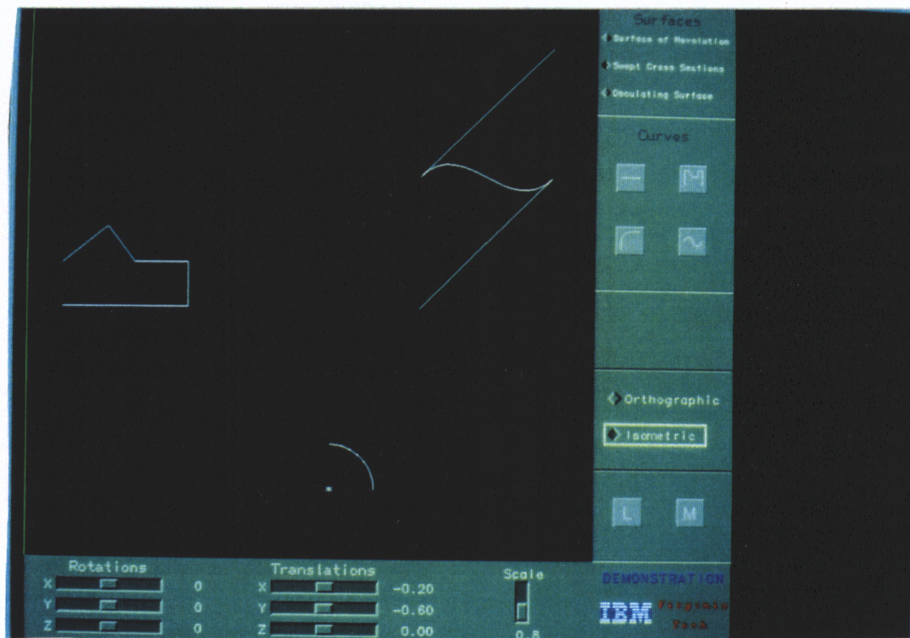


Figure 18 Simple Curves Generated Using The Framework

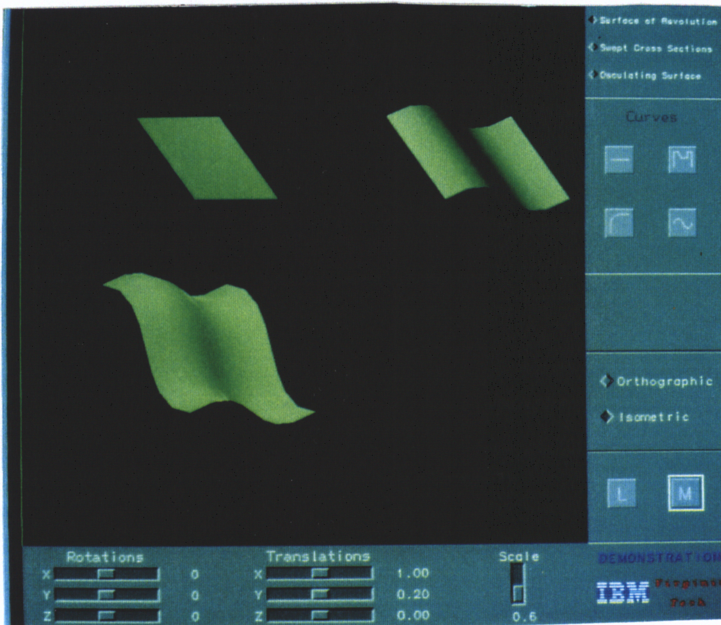


Figure 19 Simple Surfaces Generated Using The Framework

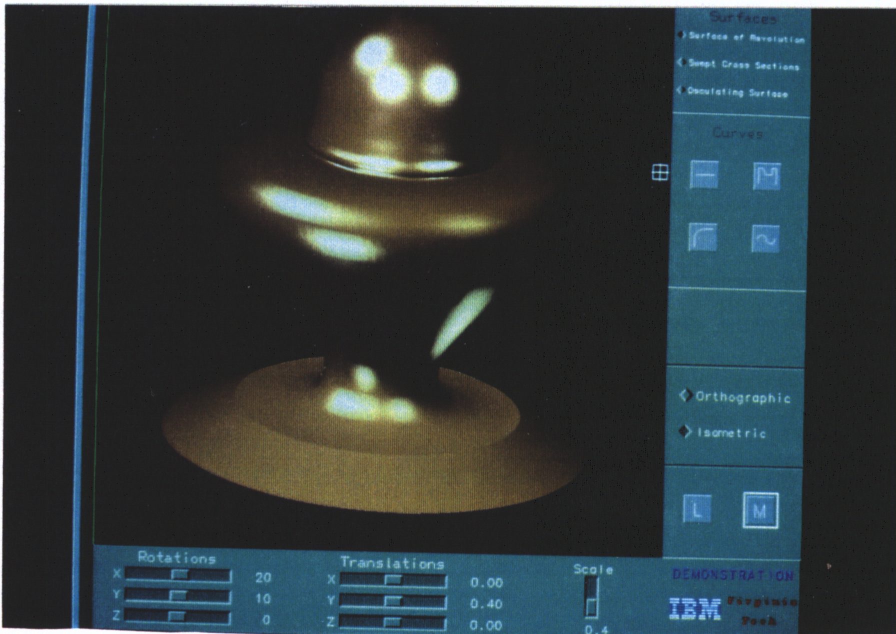


Figure 20 Surface Of Revolution Generated Using The Framework

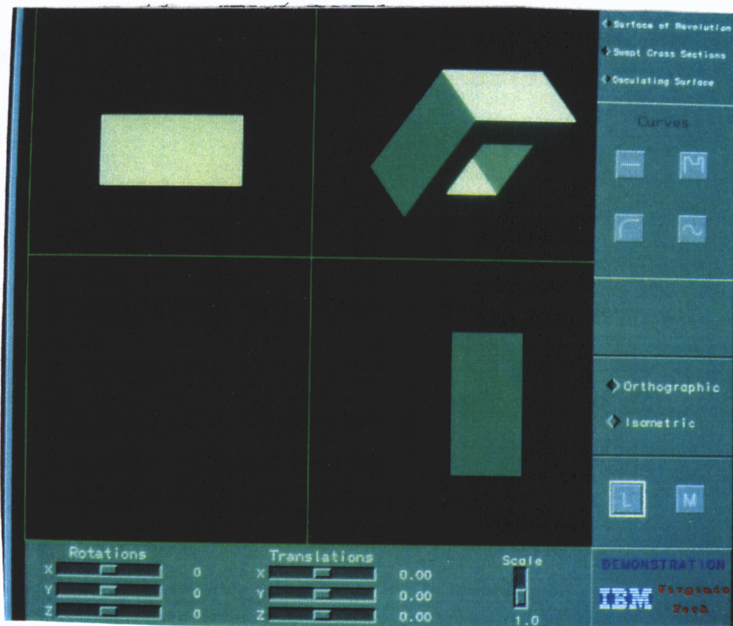


Figure 21　Swept Surface Generated Using The Framework

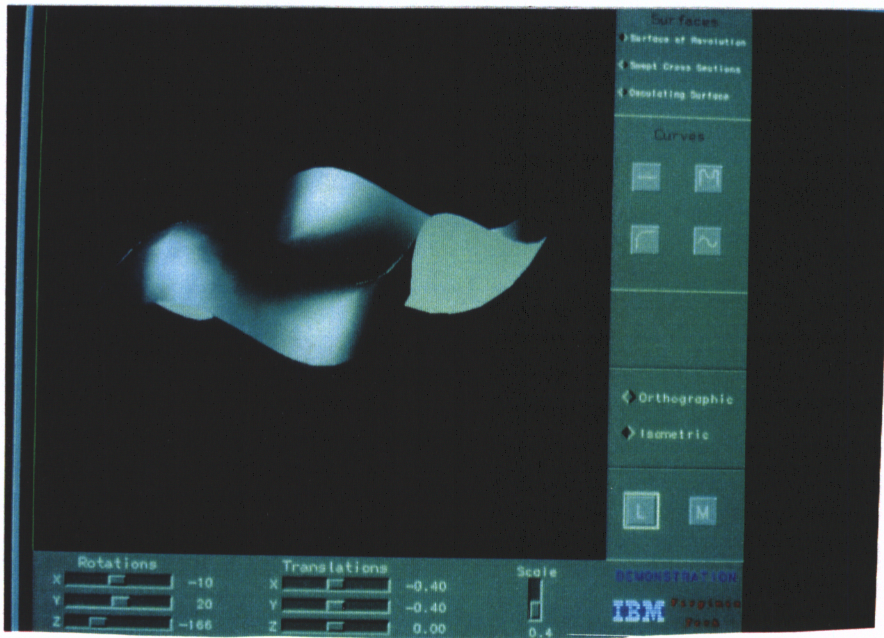


Figure 22 Free-Form Surface Generated Using The Framework

Conclusions

The Curve and Surface Framework has satisfied its original goal of facilitating CAD programmers in the development of curve and surface graphics applications. Each of the initial goals:

- Ease of use
- Use of graphics standards
- Close analogy to physical reality
- Extensibility

are addressed and satisfied by the framework.

Seven students of the Virginia Tech CAD Laboratory performed usability testing by writing simple programs which used the framework. These students responded positively to the use of the environment and were able to learn and use the framework to create engineering models in as little as ten minutes.

The creation of the Curve and Surface Framework shows great promise. The success of the framework both used independently, and in conjunction with other frameworks demonstrates its flexibility and usability. Furthermore, the ability of inexperienced users to rapidly create working code demonstrates the framework's ease of use.

The framework enables CAD programmers to write graphics routines quickly and easily. Furthermore, programmers are able to adapt their programs to the newest developments

and techniques in Computer Aided Geometric Design (CAGD) with minimal effort through enhancements to the framework. Finally, the need for a CAD programmer to be an expert in the fields of geometric modeling, computational geometry, graphics, and data exchange has been reduced.

The framework is successfully used by several programs developed for the IBM corporation [Mykl93]. These applications show the ability of the framework to be used with other applications in the development of custom CAD software.
Using the curve and surface framework in conjunction with other object-oriented enablers developed at the Virgina Tech CAD Laboratory creates a powerful environment for the CAD programmer.

It is recognized though, that even with the current success of the framework, without ongoing research and development, the work will quickly become out-dated and unused. Further improvement of the framework, especially in the areas of abstract base classes such as the Basis Function Curve class, and the addition of curves, surfaces, primitives, and materials to the base environment will further enhance the usability of the tool.

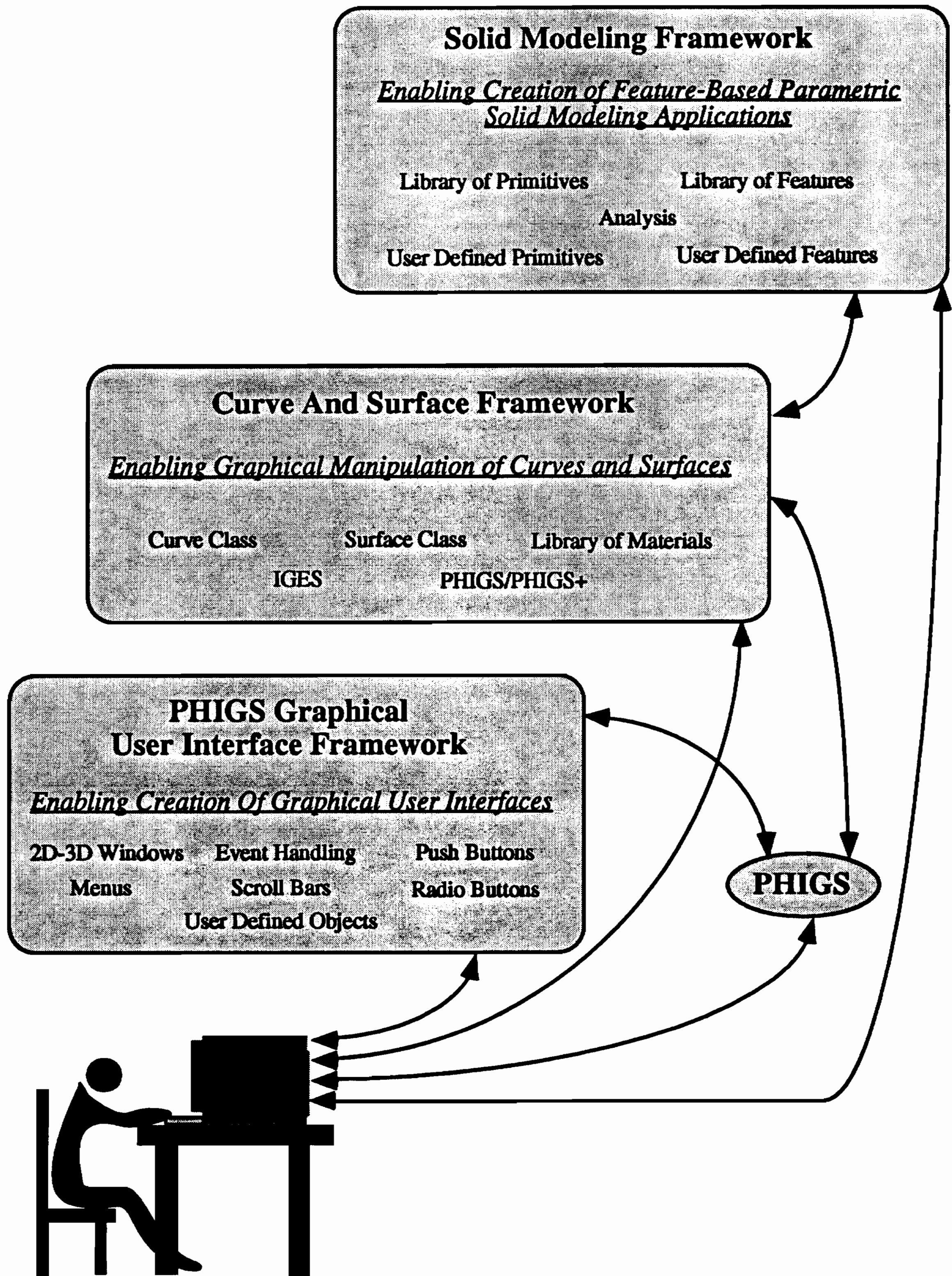


Figure 23 **Creating A Custom Engineering Design Modeler Using Object-Oriented Enablers**

References

[Bars87] RH Bartels, JC Beatty, and BA Barsky, An Introduction to Splines for Use in Computer Graphics and Geometric Modelling, Margan Kaufman Publisher 1987.

[Beac91] Beach, R.C., An Introduction To The Curves and Surfaces of Computer-Aided Design, Van Nostrand Reinhold, New York, 1991.

[Booc86] Booch, G., "Object-Oriented Development", *IEEE Transactions on Software Engineering*, Se-12, no.2, Febraruy 1986, pp. 211-221.

[Booc91] Booch, G., Object-Oriented Design With Applications, The Benjamin Cummings Publishing Company, Inc., Redwood City, Califorinia, 1991.

[Coon67] Coons, S.A., "Surfaces For Computer-Aided Design Of Space Forms", MIT Project MAC, MAC-TR-41, Massachusetts Institute of Technology, Cambridge, MA, June, 1967

[DeBo72] DeBoor, C. "On Calculating With B-Splines", *Journal of Approximation Therory*, no. 6, 1972, pp. 50-62.

[Fari88] Farin, G., Curves and Surfaces for Computer Aided Geometric Design, Academic Press, New York, 1988.

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

[Flem92a] Fleming, S., "Advanced B-Spline Modeling API", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1991.

[Flem92b] Fleming, S., A. Myklebust, "The Enhancement of PHIGS+ B-spline Functionality for Geometric Modeling", Fourth IFIP WG5.2 Workshop on Geometric Modeling in Computer Aided Design, Rensselaerville, New York, September 27 - October 1, 1992.

[Forr72] Forrest, A.R., "A New Curve Form For Computer-Aided Design", CAD Group Document No. 66. Cambridge University, England, June 1972.

[Fuax79] Faux, I.D., H. Pratt, , Computational Geometry for Design and Manufacture, John Wiley & Sons, New York, 1979.

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

[Jaya90] A. Myklebust, S. Jayaram, "Towards A Standardized Environment For The Creation Of Design And Manufacturing Software", *International Conference On Engineering Design (ICED)*, Dubrovnik, Yugoslavia, 1990.

[Jaya93] Jayaram, S. A. Myklebust, "Evaluating PHIGS for CAD Applications - A CASE Study", First Annual PHIGS Users Group Conference, Orlando, Florida, March 21-24, 1993.

[John92] Johnson, S.P., T. McReynolds, "Implementing Curves in C++", *Dr. Dobb's Journal*, December 1992, pp. 53a-64s.

[Klei92] R. Klein, P. Slusallek, "An Oject-Oriented Framework for Curves and Surfaces", *Curves and Surfaces in Computer Vision and Graphics III, Proc. SPIE 1830*, 1992, pp. 284-292.

[Knol90] Knolle N., "Object-Oriented Computer Graphics", Journal of Object-Oriented Programming, November/December, 1990, pp 71-74.

[Lin92] Lin, W.H., "An Object-Oriented Software Development Environment for Geomtric Modeling in Intelligent Computer Aided Design", Doctoral Dissertation, Mechanical Engineering, Virginial Polytechnic, 1992.

[Lin93] Lin, W.H., A. Myklebust, "A Constraint Driven Solid Modeling Open Environment", accepted for Second ACM/IEEE Symposium on Solid Modeleing and Applications, Montreal, Canada, May 19-21, 1993.

[Meye87] Meyer, B., "Reusability: The Case For Object-Oriented Design", *IEEE Software*, 4(2), March 1987, pp.50-64.

[Mort85] Mortenson, N.E., Geometric Modeling, John Wiley & Sons, New York, 1985.

[Mykl92a] Myklebust A.,W. Lin, S. Fleming, C. Feustel, A. Jacobson, S. Woyak, "A Research Report to the IBM Corportation", Computer Aided Design Laboratory of Virginia Polytechnic, February, 1992.

[Mykl92b] Myklebust A.,W. Lin, S. Fleming, A. Jacobson, S. Woyak, "A Research Report to the IBM Corportation", Computer Aided Design Laboratory of Virginia Polytechnic, July, 1992.

[Mykl93] Myklebust A.,W. Lin, A. Jacobson, S. Woyak, "A Research Report to the IBM Corportation", Computer Aided Design Laboratory of Virginia Polytechnic, May, 1993.

[Rank92] J.J. Rankin, D.A. Ott, "Exploring the Open Approach to FEA Integration In The Design Process", *The PRO Exchange*, Fairfax, Virginia, November/December 1992.

[Ross67] Ross, D.T., "The AED Approach to Generalized Computer-Aided Design", *Proceedings of the ACM National Conference*, 1967.

[Schi92] Schildt, Herbert, Teach Yourself C++, Osborne McGraw-Hill, Berkeley, 1992.

[Stro88] Stroustrup, B., "What Is Object-Oriented Programming?", *IEEE Software*, May 1988, pp. 10-20.

[Stro91] Stroustrup, B., The C++ Programming Language 2nd ed., Addison Wesley Publishing Company, New York, 1991.

[Wamp91] Wampler, S., "Development of a graPHIGS Based Object-Oriented Graphics System", *Proceedings of the 2nd International graPHIGS User's Group Conference and Workshop*, October, 1991, pp. 145-155.

[Wegn86] Wegner, P., "Classification In Object-Oriented Systems", *ACM: Sigplan Notices*, October 1986, pp. 173-182.

[Woya92] Woyak, S., "A Motif-Like Object-Oriented Interface Framework Using PHIGS", Thesis - Master of Science in Mechanical Engineering, Virginia Polytechnic Institute and State University, 1992.

[Woya93] Woyak, S., A. Myklebust, "A Motif-Like Object-Oriented Interface Framework Using PHIGS", First Annual PHIGS Users Group Conference, Orlando, Florida, March 21-24, 1993.

[Yama88] Yamaguchi, F., Curves and Surfaces in Computer Aided Geometric Design, Springer-Verlag, Berlin, Germany, 1988.

Appendix A - Curve Class Implementation

Curve Class

The abstract Curve base class defines the methods which are virtual to each of the derived Curve classes.

Variables

NONE

Methods

virtual	int	get_number_of_points
virtual	Point	get_points()
virtual	Vector	get_tangents()
virtual	int*	get_continuity()
virtual	void	set_number_of_points(int number_of_points)

Arc Class

The Arc class may be used to construct circular arcs, or a full circle. The Arc class uses the Point and Vector classes extensively both for the data structure and methods within.

Variables

```
Point   start_point
Point   center_point
Point   end_point
float   angle
Vector  normal
int     number_of_points;
```

Constructors

This constructor fully defines the arc. The magnitude of the normal is not important, however the sign and direction is.

```
Arc(  Point     start_point,
      Point     center_point,
      Vector    normal,
      float     angle)
```

This constructor is not a reliable method for creating an Arc because it does not specify the direction that the arc should follow from the start point to the end point.

```
Arc(  Point     start_point,
      Point     center_point,
      Point     end_point)
```

Set Methods

void	set_start_point(Point	start_point)
void	set_center_point(Point	center_point)
void	set_end_point(Point	end_point)
void	set_normal(Vector	normal)
void	set_continuity()	
void	set_number_of_points(int	number_of_points)

Get Methods

int*	get_continuity()	Returns an array of continuities (one for each point)
int	get_number_of_points()	
Point*	get_points()	
Vector*	get_tangents()	
Point	get_start_point()	
Point	get_center_point()	
Point	get_end_point()	
float	get_angle()	
Vector	get_normal()	

Utililty Methods

void	print()

PHIGS Methods

create_PHIGS_structure()
create_PHIGS_start_point()
create_PHIGS_center_point()
create_PHIGS_end_point()
create_PHIGS_structure(PHIGS_Structure* phigs_structure*)
create_PHIGS_start_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_center_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_end_point(PHIGS_Structure* phigs_structure*)

Cubic Bézier Class

The bézier curve is a fundamental curve in geometric modeling. The class is divided into four major sections: Construction, Virtual Set Methods, PHIGS Functions, and Utility Functions.

Variables

Point	start_point
Point	end_point
Vector	start_tangent
Vector	end_tangent
int*	continuity
int	number_of_points

Construction

The construction of the cubic bézier curve is traditionally performed through the specification of the control points which form its control polygon. Although other methods exist, this is the only on currently provided. The first and last points correspond to the start and end of the curve, and lie on the curve at the beginning and ending points. The inner control points relate to the tangent (3 x tangent) in both magnitude and direction. These points are specified also.

```
Cubic_Bezier( Point start_point, Vector start_tangent, Vector end_tangent, Point
              end_point)

Cubic_Bezier( Point control_point_1, Point control_point_2, Point
              control_point_3, Point control_point_4)
```

Set Methods

The set_number_of_points() function must be provided due to its inclusion in the base class. The use of a subdivision technique to create a bezier curve with the given number of control points may be implemented within this method; however, this has not been done as of yet.

```
set_number_of_points( number_of_points)

set_start_point( Point start_point)
set_end_point( Point end_point)
set_start_tangent( Vector start_tangent)
set_end_tangent( Vector end_tangent)
set_continuity()
```

Get Methods

The following virtual functions return the values contained within the class data structure. The curve may be approximated by the NUBS curve using the start points, and start tangent conditions.

```
int*        get_continuity()
int         get_number_of_points()
Point*      get_points()
Vector*     get_tangents()

Point       get_start_point()
Point       get_end_point()
Vector      get_start_tangent()
Vector      get_end_tangent()
```

PHIGS Methods

Because PHIGS does not include a method for representing the cubic bezier curve, in order to display the curve it must be placed into a form that PHIGS will allow. For accurate representations, the NUBS curve class is used. This should provide and exact representation of the bezier curve. In addition, several other display functions are also provided. When points are being displayed the polymarker is used. The display of the control hull utilizes the polyline, and the tangents use the line segment primitive.

```
create_PHIGS_structure()
create_PHIGS_control_polygon()
create_PHIGS_start_point()
create_PHIGS_start_tangent_point()
create_PHIGS_end_tangent_point()
create_PHIGS_end_point()
create_PHIGS_start_tangent()
create_PHIGS_end_tangent()
create_PHIGS_structure(PHIGS_Structure* phigs_structure*)
create_PHIGS_control_polygon(PHIGS_Structure* phigs_structure*)
create_PHIGS_start_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_start_tangent_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_end_tangent_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_end_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_start_tangent(PHIGS_Structure* phigs_structure*)
create_PHIGS_end_tangent(PHIGS_Structure* phigs_structure*)
```

Utility Methods

The print function allows all of the control points to be displayed. The points are printed in normal Cartesian form (x,y,z) from first to last control point.

```
print()
```

Non-Uniform B-Spline (NUBS) Curve Class

The NUBS curve is a versatile curve which may be constructed in many ways. Through the use of the Advanced Modeling B-Spline Toolkit [Flem92] a NUBS curve may be defined by data points, continuity conditions, and tangents. Using this methodology, the NUBS curve may be used to represent other curves which are of the Curve class type. More classical NUBS curve definitions may also be made through the specification of control points and knots.

Variables

Point*	data_point
int*	continuity
int	number_of_points
int	number_of_tangents
Vector*	tangent
int	paramaterization
float*	control_point
float*	knot
int	number_of_control_points
int	number_of_knots

Construction

NUBS_Curve(Curve** array_of_data_points, Vector* array_of_tangents, int paramatization_method, int number_of_points, int number_of_tangents, int* array_of_continuities)

NUBS_Curve(int number_of_curves, Curve** array_of_curves)

NUBS_Curve(Point* array_of_control_points, int* array_of_knots, int number_of_control_points, int number_of_knots)

Internal Methods

The allocate_space method uses the number of curves and the array of input curves to allocate the necessary space for the total number of data points, tangents, and continuities. This method should not be invoked unless these input curves have been constructed.

```
allocate_space( int number_of_curves, Curve** array_of_input_curves)
```

Each of the following set functions require that the required space has been allocated for each variable before being called (allocate_space() method). At each point, the continuity conditions may be determined by checking the continuity specified to each side of the point.

```
set_continuities( int number_of_curves, Curve** array_of_input_curves)
set_data_points( int number_of_curves, Curve** array_of_input_curves)
set_number_of_points( int number_of_curves, Curve** array_of_input_curves)
```

Set Methods

The set parameterization currently accepts an integer (1,2, or 3) which represents the type of paramatization being used. The main use for setting the paramatization is for the use of the Advanced Modeling B-Spline Toolkit [Flem91]. When this toolkit is invoked, these three parameterization may be used to yield different results [see Table 2].

The parameterization value is a private variable in the class and may be set during construction, or using the set_parameterization call. If the set_parameterization call is used, and the value of the parameterization variable has changed, the curve must be inverted

again, and control points solved. Thus, if the type of parameterization is known, it should be specified in the constructor. If no parameterization is chosen, and one is needed for inversion, the Chord Length (2) method is used as a default.

```
set_parameterization( int parameterization_type)
```

The set_number_of_points() call must be defined due to the fact that the class is a type of Curve. However, methods for subdividing the curve into any given number of sections have not been implemented at this time. For this reason, the set_number_of_points() method is a blank function, which has no effect.

```
set_number_of_points( int number_of_points)
```

Virtual Get Methods

Each of the following get functions are required by the Curve base class. Each of the functions returns the values stored in the private data area of the class. Because each of these functions may be called at any time after construction, all of the values must be solved for during the construction process. Currently, the use of the control point constructor, where the control points and knots are given by the user may cause a problem if the curve is approximated by the NUBS_Curve. Since no data points are given, the get_points() function will not return the appropriate values. This problem may be solved with the addition of virtual functions to the base Curve class, or by solving for the corresponding control points on the curve.

```
int*        get_continuity()
int         get_number_of_points()
```

Point*	get_points()
Vector*	get_tangents()
int	get_number_of_tangents()
int	get_parameterization()
float*	get_parametric_values()

Curve Inquiry

In several analysis routines, different values on the curve are needed. These range from control points values, data point values, to knot values. The get_parametric_values() function, an internal method, must be called if the curve was not defined using parametric values, in order to use the following functions:

int	get_number_of_control_points()
int	get_number_of_knots()
float*	get_control_points()
float*	get_knots()
Point*	get_data_points()

The get_data_points() method is only effective when the constructor used to create the curve uses data points as opposed to control points.

Analysis Methods

The following methods use algebraic equations to solve for the respective values. For each, the derivatives of the blending functions are used at the corresponding control point values. Left and right simply implies the direction in which the tangent is taken, from the left implies that the point is approached from the lower control value towards the higher control point value.

```
Vector*      get_left_tangent( Point data_point)
Vector*      get_right_tangent( Point data_point)
Vector*      get_left_acceleration( Point data_point)
Vector*      get_right_acceleration( Point data_point)
Vector*      get_left_normal( Point data_point)
Vector*      get_right_normal( Point data_point)
```

"Geometric Modeling" by Mortenson is used as a reference for each of these routines [Mort85]. The corresponding page numbers are referred to in the coding, and variable naming has followed Mortenson's convention.

PHIGS Methods

Several PHIGS functions may be invoked on the constructed curve for visualization. The create structure method uses the NUBS functionality to create a representation. Currently the composite fill area is not working properly, and thus it has been disabled, however, this method will produced a filled area for planar closed curves. The tangents and accelerations utilize a PHIGS line segment and the corresponding anlaysis functions. Both the left and right values are displayed. As with other classes, if the object oriented phigs structure is not used, the structure open state must be present.

```
create_PHIGS_structure()
create_PHIGS_control_polygon()
create_PHIGS_tangents()
create_PHIGS_accelerations()
create_PHIGS_fill_area()
create_PHIGS_structure( PHIGS_Structure phigs_structure*)
create_PHIGS_control_polygon( PHIGS_Structure phigs_structure*)
create_PHIGS_tangents( PHIGS_Structure phigs_structure*)
create_PHIGS_accelerations( PHIGS_Structure phigs_structure*)
create_PHIGS_fill_area( PHIGS_Structure phigs_structure*)
```

Utility Methods

The print function allows all of the control points to be displayed. The points are printed in normal Cartesian form (x,y,z) from first to last control point.

```
print()
```

Operators

The equal operator has been provided for ease in copying NUBS to new structures. All of the elements in the classes data structure are copied to the new structure (by value).

```
operator=(copy_of_NUBS)
```

Polyline Class

The Polyline is a very widely used curve in geometric modeling due primarily to its simplicity. The curve consits of line segments joined in a head to tail manner. This curve may be used to approximate other members of the Curve class. The class is divided into five major areas: Construction, Internal Methods, Virtual Set Methods, Virtual Get Functions, and PHIGS Functions.

Variables

```
Point   point
int*    continuity
int     number_of_points
```

Construction

The polyline is a set of line segments connected in a head to tail manner. The class has the ability to take other curves which are members of the Curve class, and approximate them with a polyline. This methodology uses the number of points set in the base curve, and currently has no iterative methodology employed for adjusting the number of points to acheive an error tolerant approximation. The second method of construction shown simply places line segments through the ordered array of points in a 'connect the dots' manner.

```
Polyline( int number_of_curves, Curve** array_of_curves)
Polyline( int number_of_points, Curve** array_of_points)
```

Internal Methods

The set_continuity() function is given to automatically set the continuity of each of the internal points to C^0, and the end points to C^1. The end points will remain C^1 continuos until the adjoining sections (if any are applied) are known. If these are also Polylines, the two C^1 ends, joined together will form a C^0 point (see NUBS Curve Class Continuity Table 1). This method uses the number of points stored in the class data structure to set each of the continuities.

```
set_continuity()
```

Virtual Set Methods

The set number of points operation is present, however, this value is automatically set during construction to be one greater than the number of line segments. To create more, or less points along the curve will require the implementation of an additional methodology. Currently, no use for this has been seen.

```
set_number_of_points( int number_of_points)
```

Virtual Get Methods

The following functions are supplied for satisfaction of the base class. The get_tangents() call returns an array of zeros, which yields the correct results. Methods may be incorporated to calculate the correct magnitude of the parametric tangent. Unfortunately, without new paramatization techniques, this may hinder the ability to mesh with other

curves when the composite cuve is combined with other curve types in the creation of a new curve (such as a NUBS curve). When the curves are reparamatized, the tangents will change, unless each curve has a unit paramaterization of some sort.

```
get_continuity()
get_number_of_points()
get_points()
get_tangents()
```

Analysis Methods

The get length analysis method is included in the Polyline class. This method uses the same get length function provided in the Line class iteratively. The total length is along the curve.

```
float    get_length()
```

PHIGS Methods

Several PHIGS operations are provided to assist in the visualization of the completed curve. The create_PHIGS_structure() uses the PHIGS polyline call. To create points, the three dimensional polymarker is employed.

```
create_PHIGS_structure()
create_PHIGS_start_point()
create_PHIGS_end_point()
create_PHIGS_points()
create_PHIGS_structure( PHIGS_Structure* phigs_structure*)
create_PHIGS_start_point( PHIGS_Structure* phigs_structure*)
create_PHIGS_end_point( PHIGS_Structure* phigs_structure*)
create_PHIGS_points( PHIGS_Structure* phigs_structure*)
```

Line Class

The Line class is perhaps the simplest curve class. The Line class defines a line segment by the definition of a start and end point. The class is divided into six major areas: Construction, Virtual Set Methods, Virtual Get Functions, Set Methods, PHIGS Functions, and Analsis Functions.

Variables

```
Point        start_point
Point        end_point
int*         continuity
int          number_of_points
```

Construction

Two methods are given for the construction of a line segment. The most commonly used, two point definition, and the point and vector form.

```
Line( Point start_point, Point end_point)
Line( Point start_point, Vector vector)
```

Virtual Set Methods

The set number of points operation divides the line segment into equal segments. The number of points may be set to values greater than one. However, the usefulness of increasing the number of points on a line segment for accuracy in approximation may not be present, this method may be utilized to match the number of points of one cross section with another. For example, if sweeping from a line to a polyline with four sections, the

line may be divided into for equal sections to yield the correct number of points. If this spacing is not correct, the Polyline may be used to give the correct spacing.

set_number_of_points()

Virtual Get Methods

Each of the following get functions returns a value stored in the class data structure with the exception of the get_tangents operation. This operations returns a zero magnitude tangent. As seen in the Composite Line class, the use of the zero tangent vector is necessary until new paramatization techniques are employed in the translation of groups of curves.

```
int*        get_continuity()
int         get_number_of_points()
Point*      get_points()
Vector*     get_tangents()
```

Set Methods

To aid the editing of a line segment, each of the following functions is included. Neither method will generate an error, and both will result in a line segment being drawn. In the case of a zero magnitued line, PHIGS does not generate an error, and neither does the Line class.

```
set_start_point( Point new_point)
set_end_point( Point end_point)
```

PHIGS Methods

Each of the following PHIGS methods utilizes PHIGS polyline and polymarker methods.

```
create_PHIGS_structure()
create_PHIGS_start_point()
create_PHIGS_end_point()
create_PHIGS_structure( PHIGS_Structure* phigs_structure*)
create_PHIGS_start_point(PHIGS_Structure* phigs_structure*)
create_PHIGS_end_point(PHIGS_Structure* phigs_structure*)
```

Analysis Methods

Frequently, the distance between a point and a line is necessary. This functionality is included in the Line class. The get_distance() method finds the shortest (perpendicular) distance between a point and the given line. The function returns both the distance and the direction in a Vector value. This method is well utilized in the creation of surfaces of revolution. In these functions, the distance from a profile curve point, to the axis of rotation (a Line) is found. This distance is then used as a vector to be rotated as additional cross sections are determined. Using this methodology, the line segment is treated as a line. If the shortest distance does not fall on the existing line segment, no error is generated, instead the value of the vector which is found is returned. If the magnitude of the line segment is zero, an error will be generated (division by zero), unable to determine direction of line segment.

```
float    get_distance( Point point)
```

Appendix B - Surface Class Implementation

Surface Class

The abstract base Surface class defines all of the virtual methods which must be defined in the derived surface classes. All surfaces are defined by a rectangular array of points and stores as isoparametric curves along one direction. Material properties and colors may be set to each surface. Additional surface properties may be added in future versions.

Variables

NONE

Methods

```
virtual    void           set_material( Material* material)
virtual    void           set_color( Direct_Color* color)
virtual    Material*      get_material()
virtual    Direct_Color*  get_color()

virtual    int            get_number_of_curves()
virtual    Curve**        get_curves()
virtual    int**          get_continuity()
virtual    Vector*        get_tangents()
```

Non-Uniform B-Spline (NUBS) Surface Class

The NUBS surface class utilized the NUBS curve class. The NUBS surface may be generated using either points on the surface along with the desired continuity conditions and corresponding tangents, or using control points and knots values. The class allows surface to be created using a variety of methods described in the construction section. The six major areas of the NUBS surface class are as follows: Construction, Internal Methods, Virtual Set Methods, Virtual Get Functions, PHIGS Functions, and Analysis Functions.

Variables

int	number_of_curves
NUBS_Curve**	curves
Vector*	tangent_v
int*	continuity_v
int	parameterization
int	points_per_curve
int	number_of_u_tangents
int	number_of_v_tangents
Material*	material
Direct_Color*	direct_color
PHIGS_Structure*	phigs_structure

Constructors

The NUBS Surface may be created using one of several methods. These methods include sweep operations, surface of revolution, and control point definitions. When curves are being input to the constructor, they are accepted as an array of Curve pointers. Thus, each curve in the array must be a member of the Curve class. Because the constructor contains pointers to the curves, any of the Curve class virtual functions may be accessed. This allows additional curves to be added to the base class without effecting the surface classes.

NUBS_Surface(int number_of_cross_sections, Curve** array_of_curves, int* continuities)

Using this constructor, two or more cross sectional curves are specified. These curves are then connected using a NUBS Surface. If the continuity variable is set to zero, then sweep is obtained by creating four sided patches which use linear segments between each cross section. If the continuity is set to equal two, then the resulting surface is rounded, and the continuity at each cross section is maintained as C^2 continuous.

Sweeping along a guide curve yields similar results to those shown above; however, using this constructor, the curve shape between sections is more easily controlled and the continuities at a cross section may be set to C^1.

NUBS_Surface(Curve** cross_sectional_curves, Curve** guide_curve)

Using the surface of revolution construction, a surface may be created by specifying a profile curve and an axis of revolution. The profile curve is created similar to cross sectional curves (as an array of Curve pointers). The profile curve is defined using the Line class.

NUBS_Surface(Curve** profile_curve_array, Line axis_of_revolution)

NUBS_Surface(Curve** profile_curve_array, Line* axis_of_revolution, int number_of_cross_sections)

Internal Methods

Depending on the method being used to construct the surface, a number of curves must be created. For example, if the surface of revolution is being used, each of the transformed

profile curves must be calculated. With the swept cross section along a guide curve, again the transformed curves must be solved. Internal methods for these operations enable the user to create new surfaces readily. Depending on the input arguements, different methods are applied.

```
create_NUBS_curves( Curve** array_of_curves)
create_NUBS_curves( Curve** profile_curve, Line* axis_of_revolution)
create_NUBS_curves( Curve** cross_section, Curve** guide_curve)
```

Additional functions are provided for allocating the correct amount of memory, setting the tangents along the v-direction (cross sectional curves are put into the u-direction), and the continuities in the v-direction. Again, operator overloading is used.

```
set_v_tangents()
set_v_tangents( Curve** guide_curve)
set_v_continuity( int* array_of_continuities)
set_v_continuity( Curve** guide_curve)
```

Virtual Set Methods

Two set functions are defined as virtual functions in the Surface base class. These functions must be satisfied in each of the derived classes. The set_material() function sets a pointer to the material properties of the surface. If no material properties are set, a default property is used. This material property is currenlty being used to create aesthetically pleasing displays; however, engineering material constants could be added to increase the functionality of the material properties.

```
set_material( Material* material)
```

The set_color() operation also is used for display purposes. This function sets a pointer to a direct color value (RGB).

set_color(Direct_Color* color)

Virtual Get Methods

Similar to the Curve class, the Surface class incorporates several virtual functions so that the interpretation of surfaces may be made in a general way. This allows additional curves and surfaces to be added to the framework without effecting current classes.

```
Material*          get_material_property()
Direct_Color*      get_color()
Curve*             get_curves()
int*               get_continuity()
Vector*            get_tangents()
```

The get_continuity() and get_tangents() functions return values in the v-direction only. The values for the u-components of each of these calls is contained in the curves. By keeping a pointer to each curve used to create the surface, the u values are not needed.

PHIGS Methods

When creating a surface using PHIGS, several variables control the appearance of the displayed results. The Material Property class controls many of the properties associated wth lighting and shading. This is done to simplify the programmers task. The surface may be constructed using the material properties contained in the Surface class, or without those properties. However, if the material properties are not used, the user will need to insert surface properties into the PHIGS structure to view the object. This latter methodology enables the user to create surfaces without using the surface properties contained within the Material class. If the user wishes to not use the phigs_structure class provided, the initial function is provided which does not use this class.

```
create_PHIGS_structure()
create_PHIGS_control_net()
create_PHIGS_u_tangents()
create_PHIGS_v_tangents()
create_PHIGS_u_acceleration()
create_PHIGS_v_acceleration()
create_PHIGS_structure( PHIGS_Structure phigs_structure*)
create_PHIGS_control_net( PHIGS_Structure* phigs_structure*)
create_PHIGS_u_tangents( PHIGS_Structure* phigs_structure*)
create_PHIGS_v_tangents( PHIGS_Structure* phigs_structure*)
create_PHIGS_u_acceleration( PHIGS_Structure* phigs_structure*)
create_PHIGS_v_acceleration( PHIGS_Structure* phigs_structure*)
create_PHIGS_structure_without_surface_properties( PHIGS_Structure*
                    phigs_structure)
create_PHIGS_structure_with_surface_properties( PHIGS_Structure*
                    phigs_structure)
set_PHIGS_surface_properties( PHIGS_Structure* phigs_structure)
```

Analysis Methods

The ability to analyze a surface is possible through the use of the NUBS curve class. Each curve may be analyzed using by getting the specific curve, or the entire surface may be analyzed using one of the following functions:

```
get_tangents_u_direction()
get_tangents_v_direction()
get_acceleration_u_direction()
get_acceleration_v_direction
```

Planar Surface Class

The planar suface is a widely used surface in geometric modeling due to its simplicity. The triangular patch is used to represent the surface because by definition, it may be defined as a planar surface (three points defines a plane). The five major areas of the Planar Surface class are: Construction, Internal Methods, Virtual Set Methods, Virtual Get Functions, and PHIGS Funtions.

Variables

int	number_of_curves
Polyline**	curves
int	parameterization
int	points_per_curve
int*	continuity_v
Vector*	tangent_v
Material*	material
Direct_Color*	color
PHIGS_Structure*	phigs_structure

Construction

The Planar Surface class includes constructors for swept surfaces and surfaces of revolution. The swept surface construction may include the specification of multiple cross-sections, or a single cross section and a guide curve. Each of the curves used in the constructors must be derived from the Curve base class. The Polyine class is used to approximate the input curves with linear segments. Currently iterative methods are not being used to increase the accuracy of the intial approximation; however, the ability to set the number of points along each of the input curves is a virtual function in the Curve base class, and thus could be added in the future.

```
Planar_Surface( int number_of_cross_sections, Curve** array_of_curves, int*
                continuities)
```

Using this constructor, two or more cross-sectional curves are specified. These curves are then connected using a Planar Surface.

Sweeping along a guide curve yields similar results to those shown above, since the surface is created by using linear segments between sections, the guide curve is first approximated using the Polyline class.

```
Planar_Surface( Curve** cross_sectional_curves, Curve** guide_curve)
```

Using the surface of revolution construction, a surface may be created by specifying a profile curve and an axis of revolution. The profile curve is created similar to cross-sectional curves (as an array of Curve pointers). The profile curve is defined using the Line class.

```
Planar_Surface( Curve** profile_curve_array, Line* axis_of_revolution)
Planar_Surface( Curve** profile_curve_array, Line* axis_of_revolution, int
                number_of_cross_sections)
```

Internal Methods

Depending on the method being used to construct the surface, a number of curves must be created. For example, if the surface of revolution is being used, each of the transformed profile curves must be calculated. With the swept cross-section along a guide curve, again the transformed curves must be solved. Internal methods for these operations enable the user to create new surfaces readily. Depending on the input arguements, different methods are applied.

```
create_linear_curves( Curve** array_of_curves)
create_linear_curves( Curve** array_of_curves, Curve** guide_curve)
create_linear_curves( Curve** profile_curve, Line* axis_of_revolution)
```

Additional functions are provided for allocating the correct amount of memory, setting the tangents along the v-direction (cross sectional curves are put into the u-direction), and the continuities in the v-direction. Again, operator overloading is used.

```
set_v_tangents()
set_v_tangents( Curve** guide_curve)
set_v_continuity( int* array_of_continuities)
set_v_continuity( Curve** guide_curve)
```

Virtual Set Methods

Two set functions are defined as virtual functions in the Surface base class. These functions must be satisfied in each of the derived classes. The set_material() function sets a pointer to the material properties of the surface. If no material properties are set, a default property is used. This material property is currenlty being used to create aesthetically pleasing displays; however, engineering material constants could be added to increase the functionality of the material properties.

```
set_material( Material* material)
```

The set_color() operation also is used for display purposes. This function sets a pointer to a direct color value (RGB).

```
set_color( Direct_Color* color)
```

Virtual Get Methods

Similar to the Curve class, the Surface class incorporates several virtual functions so that the interpretation of surfaces may be made in a general way. This allows additional curves and surfaces to be added to the framework without effecting current classes.

```
Material*        get_material_property()
Direct_Color*    get_color()
Curve*           get_curves()
int*             get_continuity()
Vector*          get_tangents()
```

The get_continuity() and get_tangents() functions return values in the v-direction only. The values for the u-components of each of these calls is contained in the curves. By keeping a pointer to each curve used to create the surface, the u values are not needed.

PHIGS Methods

Display of the surface is enabled using PHIGS functions. Due to the fact that the surface is planar in nature, filled polygons may be used to represent the final shaded image.
Each of the planar surfaces use three sided patches to guarantee that each facet is planar, although many surfaces could use planar four sided patches this would not guarantee that the resulting surface was planar. If the patches are relatively large, this may not yield an aesthetically appealing image due to the large triangular facets created. This problem may be solved by increasing the number of cross sections, or points along each curve.

When creating a surface using PHIGS, several variables control the appearance of the displayed results. The Material Property class controls many of the properties associated wth lighting and shading. This is done to simplify the programmers task. The surface may be constructed using the material properties contained in the Surface class, or without those properties. However, if the material properties are not used, the user will need to insert

surface properties into the PHIGS structure to view the object. This latter methodology enables the user to create surfaces without using the surface properties contained within the Material class. If the user wishes to not use the phigs_structure class provided, the initial function is provided which does not use this class.

```
create_PHIGS_structure()
create_PHIGS_structure_without_surface_properties( PHIGS_Structure*
                phigs_structure)
create_PHIGS_structure_with_surface_properties( PHIGS_Structure*
                phigs_structure)

set_PHIGS_surface_properties( PHIGS_Structure* phigs_structure)
```

Appendix C - Auxiliary Class Implementation

Point Class

The point class consists of the data and methods necessary for the creation and manipulation of three dimensional, cartesian coordinates. The constructors for the point class are as follows:

Variables

```
float   x
float   y
float   z
```

Constructors

```
Point( float x_value, float y_value, float z_value)
Point()
```

The latter constructor may be used in order to allocate the memory for the point, and values may then be set later using the provided set methods.

Operators

Vector arithmetic is provided with the addition and subtraction operators. Adding a vector to a point yields another point. Subtracting two points however, yields a vector. Finally an equals operator allows new points to be set to other points.

```
void        operator=( Point new_point)
```

Point	operator+(Vector added_vector)
Vector	operator-(Point subtracted_point)

Set and Get Methods

Set and get operations are also provided for each of the data elements in standard format get_element() or set_element(). These include:

float	get_x()
float	get_y
float	get_z()
void	set_x(float new_x)
void	set_y(float new_y)
void	set_z(float new_z)
void	set_xyz(float new_x, float new_y, float new_z)

A print utility is provided to allow quick output of vectors. This operation will output the three components in the form (x_value, y_value, z_value). No line feeds are placed into this print statement so that it may be more easily incorporated into other output routines.

print()

Vector Class

The vector class provides for many standard vector operations to be performed. Vectors are represented by three data elements (x,y,z). The constructors for the vector class allow for the full specification of the vector or just the allocation of the necessary memory.

Variables

```
float   x
float   y
float   z
```

Constructors

```
Vector( float x_value, float y_value, float z_value)
Vector()
```

Set and Get Methods

Operations for setting and getting each are similar to those shown for the Point class. The element to be changed, or returned is preceded by the key word get or set.

```
float     get_x()
float     get_y()
float     get_z()

void      set_x( float new_x)
void      set_y( float new_y)
void      set_z( float new_z)
void      set_xyz( float new_x, float new_y, float new_z)
```

Vector Manipulation Methods

The vector operations provided for the vector class are more extensive than those provided for the Point class. These include the vector arithmetic dot and cross operations, as well as the ability to calculate the magnitude of a vector, normalize a vector, or rotate a vector about an arbitrary axis. The dot and cross operations are also included outside of the Vector class. This allows the syntax for the method to be more intuitive to the user.

```
Vector    cross( vector_1, vector_2)
float     dot( vector_1, vector_2)
Vector    rotate( normal_vector, delta_radians)
void      normalize()
float     get_magnitude()
void      operator=( Vector new_vector)
void      operator=(Vector* new_vector)
Vector    operator+( Vector added_vector)
Vector    operator-( Vector subtracted_vector)
Vector    operator*( float scalar_value)
Vector    operator/( float scalar_value)
```

Utility Methods

A print utility is provided to allow quick output of vectors. This operation will output the three components in the form (x_value, y_value, z_value). No line feeds are placed into this print statement so that it may be more easily incorporated into other output routines.

```
print()
```

Material Property Class

The material class is the base class for materials to be created. This class is then used in the creation of material libraries. The user may use the methods in this class to modify materials from a material library, create new materials, or to create new libraries. Currently the material properties stored within the Material class only include PHIGS properties. This may change in the future to allow the user to store engineering information about the material, such as the relevant material constants. This may be very useful when analyzing objects created on a variety of CAD systems.

Variables

```
char*          name
Direct_Color*  surface_color
Direct_Color*  specular_color
float          ambient_reflection_coefficient
float          diffuse_reflection_coefficient
float          specular_coefficient
float          specular_exponent
float          transparency_coefficient
```

Constructors

In order to construct a material, each of the properties is set. Future modification may allow the specification of any number of values, with the rest being set to default values.

Each of the data fields may be modified using set methods, and the current values may be found using the get methods. The class may be used to set the values into a PHIGS

structure by utilizing the set_PHIGS_property() function. This function requires that a pointer to a PHIGS structure be passed into the routine so that the structure may be opened, and the current values for each of the material properties data elements is updated. These properties may be removed using the remove_PHIGS_properties() function during a structure open state.

```
set_PHIGS_properties( PHIGS_Structure* phigs_structure)
remove_PHIGS_properties()
```

The PHIGS structure used in this class is an object oriented version of the PHIGS structure.

Material Library Class

The Material Libary class is provided as an example of how to create material libraries using the Material Property class. The Material Library is activated by invoking the constructor during a program. By activating the library in this way, the flexibility and overhead of using the library is reduced. Several libraries may be created and destroyed during the operation of a program. Once the library is constructed, each of the materials may be used by simply referring to its object name (ex. steel, glass, rubber, etc...). Methods for manipulating these objects are contained in the Material Property class, and will alter the Material for only that use of the library. If the library is re-constructed, each of the values will return to its original state.

Direct Color Class

The Direct Color class provides an alternative to the restrictive color table which is provided by PHIGS. Using the Direct Color class, any number (within memor limitations) of colors may be defined. These colors may then be referred to using the objects name. Thus, the user has available an ability to create an endless array of colors for future use. The data elements of the color class inclue the red, green, and blue values which define the color, and the character string name which describes the color.

```
char*     name
float     red
float     green
float     blue
```

Each of the red, green, and blue values should range between zero and one. The class allows for the modification of each of the data fields using the standard set operations, and obtaining current values through the use of the get functions. Three additional methods are of significant merit. The equal operator sets a new color equal to a previous color, in all respects except for the name of the color. The change_intensity method allows the user to brighten or darken a color without changing the shade. And finally, the get_PHIGS_form returns a three dimensional arrary of floats (pointer) which contains the RGB value of the color.

```
void      operator=( new_direct_color)
void      change_intensity( scalar)
float*    get_PHIGS_form()
```

Direct Color Library Class

The Direct Color Library is very similar to the Material Property Library. Again, the library is constructed during the operation of the program through the use of the class constructor. Modification of the class elements utilizes the Direct Color class, and the individual member of the library which is to be changed. Modification of elements within the class may be replaced to the original values by re-constructing the library.

Vita

Alan Jacobson was born January 19, 1969 in Charlottesville, Virginia. He received his primary and secondary educations in the Albemarle County school systems. During his High School education he was introduced to science by a Biology teacher William Banks. Through this teacher, Alan was challenged to explore education beyond the classroom. This climaxed during his junior year in an award winning, International Science Fair project on "The Gravitational Preference of Apterous Drosophilia". Alan later attended the University of New Hampshire and received an Honors Degree in Mechanical Engineering with a minor in Electrical Engineering.

Currently Alan is deciding whether to further his engineering education in academia or in industry; or to switch fields back to his first interest, medicine.