Rule-Based Fuselage and Spine and Cross-Section Methods for Computer Aided Design of Aircraft Components

by

John H. Kelly

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

[Signatures]

Dr. Sankar Jayaram, Chairman

Dr. A. Myklebust

Dr. W.F. O'Brien

May 4, 1993

Blacksburg, VA
Abstract

In recent years, the use of computer-aided design (CAD) systems for conceptual aircraft design has greatly increased. As a result, new and better methods for creating surface models of aircraft geometry using dimensional parameters are needed.

One such method, the Rule-Based Fuselage method, was suggested by Lockheed. The Rule-Based Fuselage method allows an aircraft designer to define complex aircraft fuselage geometry by specifying the fuselage profile and individual parametric cross-sections along the fuselage.

This thesis describes the Rule-Based Fuselage method and discusses the implementation of the method in an interactive, object-oriented environment. Also included in this system is the Spine and Cross-Section method for creating arbitrarily shaped aircraft components.

The design and implementation of both the Rule-Based Fuselage and Spine and Cross-Section methods are described. The integration of these methods with the conceptual aircraft design code, ACSYNT, is also discussed.
Acknowledgments

First of all, I would like to thank both of my parents for all of their love and support throughout the years. I would also like to thank them on behalf of the Treasurer of Virginia Tech who has taken enough of their money to put off their retirement plans for another 50 years.

Thank you to my advisor, Dr. Jayaram, for having the faith and patience to make me one of your graduate students - good luck wherever you go in the future. And thank you to my other committee members Dr. Myklebust and Dr. O'Brien without whom this thesis would not be possible.

I must also thank the other Dr. Jayaram, Dr. Uma Jayaram, whose invaluable time and effort were the only reason that this thesis was published on time. Thank you for everything.

Without further adieu, I want to thank all of the people I have met over my 6 years (but hey, who’s counting?) at Virginia Tech, starting with my fellow graduate students:

Francisco Rivera  Although your knowledge of C, C++, and Macintosh computers may not have enriched my life, it certainly helped me get out of here! Thanks for all the late night coding sessions, good fun, and intense card games. Have fun back in California.

P-man  Parasuram Narayanan in real life. My good Indian friend and Unix guru, thanks for all your help and for teaching me a little about India (just stay away from that Bombay cocktail). Good luck to you at Ford (or wherever
you may go).

SV  Srinivas Dhulipala, world’s greatest Indian chef. I look forward to being in tears once again because you used too much spice (that’s OK, I’ll get you back with some of those sour balls)! Thanks for all the good times and good luck up in Michigan (keep an eye on the P-man for me). By the way, I still don’t like FEM.

Andy Steude  “So what do you wanna hear?” Thanks for coming in and bringing your CD player with you for those all night coding sessions. I don’t suppose you’ll be eating at Hardee’s or Kroger’s anytime soon.

Alan Jacobson  Thanks for the cool Blue Oyster Cult background - You see me now a veteran of 1000 psychic wars...

Jim Pascoe  The guy who got me hooked on Hearts and Spades. Thanks for everything, especially my bachelor party at your place.

I would also like to extend my thanks to all of the other CAD Lab ACSYNT geeks (and I mean that most affectionately): David Coe, Brett Malone (thanks for the Mac usage - sorry I kept locking it up!), Steve Uhorchak, Scott Woyak (cool menus), Shahab Hasan (supergeeky - BAD SU at ya), Dee Meier, and Darrell Early.

Now, I wish to express my gratitude to the people who have made my stay at Virginia Tech a whole lot of fun - the TUBAS and those who are tubas at heart: Steve (the Grand Tuba), JJ, Tommy, Jeff, Floozy, Lil’ Al, Belcher, Elroy, Pam Speed, and Stork. I especially want to thank my roommates at the Tuba House who have had to put up with my unusual hours: C.L., Evan,
Joby, and most of all, Dave Whiteside whom I’ve know more than a quarter of my life since I came to Virginia Tech - time for both of us to leave.

I also want to thank my best friend, Dave Paulsen, for helping me get where I am and making life lots of fun.

Thanks also to my 5 cats (don’t ask) who always make life exciting, although not necessarily in a positive way: Malakai, Zoey, Samantha, Flea, and Peanut.

Finally, I wish to express my deepest gratitude to my wife, Rebecca, who has endured living apart from me for five long months while I finished my degree. Throughout the lonely and endless days of coding, writing, and core dumps, your love gave me the inspiration and determination that I needed to pull through. Thank you for all of the weekend visits which picked me up when I needed it most. This thesis is for you and now that it’s done, I’m coming home for good...
Table of Contents

List of Illustrations ......................................................................................... ix

1. Introduction ................................................................................................. 1
   Aircraft Design ............................................................................................ 1
   Aircraft Fuselage Design ......................................................................... 2

2. Literature Review ........................................................................................ 4
   Aircraft Design .......................................................................................... 4
   Parametric Design ......................................................................................... 5
   Aircraft Conceptual Design Code ............................................................. 6
   User Interface ............................................................................................... 7
   Object-Oriented Programming and Design .................................................. 8
   Object-Oriented Design in Graphics and CAD ............................................ 9

3. Thesis Objectives ......................................................................................... 11

4. Object-Oriented Design and Programming .............................................. 12

5. Motif-Like Interface Framework ............................................................... 14
   Motif ........................................................................................................... 14
   Interface Framework .................................................................................... 14

6. Aircraft Fuselage Design ........................................................................... 18
   ACSYNT Fuselage Design ....................................................................... 21

7. Rule-Based Fuselage Method .................................................................... 26
   Rule-Based Fuselage Profile - Lockheed .................................................. 26
   Rule-Based Fuselage Cross-Sections - Lockheed ..................................... 30
   Rule-Based Fuselage - Lockheed Derivation ............................................. 30
   Enhanced Rule-Based Fuselage Method .................................................... 41
   Rule-Based Fuselage Profile Curves .......................................................... 41
   Rule-Based Fuselage Cross-Sections ........................................................... 42

8. Spine and Cross-Section Method ............................................................... 44

9. Object-Oriented Design of the System ..................................................... 46
   Software Design ......................................................................................... 46
   Class Design ............................................................................................... 46
Objects of Derived Classes ................................................. 51

10. Class Descriptions .......................................................... 55
    Base Classes .................................................................. 55
    Derived Classes ................................................................ 67
    Utility Classes .................................................................. 100

11. User Interface ...................................................................... 120
    User Friendliness .......................................................... 120
    Look and Feel .................................................................... 120

12. Implementation and Examples of Results ......................... 122
    PHIGS .......................................................................... 122
    C++ ............................................................................. 122
    Rule-Based Fuselage Components .................................... 122
    Spine and Cross-Section Components ............................... 131
    System Cross-Sections .................................................... 135

13. ACSYNT Integration ............................................................ 137
    Create New Component .................................................... 137
    Modify Component ........................................................ 138
    ACSYNT Geometry File ................................................... 139
    Examples of Results ........................................................ 139
    Limitations ....................................................................... 140

14. Conclusions and Recommendations ................................... 145

15. References .......................................................................... 147

Appendix A. User Guide ......................................................... 152
    Overview ......................................................................... 154
    Main Menu ....................................................................... 155
    EDIT Main Menu Options ................................................ 159
    Create Segments ............................................................. 165
    Create Rule-Based Fuselage Component ......................... 175
    Modify RBF Profile Curve .............................................. 184
    Create Spine and Cross-Section Component .................... 185
    Modify Profile Curve Segments ....................................... 187
    Create Cross-Sections ..................................................... 201
    Modify Component ........................................................ 210
    Viewing Options ............................................................ 211
Appendix B. Detailed Class Descriptions ........................................ 217

Overview ........................................................................ 218
Component Class ......................................................... 219
Component Manager Class ............................................ 225
Cross-Section Class ....................................................... 228
Crown Curve Class ....................................................... 233
Cubic Segment Class ..................................................... 236
File Manager Class ....................................................... 243
Fillet Segment Class ..................................................... 248
Grid Class .................................................................. 250
Input/Display Manager Class .......................................... 252
Keel Curve Class .......................................................... 255
Linear Segment Class ................................................... 258
Menu Manager Class .................................................... 261
Piecewise Cross-Section Class ....................................... 264
Plan Curve Class .......................................................... 268
Point Class ................................................................ 271
Profile Curve Class ....................................................... 274
RBF Profile Curve Manager .......................................... 277
Rule-Based Fuselage Class ............................................ 278
Segment Class .............................................................. 285
Segment Manager Class ................................................ 291
Simple Cross-Section Class ............................................ 297
Spine Curve Class ........................................................ 301
Spine and Cross-Section Class ....................................... 302
View Manager Class ..................................................... 306

Appendix C. Cross-Section Orientation Point .......................... 311

Appendix D. Cubic Segments ............................................. 318
Cubic Spline Parameters ............................................... 319
Evaluating Spline Parameters ......................................... 321
Evaluating the Knot (Breakpoint) Sequence .................... 321
Evaluating the Spline .................................................... 321

Appendix E. cubic_segment_tangents() Source Code Listing ....... 323

Vita .............................................................................. 331
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Simplified Class Organization for Motif-Like Interface Framework</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Longitudinal Control Lines</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>Conic Fuselage Lofting Example</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>Parameters and Characteristic Points of the Nose</td>
<td>23</td>
</tr>
<tr>
<td>5</td>
<td>Parameters and Characteristic Points of the Mid-section</td>
<td>24</td>
</tr>
<tr>
<td>6</td>
<td>Parameters and Characteristic Points of the Afterbody</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>Rule-Based Fuselage Profile Input Parameters</td>
<td>27</td>
</tr>
<tr>
<td>8</td>
<td>Example Rule-Based Fuselage Profile</td>
<td>28</td>
</tr>
<tr>
<td>9</td>
<td>Rule-Based Fuselage Profile Associated Input Parameters</td>
<td>29</td>
</tr>
<tr>
<td>10</td>
<td>Rule-Based Fuselage Cross-Section Input Parameters</td>
<td>31</td>
</tr>
<tr>
<td>11</td>
<td>Rule-Based Fuselage Cross-Section with Input Parameters</td>
<td>32</td>
</tr>
<tr>
<td>12</td>
<td>Rule-Based Fuselage Cross-Section</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>Example Rule-Based Fuselage Cross-Sections</td>
<td>40</td>
</tr>
<tr>
<td>14</td>
<td>Spine and Cross-Section Component</td>
<td>45</td>
</tr>
<tr>
<td>15</td>
<td>Class Hierarchy - Utility Classes</td>
<td>47</td>
</tr>
<tr>
<td>16</td>
<td>Class Hierarchy - Component Class</td>
<td>48</td>
</tr>
<tr>
<td>17</td>
<td>Class Hierarchy - Segment Classes</td>
<td>49</td>
</tr>
<tr>
<td>18</td>
<td>Linked List Example without Inserted Point</td>
<td>52</td>
</tr>
<tr>
<td>19</td>
<td>Linked List Example with Inserted Point</td>
<td>53</td>
</tr>
<tr>
<td>20</td>
<td>Fillet Geometrical Relationships</td>
<td>99</td>
</tr>
<tr>
<td>21</td>
<td>Example Rule-Based Fuselage Component in 4-View Display</td>
<td>123</td>
</tr>
<tr>
<td>22</td>
<td>Example Rule-Based Fuselage Component in Isometric View</td>
<td>124</td>
</tr>
<tr>
<td>23</td>
<td>Example Rule-Based Fuselage Crown Curve</td>
<td>126</td>
</tr>
<tr>
<td>24</td>
<td>Example Rule-Based Fuselage Keel Curve</td>
<td>127</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>25</td>
<td>Example Rule-Based Fuselage Plan Curve</td>
<td>128</td>
</tr>
<tr>
<td>26</td>
<td>Example Rule-Based Fuselage Cross-Sections</td>
<td>129</td>
</tr>
<tr>
<td>27</td>
<td>Example Rule-Based Fuselage Cross-Section</td>
<td>130</td>
</tr>
<tr>
<td>28</td>
<td>Spine and Cross-Section Component in 4-View Display</td>
<td>132</td>
</tr>
<tr>
<td>29</td>
<td>Spine and Cross-Section Component in Isometric View</td>
<td>133</td>
</tr>
<tr>
<td>30</td>
<td>Spine Profile Curve in Side View</td>
<td>134</td>
</tr>
<tr>
<td>31</td>
<td>ACSYNT Geometry Module</td>
<td>141</td>
</tr>
<tr>
<td>32</td>
<td>ACSYNT Rule-Based Fuselage Component</td>
<td>142</td>
</tr>
<tr>
<td>33</td>
<td>ACSYNT Spine and Cross-Section Component</td>
<td>143</td>
</tr>
<tr>
<td>34</td>
<td>Copied ACSYNT Spine and Cross-Section Component</td>
<td>144</td>
</tr>
<tr>
<td>35</td>
<td>Parametric Cubic Spline for Bay n</td>
<td>320</td>
</tr>
</tbody>
</table>
1. Introduction

Aircraft Design

The design of an aircraft begins with a set of requirements such as aircraft range, takeoff and landing distances, maneuverability, and speed. Typically, these requirements are specified by the particular mission that the aircraft must perform. The design process consists of three main phases: (1) conceptual design, (2) preliminary design, and (3) detail design [Raym89].

In the conceptual design phase, the initial layout of the aircraft is determined from weight and size estimations of a rough sketch of the aircraft. An analysis of the aerodynamics, weights, and propulsion of the initial layout are used to optimize the size and performance of the aircraft. This optimization results in a more detailed aircraft layout which undergoes a refined analysis and optimization process to find the lightest or lowest-cost aircraft that will meet the design requirements.

In the preliminary design phase, only minor alterations to the aircraft configuration are allowed. An important part of preliminary design is to describe the outer surface geometry of the aircraft by a process know as “lofting”, the mathematical modeling of the outside surface of the aircraft. The aircraft geometry is defined by a series of conical cross-sections connected together by lofted surface patches. This geometric description of the aircraft surface may then be used to perform finite element and CFD (computational fluid dynamics) analyses.

Finally, in the detail design phase, the detailed structural design and interior layout configuration are completed. An important design consideration in this phase is the design of the parts of the
aircraft which are to be manufactured. The aircraft structure is fabricated and flight simulators are developed to test the design. This phase leads into the actual fabrication of the entire aircraft.

**Aircraft Fuselage Design**

The fuselage is a critical design component of any aircraft. An accurate description of the fuselage geometry is necessary to allow the designer to minimize aircraft surface area and volume, thus minimizing aircraft weight. The surface geometry of the fuselage must be precise in order to perform aerodynamic analysis.

An accurate surface model of the aircraft fuselage can be obtained using geometric modeling. However, it is unrealistic to expect aircraft designers to have enough expertise in geometric modeling to develop their own methods for creating the surface geometry of an aircraft.

For this reason, geometric modeling tools must be developed to allow aircraft designers to create aircraft components which fully describe the surface geometry. These tools must enable the designer to parametrically create aircraft components. This allows the designer to easily modify the geometry by changing only the parameters associated with the component, such as fuselage diameter and length. For greater accuracy, the designer should have complete control over important geometric design features, such as the individual cross-sections which define an aircraft fuselage.

The focus of this thesis is to describe the design and implementation of a computer aided design (CAD) system which allows a designer to interactively create an aircraft fuselage using the Rule-Based Fuselage method proposed by Lockheed. The system also provides methods to create arbitrary geometric shapes for aircraft components such as inlets and ducts using the Spine and
Cross-Section method. Discussed in detail are both the Rule-Based Fuselage and the Spine and Cross-Section methods.

The system was designed using object-oriented design techniques. A discussion of the system design and implementation is included. Also discussed is the integration of these methods with the parametric, computer aided conceptual aircraft design code, ACSYNT, developed at NASA and Virginia Polytechnic Institute and State University (VPI & SU) [Wamp88], [Jaya92a].
2. Literature Review

Aircraft Design

The aircraft design process consists of the conceptual, preliminary, and detail design phases. Since the system described in this thesis is integrated with the conceptual aircraft design code, ACSYNT, the discussion of the aircraft design process will be limited to the conceptual design phase, particularly the sizing of the aircraft fuselage.

The book, Fundamentals of Aircraft Design, provides a description of the iterative processes involved in the conceptual design phases for both civilian and military aircraft [Nico84]. The preliminary fuselage sizing and design are discussed. Included are typical internal components which influence fuselage size and internal configuration as well as a discussion of the fuselage fineness ratio (fuselage length over fuselage diameter) and how it differs for subsonic and supersonic flow.

A book which concentrates on conceptual design, Aircraft Design: A Conceptual Approach, describes the process of lofting to size the aircraft fuselage during the conceptual design phase [Raym89]. Included is a discussion of conical fuselage development, a method in which the fuselage is geometrically constructed from conical cross-sections which are connected together by longitudinal control lines.

The series of books, Airplane Design, parts I - VIII, covers aircraft design from the conceptual design stage to production [Rosk89]. Fuselage sizing is discussed in the preliminary design phase. Included are tables of typical numerical values of fuselage parameters and equations used
for preliminary sizing of the fuselage.

Parametric Design

The surface geometry of a fuselage is typically defined by first creating the fuselage cross-sections from a set of cross-section parameters. The fuselage surface is then lofted over the cross-sections by various geometric modeling and interpolation techniques.

The paper, “Parametric Component Modeling for Computer Aided Design”, presents methods used by the conceptual aircraft design code, ACSYNT, to support modeling of parametric aircraft components [Jaya92b]. These components are described by a set of "characteristic points" which represent the minimum amount of data (points and other parameters) required to completely define the component geometry. The intermediate cross-sections of an aircraft component can be determined by interpolating the characteristic points at the component ends. The surface model of the component is then created by "lofting" surface patches over the component cross-sections.

A method of B-Spline surface interpolation which is used to model the surface of the fuselage of a fighter aircraft is described in the paper, “Surface Modeling of a Fighter Aircraft Fuselage with B-Spline Skeletal Lines”, [Kank91]. The method uses an iterative scheme to obtain the control points for the recursive interpolation algorithm. Cubic splines which pass through equally spaced points along one axis with nearly equal spacing between neighboring lines, called skeletal lines, are used to select the appropriate surface control points for the surface interpolation.

To create an arbitrary geometric shape, one method is to define cross-sections oriented normal to a guiding curve, referred to as the spine curve. Again, the cross-sections must be joined by geometric modeling interpolation methods.
In the paper, "Skinning Techniques for Interactive B-Spline Surface Interpolation", a different technique for the skinning method of interpolating surface sections by using projection curve techniques for interactive B-spline surface interpolation is presented [Wood88]. This technique increases the shape control in the longitudinal surface direction.

Aircraft Conceptual Design Code

The use of CAD in aircraft conceptual design first began with the Computer graphics Augmented Design and Manufacturing system (CADAM) developed by Lockheed in 1965 [Corn88]. Around 1980, the Avions Marcel Dassault-Breguet Aviation company developed the Computer-graphics Aided Three-dimensional Interactive Application system (CATIA) which was used to extend the capabilities of CAD in aircraft conceptual design beyond those of CADAM [Corn88].

In 1982, Rockwell International developed the Configuration Design System (CDS) which allows the designer to create three-dimensional lofted surface aircraft components [Raym82a]. CDS can also be used to arrange internal aircraft components and perform conceptual analysis of weight and drag. Under the supervision of Daniel Raymer, CDS was made a part of the Integrated Design and Analysis System (IDAS) and became known as the Configuration Definition Module (CDM) [Raym82b]. CDM allows up to 45 lines of numerical data for each aircraft component which must be entered line by line.

In 1986, an interface was developed for the Helicopter Sizing and Performance Computer Program (HESCOMP) which created a geometric model from the numerical data generated by the helicopter program developed at Boeing Vertol under NASA and U.S. Navy contracts. This geometric model enabled CAD/CAM tools to assess mission equipment packages (such as avionics) on the
geometry, weight, and performance of a helicopter. The interface for HESCOMP is described in the paper, “Integration of a Helicopter Sizing Code with a Computer-Aided Design System” [Mykl86].

The AirCraft SYNThesis (ACSYNT) code is a feature-based, parametric computer-aided aircraft conceptual design code developed at VPI & SU in conjunction with NASA Ames Research Center since 1987. ACSYNT combines a graphical user interface based on the ISO standard PHIGS and the analysis code, ACSYNT, which was developed by NASA Ames Research Center [Wamp88], [Jaya92a].

The Aircraft Design and Analysis System (ADAS) was developed at Delft University, Netherlands in 1988 and is primarily intended for aircraft configuration development [Corn88].

Jan Roskam developed the Advanced Aircraft Analysis (AAA) conceptual aircraft design code for use on Apollo workstations [Kohl90]. AAA is based on his series of aircraft design books and is used primarily for educational purposes [Rosk89].

The Aircraft Computer Aided Design System (ACADS), developed by General Dynamics, is a fully three-dimensional drafting system for aircraft design [ACAD89]. The designer creates aircraft geometry by creating the profile of the geometry through which conic cross-sections are lofted. ACADS geometry may be used for finite element as well as radar cross-section analysis.

**User Interface**

In order to overcome the difficulties involved with the creation of input commands and data for spatial-mechanism design programs, a graphics preprocessor was developed to interactively assist
with such problems. The preprocessor was created using the three-dimensional graphics standard, PHIGS (Programmer’s Hierarchical Interactive Graphics System), in order to achieve graphics device independence. The preprocessor is described in the paper, “A PHIGS-Based Graphics Input Interface for Spatial-Mechanism Design” [That88].

The principles of user interface design and implementation such as user input and menus are introduced in the book, Programming the User Interface [Brow89]. Techniques used to maintain consistency throughout the user interface are described. The importance of color, providing the user with information during program execution, and reducing and controlling program errors is stressed. The book also provides examples of user interface tools written in the C programming language.

Object-Oriented Programming and Design

The book, The Tao of Objects: A Beginner’s Guide to Object-Oriented Programming, introduces the object-oriented programming (OOP) concepts of encapsulation, inheritance, and polymorphism which are used to create software that is both simple in design and easy to maintain. Also discussed is the use of OOP for creating a graphical user interface (GUI) to hide the complexity of the GUI from the user and reduce the amount of programming required to process user input.

These OOP concepts are described in greater detail in the book, Object-Oriented Software Engineering with C++ [Darr91]. The software engineering process is also described. The C++ programming language is used exclusively for illustrating the OOP principles because the author believes that C++ will be the dominant object-oriented language used for industrial applications.

The basic concepts of object-oriented design which can be used to reduce the complexity of
software, are introduced in the book, *Object-Oriented Design with Applications* [Booc91]. Particular attention is given to the identification, design, and implementation of object classes.

**Object-Oriented Design in Graphics and CAD**

The book, *Computer Graphics using Object-Oriented Programming*, describes the advantages of using object-oriented programming languages to develop and maintain computer graphics software and how to implement object-oriented programming with computer graphics [Cunn91]. Several examples of the uses of object-oriented programming in solving practical computer graphics problems are provided.

The paper, “A Constraint-Driven Solid Modeling Open Environment”, describes the design and implementation of an object-oriented system which provides classes to represent geometry either by form features and constraints or boundary representation solid model [LinW93].

An interface framework which emulates the windows and menus of such applications as Motif, OS/2, and Macintosh is described in the paper, “A Motif-Like Object-Oriented Interface Framework Using PHIGS” [Woya93]. The framework is object-oriented, written in the C++ programming language, and uses the three-dimensional graphics language, PHIGS. The use of this framework to create the GUI for the system described in this thesis is discussed later.

Another object-oriented system is presented in the paper, “An Object-Oriented Class Library for Creating Engineering Graphs Using PHIGS” [Uhor93]. This paper discusses the design and implementation of a library of object-oriented classes which can be used to create engineering
graphs for GUI applications using PHIGS. This library of classes has been implemented in a GUI which was developed using the interface framework described above.

Two more papers on object-oriented design and computer graphics are "Higher Level Notions of PHIGS Objects: Constructors and Agents for Supporting Highly Interactive Graphics" [Hart93] and "An Object-Oriented System Using PHIGS" [Neal93]. Both of these papers describe the benefits of using object-oriented systems in the PHIGS environment to create better GUIs.
3. Thesis Objectives

In 1988, Lockheed Corporation described a new approach for creating an aircraft fuselage, the Rule-Based Fuselage method, in their systems specification report [Sher88]. The Rule-Based Fuselage method was intended to be part of an in-house aircraft conceptual design graphical user interface being developed at Lockheed. However, as mentioned in the introduction, aircraft designers typically lack the geometric modeling knowledge necessary to create a graphical user interface of this nature. As a result, the Rule-Based Fuselage method was presented to the ACSYNT Institute at VPI & SU to be integrated with the existing conceptual aircraft design code, ACSYNT.

The objectives of this thesis were to:

- Create a method to support the specification of an aircraft fuselage using the Rule-Based Fuselage procedures outlined by Lockheed.

- Improve the Lockheed Rule-Based Fuselage method by simplifying the method and allowing greater flexibility for creating fuselage cross-sections.

- From the methods developed for the Rule-Based Fuselage system, develop methods to create an arbitrary geometric shape, referred to as the Spine and Cross-Section method.

- Enhance the current aircraft fuselage design in ACSYNT by integrating the developed system with ACSYNT.
4. Object-Oriented Design and Programming

Software should be designed to ensure a high degree of reusability, modular design, and ease of maintenance. Software designed in this manner allows other software engineers to easily modify the code. This is particularly important for maintaining large software applications such as ACSYNT, to which new modules are continuously being developed at VPI & SU and NASA. At the same time, previously developed modules of ACSYNT must also be enhanced by persons other than the original module designer.

Although it is possible to implement modular design in any programming language, currently the most structured program design can be achieved using an object-oriented design approach. Since the system described in this thesis is designed using an object-oriented approach, it is important to briefly discuss some of the concepts of object-oriented design (OOD) and object-oriented programming (OOP).

OOP differs from traditional programming in the sense that traditional programming is a collection of routines organized by the function they perform. The programmer thinks in terms of actions, whereas in OOP, the program designer must think in terms of abstract objects. Within each object are the data and functions used to create or modify itself.

The building blocks of OOP are classes and objects. A class is a collection of objects with similar attributes, and an object is a specific instance of its class [Schi92]. The data and functions of a class are bound together by encapsulation. Encapsulation protects the class data and ensures that no misuse or mishandling of the data can occur.
Classes may be designed hierarchically using inheritance, the process by which an object can inherit a general set of properties to which it can add features specific only to itself [Schi92]. The general class is referred to as the base class of the more specific derived class.

As an example, consider an airplane class which is the base class of the derived fighter and commercial airplane classes. The fighter and commercial airplane share many attributes with each other (contained in the airplane class), but each has its own set of attributes unique to that type of aircraft (contained in the fighter and commercial airplane classes, respectively). The F-16 would be an object of the fighter class and the Boeing 747 an object of the commercial airplane class.

The attributes refer to the data and functions encapsulated within the base and derived classes. For example, data such as the fuselage length and diameter would be contained in the base airplane class because each aircraft type has these parameters. However, the methods used to perform weight analysis would be contained in the derived classes. An important factor for determining the weight of an F-16 fighter is the number of weapons whereas the commercial aircraft weight is dependent on the number of people to be carried.

The approach to OOD is to first identify the classes which must be developed. This is done by determining the most general objects which must be created. These objects will be used to define the base classes. Next, the derived classes are developed to create specific objects which use the attributes of the base classes. Finally, utility classes must be developed which are used to organize the base and derived classes.

In developing the base and derived classes, it is important to identify which methods are shared by the different types of derived classes and which methods are specific to each derived class. In this manner, new derived classes can be quickly developed simply by adding only those features not shared with the other derived classes.
5. Motif-Like Interface Framework

Motif

Many existing CAD systems use Motif to design their user interface. Motif provides functions for creating, displaying, and processing 2D windows and menus of a software application. The windows may be dynamically sized during program execution. The menus are also moveable (i.e. the window can be moved by clicking on it with a mouse and dragging it to another location on the graphics display device) and contain Widgets (push buttons, radio buttons, etc.). By using Motif, the software designer can easily develop a program application with a user interface that resembles other software, thus ensuring that the user will be familiar with the interface menus and input processing methods.

Although Motif is widely used for CAD user interfaces, there are several limitations among which are [Woya93]:

- Motif can only support 2D graphics
- Motif is not truly object-oriented, making it difficult to integrate with object-oriented software applications
- Motif cannot be easily expanded by the software designer

Interface Framework

To resolve these limitations, the windows and menus developed for the system described in this thesis use a PHIGS-based “Motif-like” object-oriented interface framework developed by Scott
Woyak at VPI & SU [Woya92], [Woya93]. By using the ISO 3D graphics standard, PHIGS (Programmers Hierarchical Interactive Graphics System), the interface framework allows the software designer to create a user interface for a 3D graphics environment and supports graphics device independence.

Since the interface framework is object-oriented, it is compatible with object-oriented software applications and may easily be expanded by the software designer by adding new classes to the existing class organization.

Figure 1 on page 16 shows a simplified version of the class organization of the interface framework [Woya93]. The Window class is used for displaying menus and user-defined geometry. Windows can be displayed overlapping one another and the raising and lowering of the windows may be controlled by the user. There are several types of windows that can be created from the Window class derived classes: the Simple View class, the Geometry Manager class, and the Pop-Up Menu class.

The Simple View class is a 3D PHIGS view which contains functions to control the view size and modify the view orientation. The Geometry Manager class is a more complicated version of the Simple View class and has many of the same capabilities as the windows used by X Windows, OSF/Motif, Microsoft Windows, and OS/2. The Geometry Manager window has a rectangular frame to which scroll bars and push buttons may be added for zooming and panning within the window. A Geometry Manager is also moveable.

The Pop-Up Menu class is used to create “Motif-like” pop-up menus. The Menu Item class derived classes (Push Button, Slider, and Radio Button classes) are used to create menu items such as push buttons, check boxes, and text input fields. These menu items are similar to the Widgets used by Motif. The pop-up menus may also be dynamically moved to another location.
Figure 1. Simplified Class Organization for Motif-Like Interface Framework [Woya93].
during program execution. The Menu Manager controls the operation of a set menu items.

The Static Menu class (not shown) is similar to the Pop-Up Menu class except that a static menu is immovable and is displayed for the duration of the application program.

The Interface Manager class connects the application program, the application windows, and the user and is responsible for processing user input. The Interface Manager class also controls the visibility of a window by raising and lowering the windows.

For more details concerning the interface framework classes and their corresponding data and functions, the reader should refer to [Woya92], [Woya93].
6. Aircraft Fuselage Design

In the conceptual design phase, the overall aircraft configuration and size must be determined. One of the most important aircraft components to be sized is the fuselage. Initially, the fuselage is sized based on an estimate of gross takeoff weight. Later in the design process, the fuselage length and diameter are established by packaging and internal components.

Once the fuselage has been sized, a more detailed description of the fuselage geometry may be generated for aircraft weight estimations and aerodynamic analysis.

One technique used to create the fuselage geometry is "lofting", the mathematical modeling of the outside surface of the aircraft [Raym89]. This technique uses conic cross-sections connected by smooth longitudinal lines to define the fuselage geometry. Figure 2 on page 19 shows the upper half of an aircraft fuselage in which the points A, B, C, and S of each of the three cross-sections are connected by smooth longitudinal lines.

Figure 3 on page 20 shows an example of conic fuselage lofting in which the designer creates "control cross-sections" or "control stations" along the fuselage to develop the "longitudinal control lines" which define the fuselage shape. Initially, the designer defines the control stations (0, 120, 240, 370, and 500). Station 0 is the nose and is always a single point. The other control stations represent cross-sections along the fuselage located at critical design points. For example, station 240 has a flat side to provide for a side-mounted inlet typical of such aircraft as the F-4 and MiG-23.

The longitudinal control lines are then created from the control stations. Once the fuselage shape
Figure 2. Longitudinal Control Lines [Raym89].
Figure 3. Conic Fuselage Lofting Example [Raym89].
has been defined by the control stations and longitudinal control lines, the designer may create intermediate conic cross-sections between the control stations. An intermediate cross-section is created by measuring the positions of points A, B, C, and S (see Figure 2 on page 19) from the longitudinal control lines to the new cross-section location. Control station 290 of Figure 3 on page 20 is an example of a cross-section created in this manner.

This process must be repeated for each new intermediate cross-section along the aircraft fuselage. However, CAD systems have been developed for aircraft design which can automate the process and provide immediate visual feedback. For example, the conceptual aircraft design code, ACSYNT, allows designers to create an aircraft fuselage consisting of conic and/or elliptic cross-sections simply by entering a few parameters associated with the fuselage.

**ACSYNT Fuselage Design**

The geometric model of an aircraft fuselage created within ACSYNT consists of three components: the nose, mid-section, and afterbody. These three components are each described by their own set of characteristic points which represent the minimum number of data points required to fully define the component geometry. The characteristic points are interpolated to determine the intermediate cross-sections of the nose, mid-section, and afterbody components. Lofted surface patches are then used to connect the cross-sections together to create a smooth fuselage surface from the three components.

Associated with each component are the parameters which are used to calculate the characteristic points [Jaya91]. An aircraft designer using ACSYNT is only concerned with the aircraft parameters since the characteristic points and their calculation are invisible to the user.
Figures 4 through 6 on pages 23 through 25 show the associated parameters and characteristic points of the nose, mid-section, and afterbody components respectively. By modifying the parameters of the three components, the designer can change the shape and size of the fuselage.

From within the geometry module of ACSYNT, a user may create an aircraft fuselage by adding the nose, mid-section, and afterbody components to the current geometry. The added components are displayed in their default locations, connected together to form the aircraft fuselage.

ACSYNT allows the user to modify the fuselage components by selecting the component to be modified from the current geometry. The user can modify the component location and/or parameters (see Figures 4-6 for the fuselage parameters). Once the location or a parameter of a component is modified, the change is immediately reflected in the displayed geometry.

ACSYNT also provides parent/child relationships which may be used to group together the components of the fuselage. This allows the user to modify all three components simultaneously.

Although the current fuselage design in ACSYNT allows the user to define a variety of cross-sections, some complex aircraft fuselages cannot be defined by elliptic or conic cross-sections. Furthermore, the cross-sections must be the same shape over the entire component. It is possible to define a fuselage with more than one cross-section shape by using different cross-section shapes for the mid-section and afterbody components. However, the fuselage is still limited to only two different cross-section shapes.
Figure 4. Parameters and Characteristic Points of the Nose [Jaya91].
Figure 5. Parameters and Characteristic Points of the Mid-section [Jaya91].
Figure 6. Parameters and Characteristic Points of the Afterbody [Jaya91].
7. Rule-Based Fuselage Method

The Rule-Based Fuselage method enables designers to create aircraft fuselage using complex cross-sections and allows an aircraft fuselage to be defined by several different cross-section shapes. The Rule-Based Fuselage method was developed by Lockheed as an aircraft design tool for a forthcoming aircraft conceptual design user interface [Sher88]. The details and derivation of the Lockheed Rule-Based Fuselage method are included in this section.

Rule-Based Fuselage Profile - Lockheed

In order to create an aircraft fuselage using the Rule-Based Fuselage method, the designer must first define the fuselage profile in the top (plan) and side views. Figure 7 on page 27 shows the input parameters that the user must enter which are used to create the fuselage profile. The designer creates the fuselage profile in the plan and side views based on the input parameters (a complete description of the relation between the fuselage profile and the input parameters is given in Chapter 10 - see Crown, Keel, and Plan classes) by drawing the fuselage profile points and using a curve-fitting scheme to draw the aircraft fuselage profile through the points. Figure 8 on page 28 shows an example of a fuselage profile and Figure 9 on page 29 shows the associated input parameters of the fuselage profile.
# INPUT MENU

<table>
<thead>
<tr>
<th>FUSELAGE (RULED-BASED)</th>
<th>INPUT</th>
<th>UNIT</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOSE POSITION</td>
<td>IN.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUSELAGE LENGTH</td>
<td>IN.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PLAN VIEW**

| NOSE RADIUS          | IN.  |       |
| NOSE ANGLE           | DEG. |       |
| AFT RADIUS           | IN.  |       |

**SIDE VIEW**

| NOSE CROWN ANGLE     | DEG. |       |
| NOSE KEEL ANGLE      | DEG. |       |
| AFT OFF-SET          | IN.  |       |
| AFT BODY UPSWEEP ANGLE | DEG. |       |

---

**Figure 7.** Rule-Based Fuselage Profile Input Parameters courtesy Lockheed.
Figure 8. Example Rule-Based Fuselage Profile courtesy Lockheed.
Figure 9. Rule-Based Fuselage Profile Associated Input Parameters courtesy Lockheed.
**Rule-Based Fuselage Cross-Sections - Lockheed**

Once the fuselage profile has been defined, the designer must create each individual cross-section along the fuselage. Figure 10 on page 31 shows the input the parameters for a cross-section. These parameters have to be entered for each individual cross-section along the aircraft fuselage. Figure 11 on page 32 shows an example cross-section with its associated parameters. Only half of the cross-section is shown because it is symmetrical about the vertical axis. This is because the aircraft fuselage is symmetrical in the plan view.

The crown and keel points are the points at which the cross-section intersects the fuselage profile in the side view and the plan point (half breadth) is the point at which the cross-section intersects the fuselage profile in the plan view. These three points define the cross-section boundary and are used to calculate the unknown points and lengths of the fuselage cross-section.

**Rule-Based Fuselage - Lockheed Derivation**

Figure 12 on page 33 shows a fuselage cross-section with the points, lengths, and angles labeled. Note that point $P_1$ is the crown point, $P_3$ is the plan point, and $P_5$ is the keel point. From the input parameters and the crown, plan, and keel points, the unknown cross-section lengths and points may be calculated using the geometry of the cross-section.

The derivation of the Rule-Based Fuselage method as outlined by Lockheed is as follows:

**Given:**

$P_1, P_2, P_3$

$A_1, A_2, A_3, A_4, A_5$

$L_3$
**INPUT MENU**

**CROSS-SECTIONS**

SECTIO N NO. X OF YY
AT F.S. XXXX IN. (XXX.X %)

<table>
<thead>
<tr>
<th>SIDE</th>
<th>ANGLE</th>
<th>RADIUS</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>$A_1 =$</td>
<td>$R_1 =$</td>
<td>$L_3 =$</td>
</tr>
<tr>
<td>$S_2$</td>
<td>$A_2 =$</td>
<td>$R_2 =$</td>
<td></td>
</tr>
<tr>
<td>$S_3$</td>
<td>$A_3 =$</td>
<td>$R_3 =$</td>
<td></td>
</tr>
<tr>
<td>$S_4$</td>
<td>$A_4 =$</td>
<td></td>
<td>$R_4 =$</td>
</tr>
<tr>
<td>$S_5$</td>
<td>$A_5 =$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10. Rule-Based Fuselage Cross-Section Input Parameters
courtesy Lockheed.

7. Lockheed Ruled-Based Fuselage Method
Figure 11. Rule-Based Fuselage Cross-Section with Input Parameters
courtesy Lockheed.
Figure 12. Rule-Based Fuselage Cross-Section courtesy Lockheed.
Required: \( P_2, P_4 \)
\( L_1, L_2, L_4, L_5 \)

Solution: Solve for \( L_1 \) and \( L_2 \) first,

\[
L_2\sin(A_2) - L_4\sin(A_1) + \frac{L_3}{2} = Y_{13} \quad (1)
\]
\[
L_1\cos(A_1) + L_2\cos(A_2) = X_{13} \quad (2)
\]

where
\[
X_{13} = P_{3x} - P_{1x}
\]
\[
Y_{13} = P_{1y} - P_{3y}
\]

From (2),

\[
L_1 = \frac{X_{13} - L_2\cos(A_2)}{\cos(A_1)} \quad (3)
\]

Substituting (3) into (1),

\[
L_2\sin(A_2) - \frac{X_{13} - L_2\cos(A_2)}{\cos(A_1)}\sin(A_1) + \frac{L_3}{2} = Y_{13}
\]

\[
L_2\sin(A_2) + L_2\cos(A_2)\tan(A_1) - X_{13}\tan(A_1) = Y_{13} - \frac{L_3}{2}
\]

\[
L_2(\sin(A_2) + \cos(A_2)\tan(A_1)) = Y_{13} - \frac{L_3}{2} + X_{13}\tan(A_1)
\]
Therefore,

\[ L_2 = \frac{Y_{13} + X_{13} \tan(A_1) - L_3 / 2}{\sin(A_2) + \cos(A_2) \tan(A_1)} \] (4)

Solving for \( P_2 \),

\[ P_{2x} = L_4 \cos(A_1) + P_{1x} \] (5)
\[ P_{2y} = L_4 \cos(A_1) + P_{1y} \] (6)

Solve for \( L_4 \) and \( L_5 \),

\[ L_4 \sin(A_4) - L_2 \sin(A_4) + L_3 / 2 = -Y_{53} \] (7)
\[ L_4 \cos(A_4) + L_5 \cos(A_5) = X_{53} \] (8)

where

\[ X_{53} = P_{3x} - P_{5x} \]
\[ Y_{53} = P_{5y} - P_{3y} \]

From (8),

\[ L_5 \frac{X_{53} - L_4 \cos(A_4)}{\cos(A_5)} \] (9)

7. Lockheed Ruled-Based Fuselage Method
Substituting (9) into (7),

\[ L_4 \sin(A_4) - \frac{X_{53} - L_4 \cos(A_4)}{\cos(A_5)} \sin(A_5) + L_3 / 2 = -Y_{53} \]

\[ L_4 \sin(A_4) + L_4 \cos(A_4) \tan(A_2) - X_{53} \tan(A_2) = -Y_{53} - L_3 / 2 \]

\[ L_4 (\sin(A_4) + \cos(A_4) \tan(A_5)) = -Y_{53} - L_3 / 2 + X_{53} \tan(A_5) \]

Therefore,

\[ L_4 = \frac{X_{53} \tan(A_4) - Y_{53} - L_3 / 2}{\sin(A_4) + \cos(A_4) \tan(A_5)} \]  \hspace{1cm} (10)

Solving for P4,

\[ P_{4x} = L_5 \cos(A_5) + P_{5x} \]  \hspace{1cm} (11)

\[ P_{4y} = L_5 \cos(A_5) + P_{5y} \]  \hspace{1cm} (12)

Results:

\[ P_{2x} = L_1 \cos(A_1) + P_{1x} \]

\[ P_{2y} = L_1 \cos(A_1) + P_{1y} \]

\[ P_{4x} = L_5 \cos(A_5) + P_{5x} \]

\[ P_{4y} = L_5 \cos(A_5) + P_{5y} \]

\[ L_1 = \frac{X_{13} - L_2 \cos(A_2)}{\cos(A_1)} \]
\[ L_2 = \frac{Y_{13} + X_{13}\tan(A_1) - L_3 / 2}{\sin(A_2) + \cos(A_2)\tan(A_1)} \]

\[ L_4 = \frac{X_{53}\tan(A_4) - Y_{53} - L_2 / 2}{\sin(A_4) + \cos(A_4)\tan(A_5)} \]

\[ L_5 = \frac{X_{53} - L_4\cos(A_4)}{\cos(A_5)} \]

where

\[ X_{13} = P_{3x} - P_{1x} \]
\[ Y_{13} = P_{1y} - P_{3y} \]
\[ X_{53} = P_{3x} - P_{5x} \]
\[ Y_{53} = P_{5y} - P_{3y} \]

Note: these equations exhibit a singularity at \( A_1 = A_2 \).

This derivation is only valid for a cross-section with symmetry about the vertical axis consisting of five linear segments. In addition to these limitations, the Lockheed derivation is based on the assumption that the angle, \( A_3 \), is 90 degrees.

In order for these equations to generate a cross-section shape similar to those given in Figures 11 and 12 on pages 32 and 33, the length, \( L_3 \), must be specified within a certain range of values determined by the angles \( A_1, A_2, A_4, \) and \( A_5 \). The equations which constrain \( L_3 \) are as follows:

\[ L_2 \text{ must be } > 0: \]
\[ P_{1y} + P_{3x}\tan(A_1) > L_3 / 2 \]
\[ L_3 < 2(P_{1y} + P_{3x}\tan(A_1)) \]
$L_1$ must be $> 0$:

$$L_2 < \frac{P_{3x}}{\cos(A_2)}$$

$$\frac{P_{1y} + P_{3x} \tan(A_1) - L_3 / 2}{\sin(A_2) + \cos(A_2) \tan(A_1)} < \frac{P_{3x}}{\cos(A_2)}$$

$$L_3 / 2 - P_{1y} - P_{3x} \tan(A_1) > -\frac{P_{3x} \left( \sin(A_2) + \cos(A_2) \tan(A_1) \right)}{\cos(A_2)}$$

$$L_3 / 2 - P_{1y} - P_{3x} \tan(A_1) > -P_{3x} \left( \tan(A_2) + \tan(A_1) \right)$$

$$L_3 / 2 > P_{1y} + P_{3x} \tan(A_1) - P_{3x} \tan(A_2) - P_{3x} \tan(A_1)$$

$$L_3 > 2(P_{1y} - P_{3x} \tan(A_2))$$

therefore,

$$2(P_{1y} - P_{3x} \tan(A_2)) < L_3 < 2(P_{1y} + P_{3x} \tan(A_1))$$

$L_4$ must be $> 0$:

$$-P_{5y} + P_{3x} \tan(A_2) > L_3 / 2$$

$$L_3 < 2(-P_{5y} + P_{3x} \tan(A_2))$$

$L_5$ must be $> 0$:

$$L_4 < \frac{P_{3x}}{\cos(A_4)}$$

$$\frac{-P_{5y} + P_{3x} \tan(A_2) - L_3 / 2}{\sin(A_4) + \cos(A_4) \tan(A_5)} < \frac{P_{3x}}{\cos(A_4)}$$

$$L_3 / 2 + P_{5y} - P_{3x} \tan(A_5) > -\frac{P_{3x} \left( \sin(A_4) + \cos(A_4) \tan(A_5) \right)}{\cos(A_4)}$$

$$L_3 / 2 + P_{5y} - P_{3x} \tan(A_5) > -P_{3x} \left( \tan(A_4) + \tan(A_5) \right)$$

$$L_3 / 2 > -P_{5y} + P_{3x} \tan(A_5) - P_{3x} \tan(A_4) - P_{3x} \tan(A_5)$$

$$L_3 > 2(-P_{5y} - P_{3x} \tan(A_4))$$
therefore,
\[ 2(-P_{5y} - P_{3x}(\tan(A_4))) < L_3 < 2(-P_{5y} + P_{3x}(\tan(A_5))) \]

Figure 13 on page 40 shows some sample cross-sections generated using the Rule-Based Fuselage method and these equations. Although a wide variety of fuselage cross-sections may be created using this method, the process is very tedious since the user must specify all of the cross-section parameters for each cross-section. Furthermore, this method does not allow the user to take advantage of any existing symmetry about the horizontal axis.
Figure 13. Example Rule-Based Fuselage Cross-Sections courtesy Lockheed.
Enhanced Rule-Based Fuselage Method

The Rule-Based Fuselage method used to develop the system for the ACSYNT aircraft fuselage design is actually an enhanced version of the Rule-Based Fuselage method developed by Lockheed [Sher88]. The requirement of entering the parameters for each individual cross-section has been eliminated. The cross-sections may be defined using any number of segments, linear or cubic, and the user is given the option to take advantage of any existing cross-section symmetry. Following is a discussion of the enhanced Rule-Based Fuselage method. A more detailed discussion of how to create an aircraft fuselage using this enhanced Rule-Based Fuselage method may be found in the User Guide (see Appendix A).

Rule-Based Fuselage Profile Curves

The fuselage profile is comprised of three profile curves: the crown, keel, and plan profile curves. The crown and keel curves are defined in the side view and the plan curve is defined in the top (plan) view. Each profile curve is associated with the parameters shown in Figure 7 on page 27.

The user has the option of defining the profile curves with or without the profile curve parameters. If the profile curves are created without the profile curve parameters, the parameters will automatically be calculated. If the profile curves are created with the parameters, many of the points along the profile curves are automatically generated based on the profile curve parameters and the type of profile curve segments (Chapter 10 - see Crown, Keel, and Plan classes).

The Rule-Based Fuselage profile curves may be created using linear segments, cubic segments, or
a combination of both linear and cubic segments. The last segment of a Rule-Based Fuselage profile curve is always linear by default if the profile curve parameters are used to create the profile curve and the aft offset is non-zero (crown and keel curves) or the aft radius is non-zero (plan curve). In this case, the last segment of the profile curve is automatically created.

Once a profile curve has been created, the user may modify the profile curve either by modifying the actual segments which define the profile curve or by changing the values of the profile curve parameters.

**Rule-Based Fuselage Cross-Sections**

Once the Rule-Based Fuselage profile curves have been defined, the user may create the fuselage cross-sections by either defining each individual cross-section or by defining a single cross-section which can be scaled to create a specified number of other cross-sections. The user may use a combination of both of these methods to allow greater flexibility in cross-section design.

To facilitate the creation of Rule-Based Fuselage cross-sections, the three points at which the cross-section intersects the crown, keel, and plan profile curves are presented to the user. These points are calculated from the cross-section depth (see Appendix C) and represent the cross-section limits (see Figure 11 on page 32). If the user does not exactly pick/enter these points, the cross-section will automatically be updated so that it intersects the crown, keel, and plan profile curves.

Once a cross-section is defined, the user may modify the cross-section. However, the modification will affect only that cross-section, not any cross-sections that may have been scaled from that cross-section. Furthermore, the Rule-Based Fuselage profile curves will not be changed. Instead, the cross-section will be updated so that it intersects the profile curves. However, if any
of the profile curves are modified, the cross-sections will automatically adjust so that the intersection is maintained.
8. Spine and Cross-Section Method

The development of the methods for creating an aircraft using the system Rule-Based Fuselage method led to the development of methods to create an arbitrary shape, the Spine and Cross-Section method. This arbitrary shape is created from only one profile curve, referred to as the spine curve, with cross-sections that are oriented normal to the spine profile curve.

As with the Rule-Based Fuselage method, to create a component, the profile curve (spine curve) must first be defined using linear and/or cubic segments. Other than the fact that there are no parameters associated with the spine curve, the methods to create and modify the spine curve are exactly the same as those created for the Rule-Based Fuselage profile curves.

Once the spine curve has been defined, the cross-sections may be created using methods similar to those developed for the Rule-Based Fuselage method. However, rather than intersecting the profile curves as in the Rule-Based Fuselage method, the cross-sections are centered about the spine curve and oriented such that they are normal to the profile curve at the cross-section depth. Figure 14 on page 45 shows an example of a Spine and Cross-Section component.

A more detailed description of how to create and modify an arbitrary component using the Spine and Cross-Section method is given in the User Guide (Appendix A).
Figure 14. Spine and Cross-Section Component.
9. Object-Oriented Design of the System

**Software Design**

The software design of the system incorporating the Rule-Based Fuselage and Spine and Cross-Section methods is based on an object-oriented design (OOD) approach. As previously mentioned, OOD ensures modularity, reuse, and maintainability of the software which is necessary for integration with a much larger software application, ACSYNT. Such integration is only possible with software that is modular and easily manageable.

**Class Design**

As discussed earlier, the building blocks of object-oriented software are the classes. Figures 15 through 17 on pages 47 through 49 show the class hierarchy of the object-oriented design of the methods described in this thesis. The shaded classes are part of the interface framework developed by Woyak and Myklebust [Woya92], [Woya93]. The symbols connecting the classes together represent the following:

\[
\text{A} \rightarrow \text{B}
\]

Class A inherits class B (i.e. class B is the base class of class A)

\[
\text{A} \rightarrow \text{B}
\]

Class B creates class A
Figure 15. Class Hierarchy - Utility Classes.
Figure 16. Class Hierarchy - Component Class.
Figure 17. Class Hierarchy - Segment Classes.
The design process for the system begins with the most general class, the component. By identifying the objects that are required to define a component (Rule-Based Fuselage or Spine and Cross-Section), the profile curve and cross-section classes are developed. Profile curves and cross-sections can be further broken down into segments which can, in turn, be reduced to a series of points. From this hierarchical decomposition, the segment and point classes are developed.

Once the general base classes are identified and developed, the more specific derived classes can be designed based on the types of objects that must be created within the system. Since the system must enable a designer to create components using the Rule-Based Fuselage and Spine and Cross-Section methods, two different derived classes of the component base class are developed to create the two types of components.

Since there are three different profile curves for a Rule-Based Fuselage component, three separate derived classes of the base profile curve class are developed for the crown, keel, and plan profile curves. The spine profile curve derived class is also developed for the Spine and Cross-Section component.

Components may be defined using either simple or piecewise cross-sections which are derived classes of the cross-section class. A simple cross-section is a simple geometric shape (rectangle, circle, or ellipse) which may be created by entering the parameters associated with that shape. The following parameters are associated with each simple cross-section type:

- rectangle: width, height
- circle: radius
- ellipse: horizontal radius, vertical radius

A piecewise cross-section consists of linear and/or cubic segments and uses many of the same
creation and modification methods developed for the profile curve class.

The preceding base and derived classes are developed to create an individual Rule-Based Fuselage or Spine and Cross-Section component. The component manager utility class is developed to create several components of either type.

Other utility classes such as the menu manager, view manager, and input/display manager are developed to provide the system with access to the functions provided by the interface framework developed by Woyak [Woya92], [Woya93]. These functions are used to display menus and other geometry and to process user input.

**Objects of Derived Classes**

Objects of a derived class are maintained by linked lists. A linked list is a means by which objects are connected such that each object can quickly access the previous and/or next object within the list. Figure 18 on page 52 shows an example of a linked list of cubic segment points. By changing the previous and next values of the second point, a fourth point may be inserted into the segment linked list as shown in Figure 19 on page 53.

Access to the beginning of the linked list of Figure 18 is provided by a pointer to Point 1. Access to the previous point of a point in the linked list is provided by the variable previous_point. The variable, next_point, provides access to the next point of a point in the linked list. A next_point value of NULL indicates that the point is the last point in the linked list.

Traversal of a linked list provides an easy and efficient means of modifying an object. For
Figure 18. Linked List Example without Inserted Point.
Figure 19. Linked List Example with Inserted Point.
example, if all of the points of the cubic segment are to be changed by a specific amount, the linked list of cubic segment points may be traversed from the beginning of the list, adding the amount to each point in the list. The pseudo-code for this type of modification is as follows:

```
point = first point of segment

process loop until last point is reached (next_point = NULL)

    point x-value = point x-value + amount
    point y-value = point y-value + amount
    point z-value = point z-value + amount

    point = next_point (this accesses the next point in the linked list)

end processing loop
```

It is possible to maintain a linked list of objects of different derived classes provided that the objects are created using the base class type. For example, the segments of a spine profile curve may consist of both linear and cubic segments. By creating objects of the linear and cubic segment classes declared with the segment class type, a linked list of both linear and cubic segments can be created.
10. Class Descriptions

This chapter presents a detailed description of each of the system classes shown in Figures 15 through 17 on pages 47 through 49. For more detailed information on the variables and functions contained within each class, the reader should refer to Appendix B.

Base Classes

Component

The component class is the base class of the Rule-Based Fuselage and Spine and Cross-Section derived classes and contains functions to display both types of components.

The component class also maintains a linked list of the cross-sections contained within a component object. Methods for creating, modifying, and displaying all of the cross-sections of a component are provided by the component class. Functions used to set the symmetry and spacing method for all component cross-sections are also included in the component class.

Display Component Functions:

display_component()
remove_component()
autocenter_component()
The display_component() function displays the component PHIGS structure autocentered in the current active view(s). The PHIGS structure may be removed from the current active view(s) using the remove_component() function. The autocenter_component() function determines the location of the minimum and maximum extremes of the component in order to autocenter the component PHIGS structure in all active views.

The component class maintains the linked list of cross-sections for the Rule-Based Fuselage and Spine and Cross-Section classes by using the following data and functions:

**Private Variables:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>number_of_cross_sections</td>
</tr>
<tr>
<td>Cross_Section*</td>
<td>first_cross_section</td>
</tr>
<tr>
<td>Cross_Section*</td>
<td>last_cross_section</td>
</tr>
<tr>
<td>int</td>
<td>cross_section_symmetry</td>
</tr>
<tr>
<td>int</td>
<td>cross_section_spacing</td>
</tr>
</tbody>
</table>

**Create Cross-Sections Functions:**

- void change_symmetry()
- void change_spacing()
- void link_cross_sections()
- void auto_cross_sections()
- void copy_cross_sections()

The component class keeps track of the total number of cross-sections contained within a component. Pointers to the first and last cross-sections of the component provide access to the cross-sections linked list. The overall symmetry and spacing method for each component cross-section is stored within the component class. These variables can be changed using the change_symmetry() and change_spacing() functions. The link_cross_sections() function is used to link together two cross-sections in the linked list.
The auto_cross_sections() function allows a user to create a specified number of cross-sections by scaling a previously defined cross-section, referred to as the scaling cross-section. When a component is duplicated to create a new component, the copy_cross_sections() function is used to duplicate all of the component cross-sections.

Modify Component Cross-Sections Functions:

void modify_cross_sections()
void delete_cross_sections()
void delete_one_cross_section()
void add_cross_section()
void modify_cross_section()

The user has the option to delete all of the cross-sections, delete one cross-section, add a new cross-section, or modify the parameters of an individual cross-section.

Display Component Cross-Sections:

void draw_cross_sections()
void display_cross_sections()
void remove_cross_sections()
void delete_cross_sections_structures()

The draw_cross_sections() function is used to create the PHIGS structure of each component cross-section. The display_cross_sections() function displays all of the component cross-sections PHIGS structures autocentered in the current active view(s). The PHIGS structures of the cross-sections may be removed from the current active view(s) using the remove_cross_sections() function. When the destructor of a component is called, the delete_cross_sections_structures() function is called to delete all of the component cross-sections PHIGS structures.

ACSYNT Integration Functions:

void create_surface()
void create_cross_section_surface()

The create_surface() and create_cross_section_surface() functions are used to send arrays of cross-section points to ACSYNT via C++/FORTRAN bridge function calls. These points are used by ACSYNT to create the surface model of the component.

Profile Curve

The profile curve class is the base class for the crown curve, keel curve, and plan curve derived classes (for Rule-Based Fuselage components) as well as the spine curve derived class (for Spine and Cross-Section components). Included in the profile curve class are functions used to modify, display, highlight, and autocenter profile curves in the current view.

When a new component is to be created by duplicating a previously defined component, the following function duplicates the profile curve(s) of the component to be duplicated and returns a pointer to the duplicated profile curve:

```c
    Profile_Curve* copy_profile_curve();
```

The profile curve class is also responsible for displaying the PHIGS structures of a profile curve (crown, keel, and plan curves for Rule-Based Fuselage components and spine curves for Spine and Cross-Section components).

**Display Profile Curve Functions:**

```c
    void highlight_profile_curve()
    void draw_profile_curve()
    void display_profile_curve()
```
void remove_profile_curve()
void delete_profile_curve_structure()
void autocenter_profile_curve()

The highlight_profile_curve() changes the color of the profile curve PHIGS primitives to either highlight or remove the highlighting from the profile curve. A profile curve is highlighted to indicate which profile curve of a Rule-Based Fuselage is being modified or the function is used to highlight an entire component.

The draw_profile_curve() function is used to create the PHIGS structure of a profile curve. The display_profile_curve() function displays the PHIGS structure of a profile curve. The PHIGS structure of a profile curve may be removed from the current active view(s) using the remove_profile_curve() function. When the destructor of a profile curve is called, the delete_profile_curve_structure() function is called to delete the PHIGS structure of the profile curve. The autocenter_profile_curve() function autocenters the PHIGS structure of the profile curve in the current active view(s) by determining the minimum and maximum point values along the profile curve.

The profile curve class also contains a function which is used to determine the orientation point of a cross-section which represents the intersection of the cross-section and a Rule-Based Fuselage profile curve, or the point about which a cross-section of a Spine and Cross-Section component is to be oriented normal to the spine profile curve (see Appendix C):

Point* orientation_point()

The orientation_point() function determines on which segment of the profile curve and between which two points of that segment the orientation point lies by comparing the x-values of the profile curve segment points with the cross-section depth. Once these values have been determined, based
on the segment type in which the orientation point lies, the orientation_point() function calls the appropriate functions to evaluate the orientation point.

The functions for determining the orientation point if the orientation point lies on a linear segment are members of the linear segment class (see Linear Segment). The functions for evaluating the orientation point of a cubic segment and for calculating the tangent vectors at the orientation point are members of the cubic segment class (see Cubic Segment).

**Cross-Section**

The cross-section class is the base class for the simple cross-section and piecewise cross-section derived classes and contains functions for setting the cross-section symmetry and depth of an individual cross-section as well as functions for displaying a cross-section.

The cross-section class contains the following pointers used by the component class to maintain a linked list of cross-sections:

```
Cross_Section* next
Cross_Section* previous
```

An important feature of the system Rule-Based Fuselage method is the reduction of the number of parameters that a designer must define for each cross-section. The parameters which are associated with both simple and piecewise cross-sections are as follows:

```
int    cross_section_symmetry
int    cross_section_spacing
float  cross_section_depth
```
The symmetry and spacing method of an individual cross-section are initially set by the current cross-section symmetry and spacing method of the component for which the cross-section is to be created. These parameters may be changed by the user. The cross-section depth is the position along the fuselage (for Rule-Based Fuselage component cross-sections) or along the spine curve (for Spine and Cross-Section component cross-sections). The set_cross_section_depth() function is used to determine the cross-section depth based on the spacing method.

When a Rule-Based Fuselage component cross-section is created, the cross-section must intersect the crown, keel, and plan profile curves at the cross-section depth. Pointers to these points of intersection are contained in the cross-section class:

```c
Point* crown_point
Point* keel_point
Point* plan_point
```

The cross-section class also has a pointer to the point about which a Spine and Cross-Section component cross-section is oriented normal to the spine curve:

```c
Point* orient_point
```

The crown_point, keel_point, plan_point, and orient_point are referred to as the orientation point of a given cross-section throughout this thesis. For details on the calculation of the orientation point, refer to Appendix C.

**Display Cross-Section Functions:**

```c
void display_cross_section()
void remove_cross_section()
void delete_cross_section_structure()
```
void autocenter_cross_section()

The display_cross_section() function displays the cross-section PHIGS structure autocentered in the current active view(s). The PHIGS structure of the cross-section may be removed from the current active view(s) using the remove_cross_section() function. When the destructor of a cross-section is called, the delete_cross_section_structure() function is called to delete the cross-section PHIGS structure. The autocenter_cross_section() function determines the location of the minimum and maximum extremes of the cross-section in order to autocenter the cross-section PHIGS structure in all active views.

Segment

The segment class is the base class for the linear segment and cubic segment derived classes. Included are functions to create, modify, and display linear, cubic segments, and fillet segments. Also included are functions to create, modify, and display the points of linear and cubic segments.

The parameters associated with linear, cubic, and fillet segments are as follows:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>number_of_points</td>
</tr>
<tr>
<td>Point*</td>
<td>first_point</td>
</tr>
<tr>
<td>Point*</td>
<td>last_point</td>
</tr>
</tbody>
</table>

The number of points for a linear segment is always two and must always be greater than or equal to two for a cubic segment. The pointers first_point and last_point are used to access the first and last points of the segment points linked list.
Create Segment Functions:
void set_segment_first_point()
void link_points()

When a segment other than the first segment of a profile curve or piecewise cross-section is created, the function set_segment_first_point() automatically sets the first point of the segment equal to the last point of the previous segment. The link_points() function is used to link together two points to maintain the segment points linked list.

Modify Segment Functions:
void move_segment_points_by_delta()
void scale_segment()

The move_segment_points_by_delta() is used to translate a segment of a profile curve or piecewise cross-section by moving the segment points a specified amount.

The scale_segment() function allows the user to change the size of a segment by scaling the segment in the x, y, or z directions. A PHIGS scaling matrix is created from the scaling factors input by the user. This matrix is applied to the segment PHIGS structure and the segment points values are evaluated from the matrix. This function was developed because the piecewise cross-section class needs functions to scale the individual segments of a piecewise cross-section when creating auto cross-sections (see Cross-Section class and Piecewise Cross-Section class).

When a segment of a profile curve or piecewise cross-section is modified, the following functions are used to automatically update any existing adjacent segments:

void update_adjacent_segments()
void check_previous_segment()
void check_next_segment()
void update_previous_segment()
void update_next_segment()

If the first point of the modified segment was changed, the last point of the previous segment (if it exists) must be set equal to the new first point of the modified segment. If the last point of the modified segment was changed, the first point of the next segment (if it exists) must be set equal to the new last point of the modified segment. These functions ensure that the segments of a profile curve or piecewise cross-section are always connected despite any modification of the individual segments.

**Display Segment Functions:**
void display_segment()
void remove_segment()
void delete_segment_structure()

The display_segment() function displays the PHIGS structure of a segment. The PHIGS structure of a segment may be removed from the current active view(s) using the remove_segment() function. When the destructor of a segment is called, the delete_segment_structure() function is called to delete the PHIGS structure of the segment.

**Modify Segment Points Functions:**
void delete_points()
void move_point()

The delete_points() destroys all of the segment points when the segment destructor is called. The function move_point() allows the user to move the point of a segment to a new location. When a segment point is moved, the segment is automatically updated as are any existing adjacent segments (see update_adjacent_segments() above).
Display Segment Points Functions:

void draw_segment_points()
void display_segment_points()
void remove_segment_points()
void delete_points_structures()
void display_segment_end_points()
void remove_segment_end_points()
void display_segment_intermediate_points()
void remove_segment_intermediate_points()

Several functions are provided by the segment class to create and display the points of linear and cubic segments. The endpoints of a segment are always displayed when the segments of a profile curve or piecewise cross-section are to be modified. This is done so that the user can easily see all of the different segments which define the profile curve or piecewise cross-section. The intermediate points of a cubic segment must be displayed when the user modifies the cubic segment points.

Point

The point class contains all of the data and functions to create and display an individual point of a linear, cubic, or fillet segment. Points may be created using keyboard input, by selecting points from a grid, or by automatic calculations in the case of parametric geometry creation. The point class is also used to define cross-section intersection and orientation points.

The following parameters (private variables) are associated with a point:

**XYZ Values:**

float x
float y
float  z

Input  Tangents (Slopes):
  float  dx
  float  dy
  float  dz

Tangent Vectors:
  float  x_tangent
  float  y_tangent
  float  z_tangent

  float  chord_length
  int    tangent_specification

The chord length is the Euclidean distance between consecutive points of a cubic segment and is used by the calculate_tangents() function (see Appendix D). The tangent specification of a point is set within the Cubic Segment class when a cubic segment is created. This tangent specification is used to determine whether or not the user must enter an input tangent (slope) for the current cubic segment point.

Create Point Functions:
  void  create_point()
  void  get_xyz_values()
  void  get_xyz_tangents()

The xyz values of the point must be entered by the user and the get_xyz_values() function processes the user input based on the current input method (keyboard or grid). For a cubic segment, the user must also input the xyz tangents (slopes) of a point depending on the point tangent specification. This input is processed by the get_xyz_tangents() function.
Display Point Functions:

void draw_segment_point()
void display_point()
void remove_point()
void delete_point_structure()

The draw_point() function is used to create the PHIGS structure of a segment point. The display_point() function displays the PHIGS structure of a segment point. The PHIGS structure of a segment point may be removed from the current active view(s) using the remove_point() function. When the destructor of a point is called, the delete_point_structure() function is called to delete the PHIGS structure of the segment point.

Derived Classes

Note that the protocol used to create objects of the derived classes requires that the objects must be declared the same type as the corresponding base class in order to maintain linked lists of different types of objects.

Rule-Based Fuselage

The Rule-Based Fuselage class contains functions specifically designed to create, modify, and display Rule-Based Fuselage components. Also contained in this class are functions to create, modify, and display Rule-Based Fuselage profile curves (crown, keel, and plan curve). Functions to create, display, and orient Rule-Based Fuselage cross-sections are also included in this class.
To orient a Rule-Based Fuselage cross-section means to ensure that the cross-section intersects each of the three profile curves at the cross-section depth.

To create a Rule-Based Fuselage component, the following protocol is used:

```c
Component* component; // Pointer to the base component class.
component = new Ruled_Based_Fuselage();
```

**Rule-Based Fuselage Component Functions:**

```c
void create_component()
void translate_component()
void highlight_component()
void draw_component()
void delete_component_structure()
```

The `create_component()` function allows the user to create and modify the profile curves and cross-sections of a Rule-Based Fuselage component. The `translate_component()` function may be used to move a Rule-Based Fuselage component to a new location by translating the component profile curves and cross-sections. When a component is selected to be deleted, the `highlight_component()` function highlights that component by highlighting the profile curves and cross-sections.

The `draw_component()` function creates the PHIGS structure of a Rule-Based Fuselage component which is deleted by the `delete_component_structure()` function when the component destructor is called.

**Rule-Based Fuselage Profile Curve Functions:**

```c
void create_crown_curve()
void create_keel_curve()
void create_plan_curve()
```
The Rule-Based Fuselage class contains functions to create the segments of the crown, keel, and plan profile curves. These functions are in the Rule-Based Fuselage class for the sake of reusability since all three profile curves use the same functions to create their segments.

```c
void modify_profile_curve()
void modify_profile_curve_segments()
void update_same_rbf_parameters()
```

The user has the option to modify the parameters or the individual segments of a Rule-Based Fuselage profile curve. Once a profile curve has been modified, the `update_same_rbf_parameters()` ensures that the nose position and fuselage length are the same for all three profile curves.

```c
void draw_profile_curve_structures()
void remove_profile_curve_structures()
void display_profile_curves()
void remove_profile_curves()
```

Since a Rule-Based Fuselage is defined by three profile curves (crown, keel, and plan), functions to create and display the PHIGS structures of all three profile curves at once are provided by the Rule-Based Fuselage class.

**Rule-Based Fuselage Cross-Sections Functions:**

```c
void create_cross_sections()
void initialize_cross_section()
void auto_cross_section()
void scale_cross_section()
void display_rbf_cross_section_points()
void orient_cross_section()
```

The user has the option to create individual cross-sections of a Rule-Based Fuselage component.
(provided by the create_cross_sections() function), or to create several cross-sections of the same shape (auto_cross_section() function). An auto cross-section is created by scaling a previously defined cross-section (scale_cross_section() function). This option provides an easy method of creating several cross-sections at once, previously not available in the Lockheed Rule-Based Fuselage method.

Cross-section parameters such as symmetry and depth are determined by the initialize_cross_section() function.

In order to allow the user to create a cross-section within the profile defined by the crown, keel, and plan profile curves, the points at which the cross-section intersects each of the profile curves are displayed using the display_rbf_cross_section_points() function. These points may be selected just like a point from a grid by the user when creating the cross-section segments.

The orient_cross_section() function orients the cross-section at the proper depth along the fuselage and uses functions within the cross-section derived classes that are provided to automatically ensure that the cross-section intersects the Rule-Based Fuselage profile curves. The cross-section symmetry is used to determine whether or not the cross-section PHIGS structure needs to be mirrored.

**Spine and Cross-Section**

The Spine and Cross-Section class contains functions specifically designed to create, modify, and display a Spine and Cross-Section component. This class also contains functions to create and orient Spine and Cross-Section component cross-sections normal to the spine profile curve.
To create a Spine and Cross-Section component, the following protocol is used:

```c
Component* component;  // Pointer to the base component class.
component = new Spine_and_Cross_Section();
```

**Spine and Cross-Section Component Functions:**

- `void create_component()`
- `void translate_component()`
- `void highlight_component()`
- `void draw_component()`
- `void delete_component_structure()`

The `create_component()` function allows the user to create and modify the spine profile curve and cross-sections of a Spine and Cross-Section component. The `translate_component()` function may be used to move a Spine and Cross-Section component to a new location by translating the component spine profile curve and cross-sections. When a component is selected to be deleted, the `highlight_component()` function highlights that component by highlighting the spine profile curve and cross-sections.

The `draw_component()` function creates the PHIGS structure of a Spine and Cross-Section component which is deleted by the `delete_component_structure()` function when the component destructor is called.

**Spine and Cross-Section Component Cross-Sections Functions:**

- `void create_cross_sections()`
- `void initialize_cross_section()`
- `void auto_cross_section()`
- `void scale_cross_section()`
- `void orient_cross_section()`
The user has the option to create individual cross-sections of a Spine and Cross-Section component (provided by the create_cross_sections() function), or to create several cross-sections of the same shape (auto_cross_section() function). An auto cross-section is created by scaling a previously defined cross-section (scale_cross_section() function).

Cross-section parameters such as symmetry and depth are determined by the initialize_cross_section() function. The orient_cross_section() function orients the cross-section normal to the spine profile curve at the cross-section depth. The cross-section symmetry is used to determine whether or not the cross-section PHIGS structure needs to be mirrored.

Crown Curve

The crown curve class contains functions to create the crown profile curve of a Rule-Based Fuselage component. Included are functions which allow the user to enter/modify the crown curve parameters and functions which automatically generate points along the crown curve from the parameters.

The parameters associated with the crown profile curve as follows (refer to Figures 8 and 9 on pages 28 and 29):

- nose crown angle
- crown aft offset

The nose crown angle automatically calculates different point values of the first crown curve segment depending on whether the segment is linear or cubic:
Linear Segment

\[
\begin{align*}
x &= \text{value input by user} \\
y &= 0.0 \\
z &= (x - \text{nose position x-value})\tan(\text{nose crown angle})
\end{align*}
\]

where

\[x, y, \text{and } z\] are the coordinates of the last point of the first segment of the crown profile curve.

Cubic Segment

\[
\begin{align*}
dx &= \cos(\text{nose crown angle}) \\
dy &= 0.0 \\
dz &= \sin(\text{nose crown angle})
\end{align*}
\]

where

\[dx, dy, \text{and } dz\] are the input tangents (slopes) of the first point of the first segment of the crown profile curve.

Note: these equations are valid only if the nose crown angle is non-zero.

The crown aft off-set is used to calculate the last point of the last user-defined segment (linear or cubic) if the crown aft-offset is non-zero:

\[
\begin{align*}
x &= \text{value set by user} \\
y &= 0.0 \\
z &= \text{crown aft offset}
\end{align*}
\]

where

\[x, y, \text{and } z\] are the coordinates of the last user-defined segment of the crown profile curve.
If the crown aft offset is non-zero, the last segment of the crown profile is a linear segment automatically created between the point set above by the crown aft offset and the last point of the crown profile curve set by the fuselage length.

The fuselage length and nose position are also parameters of the crown curve, but these parameters are not exclusive to the crown profile curve (i.e. the keel and plan profile curves also share these parameters).

The following equations apply to all three Rule-Based Fuselage profile curves (crown, keel, and plan curves) which relate the fuselage length and nose position to the profile curve segments (refer to Figures 8 and 9 on pages 28 and 29):

The x-value of the nose position is used to calculate the xyz values of the first point of the first segment (linear or cubic) of a Rule-Based Fuselage profile curve:

\[
\begin{align*}
x & = \text{nose position x-value} \\
y & = 0.0 \\
z & = 0.0
\end{align*}
\]

where

\[x, y, \text{ and } z \text{ are the coordinates of the first point of the first segment of the Rule-Based Fuselage profile curve.}\]

The fuselage length is used to calculate the xyz values of the last point of the last segment (linear or cubic) of a Rule-Based Fuselage profile curve:

\[
\begin{align*}
x & = \text{nose position x-value} + \text{fuselage length} \\
y & = 0.0 \\
z & = 0.0
\end{align*}
\]
where

\[ x, y, \text{and } z \text{ are the coordinates of the last point of the last segment of the} \]
\[ \text{Rule-Based Fuselage profile curve.} \]

To create a crown profile curve, the following protocol is used:

```c
Profile_Curve* curve; // Pointer to the base profile curve class.

curve = new Crown_Curve();
```

Create Crown Profile Curve Functions:

```c
void create_profile_curve()
void draw_crown_curve_auto_points()
```

The `create_profile_curve()` function initializes the crown profile curve parameters used to create the crown profile curve segments (linear and/or cubic) with or without the crown curve parameters. If the parameters are used, the crown curve minimum and maximum points are displayed using the `draw_crown_curve_auto_points()` function.

The following functions are used to automatically generate some of the crown profile curve segment points depending on the segment type (linear or cubic) if the crown curve parameters are used (see above equations):

```c
void set_last_point_rbf_first_linear_segment()
void set_last_rbf_linear_segment_point()
void set_first_point_rbf_first_cubic_segment()
void check_last_point_cubic_segment()
```

Modify Crown Profile Curve Functions:

```c
void modify_curve_parameters()
void update_curve_segments()
void update_curve_parameters()
```
By changing the crown profile curve parameters using the `modify_curve_parameters()` function, the crown profile curve segments are automatically updated using the `update_curve_segments()` functions. The `update_curve_parameters()` function automatically updates the crown profile curve parameters after the crown curve segments have been modified.

**Keel Curve**

The keel curve class contains functions to create the keel profile curve of a Rule-Based Fuselage component. Included are functions which allow the user to enter/modify the keel curve parameters and functions which automatically generate points along the keel curve from the parameters.

The parameters associated with the keel profile curve as follows (refer to Figures 8 and 9 on pages 28 and 29):

- nose keel angle
- aftbody upsweep angle
- keel aft offset

The nose keel angle automatically calculates different point values for the first segment of the keel curve depending on whether the segment is linear or cubic:

**Linear Segment**

\[
\begin{align*}
x & = \text{value input by user} \\
y & = 0.0 \\
z & = -(x - \text{nose position} \ x\text{-value}) \tan(\text{nose keel angle})
\end{align*}
\]
where

x, y, and z are the coordinates of the last point of the first segment of the keel profile curve.

Cubic Segment

\[ dx = \cos(\text{nose keel angle}) \]
\[ dy = 0.0 \]
\[ dz = \sin(\text{nose keel angle}) \]

where

dx, dy, and dz are the input tangents (slopes) of the first point of the first segment of the keel profile curve.

Note: these equations are valid only if the nose keel angle is non-zero.

The keel aftbody upsweep angle automatically calculates different point values for the last user-defined segment of the keel curve depending on whether the segment is linear or cubic:

Linear Segment

\[ x = \text{value set by user} \]
\[ y = 0.0 \]
\[ z = (x - (\text{fuselage length} + \text{nose position x-value})) \times \]
\[ \tan(\text{aftbody upsweep angle}) - \text{keel aft offset} \]

where

x, y, and z are the coordinates of the first point of the last user-defined segment of the keel profile curve.
**Cubic Segment**

\[
\begin{align*}
dx & = \cos(\text{nose keel angle}) \\
dy & = 0.0 \\
dz & = \sin(\text{nose keel angle})
\end{align*}
\]

where

\[
dx, dy, and dz are the input tangents (slopes) of the last point of the last user-defined segment of the keel profile curve.
\]

Note: these equations are valid only if the keel aftbody upsweep angle is non-zero.

The keel aft off-set is used to calculate the last point of the last user-defined segment (linear or cubic) if the keel aft-offset is non-zero:

\[
\begin{align*}
x & = \text{value input by user} \\
y & = 0.0 \\
z & = -(\text{keel aft offset})
\end{align*}
\]

where

\[
x, y, and z are the coordinates of the last user-defined segment of the keel profile curve.
\]

If the keel aft offset is non-zero, the last segment of the keel profile is a linear segment automatically created between the point set above by the keel aft offset and the last point of the keel profile curve set by the fuselage length.

The fuselage length and nose position are also parameters of the keel curve, but these parameters are not exclusive to the keel profile curve (i.e. the crown and plan profile curves also share these parameters). The equations relating the fuselage length and nose position to the segments of a
Rule-Based Fuselage profile curve are given in the Crown Curve class section.

To create a keel profile curve, the following protocol is used:

```
Profile_Curve* curve; // Pointer to the base profile curve class.

curve = new Keel_Curve();
```

**Create Keel Profile Curve Functions:**

```c
void create_profile_curve()
void draw_keel_curve_auto_points()
```

The `create_profile_curve()` allows the user to create the keel profile curve segments (linear and/or cubic) with or without the keel curve parameters. If the parameters are used, the keel curve minimum and maximum points are displayed using the `draw_crown_curve_auto_points()` function.

The following functions are used to automatically generate some of the keel profile curve segment points depending on the segment type (linear or cubic) if the keel curve parameters are used (see above equations):

```c
void set_last_point_rbf_first_linear_segment()
void set_last_rbf_linear_segment_point()
void update_previous_keel_segment()
void set_first_point_rbf_first_cubic_segment()
void check_last_point_cubic_segment()
```

**Modify Keel Profile Curve Functions:**

```c
void modify_curve_parameters()
void update_curve_segments()
void update_curve_parameters()
```
By changing the keel profile curve parameters using the modify_curve_parameters() function, the keel profile curve segments are automatically updated using the update_curve_segments() functions. The update_curve_parameters() function automatically updates the keel profile curve parameters after the keel curve segments have been modified.

**Plan Curve**

The plan curve class contains functions to create the plan profile curve of a Rule-Based Fuselage component. Included are functions which allow the user to enter/modify the plan curve parameters and functions which automatically generate points along the plan curve from the parameters.

The parameters associated with the plan profile curve as follows (refer to Figures 8 and 9 on pages 28 and 29):

- nose angle
- nose radius
- aft radius

The nose angle and radius automatically calculate different point values of the first segment of the plan curve depending on whether the segment is linear or cubic:

**Linear Segment**

\[
\begin{align*}
x &= \frac{\text{nose radius}}{\tan(\text{nose angle})} + \text{nose position x-value} \\
y &= \text{nose radius} \\
z &= 0.0
\end{align*}
\]
where

x, y, and z are the coordinates of the last point of the first segment of the plan profile curve.

\textbf{Cubic Segment}

\begin{align*}
dx & = \cos(\text{ nose angle }) \\
dy & = \sin(\text{ nose angle }) \\
dz & = 0.0
\end{align*}

where

dx, dy, and dz are the input tangents (slopes) of the first point of the first segment of the plan profile curve.

\begin{align*}
dx & = \text{value input by user} \\
dy & = \text{nose radius} \\
dz & = 0.0
\end{align*}

where

dx, dy, and dz are the input tangents (slopes) of the second point of the first segment of the plan profile curve.

\textbf{Note}: these equations are valid only if the nose angle and radius are non-zero.

The aft radius is used to calculate the last point of the last user-defined segment (linear or cubic) if the aft radius is non-zero:

\begin{align*}
x & = \text{value input by user} \\
y & = \text{aft radius} \\
z & = 0.0
\end{align*}

where

x, y, and z are the coordinates of the last user-defined segment of the plan profile.
curve.

If the aft radius is non-zero, the last segment of the plan profile is a linear segment automatically created between the point set above by the aft radius and the last point of the plan profile curve set by the fuselage length.

The fuselage length and nose position are also parameters of the plan curve, but these parameters are not exclusive to the plan profile curve (i.e. the keel and crown profile curves also share these parameters). The equations relating the fuselage length and nose position to the segments of a Rule-Based Fuselage profile curve are given in the Crown Curve class section.

To create a plan profile curve, the following protocol is used:

```cpp
Profile_Curve* curve; // Pointer to the base profile curve class.
curve = new Plan_Curve();
```

**Create Plan Profile Curve Functions:**

```cpp
void create_profile_curve()
void draw_plan_curve_auto_points()
```

The create_profile_curve() allows the user to create the plan profile curve segments (linear and/or cubic) with or without the keel curve parameters. If the parameters are used, the plan curve minimum and maximum points are displayed using the draw_crown_curve_auto_points() function.

The following functions are used to automatically generate some of the plan profile curve segment points depending on the segment type (linear or cubic) if the plan curve parameters are used (see above equations):
void set_last_point_rbf_first_linear_segment()
void set_last_rbf_linear_segment_point()
void set_first_point_rbf_first_cubic_segment()
void set_second_point_rbf_first_cubic_segment()
void check_last_point_cubic_segment()

Modify Plan Profile Curve Functions:
void modify_curve_parameters()
void update_curve_segments()
void update_curve_parameters()

By changing the plan profile curve parameters using the modify_curve_parameters() function, the plan profile curve segments are automatically updated using the update_curve_segments() functions. The update_curve_parameters() function automatically updates the plan profile curve parameters after the plan curve segments have been modified.

Spine Curve

The spine curve class contains functions to create and modify the segments of the spine profile curve of a Spine and Cross-Section component.

To create a spine profile curve, the following protocol is used:

```
Profile_Curve* curve; // Pointer to the base profile curve class.
curve = new Spine_Curve();
```

Spine Profile Curve Functions:
void create_profile_curve()
void modify_profile_curve()
The create_profile_curve() function allows the user to create the segments of the spine profile curve and the modify_profile_curve() function allows the user to modify the segments of the spine profile curve.

In addition to the functions within the base profile curve class, the spine curve class provides a function to determine the orientation point if it lies exactly on the endpoint of a linear segment of a spine profile curve:

```c
void avg_spine_orient_pt()
```

This function checks to see if the orientation point is at the second point of the linear segment and the next segment is also linear. If this is the case, the tangent vectors of the orientation point must be the average of the tangent vectors using the last point of the current segment and the first point of the next segment. It is necessary to orient the cross-section halfway between the normals of the adjacent linear segments in order to create a smoothly blended surface between the two segments.

**Simple Cross-Section**

The simple cross-section class contains functions to create and modify rectangular, circular, and elliptical cross-sections. Included are methods to create individual cross-sections or several cross-sections by scaling a previously defined cross-section, referred to as the scaling cross-section.

The simple cross-section class provides functions which can mirror the cross-section about the Z-axis (simple cross-sections always have Z-symmetry). Other functions include those which ensure that the simple cross-section of a Rule-Based Fuselage intersects the crown, keel, and profile
curves.

The following parameters (private variables) are associated with the different simple cross-sections:

**Rectangular Cross-Section Parameters:**
- float `cross_section_height`
- float `cross_section_width`

**Circular Cross-Section Parameters:**
- float `cross_section_radius`

**Elliptical Cross-Section Parameters:**
- float `cross_section_horizontal_radius`
- float `cross_section_vertical_radius`

To create a simple cross-section, the following protocol is used:

```c
int symmetry_flag; // Flag indicating if the user has changed the current
                   // cross-section symmetry.
int spacing_flag;  // Flag indicating if the user has changed the current
                   // cross-section spacing.
CrossSection* cross_section; // Pointer to the base cross-section class.
cross_section = new SimpleCrossSection( symmetry_flag, spacing_flag );
```

**Create Simple Cross-Section Functions:**
- void `create_cross_section()`
- void `create_rectangle_cross_section()`
- void `create_circle_cross_section()`
- void `create_ellipse_cross_section()`
- void `scale_cross_section()`

The `create_cross_section()` function allows the user to create rectangular, circular, and elliptical
simple cross-sections. Each type of simple cross-section uses its own function for entering the cross-section parameters (these are the create_<type>_cross_section() functions).

The scale_cross_section() is an overloaded function that creates a cross-section from a previously defined cross-section. This function is used to create auto cross-sections for both Rule-Based Fuselage and Spine and Cross-Section components.

In the case of a Rule-Based Fuselage component, the scale_cross_section() function adjusts the simple cross-section such that the cross-section intersects the crown profile curve at the given cross-section depth. If the component is not symmetrical, a simple cross-section cannot intersect all three profile curves.

For a Spine and Cross-Section component, the simple cross-section parameters are multiplied by the scaling factor input by the user. For example, for a circular cross-section, a scaling factor of two would double the cross-section radius.

**Modify Simple Cross-Section Functions:**
```c
void modify_cross_section()
```

The modify_cross_section() function allows the use to modify the parameters of a simple cross-section, depending on the type of simple cross-section (rectangle, circle, or ellipse).

**Display Simple Cross-Section Functions:**
```c
void draw_cross_section()
void draw_rectangle_cross_section()
void draw_circle_cross_section()
void draw_ellipse_cross_section()
void mirror_z_plane()
```
The `draw_cross_section()` function determines which function must be used (`draw_<type>_cross_section`) to create the PHIGS structure of simple cross-section. The `mirror_z_plane()` function is used to mirror the cross-section based on the cross-section symmetry (z-symmetry only for a simple cross-section).

**Simple Cross-Section Profile Curve Intersection Functions:**

```c
void check_rbf_cross_section()
```

This function ensures that a simple cross-section of a Rule-Based Fuselage component will intersect the crown, keel, and plan profile curves. This function is necessary because the user may create a simple cross-section that does not exactly intersect the fuselage profile curves. Rather than force the user to modify the cross-section by trial and error, this function automatically corrects the problem.

**Piecewise Cross-Section**

The piecewise cross-section class contains functions to create, modify, and display cross-sections which are comprised of linear and/or cubic segments. Included are methods to create individual cross-sections or several cross-sections by scaling a previously defined cross-section, referred to as the scaling cross-section.

The piecewise cross-section class provides functions which can mirror the cross-section segments based on the cross-section symmetry. Other functions include those which ensure that the piecewise cross-section of a Rule-Based Fuselage intersects the crown, keel, and profile curves.

To create a piecewise cross-section, the following protocol is used:
int symmetry_flag; // Flag indicating if the user has changed the current
               // cross-section symmetry.
int spacing_flag;  // Flag indicating if the user has changed the current
               // cross-section spacing.

Cross_Section* cross_section;  // Pointer to the base cross-section class.
cross_section = new Piecewise_Cross_Section( symmetry_flag, spacing_flag );

Create Piecewise Cross-Section Functions:
void create_cross_section()
void scale_cross_section()

The create_cross_section() function allows the user to create the segments of a piecewise cross-section. The scale_cross_section() is an overloaded function that creates a cross-section from a previously defined cross-section, the scaling cross-section. This function is used to create auto cross-sections for both Rule-Based Fuselage and Spine and Cross-Section components.

For a Spine and Cross-Section component, the scale_cross_section() function scales each segment of the scaling cross-section. This is done by applying a scaling matrix to each segment calculated by the scale_segment() function which is a member of the segment class (see Segment).

For a Rule-Based Fuselage component, the scale_cross_section() function first calculates the following delta amounts:

\[
\begin{align*}
\text{delta crown} &= z\text{-value of crown point 2} - z\text{-value of crown point 1} \\
\text{delta keel} &= z\text{-value of keel point 2} - z\text{-value of keel point 1} \\
\text{delta plan} &= y\text{-value of plan point 2} - y\text{-value of plan point 1}
\end{align*}
\]

where

- point 2 represents an intersection point of the cross-section to be created and point 1 represents an intersection point of the scaling cross-section.
These delta amounts are then used to shift the scaling cross-section points to create a new cross-section such that the overall shape is preserved while the cross-section still intersects the Rule-Based Fuselage profile curves. If a scaling matrix were applied to the cross-section segments instead of this method, the cross-section would not intersect the profile curves.

**Modify Piecewise Cross-Section Functions:**

```cpp
void modify_cross_section()
```

The `modify_cross_section()` function allows the user to modify the segments of a piecewise cross-section.

**Display Piecewise Cross-Section Functions:**

```cpp
void draw_cross_section()
void mirror_y_and_z_plane()
void mirror_y_plane()
void mirror_z_plane()
void display_cross_section_segments()
void remove_cross_section_segments()
```

The `draw_cross_section()` function creates the PHIGS structure of the piecewise cross-section. The mirror functions are used to mirror the cross-section segments based on the cross-section symmetry.

The PHIGS structures of the segments of a piecewise cross-section can be displayed autocentered in the current active view(s) or removed from the current active view(s) using the `display_cross_section_segments()` and `remove_cross_section_segments()` functions.
Piecewise Cross-Section Profile Curve Intersection Functions:

```c
void check_rbf_cross_section()
```

This function ensures that a piecewise cross-section of a Rule-Based Fuselage component will intersect the crown, keel, and plan profile curves. This function is necessary because the user may create a piecewise cross-section that does not exactly intersect the fuselage profile curves. Rather than force the user to modify the cross-section by trial and error, this function automatically corrects the problem.

**Linear Segment**

The linear segment class contains functions to create and modify linear segments of profile curves and piecewise cross-sections. Also included are overloaded functions to create linear segments of Rule-Based Fuselage profile curves based on the profile curve parameters.

Since the user has the option of creating linear segments using points only or by using the first point, angle, and length, the linear segment class has functions which calculate the last point of a linear segment from the first point, angle, and length. Functions which calculate the angle and length of a linear segment from the endpoints are also provided by the linear segment class.

The following parameters (private variables) are associated with a linear segment:

```c
float linear_segment_length
float linear_segment_angle
```

The angle of a linear segment is measured counterclockwise from the horizontal to the linear segment.
To create a linear segment, the following protocol is used:

```java
Segment* segment;  // Pointer to the base segment class.
segment = new Linear_Segment();
```

**Create Linear Segment Functions:**
- `void create_segment()`
- `void set_linear_segment_point()`
- `void set_linear_segment_angle_length()`

The function `create_segment()` is an overloaded function which can be used to create a linear segment for a Rule-Based Fuselage profile curve using the profile curve parameters or to create a linear segment for profile curves which do not use parameters (including the spine profile curve of a Spine and Cross-Section component).

The system provides the option of creating linear segments using the segment endpoints only or the first point, angle, and length of the segment (see Input/Display Manager). If the latter option is used, the function `set_linear_segment_point()` calculates the second point of the linear segment from the angle and length. The function `set_linear_segment_angle_length()` calculates the angle and length of the segment from the segment endpoints.

**Modify Linear Segment Functions:**
- `void modify_segment()`
- `void change_angle_length()`

The linear segment class contains functions which allow the user to modify a linear segment by changing the segment angle and/or length.
Display Linear Segment Functions:
    void draw_segment()

The function draw_segment() creates the PHIGS structure of a linear segment.

Cross-Section Orientation Functions:
    Point* linear_orient_point()
    Point* special_linear_orient_point()

These functions are used to determine the crown_point, keel_point, plan_point, and orient_point which are used to orient cross-sections of Rule-Based Fuselage and Spine and Cross-Section components (see Cross-Section).

If the orientation point lies between the endpoints of a linear segment, the linear_orient_point() function determines the xyz values of the orientation point using the slope of the linear segment. If the orientation point lies exactly on one of the linear segment endpoints, the xyz values and tangent vector are determined using the special_linear_orient_point() function.

Cubic Segment

The cubic segment class contains data and functions to create and modify cubic segments of profile curves and piecewise cross-sections of Rule-Based Fuselage and Spine and Cross-Section components. Also included are overloaded functions to create cubic segments of Rule-Based Fuselage profile curves based on the profile curve parameters.

The cubic segment class contains functions which are used to calculate the tangent vector at any point along a profile curve segment (linear and/or cubic). This tangent vector is used to orient a
cross-section normal to the spine profile curve at the cross-section depth for Spine and Cross-Section components. The tangent vector may also be used to determine the points at which a cross-section intersects the profile curves of a Rule-Based Fuselage component at the cross-section depth.

The parameters (private variables) associated with a cubic segment are as follows:

```c
float* cubic_points
int tangent_specification
    1 = no tangents specified
    2 = all tangents specified
    3 = first tangent specified
    4 = last tangent specified
    5 = first and last tangent specified
    6 = user can specify tangents at any point along the cubic segment
```

The variable cubic_points is a dynamically allocated array containing the intermediate points of the cubic segment. The cubic segment tangent specification indicates for which points of the cubic segment the user must input tangent (slope) values.

To create a cubic segment, the following protocol is used:

```c
Segment* segment; // Pointer to the base segment class.
segment = new Cubic_Segment();
```

**Create Cubic Segment Functions:**

```c
void create_segment()
void calculate_tangents()
void calculate_chord_lengths()
void calculate_intermediate_points()
```
The function create_segment() is an overloaded function which can be used to create a cubic segment for a Rule-Based Fuselage profile curve using the profile curve parameters or to create a cubic segment for profile curves which do not use parameters (including the spine profile curve of a Spine and Cross-Section component).

The cubic segment class creates a cubic segment which is actually a parametric cubic spline that uses chord length parametrization. In order to evaluate the spline (i.e. calculate the intermediate points), the tangent vectors of each cubic segment point must first be calculated. The intermediate points may then be determined using normalized cubic Hermite blending functions (see Appendix D).

The overloaded cubic_segment_tangents() functions are used to calculate the tangent vectors of the cubic segment points for a cubic segment of two or more points (one of the cubic_segment_tangents() functions handles the special case of two points and the other function handles cubic segments with more than two points). Appendix E provides the source code listing for these two functions.

However, these functions can only calculate the tangent vectors for cubic segments with the following tangent specifications:

- no tangents specified
- first tangent specified
- last tangent specified
- first and last tangents specified

Note that the all tangents specified and any tangents specified (tangent_specification = 2 and 6, respectively) options are not allowable. In order to allow the user to create cubic segments with these additional options, the calculate_tangents() function must break down the cubic segment into
portions which satisfy one of the four above tangent specifications.

For all tangents specified, the cubic segment is broken down into cubic segments of two points. For the case of any tangents specified, the following functions are used by the calculate_tangents() function to break down the cubic segment:

```c
int check_first_point_tangents()
int check_tangents()
int check_last_point_tangents()
```

These functions are used to check if a cubic segment has a non-zero tangent specified to set the tangent specification of a portion of the cubic segment accordingly. The check_first_point_tangents() function determines whether or not the first point of the cubic segment has any non-zero tangents (slopes) specified. Once the check_tangents() function detects a point (other than the first point) with non-zero tangents (slopes) specified, that point becomes the last point of the current cubic segment portion and the tangent vectors are calculated for that portion.

This process is repeated until the check_last_point_tangents() function determines if any non-zero tangents (slopes) have been specified for the last point of the last cubic segment portion. The tangent vectors are then calculated for the last cubic segment portion.

**Modify Cubic Segment Functions:**

```c
void modify_segment()
void delete_one_point()
void add_point()
void change_tangents()
```

The user has the option to delete a point (until there are only 2 points left), add a new point, or
change the input tangent (slope) values of a point.

**Display Cubic Segment Functions:**
void adjust_tangents()
void display_cubic_segment()

The system gives the user the option to use automatic tangents updating for cubic segments (see Segment Manager). If this option is used, the function adjust_tangents() automatically sets the input tangent (slope) values of the endpoints of the cubic segment equal to the slope of any existing adjacent linear segments. The magnitude of the end tangent vectors, however, are not set equal to the magnitude of the adjacent linear segments. Instead, the magnitude of the tangent vector of a cubic segment is equal to the chord length corresponding to that point (see Appendix D). The function display_cubic_segment() is then used to create and display the PHIGS structure of the cubic segment.

**Cross-Section Orientation Functions:**
Point* linear_orient_tangents()
Point* cubic_orient_point()
Point* cubic_orient_tangents()

These functions are used to determine the crown_point, keel_point, plan_point, and orient_point which are used to calculate the intersection of a Rule-Based Fuselage cross-section and the crown, keel, and plan profile curves. This function is also used to calculate the point about which a cross-section of a Spine and Cross-Section component is oriented normal to the spine profile curve (see Cross-Section). These functions are members of the cubic segment class because in order to orient a cross-section of a Spine and Cross-Section component normal to the spine profile curve, the tangent vector must be calculated at the point of orientation. This is done by creating a dummy cubic segment which uses the calculate_tangents() function.
The cubic_orient_point() function determines the crown_point, keel_point, plan_point, and orient_point by iteratively comparing the cross-section depth with the intermediate cubic segment point x-value determined by the calculate_intermediate_points() functions (see Cross-Section).

Since both the calculate_tangents() and calculate_intermediate_points() functions are members of the cubic segment class, it was necessary to make the orientation functions members of this class as well.

For a detailed description of these orientation functions, refer to Appendix C.

**Fillet Segment**

The fillet segment class contains the data and functions which can be used to create a circular arc fillet between two linear segments by specifying only the fillet radius. Using the geometry of the angles between the linear segments to be filleted, the following fillet parameters (private variables) may be calculated:

```c
Point* fillet_start_point
Point* fillet_end_point
Point* fillet_center_point
float fillet_arc_angle
float start_angle
```

These parameters are calculated within the following functions:

```c
void create_segment()
void calculate_start_angle()
```
Once the fillet parameters have been determined, the `draw_segment()` function is used to create and display the PHIGS structure of the fillet segment.

The fillet segment is linked with the two linear segments that are filleted by the fillet arc the same way that a cubic segment or another linear segment would be inserted into the segments linked list. Both of the linear segments are equally trimmed by the fillet. Their endpoints are adjusted to the fillet arc using the `update_adjacent_segments()` function (see Segment).

Figure 20 on page 99 shows the geometric relationships used to calculate the parameters of the fillet between the two segments of lengths $L_1$ and $L_2$. The fillet arc trims an equal length from both of the segments, the trim length shown on Figure 20. The fillet radius, $r$, is specified by the user. The angles, $A_1$ and $A_2$, used in the following equations are the slopes of the linear segments of lengths, $L_1$ and $L_2$, respectively. The calculations for the fillet arc are as follows:

$$\text{fillet}_\text{-}\text{arc}\_\text{angle}, \beta:\ $$

$$\beta = \cos^{-1}\left(\frac{L_1^2 + L_2^2 - L_3^2}{2L_1L_2}\right)$$

equal length trimmed by the fillet:

$$\text{trim}\_\text{-}\text{length} = \frac{\text{fillet}\_\text{-}\text{radius}}{\tan^{-1}\left(\frac{\beta}{2}\right)}$$

$$\text{fillet}_\text{-}\text{start}_\text{-}\text{point}, P_1:\ $$

$$P_{1y} = Y_1 + (L_1 - \text{trim}\_\text{-}\text{length})\cos(A_1)$$

$$P_{1z} = Z_1 + (L_1 - \text{trim}\_\text{-}\text{length})\sin(A_1)$$
Figure 20. Fillet Geometrical Relationships.
fillet_end_point, P₂:

P₂y = Y₂ + (trim_length)cos(A₂)
P₂z = Z₂ + (trim_length)sin(A₂)

fillet_center_point, C:

\[ D = \sqrt{(L₁ - \text{trim\_length})^2 + \text{fillet\_radius}^2} \]

\[ \vartheta = \sin^{-1}\left(\frac{\text{fillet\_radius}}{D}\right) \]

\[ C_y = Y₁ + (D)\cos(A₁ - \vartheta) \]
\[ C_z = Z₁ + (D)\sin(A₁ - \vartheta) \]

fillet arc starting angle:

\[ \text{start\_angle} = \tan^{-1}\left(\frac{P₁_{y} - C_y}{P₁_{z} - C_z}\right) \]

Utility Classes

The utility classes are one level above the base classes developed for this system (refer to Figure 15 on page 47). Many of the utility classes are designed to provide the tools created for the system with access to the interface framework (menu manager, view manager, and input/display manager) [Woya92].
Menu Manager

The menu manager class contains menu data and functions which are inherited by the view manager and, consequently, the input/display manager classes. This menu manager is not the same as the Menu_Manager class for the interface framework developed by Woyak [Woya92]. Any references made to the menu manager class are for the system menu manager created for this research unless explicitly stated.

The menu manager initializes the interface manager which controls the raising and lowering of all interface windows (menus and views). The interface manager is a pointer to the Interface_Manager class which is part of the interface framework [Woya92]. For example, the following would be used to raise a menu:

```c
Pop_Up_Menu menu;
Interface_Manager* manager;

manager = get_interface_manager();
manager->raise_window( &menu );
```

Menu is an object of the Pop_Up_Menu class and manager is a pointer to the Interface_Manager class. The function get_interface_manager() is a member of the system menu manager class and returns a pointer to the Interface_Manager class. The function raise_window() is a member of the framework Interface_Manager class and raises the specified menu.

Pointers to the main menu and the current active menu are maintained by the menu manager. These pointers are used to raise or lower the main menu and current active menu when necessary via pointer to the input/display manager class:

```c
Static_Menu* main_menu
Pop_Up_Menu* current_menu
```
The menu functions provided by the menu manager are text input menus and a confirm menu. These menus are closed control menus which means that the user must perform whatever action or actions that the menu requires before the menu releases control (i.e. the user cannot select anything or perform any actions outside of the menu window until a selection has been made).

**Menu Manager Menu Functions:**

```c
int input_menu()
int message_menu()
```

The input menus allow the user to enter character strings, floating point, and integer values and the `message_menu()` function is a confirm menu which displays messages to the user.

The menu manager menu functions create the menu, process the menu, and return the and identifier identifying the menu selection. For example, the following would be used to process a confirm menu:

```c
int menu_selection = message_menu( message, 2 );
```

In this example, `message` is a character array containing the message to be displayed to the user and the text for the two push buttons. The number 2 indicates that the menu is to have two push buttons (typically OK and CANCEL).

The following functions are used to dynamically set the menu dimensions based on the number of menu items and the menu item text:

```c
float set_pop_up_menu_width()
float set_pop_up_menu_height()
float set_input_menu_width()
```
float set_input_menu_height()
float set_frame_x()
float set_frame_y()
float set_frame_width()
float set_max_string_length()

These functions are also used by other classes which need to create their own specific menus such as the main menu created in the component manager or the regular_menu() function of the input/display manager.

**View Manager**

The view manager class contains all of the system viewing data and functions which are inherited by the input/display manager class. All system views are created using the Simple_View class of the interface framework [Woya92]. This view manager is not the same as the View_Manager class for the interface framework. Any references made to the view manager class are for the system view manager created for this research unless explicitly stated.

The view manager initializes the eight views of the system. The first four views are full display views of the front, side, top, and isometric views respectively. The next four views are used to define a four-view display area which shows the front, top, side, and isometric views simultaneously. All eight views are standard PHIGS 3D views.

The view manager is responsible for displaying all PHIGS geometry and provides functions to toggle between different views as well as autocentering the PHIGS geometry within the current active view.
When a PHIGS structure is displayed, it is displayed in all eight views at once. This is done so that the structure is displayed if the user toggles to a view other than the current active view. The PHIGS structures associated with each of the eight views are actually copies of the original PHIGS structure since standard PHIGS does not allow a structure to be associated with more than one view at a time.

For example, the following would be used to display a PHIGS structure:

```
PHIGS_Structure_ID* structure_id[8];

display_structure( structure_id );
```

PHIGS_Structure_ID is a class defined in the interface framework which provides functions for a standard PHIGS structure such as open() and PHIGS_delete(). Structure_id is an array of eight PHIGS structures. Only the first structure (structure_id[0]) is actually created and modified. The other seven are copies for displaying the structure in all eight views.

The view manager also sets the grid horizontal minimum and maximum values based on the size of the current active view. These values are used to initialize the parameters of a new grid (described later).

The view manager contains pointers to the simple views used by the system:

```
Simple_View* full_front_view
Simple_View* full_side_view
Simple_View* full_top_view
Simple_View* full_iso_view
Simple_View* front_view
Simple_View* side_view
Simple_View* top_view
Simple_View* isometric_view
```
The view manager also maintains the xyz minimum and maximum values which represent component object limits that are used to autocenter the objects in the current active view. These values are the PHIGS viewport dimensions:

```plaintext
float x_minimum
float x_maximum
float y_minimum
float y_maximum
float z_minimum
float z_maximum
```

The current active view and plane in which that view is defined are also members of the view manager:

```plaintext
int current_view
int current_plane
```

Although the user can toggle between the active views, the system automatically activates different views depending upon the component object which is being created or modified (i.e. entire component, profile curve, or cross-section). These views are activated or deactivated by the following functions:

```plaintext
void profile_curve_view()
void cross_section_view()
void plan_curve_view()
void component_view()
void isometric_view()
```

These functions automatically set the current_view and current_plane parameters. Other viewing
functions of the view manager class are as follows:

```c
void set_view_parameters()
void deactivate_current_view()
void autocenter_display()
```

The `set_view_parameters()` sets the view orientation and translation matrices of the system views. The orientation matrix is used to define different planes for each view and the translation matrix is used to autocenter PHIGS structures within the view. When a view is activated, the current active view is deactivated by the `deactivate_current_view()` function. The `autocenter_display()` sets the viewport dimensions and the translation matrix for a view in which PHIGS structures are to be autocentered.

The view manager is also responsible for displaying PHIGS structures in the current active view. The following functions are used by all other classes to display/remove PHIGS structures:

```c
void display_structure()
void remove_structure()
```

**Input/Display Manager**

The input/display manager provides all of the other classes with access to the menu manager and view manager functions. For example, to autocenter a structure in the current view, the following protocol would be used:

```c
Input_Display_Manager* manager;
manager->autocenter_display();
```
The function autocenter_display() is a member of the view manager class, but is made a member of the input/display manager class through inheritance.

In addition to the functions inherited from the menu manager and view manager, the input/display manager also contains data and functions which are used to select the input methods used by the system. The user has the option to enter points data for segments (linear and cubic) either by keyboard or selecting points from a grid. Also, the user has the option to create linear segments either by providing two endpoints only or by specifying the first point, angle, and length of the linear segment.

The following variables and functions enable the user to set the input options provided by the system:

**Variables:**

<table>
<thead>
<tr>
<th>int</th>
<th>input_method</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>keyboard</td>
</tr>
<tr>
<td>2</td>
<td>grid</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int</th>
<th>linear_segment_input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>points only</td>
</tr>
<tr>
<td>2</td>
<td>first point, angle, and length</td>
</tr>
</tbody>
</table>

**Functions:**

<table>
<thead>
<tr>
<th>void</th>
<th>process_input_method_menu()</th>
</tr>
</thead>
<tbody>
<tr>
<td>void</td>
<td>process_linear_segment_input_menu()</td>
</tr>
</tbody>
</table>

These functions are used to process menus which allow the user to choose the input options provided by this system.

Since one of the point input options is via grid, the input/display manager provides access to the grid class which allows the user to create a grid in either the front, side, or top views.
The following variables and functions are used to provide other classes with access to the grid class via pointer to the input/display manager class:

Variables:
Grid* XY_grid_pointer
Grid* YZ_grid_pointer
Grid* XZ_grid_pointer

Functions:
void process_grid_menu()
void grid_off()
Point* get_grid_point()

The process_grid_menu() function allows the user to create a grid in the front, side, or top view (see Grid). The grid_off() function removes the grid from the current active view(s). The get_grid_point() function enables the user to select a point from the grid and returns a pointer to that point.

The following variables are used by the get_grid_point() function to determine whether the user selected a point from the grid, a point from a Rule-Based Fuselage profile curve auto point, or a cross-section intersection point:

Profile_Curve* crown_pointer
Profile_Curve* keel_pointer
Profile_Curve* plan_pointer
PHIGS_Structure_ID* rbf_cross_section_points_structure_id[8]
PHIGS_Structure_ID* rbf_curve_auto_points_structure_id[8]

These variables are in the input/display manager class because they must be compared with grid variables that are only accessible by the input/display manager class.
The input/display manager also provides the system with an open control menu. An open control menu is a pop-up menu which allows the user to select outside of the menu window in order to access the main menu options which are also controlled by the input/display manager.

**Input/Display Manager Menu Functions:**

```c
int regular_menu()
void process_main_menu()
void process_views_menu()
```

The `regular_menu()` function creates and processes open-control pop-up menus and returns an identifier identifying the menu selection. The `process_main_menu()` function processes the main menu created by the component manager class (note that the menu options change for this main menu). The `process_views_menu()` allows the user to toggle between the different views of the system.

**Grid**

The grid class is used by the input/display manager to allow the user to create a grid in the front, side, or top view. The grid is used to create and modify profile curve and piecewise cross-section segments as well as to move geometry from one location to another.

There are three different types of grids: (1) Auto Grid, (2) New Grid, and (3) Center Grid. The Auto Grid automatically creates a grid in the current active view. A New Grid allows the user to enter the minimum and maximum horizontal limits of the grid as well as the horizontal and vertical grid spacing. The Center Grid allows the user to enter the grid spacing (horizontal and vertical spacing are the same for a Center Grid) and the grid half-length. The grid automatically sets the
horizontal and vertical minimum and maximum limits such that the grid is centered about the origin in the horizontal and vertical directions. Furthermore, the grid length is set based on the grid spacing such that the origin point (0,0,0) always lies on the grid. This type of grid is always used to create cross-sections because cross-sections must be created about the origin to take advantage of their symmetry.

The grid vertical minimum and maximum are automatically set such that the grid is always centered about zero vertically. This is done to ensure that the grid is a perfect square.

Whenever a grid is created, the input method is automatically set to grid input. It is possible to have more than one grid displayed at once (up to three) provided that each grid is defined in a different view.

If the user attempts to create a grid in the isometric view or the 4-view display, the user will be prompted to enter the view in which the grid is to be oriented.

Once a grid is defined, the grid remains displayed until a new grid is created in the same plane, or the user turns the grid off. If all grids have been turned off, the input method automatically defaults back to keyboard input.

The following parameters (private variables) are used to define a grid:

```c
int grid_type
    1 = Auto Grid
    2 = New Grid
    3 = Center Grid

int grid_plane
    1 = XY plane
    2 = YZ plane
```
3 = XZ plane
float horizontal_minimum
float horizontal_maximum
float vertical_maximum
float horizontal_spacing
float vertical_spacing
float* grid_points

The variable grid_points is a dynamically allocated array of grid points. The array width is 3 (corresponding to the x, y, and z values of the grid point).

Create Grid Functions:
void draw_grid()
void grid_spacing()
int grid_points_space()
void display_grid()

The function draw_grid() creates the grid PHIGS structure and calculates the points for the grid based on the grid spacing and the plane in which the grid is to be oriented. The grid spacing is determined by the grid_spacing() function and depends on the type of grid to be created. The grid_points_space() function determines the amount of space which must be allocated for the grid_points array. The display_grid() function is used to display the grid PHIGS structure in the current active view(s).

Segment Manager

The segment manager class contains functions used by the Profile Curve and Piecewise Cross-Section classes to create, modify, and display the segments of a profile curve or piecewise cross-section.
Segments modification functions include routines which allow the user to add a segment to or delete a segment from the group of segments which define a profile curve or piecewise cross-section. It is the responsibility of the segment manager to update the segments linked list of the profile curve or piecewise cross-section when this type of modification is performed. The following example shows how the segments linked list must be updated after an intermediate segment has been deleted:

```c
// Pointer to the segment to be deleted.
Segment* segment;

// Pointer to the previous segment.
Segment* previous_segment = segment->get_previous();

// Pointer to the next segment.
Segment* next_segment = segment->get_next();

// Delete the points and their PHIGS structures of the segment and delete the PHIGS
// structure of the segment.
segment->delete_segment();

// Update the segments linked list.
previous_segment->set_next( next_segment );
next_segment->set_previous( previous_segment );

number_of_segments--;

// Call the segment object destructor.
delete segment;
```

Display functions within the Segment Manager include routines which can display/remove the endpoints of all of the segments of a profile curve or piecewise cross-section in/from the current view. The following is an example of how to display all of the segments endpoints:

```c
// Pointer to the Input/Display Manager class.
Input_Display_Manager* manager;

// Pointer to a profile curve for which the segments endpoints are to be displayed.
Profile_Curve* curve;

curve->display_segments_end_points( manager );
```
The display_segments_end_points() function is accessible to the object of the profile curve class through inheritance from the segment manager class.

The following variables and functions of a profile curve or piecewise cross-section are contained in the segment manager class:

**Protected Variables:**

- `int number_of_segments`
- `Segment* first_segment`
- `Segment* last_segment`

**Create Segments Functions:**

- `void link_segments()`

The first_segment and last_segment pointers provide access to the segments linked list of a profile curve or piecewise cross-section. The function link_segments() is used to link two segments together to maintain the segments linked list.

**Modify Segments Functions:**

- `void delete_segments()`
- `void delete_one_segment()`
- `void add_one_segment()`
- `void move_segment()`
- `void move_seg_pt_to_pt()`
- `void move_seg_by_amount()`
- `void scale_segments()`
- `void scale_one_segment()`

The user may delete all of the segments (thus deleting the entire profile curve or piecewise cross-
section), delete one segment, add a new segment, move a segment, or scale a segment.

The system provides two methods for moving the segment of a profile curve or piecewise cross-section: (1) move the segment by moving a reference point on the segment to a new location and (2) move the segment by entering an amount.

The scale_segments() function scales all of the segments of a profile curve or piecewise cross-section and is used to create an auto cross-section from another piecewise cross-section.

Display Segments Functions:

void draw_segments()
void display_segments()
void remove_segments()
void delete_segments_structures()

The segment manager class contains functions to create and display the PHIGS structures of all of the segments of a profile curve or piecewise cross-section. The segments are displayed rather than the PHIGS structure of the profile curve or piecewise cross-section when the segments are to be modified, enabling the user to select the segments.

Display Segments Points Functions:

void display_segments_points()
void remove_segments_points()
void display_segments_end_points()
void remove_segments_end_points()

Several functions are provided by the segment manager class to display the points of profile curve and piecewise cross-section segments. These functions use the point displaying functions contained in the segment class (see Segment).
The segment manager also gives the user the option to use the automatic end tangents updating feature. If a cubic segment is adjacent to a linear segment, the end tangent values (slopes) of the cubic segment are set equal to the slope of the linear segment, thus matching the tangents and preserving continuity.

The following private variables and functions enable the user to set the automatic tangents updating features:

**Private Variables:**

```c
int tangents_update
  0 = do not automatically update tangents
  1 = automatically update tangents
```

**Public Functions:**

```c
void process_tangents_update_menu()
```

This function is used to process the menu which allows the user to choose the tangents updating options provided by the system.

**Component Manager**

The component manager maintains a linked list of all of the components (Rule-Base Fuselage and Spine and Cross-Section) which have either been created by the system or read in from an archived file. The component manager also provides the system with the necessary data and functions for creating and modifying components.

Execution of the system begins by calling the constructor of the component manager and
processing the main menu:

```c
Component_Manager* manager = new Component_Manager();
manager->process_main_menu();
```

Within the `process_main_menu()` function, the component manager initializes the input/display manager class which, in turn, initializes the view manager and menu manager classes (i.e., calls the class constructors).

The component manager maintains a linked list of components (Rule-Based Fuselage and Spine and/or Cross-Section) by using the following data and functions:

**Private Variables:**

```c
int number_of_components
Component* first_component
Component* last_component
```

**Create Components Functions:**

```c
void create_component()
void link_components()
```

The component manager allows the user to create Rule-Based Fuselage and Spine and Cross-Section components and keeps track of the total number of components created within the system. Pointers to the first and last components provide access to the components linked list. The `link_components()` function is used to link together two components in the linked list.

The component manager provides the following functions to create and modify all of the components:

```c
void modify_components()
void delete_components()
```
void delete_one_component()
void draw_components()

The user has the option to delete all of the components or delete one component. The user may also create new components or modify an existing component using the modify_components() function. The draw_components() function is used to create the PHIGS structures of Rule-Based Fuselage and Spine and Cross-Section components.

The component manager also preprocesses the components data for surface model generation using the create_surface() function. This function loads all of the points data for each cross-section into arrays which are sent to the conceptual aircraft design code, ACSYNT, to create the surface geometry of the components.

**File Manager**

The file manager class provides the component manager with functions to save components to an archive file and to read components from an archive file. There are file input/output functions for all class objects that define Rule-Based Fuselage and Spine and Cross-Section components (i.e. points, segments, profile curves, and cross-sections).

The file manager functions may be used to write/read to/from an ACSYNT geometry file or to a stand-alone archive file.

Components that are read from an archive file are created within the file manager from the data contained in the file (i.e. the constructors of all class objects defining each component are called and all necessary linked lists created). Once all of the components have been created, the PHIGS
structure of each component is created and displayed and the component manager is updated.

**Save Components Functions:**

```c
void save_acsyt_component()
void save_component()
void save_ruled_based_fuselage()
void save_spine_and_cross_section()
void save_cross_sections()
void save_cross_section()
void save_segment()
void save_point()
```

These functions are used to save all existing components by writing the data needed to create the components to an archive file. The `save_acsyt_component()` function is used to archive to the ACSYNT geometry file. The function `save_ruled_based_fuselage()` saves the Rule-Based Fuselage profile curves (crown, keel, and plan curves) data and the function `save_spine_and_cross_section()` saves the spine profile curve data.

These functions save the data for all objects which define a component (profile curve(s), cross-sections, segments, and points) by traversing the linked lists of these objects and accessing the private data of each object through public functions.

**Load Components Functions:**

```c
void load_acsyt_component()
void load_component()
void load_ruled_base_fuselage()
void load_spine_and_cross_section()
void load_profile_curve()
Segment* load_profile_curve_segment()
void load_cross_sections()
Cross_Section* load_cross_section()
```
Segment* load_cross_section_segment()

These functions are used to load all existing components by reading the data needed to create the components from an archive file. The objects which define each component are created within the file manager class from the data read from the archive file. The load_acsynt_component() function is used to read components data from an ACSYNT geometry file.
11. User Interface

User Friendliness

A requirement of any user interface is that the interface should be "user friendly". To produce software that is "user friendly", the screen layout and menu structures should be simple and easy to follow. The software must also be "foolproof", meaning that if the user attempts to make the interface perform an activity that cannot be done, the interface must not allow that action to take place. The interface should also provide continuous feedback to keep the user informed of what is happening, typically in the form of messages and/or prompts.

Look and Feel

Another important aspect of the interface design is the "look and feel" of the interface. The terms "look and feel" refer to the particular color scheme, logo, menu layout, etc. associated with a given software product. If the user is familiar with a particular "look and feel", then the user will immediately be able to use any software which has the same "look and feel". For example, anyone familiar with Microsoft Windows will be able to use any windows software which uses the same kind of icons and mouse clicking routines.

Because the ultimate goal of this thesis was to implement the methods of creating Rule-Based Fuselage and Spine and Cross-Section components with the existing ACSYNT geometric modeling methods, it was necessary to maintain the same "look and feel" that currently exists within ACSYNT. However, the need to use menus that are consistent with those found in most
current software applications (moveable "motif-like" menus with push buttons) was also considered an important part of developing an updated "look and feel" for the new ACSYNT module.

In order to achieve this particular "look and feel" for the interface, the interface framework developed by Woyak was chosen [Woya92]. The interface framework was designed to have a similar color scheme and header as those found in ACSYNT so that ACSYNT users will recognize the new user interface as a part of ACSYNT. At the same time, the user interface emulates "motif-like" menus to which software users have grown accustomed.
12. Implementation and Examples of Results

**PHIGS**

The system described in this thesis includes a graphical user interface. The graphics language in which the user interface is written is the ISO 3D international graphics standard, PHIGS (Programmers Hierarchical Interactive Graphics System), which enables the user interface software to be graphics-device independent [That88].

**C++**

The user interface of this system was programmed in the C++ language because of its ability to support object-oriented programming and design. C++ is a superset of the C language and uses all of the powerful features of C as well as additional new features developed for C++.

**Rule-Based Fuselage Components**

Figures 21 and 22 on pages 123 and 124 show an example aircraft fuselage created using the Rule-Based Fuselage method. Figure 21 shows the fuselage with cross-sections in the 4-view display. The upper left-hand corner is the top view (XY-plane), the lower left-hand corner is the side view (XZ-plane), the lower right-hand corner is the front view (YZ-plane), and the upper right-hand corner is the isometric view.

Figure 22 shows the isometric view of the fuselage and cross-sections shown in Figure 21.
Figure 21. Example Rule-Based Fuselage Component in 4-View Display.
Figure 22. Example Rule-Based Fuselage Component in Isometric View.
Figure 23 on page 126 shows the crown profile curve of the fuselage in Figure 21 with its associated parameters. The crown curve consists of four segments: linear, cubic, cubic, and linear. The last linear segment was automatically generated from the aft offset. The automatic tangents updating was turned off to create the profile shape.

Figure 24 on page 127 shows the keel profile curve of the fuselage in Figure 21 with its associated parameters. The keel profile curve consists of one cubic segment.

Figure 25 on page 128 shows the plan profile curve of the fuselage in Figure 21 with its associated parameters. The plan profile curve consists of one cubic segment and one linear segment which was automatically generated from the aft radius.

Figure 26 on page 129 shows only the cross-sections of the fuselage in Figure 21 in the isometric view. Figure 27 on page 130 shows the piecewise cross-section which was scaled to create all of the other component cross-sections.
Figure 23. Example Rule-Based Fuselage Crown Curve.
Figure 24. Example Rule-Based Fuselage Keel Curve.
Figure 25. Example Rule-Based Fuselage Plan Curve.
Figure 26. Example Rule-Based Fuselage Cross-Sections.
Figure 27. Example Rule-Based Fuselage Cross-Section.
Spine and Cross-Section Components

Figure 28 on page 132 shows a Spine and Cross-Section component in the 4-view display with a variety of cross-sections. The isometric view of the same component is shown in Figure 29 on page 133. These figures illustrate the different types of cross-sections available for this system. The first three cross-sections are simple cross-sections (rectangle, circle, and ellipse). The next cross-section is piecewise, consisting of linear and cubic segments.

Figure 30 on page 134 is a spine profile curve consisting of linear segments only.
Figure 28. Spine and Cross-Section Component in 4-View Display.
Figure 29. Spine and Cross-Section Component in Isometric View.
Figure 30. Spine Profile Curve in Side View.
System Cross-Sections

The Rule-Based Fuselage component may be defined using the following cross-section types:

- Fully flexible, non-parametric
- Simple parametric
- Extended parametric

Fully Flexible, Non-Parametric Cross-Section

A fully flexible, non-parametric cross-section consists of piecewise linear and/or cubic segments. Fillets may also be created between two adjacent linear segments. The user creates the cross-section by creating each individual cross-section segment without any associated parameters. To modify the cross-section, the individual segments must be selected and modified.

Simple Parametric Cross-Section

A simple parametric cross-section is a simple geometric shape (rectangle, circle, and ellipse) with only one or two associated parameters which fully define the cross-section. Only the parameters may be changed to modify the cross-section shape.

Extended Parametric Cross-Section

An extended parametric cross-section is an extension of the cross-section type suggested by Lockheed. This cross-section also consists only of linear segments and fillets, but the user may specify any number and combination of linear segments and fillets. The cross-section may be modified by changing the associated parameters (segment lengths and angles and the fillet radii).
Since this segment is also defined by piecewise segments, the points of the segments may also be modified in which case the associated parameters are automatically adjusted.

The parametric cross-section suggested by Lockheed is also implemented within the system as a specific case of the extended parametric cross-section with the limitation of specifying only five linear segments.

The Spine and Cross-Section component may be defined using the same cross-section types as those used for the Rule-Based Fuselage component (fully flexible, non-parametric, simple parametric, and extended parametric). The Lockheed cross-section type is not applicable since it is based on the intersection between the cross-section and the three profile curves of a Rule-Based Fuselage component.
13. ACSYNT Integration

In order to validate the Rule-Based Fuselage and Spine and Cross-Section methods of creating aircraft components, the object-oriented Rule-Based Fuselage and Spine and Cross-Section (RBFSC) geometry system was integrated with the conceptual aircraft design code, ACSYNT. The user should refer to the user guide of Appendix A, specifically the ACSYNT section, for an explanation of the menu options described in the following discussion.

Create New Component

To create a Rule-Based Fuselage or Spine and Cross-Section component from within ACSYNT, the user selects "ADD" from the ACSYNT component menu and adds the appropriate component from the components list. This automatically launches the RBFSC system via FORTRAN/C++ bridge function call from ACSYNT. The ACSYNT component number must be passed as an argument through the bridge function to the RBFSC system.

In order to launch the RBFSC system in this manner, the ACSYNT default geometry file was modified for the new component types, Rule-Based Fuselage and Spine and Cross-Section. In addition to the new component types, new cross-section types were also defined for both component types. These modifications as well as the creation of additional menu items and external FORTRAN/C++ bridge functions, were made by Dr. Uma Jayaram, VPI & SU.

Once the RBFSC system has been launched, the Rule-Based Fuselage or Spine and Cross-Section component may be created using the procedures outlined in the user guide. In order to define the
component within ACSYNT, the geometric data must be sent back to ACSYNT. This is accomplished within the RBFSC system by selecting the "SURFACE" option from the RBFSC main menu. This option loads the component cross-sections points into arrays which are sent to ACSYNT through the ACSYNT FORTRAN functions ptcmp(), ptrxs(), and ptsrf().

After selecting the "SURFACE" option, the user may exit the RBFSC system and return to ACSYNT. The geometric model of the new component should be displayed in the ACSYNT geometry window.

**Modify Component**

A component created using the procedures described above is fully defined within ACSYNT and may be modified using the same menu options as any other ACSYNT component.

The component may be copied by selecting "COPY" from the ACSYNT component menu. This option invokes a FORTRAN/C++ bridge function which accesses the RBFSC system functions that duplicate the component by constructing component objects exactly the same as those of the component to be copied.

The component location and general parameters may be modified using the procedures outlined in the user guide. These modifications are only within ACSYNT and do not affect any of the component data maintained by the RBFSC system because ACSYNT cannot access this geometric information. For this reason, the user is not allowed to change the number of cross-sections or number of points per cross-section since this information is maintained by the RBFSC system and sent to ACSYNT.
The component shape may be modified by selecting “COMP SHAPE” from the ACSYNT modify component menu which automatically launches the RBFSC system. Within the RBFSC system, the component may be modified following the procedures outlined in the user guide.

**ACSYNT Geometry File**

The component may be archived along with any other existing ACSYNT components by selecting “FILE” from the ACSYNT main menu. Additional information associated with the component must also be archived for the RBFSC profile curves as well as the types of cross-sections and segments defining the component. This information is appended to the ACSYNT geometry file when the component is saved using the “FILE” option by accessing the component data via FORTRAN/C++ bridge function.

When the ACSYNT geometry file is read using the ACSYNT “FILE” option, the appended information is used to create the component objects via bridge function calls which access the RBFSC File Manager class.

The RBFSC system also provides a file archival/retrieval option which may be used to read a single component. This option allows the user to add an RBFSC component to the current ACSYNT geometry which was defined for a previously created ACSYNT geometry.

**Examples of Results**

Figure 31 on page 141 shows the ACSYNT geometry module with the new component list which now includes the Rule-Based Fuselage and Spine and Cross-Section components.
Figure 32 on page 142 shows a Rule-Based Fuselage component that has been sent back to the ACSYNT geometry module from the RBFSC system.

Figure 33 on page 143 shows a Spine and Cross-Section component that has been sent back to the ACSYNT geometry module from the RBFSC system.

Figure 34 on page 144 shows a Spine and Cross-Section component that has been copied using the ACSYNT geometry module.

**Limitations**

A limitation that must be imposed on the RBFSC system for ACSYNT integration is the number of cross-sections that can be used to define a Rule-Based Fuselage or Spine and Cross-Section component. Since the system is object-oriented and programmed in C++, all objects are created using dynamic memory allocation which means that there is no set limit for the number of objects to be created. However, the ACSYNT geometry functions which are written in FORTRAN must allocate a specified amount of array space for the component cross-sections. As a result, the number of cross-sections created within the system must be limited to this amount, which is currently only a maximum of 20 cross-sections.

The number of points per cross-section created within the system must also be limited for the same reasons of memory allocation described above. The current ACSYNT limitation is a maximum of 29 points per cross-section.
Figure 31. ACSYNT Geometry Module.
Figure 32. ACSYNT Rule-Based Fuselage Component.
Figure 33. ACSYNT Spine and Cross-Section Component.
Figure 34. Copied ACSYNT Spine and Cross-Section Component.
14. Conclusions and Recommendations

The Rule-Based Fuselage method for designing aircraft fuselage suggested by Lockheed has been implemented in an object-oriented system. This method allows an aircraft designer to create an aircraft fuselage using a wide range of cross-section shapes.

For this system, the Lockheed Rule-Based Fuselage method was enhanced to increase the method flexibility and eliminate the limitations imposed on the cross-sections consisting of only five linear segments. The enhanced method provides the capability to create cross-sections with any number of linear, cubic segments, and fillet segments as well as simple shapes such as rectangles, circles, and ellipses. This refined version of the Lockheed Rule-Based Fuselage method also enables the designer to take advantage of any existing cross-section symmetry. Moreover, the user is no longer required to specify the parameters for each individual cross-section.

The Spine and Cross-Section method of defining an arbitrary geometric shape was also developed for the system, reusing many of the classes created for the Rule-Based Fuselage method. This method may be used to create a wide variety of complex geometric components such as inlets, ducts, etc.

To validate the Rule-Based Fuselage and Spine and Cross-Section methods, the new system has been integrated with the conceptual aircraft design code, ACSYNT. Once an aircraft designer creates a fuselage using the Rule-Based Fuselage method or an inlet or duct using the Spine and Cross-Section method within the new system, the geometry can be sent to the ACSYNT geometry module.
The use of parametric design methods in conceptual aircraft design allows aircraft designers to easily modify aircraft geometry by changing only the parameters associated with the geometry. The Rule-Based Fuselage method enables aircraft designers to parametrically design an aircraft fuselage by specifying the parameters for the fuselage profile curves. The fuselage may then be modified by changing the parameters of each individual profile curve.

Parametric cross-sections are available for the Rule-Based Fuselage and Spine and Cross-Section components, the simple parametric and extended parametric cross-sections. The Lockheed parametric cross-section is also available for Rule-Based Fuselage components. Rule-Based Fuselage and Spine and Cross-Section components can also be defined using fully flexible, non-parametric cross-sections.

Because the system is object-oriented, the system may be easily expanded by simply adding new classes. For example, new cross-section types could be developed by creating a new derived cross-section class.

The flexibility of object-oriented design will enable software designers in the future to modify existing classes and create new ones. Possible future modifications should include the development of more segment derived classes. Other curve-fitting methods could be used instead of the existing parametric cubic spline methods as well as methods which allow the user to define a segment by specifying equations.
15. References


[Fari90] Farin, G., Curves and Surfaces for Computer Aided Geometric Design: A


Appendix A. User Guide
The following user guide explains how to create and modify aircraft components using the Rule-Based Fuselage and the Spine and Cross-Section methods. The user guide consists of several sections:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview</td>
<td>explains some of the terms used throughout this user guide.</td>
</tr>
<tr>
<td>Main Menu</td>
<td>describes the main menu options.</td>
</tr>
<tr>
<td>EDIT Main Menu Options</td>
<td>describes the new main menu options which replace the original main menu options when the EDIT option is selected.</td>
</tr>
<tr>
<td>Create Segments</td>
<td>describes how to create linear and cubic segments of profile curves and piecewise cross-sections.</td>
</tr>
<tr>
<td>Create Rule-Based Fuselage Component</td>
<td>describes how to create the profile curves of a Rule-Based Fuselage component.</td>
</tr>
<tr>
<td>Modify RBF Profile Curve</td>
<td>describes how to modify the profile curves of a Rule-Based Fuselage component.</td>
</tr>
<tr>
<td>Create Spine and Cross-Section Component</td>
<td>describes how to create the spine profile curve of a Spine and Cross-Section component.</td>
</tr>
<tr>
<td>Modify Profile Segments</td>
<td>describes how to modify the segments of a profile curve of a Rule-Based Fuselage or Spine and Cross-Section component.</td>
</tr>
<tr>
<td>Create Cross-Sections</td>
<td>describes how to create and modify simple and piecewise cross-sections of a Rule-Based Fuselage or Spine and Cross-Section component.</td>
</tr>
<tr>
<td>Modify Component</td>
<td>describes how to modify a Rule-Based Fuselage or Spine and Cross-Section component.</td>
</tr>
<tr>
<td>Viewing Options</td>
<td>describes the viewing options that can be used to change the viewing orientation of the component geometry.</td>
</tr>
<tr>
<td>ACSYNT User Guide</td>
<td>describes how to create and modify components within ACSYNT.</td>
</tr>
</tbody>
</table>
Overview

Menu Push Buttons

Note that when a menu push button is referred to as “not available”, this means that the push button label is not highlighted. Push buttons with labels that are not highlighted cannot be selected (if the push button is selected, nothing happens). Once the push button has become available (automatically determined by the system), the push button label will be highlighted.

Open Control Menu

An open control menu is a pop-up menu which can access the main menu options by “double-clicking” on one of the main menu options. Once a main menu option is selected by double-clicking, the current open control pop-up menu disappears and the main menu selection is processed. Once this process is complete, the open control pop-up menu reappears.

The first mouse click releases control from the pop-up menu, thereby causing it to disappear. The second mouse click is actually used to select the main menu option. If the user inadvertently clicks outside of the menu and does not wish to select any main menu options, the MENU option (see MENU under EDIT Menu Options) may be selected to redisplay the open control pop-up menu.

Closed Control Menu

A closed control menu is a pop-up menu which must be processed (i.e. a menu item must be
selected which will cause the menu to disappear) before any other user input is processed. The pop-up menus which are used to receive text and numerical input as well as the confirm menus of the system are all closed control menus.

**Main Menu**

When the RBF system is first executed, the following main menu options appear at the top of the screen:

EXIT FILES EDIT SURFACE AUTOCENTER VIEWS

Note that the SURFACE option is not available because there are no components currently defined. Following are brief descriptions of each of these menu items.

**EXIT**

To exit the system, select the EXIT option from the main menu. The following confirm menu should appear:

EXIT?
OK CANCEL

Select OK from the current menu to exit, or select CANCEL to return to the main menu of the system.
FILES

To read in components from an archive file or to save defined components to an archive file, select the FILES option from the main menu. The following menu options should appear:

    Load Component
    Save Component
    CANCEL

If no components are currently defined, the Save Component option is not available.

Load Component

To read in components from an archive file, select Load Component from the current menu. The following menu should now appear:

    Enter Filename:
    CANCEL

Enter the name of the file to be read. If the name does not exist, the following menu should appear:

    Enter New Filename
    CANCEL

To enter a new filename, select Enter New Filename. If the filename exists, the component(s) should be displayed and the current menu returned to the main menu.
**Save Component**

To save components to an archive file, select Save Component from the current menu. The following menu should now appear:

Enter Filename:

CANCEL

Enter the name of the file to be saved. If the name already exists, the following menu should appear:

Replace Filename
Enter New Filename

CANCEL

To write over the existing file, select Replace Filename. To enter a new filename, select Enter New Filename. Once the component has been successfully saved, the following confirm menu should appear:

File Saved Successfully

OK

Select OK and the current menu is returned to the main menu.

**EDIT**

To create and modify Rule-Based Fuselage and/or Spine and Cross-Section components, select EDIT from the main menu options. Note that the first four main menu options have changed. The
current main menu items should now be as follows:

MENU INPUT TANGENTS GRID AUTOCENTER VIEWS

The new menu items are described in the EDIT Main Menu Options section. Once EDIT has been selected, the following open control pop-up menu should now appear:

Create RBF Component
Create Spine Component
Modify Component
Delete Component
DONE

Note that the last two menu options are not available if no components currently exist. These menu items are discussed in detail in later sections.

SURFACE

Once a component has been defined, the surface model may be prepared by selecting the SURFACE option from the main menu. The system DOES NOT create a surface model of the component. Instead, the component data is prepared to create a surface model using another application, for example, the conceptual aircraft design code, ACSYNT.

When the SURFACE option is selected and the component data has been prepared to generate a surface model, the component cross-sections should be redrawn with a different color to indicate that the cross-section points have been successfully converted.

Note that this surface model data is not saved with the component geometry when the FILES option is used to save the component.
AUTOCENTER

To autocenter the current geometry, select the AUTOCENTER option from the main menu. The geometry should now be autocentered in the current active view(s).

VIEWS

To toggle between the views, select the VIEWS option from the main menu. The following menu options should now appear:

- All Views
- Front View
- Side View
- Top View
- CANCEL

Select the view to activate from the current menu.

EDIT Main Menu Options

As mentioned in the EDIT section (see Main Menu), when the EDIT option is selected from the main menu, some of the main menu options change. The following sections describe these main menu options.
MENU

Most of the pop-up menus in this system are open control (see Open Control Menus) to allow the user to access the main menu options by double-clicking. As mentioned earlier, when a selection is made outside of an open control menu, the menu disappears. If the user unintentionally caused the menu to disappear, selecting the MENU option from the EDIT main menu will redisplay the menu.

INPUT

The system allows input of points data either from the keyboard or by selecting from a grid. The default is keyboard input. Creating a grid automatically sets the default to grid input and turning off a grid resets the default to keyboard (see GRID). It is possible to create a grid and use keyboard input using the INPUT option.

The system also allows linear segments to be created either by creating both endpoints or by creating the first point and specifying the segment angle and length. The default method is to create both endpoints.

To change the default input options, select the INPUT option from the EDIT main menu. The following menu options should appear:

- Keyboard Input
- Grid Input
- Linear Segment
- CANCEL
If the current input method is keyboard input, the Keyboard Input option is not available. If the current input method is grid input or no grids have been defined, the Grid Input is not available either. If at least one grid has been defined and the current input method is grid input, select Keyboard Input from the current menu to change the current input method to keyboard input. Then, selecting Grid Input from this menu will return the input method to grid input.

To change the default linear segment input option, select Linear Segment from the current menu. The following menu options should appear:

Points Only
Point, Angle, and Length
CANCEL

The current linear segment input menu will not be available. Select the desired linear segment input method from the current menu.

TANGENTS

The system provides automatic tangents updating for the end tangents of cubic segments. This feature sets the input tangents (slopes) of the cubic segment endpoints equal to the slope of any existing adjacent linear segment.

The default for the automatic tangents updating is on. To change the default, select the TANGENTS option from the EDIT main menu. The following menu options should appear:

Updating On
Updating Off
CANCEL
The current tangents updating option will not be available. Select the desired option from the menu.

GRID

The system provides methods for defining a grid which may be used to create points. The grid may be oriented in the front, side, or top view.

To create a grid, select the GRID option from the EDIT main menu. If the current view is the front, side, or top view, the following create grid menu options should appear:

Auto Grid
New Grid
Center Grid
Grid Off
CANCEL

Note that the Grid Off option is not available if no grid is currently defined in the current view.

If the current view is the isometric the view or the 4-view display, selecting GRID from the secondary main menu will cause the following menu options to appear:

Front View
Side View
Top View
CANCEL

Since a grid cannot be defined in the isometric view or 4-view display, one of the views above must be selected from this menu. Once one of the above views is selected, the create grid menu
options should now appear.

**Auto Grid**

An Auto Grid is generated automatically and is sized to the current view (or defining view). To create an Auto Grid, select Auto Grid from the create grid menu. A grid should now appear on the screen.

**New Grid**

A New Grid is generated from the grid parameters entered by the user. To create a New Grid, select New Grid from the create grid menu. The following menu should now appear:

```
  Horizontal Spacing:  1.0
  Vertical Spacing:    1.0
  Horizontal Minimum:  -50.0
  Horizontal Maximum:  50.0

  DONE
```

Note that the default values may not necessarily correspond to the default values given here. This is because the default values are generated based on the size of the current view (or defining view).

Also note that the user cannot enter values for the vertical minimum and maximum. This is because the grid is automatically centered about the horizontal axis.

Enter the grid parameters by selecting the parameter value with the mouse or by tabbing to the parameter value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until all grid parameters are set to their desired values. Select DONE when finished.
entering the grid parameter values. A grid should now appear on the screen.

*Center Grid*

A Center Grid will automatically be centered about the vertical axis (in addition to being automatically centered about the horizontal axis). The Center Grid is typically used to create cross-sections. Because cross-sections must be created about the origin (0,0,0), the Center Grid sets the total grid length based on the grid spacing such that the origin is always displayed.

To create a Center Grid, select Center Grid from the create grid menu. The following menu should now appear:

```
Grid Length: 72
Grid Spacing 3
DONE
```

Enter the grid parameters by selecting the parameter value with the mouse or by tabbing to the parameter value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until all grid parameters are set to their desired values. Select DONE when finished entering the grid parameter values. A grid should now appear on the screen.

Note that the displayed grid length is not the grid length that was entered. This is because the grid length is adjusted such that it is a multiple of the grid spacing and then doubled to ensure that the origin will be displayed on the grid.
Grid Off

If the current view is the front, side, or top view and a grid exists in this view, select Grid Off from the create grid menu to remove the grid. The grid should now disappear. If no grid exists in this view, this option is not available.

If the current view is the isometric view or 4-view display and any grids exist in any of the other views (front, side, or top), select Grid Off from the create grid menu and the following menu options should now appear:

Front View
Side View
Top View
CANCEL

Note that the menu options are only available if a grid exists in that view. To remove any existing grids, select the view in which the grid exists from the current menu. The grid should now disappear.

If the current view is the isometric view or 4-view display and no grids exist in any of the other views (front, side, or top), the Grid Off option is not available.

Create Segments

The system allows the user to create linear and cubic segments. The points for the segments may be created either by keyboard or grid input (see INPUT).
Note that the first point of a segment is only created when the current segment is the first segment of the profile curve or piecewise cross-section. Otherwise, the first point of the segment is automatically created equal to the last point of the previous segment.

When creating segments of a profile curve or piecewise cross-section, the following menu options appear:

- Linear Segment
- Cubic Segment
- DONE

This menu is open control, meaning that the options from the main menu may be selected before selecting anything from the current menu. At this point, if the user wishes to create a grid to create the segment points, select GRID from the main menu and create the grid (see GRID).

Create Linear Segment

To create a linear segment, select Linear Segment from the current menu. Depending on the current input method (see INPUT), one of the following should appear on the screen:

**Keyboard Input**

A menu with the following should appear:

Enter X: 0.0
Enter Y: 0.0
Enter Z: 0.0

DONE       CANCEL
Note: this menu will not appear if the segment is not the first segment of the current profile curve or piecewise cross-section and the linear segment input option is point, angle, and length (see INPUT).

Enter the XYZ values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until all XYZ values are set to their desired values.

Select DONE when finished entering all XYZ values (note that the default for each value is always zero). A point with the entered XYZ values should now be displayed on the screen.

If the point that was just created was the first point of the first segment, the above menu should reappear. Follow the same procedure to create the second point of the segment. A line should now be displayed between the two points.

If, however, the point that was just created was the second point of the current segment, a line should now be displayed between this point and the preceding point.

The system automatically assumes that the next segment to be created will also be linear. The above menu should now reappear. If another linear segment is to be created, follow the procedures outlined above. Otherwise, select CANCEL from the current menu. The following menu options should appear:

- Linear Segment
- Cubic Segment
- DONE

Select the segment type to be created or select DONE to finish creating the segments of the current
profile curve or piecewise cross-section.

**Grid Input**

In order to create a segment using grid input, a grid must first be created (see GRID). After the grid has been created, select Linear Segment from the current menu. The following message should appear at the bottom left-hand corner:

Select Point from the Grid

Note: this message will not appear if the segment is not the first segment of the current profile curve or piecewise cross-section and the linear segment input option is point, angle, and length (see INPUT).

Use the mouse to select the desired point from the grid. A point should now be displayed at the position selected by the mouse.

If the point that was just created was the first point of the first segment, the above message should reappear. Follow the same procedure to create the second point of the segment. A line should now be displayed between the two points.

If, however, the point that was just created was the second point of the current segment, a line should now be displayed between this point and the preceding point.

The system automatically assumes that the next segment to be created will also be linear. The above message should now reappear. If another linear segment is to be created, follow the procedures outlined above. Otherwise, use the mouse to select anywhere other than on the grid. The following menu options should appear:

Linear Segment
Cubic Segment
DONE

Select the segment type to be created or select DONE to finish creating the segments of the current profile curve or piecewise cross-section.

Create Linear Segment using Angle and Length

When a linear segment is created using the first point, angle, and length, the procedure for creating the first point of the first segment is the same as outlined above depending on the current input method.

However, if the point to be created is the second point of the linear segment, the following menu appears (regardless of the current input method):

Angle: 0.0
Length: 0.0
DONE CANCEL

Enter the angle and length by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until the segment angle and length are set to their desired values.

Select DONE when finished entering the angle and length values (note that the default for each value is always zero). A point should now be displayed on the screen and a line should now be displayed between this point and the preceding point.
The system automatically assumes that the next segment to be created will also be linear. The above menu should now reappear. If another linear segment is to be created, follow the procedures outlined above. Otherwise, select CANCEL from the current menu. The following menu options should appear:

Linear Segment
Cubic Segment
DONE

Select the segment type to be created or select DONE to finish creating the segments of the current profile curve or piecewise cross-section.

Create Cubic Segment

To create a cubic segment, select Cubic Segment from the current menu. The following menu should appear on the screen:

Enter Number of Points: 3
DONE CANCEL

Enter the number of cubic segment points by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the number of points. Hit enter after entering the number of points. If the value was entered properly, the menu should now display the entered number of points.

Select DONE when finished. Note that the default number of points is 3. The following menu options should now appear:
No Tangents
All Tangents
First Tangent
Last Tangent
First and Last Tangent
Any Tangents

CANCEL

These are the cubic segment input tangent specification options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Tangents</td>
<td>user does not specify any tangents</td>
</tr>
<tr>
<td>All Tangents</td>
<td>user specifies all tangents</td>
</tr>
<tr>
<td>First Tangent</td>
<td>user specifies first tangent only</td>
</tr>
<tr>
<td>Last Tangent</td>
<td>user specifies last tangent only</td>
</tr>
<tr>
<td>First and Last Tangent</td>
<td>user specifies first and last tangents only</td>
</tr>
<tr>
<td>Any Tangents</td>
<td>user selects which tangents to specify</td>
</tr>
</tbody>
</table>

Depending on the input tangent specification, the user may or may not be prompted to enter tangent values for certain points. Select one of the 6 input tangent specification options and depending on the current input method (see INPUT), one of the following should appear on the screen:

**Keyboard Input**

A menu with the following should appear:

Enter X: 0.0
Enter Y: 0.0
Enter Z: 0.0

DONE CANCEL

Enter the XYZ values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until all XYZ values are set to their desired values.
Select DONE when finished entering all XYZ values (note that the default for each value is always zero). Depending on the input tangent specification selected above and the current point being created, one of three actions will occur:

No Tangents Specified:

A point with the entered XYZ values should now be displayed on the screen.

Any Tangents Specified:

The following menu options should now appear:

Enter Tangent Values?

YES    NO

Other tangent specifications:

Depending on the tangent specification and the current point, the following menu may appear:

Enter DX:  0.0
Enter DY:  0.0
Enter DZ:  0.0

DONE    CANCEL

Enter the XYZ input tangent values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until all XYZ input tangent values are set to their desired values.
Select DONE when finished entering all XYZ values (note that the default for each value is always zero). A point should now be displayed on the screen.

If the point that was just created was the last point of the cubic segment, a cubic segment should now be displayed. Otherwise, continue the process outlined above until all cubic segment points are created.

The system automatically assumes that the next segment to be created will also be cubic. The Enter Number of Cubic Segment Points menu should now reappear. If another cubic segment is to be created, follow the procedures outlined above. Otherwise, select CANCEL from the current menu. The following menu options should appear:

Linear Segment
Cubic Segment
DONE

Select the segment type to be created or select DONE to finish creating the segments of the current profile curve or piecewise cross-section.

**Grid Input**

In order to create a segment using grid input, a grid must first be created (see GRID). After the grid has been created, select Cubic Segment from the current menu.

The procedures for entering the number of cubic segment points and the cubic segment input tangent specification are the same as those outlined above (keyboard input).
Once the cubic segment input tangent specification is entered, the following message should appear at the bottom left-hand corner:

Select Point from the Grid

Use the mouse to select the desired point from the grid. Depending on the input tangent specification selected above and the current point being created, one of three actions will occur:

No Tangents Specified:

A point should now be displayed on the screen.

Any Tangents Specified:

The following menu options should now appear:

Enter Tangent Values?

YES NO

Other tangent specifications:

Depending on the tangent specification and the current point, the following message may appear:

Select Tangent from Grid

Use the mouse to select a point from the grid. The input tangent will be calculated from the slope of the line between the selected point and the current point of the cubic segment.
A point should now be displayed. If the point that was just created was the last point of the cubic segment, a cubic segment should now be displayed. Otherwise, continue the process outlined above until all cubic segment points are created.

The system automatically assumes that the next segment to be created will also be cubic. The Enter Number of Cubic Segment Points menu should now reappear. If another cubic segment is to be created, follow the procedures outlined above. Otherwise, select CANCEL from the current menu. The following menu options should appear:

- Linear Segment
- Cubic Segment
- DONE

Select the segment type to be created or select DONE to finish creating the segments of the current profile curve or piecewise cross-section.

*Create Rule-Based Fuselage Component*

From the main menu, select the EDIT option. The following menu options should now appear:

- Create RBF Component
- Create Spine Component
- Modify Component
- Delete Component
- DONE

Note that the last two menu options are not available if no components currently exist. Also note
that the main menu options have changed (see EDIT Main Menu Options). Select Create RBF Component from the current menu. The following menu options should now appear:

Create Crown Curve
Create Keel Curve
Create Plan Curve
Modify Crown Curve
Modify Keel Curve
Modify Plan Curve
Create Cross-Sections
Auto Cross-Sections
Modify Cross-Sections

DONE

Note that only the first three options are currently available. In order to create the Rule-Based Fuselage component, the crown, keel, and plan profile curves must be created. The order in which the profile curves are created is not important. For simplicity, create the profile curves in the order of the first three menu options.

Create Crown Curve

Select Create Crown Curve from the current menu. An input menu for the crown curve parameters should now appear:

Nose Position 0.0
Fuselage Length 0.0
Aft Offset 0.0
Nose Crown Angle 0.0

DONE

Enter the crown curve parameters by selecting the parameter with the mouse or by tabbing to the parameter with the keyboard and then enter the desired parameter value. Hit enter after entering the parameter value. If the value was entered properly, the menu should now display the entered
parameter value. Continue this process until all parameters are set to their desired values.

Note: it is not necessary to enter values for all of the parameters. However, the parameters will not be used to generate a crown profile curve if the fuselage length is zero. Also note that no parameter values need be entered. In this case, the crown profile curve parameters are automatically generated after the crown curve has been created.

When all of the desired parameter values have been entered, select the DONE menu option at the bottom of the menu. If the fuselage length is greater than zero, two or three points will be displayed on the screen.

The leftmost point indicates the first point of the crown profile curve and is located at the value entered for the nose position. The rightmost point directly across from the leftmost point indicates the last point of the crown profile curve and is located at the nose position plus the fuselage length. If the value entered for the aft-offset was greater than zero, a third point indicating the aft-offset will be displayed directly above the rightmost point.

Once the DONE menu option is selected, whether or not the fuselage length is greater than zero, the following menu options should appear:

- Linear Segment
- Cubic Segment
- DONE

Create the crown profile curve segments (see Create Segments). Note that if the crown profile curve parameters are used to create the crown curve (i.e. the fuselage length is greater than zero), some of the segment points of the crown curve are automatically created by the system.
Important note: the crown profile curve is defined in the XZ plane and must be created in the +Z half (upper half of the screen).

If the aft-offset of the crown profile curve is non-zero, creating a point with an x-value equal to that of the last point of the fuselage will cause the system to automatically create a linear segment between the last two rightmost points of the fuselage. At this point, the crown curve is fully defined.

The crown profile curve should now appear displayed in the 4-view display. The following menu options should now appear:

Create Crown Curve
Create Keel Curve
Create Plan Curve
Modify Crown Curve
Modify Keel Curve
Modify Plan Curve
Create Cross-Sections
Auto Cross-Sections
Modify Cross-Sections
DONE

Note that the Modify Crown Curve option is now available. To modify the crown profile curve, select Modify Crown Curve and refer to the Modify RBF Profile Curve section.

Create Keel Curve

Select Create Keel Curve from the current menu. An input menu for the keel curve parameters should now appear:

<table>
<thead>
<tr>
<th>Nose Position</th>
<th>X.X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuselage Length</td>
<td>X.X</td>
</tr>
</tbody>
</table>
Aft Offset  
Nose Keel Angle  0.0  
Aft-Body Upsweep Angle  0.0  

DONE

If the parameters for the crown profile curve were entered as non-zero, those values will be the same for the current keel curve parameters menu. Those values are indicated by the X's.

Enter the keel curve parameters by selecting the parameter with the mouse or by tabbing to the parameter with the keyboard and then enter the desired parameter value. Hit enter after entering the parameter value. If the value was entered properly, the menu should now display the entered parameter value. Continue this process until all parameters are set to their desired values.

Note: it is not necessary to enter values for all of the parameters. However, the parameters will not be used to generate a keel profile curve if the fuselage length is zero. Also note that no parameter values need be entered. In this case, the keel profile curve parameters are automatically generated after the keel curve has been created.

Also note: if the crown profile curve has already been created, the parameters set by the creating the crown curve (values indicated by the X's) should not be changed.

When all of the desired parameter values have been entered, select the DONE menu option at the bottom of the menu. If the fuselage length is greater than zero, two or three points will be displayed on the screen.

The leftmost point indicates the first point of the keel profile curve and is located at the value entered for the nose position. The rightmost point directly across from the leftmost point indicates the last point of the keel profile curve and is located at the nose position plus the fuselage length. If
the value entered for the aft-offset was greater than zero, a third point indicating the aft-offset will be displayed directly above the rightmost point.

Once the DONE menu option is selected, whether or not the fuselage length is greater than zero, the following menu options should appear:

Linear Segment
Cubic Segment
DONE

Create the keel profile curve segments (see Create Segments). Note that if the keel profile curve parameters are used to create the keel curve (i.e. the fuselage length is greater than zero), some of the segment points of the keel curve are automatically created by the system.

Important note: the keel profile curve is defined in the XZ plane and must be created in the -Z half (lower half of the screen).

If the aft-offset of the keel profile curve is non-zero, creating a point with an x-value equal to that of the last point of the fuselage will cause the system to automatically create a linear segment between the last two rightmost points of the fuselage. At this point, the keel curve is fully defined.

The keel profile curve should now appear displayed in the 4-view display. The following menu options should now appear:

Create Crown Curve
Create Keel Curve
Create Plan Curve
Modify Crown Curve
Modify Keel Curve
Modify Plan Curve
Create Cross-Sections
Auto Cross-Sections
Modify Cross-Sections
DONE

Note that the Modify Keel Curve option is now available. To modify the keel profile curve, select Modify Keel Curve and refer to the Modify RBF Profile Curve section.

Create Plan Curve

Select Create Plan Curve from the current menu. An input menu for the plan curve parameters should now appear:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nose Position</td>
<td>X.X</td>
</tr>
<tr>
<td>Fuselage Length</td>
<td>X.X</td>
</tr>
<tr>
<td>Nose Angle</td>
<td>0.0</td>
</tr>
<tr>
<td>Nose Radius</td>
<td>0.0</td>
</tr>
<tr>
<td>Aft Radius</td>
<td>0.0</td>
</tr>
</tbody>
</table>

DONE

If the parameters for the crown and profile curves were entered as non-zero, those values will be the same for the current plan curve parameters menu. Those values are indicated by the X’s.

Enter the plan curve parameters by selecting the parameter with the mouse or by tabbing to the parameter with the keyboard and then enter the desired parameter value. Hit enter after entering the parameter value. If the value was entered properly, the menu should now display the entered parameter value. Continue this process until all parameters are set to their desired values.

Note: it is not necessary to enter values for all of the parameters. However, the parameters will not be used to generate a plan profile curve if the fuselage length is zero. Also note that no parameter values need be entered. In this case, the plan profile curve parameters are automatically generated.
after the plan curve has been created.

Also note: if the crown and keel profile curves have already been created, the parameters set by the creating the crown and keel curves (values indicated by the X's) should not be changed.

When all of the desired parameter values have been entered, select the DONE menu option at the bottom of the menu. If the fuselage length is greater than zero, two or three points will be displayed on the screen.

The leftmost point indicates the first point of the plan profile curve and is located at the value entered for the nose position. The rightmost point directly across from the leftmost point indicates the last point of the plan profile curve and is located at the nose position plus the fuselage length. If the value entered for the aft-offset was greater than zero, a third point indicating the aft-offset will be displayed directly above the rightmost point.

Once the DONE menu option is selected, whether or not the fuselage length is greater than zero, the following menu options should appear:

    Linear Segment
    Cubic Segment
    DONE

Create the plan profile curve segments (see Create Segments). Note that if the plan profile curve parameters are used to create the keel curve (i.e. the fuselage length is greater than zero), some of the segment points of the plan curve are automatically created by the system.

Important note: the plan profile curve is defined in the XY plane and must be created in the
+Y half (upper half of the screen).

If the aft-offset of the plan profile curve is non-zero, creating a point with an x-value equal to that of the last point of the fuselage will cause the system to automatically create a linear segment between the last two rightmost points of the fuselage. At this point, the plan curve is fully defined.

The plan profile curve should now appear displayed in the 4-view display. The following menu options should now appear:

Create Crown Curve
Create Keel Curve
Create Plan Curve
Modify Crown Curve
Modify Keel Curve
Modify Plan Curve
Create Cross-Sections
Auto Cross-Sections
Modify Cross-Sections
DONE

Note that the Modify Plan Curve option is now available. To modify the keel profile curve, select Modify Keel Curve and refer to the Modify RBF Profile Curve section.

Also note that the Create Cross-Sections and Auto Cross-Sections options are now available. At this point, the Rule-Based Fuselage component is defined and may be saved and the cross-sections created.

To save the component, select DONE from the current menu and refer to the File Input/Output section. To create the component cross-sections, refer to the Creating Cross-Sections section.


Modify RBF Profile Curve

To modify the profile curves of a Rule-Based Fuselage component, the following menu should be displayed:

Create Crown Curve  
Create Keel Curve  
Create Plan Curve  
Modify Crown Curve  
Modify Keel Curve  
Modify Plan Curve  
Create Cross-Sections  
Auto Cross-Sections  
Modify Cross-Sections

DONE

This menu appears when the Create RBF Component or Modify Component option is selected under the Edit Components menu (from the EDIT option of the main menu).

Only those profile curves that have been created can be modified. If a profile curve has not been created, its corresponding Modify <curve> Curve option is not available (<curve> refers to the profile curve type - Crown, Keel, or Plaa).

Select the option Modify <curve> Curve for the curve to be modified. The following menu should now appear:

Modify Curve Segments  
Modify Curve Parameters

DONE

To modify the segments of the profile curve, select Modify Curve Segments from the current menu and refer to the Modify Profile Curve Segments section. Note that the after the segments of the
profile curve have been modified, the profile curve parameters are automatically updated. Also note that the profile curve parameters and segments of any other existing profile curves are also automatically updated.

To modify the profile curve parameters, select Modify Curve Parameters from the current menu. The same menu that was used to enter the profile curve parameters when the profile curve was created should now appear. To change the profile curve parameters, follow the same procedure of entering the parameter values described in the Create <curve> Curve section.

Note that once a parameter value is changed, the profile curve segments are automatically updated. Also note that the profile curve segments and parameters of any other existing profile curves are also automatically updated.

Create Spine and Cross-Section Component

From the main menu, select the EDIT option. The following menu options should now appear:

Create RBF Component
Create Spine Component
Modify Component
Delete Component
DONE

Note that the last two menu options are not available if no components currently exist. Also note that the main menu options have changed (see Main Menu).

Select Create Spine Component from the current menu. The following menu options should now
appear:

Create Spine Curve
Modify Spine Curve
Create Cross-Sections
Auto Cross-Sections
Modify Cross-Sections

DONE

Note that only the first option is currently available. In order to create the Spine and Cross-Section component, the spine profile curves must be created.

Create Spine Curve

Select Create Spine Curve from the current menu. The following menu options should now appear:

Linear Segment
Cubic Segment

DONE

Create the spine profile curve segments (see Create Segments). The spine profile curve should now appear displayed in the 4-view display. The following menu options should now appear:

Create Spine Curve
Modify Spine Curve
Create Cross-Sections
Auto Cross-Sections
Modify Cross-Sections

DONE

Note that all of the menu options are now available. At this point, the Spine and Cross-Section
component is defined and may be saved and the cross-sections created.

To save the component, select DONE from the current menu and refer to the File Input/Output section. To create the component cross-sections, refer to the Creating Cross-Sections section.

Also, the spine profile curve may be modified. To modify the spine profile curve, select Modify Spine Curve from the current menu and refer to the Modify Profile Curve Segments section.

**Modify Profile Curve Segments**

When the segments of a profile curve are to be modified, the following menu options should appear:

- Delete Profile Curve
- Delete Segment
- Add Segment
- Modify Segment
- Move Segment
- Scale Segment
- DONE

**Delete Profile Curve**

To delete all of the profile curve segments, thereby deleting the profile curve, select Delete Profile Curve from the current menu.

Note: deleting a profile curve automatically deletes any existing cross-sections. The current
component is also no longer defined.

Delete Segment

To delete a segment from the profile curve, select Delete Segment from the current menu. The following message will be displayed in the lower left-hand corner:

Select Segment to Delete

Use the mouse to select the segment to delete. The segment will be highlighted and the following menu will be displayed:

Delete Selected Segment?

OK CANCEL

To delete the segment, select OK from the current menu.

If the deleted segment was between two other segments, the following message will be displayed in the lower left-hand corner:

Select Segment to be Adjusted

Since a gap was created by deleting the selected segment, the gap must be filled. Use the mouse to select the segment which will be adjusted to close the gap (the segment must be adjacent to the deleted segment).

The remaining segments are now displayed without the deleted segment (with any necessary adjustments). If the segment was the only segment of the profile curve, the Delete Segment has the
same effect as the Delete Profile Curve option (see above).

**Add Segment**

To add a segment to the profile curve, select Add Segment from the current menu. The following menu options should now be displayed:

- End Segment
- Between Segments
- DONE

Note that the Between Segments option is not available if only one segment exists.

**End Segment**

To add a segment to the end of the profile curve, select End Segment from the current menu. If more than one segment exists, the following message will be displayed in the lower left-hand corner:

- Select End Segment

Use the mouse to select the end segment of the profile curve to which a segment is to be added. If only one segment exists, the following message is displayed in the lower left-hand corner:

- Select Segment End

Use the mouse to select the endpoint of the segment to indicate to which end a segment is to be
added. The following menu options should now be displayed:

    Linear Segment
    Cubic Segment
    CANCEL

Select the type of segment to be added from the current menu and create the segment (see Create Segments). Note that the first point of the segment is automatically created if the segment is added after the current last segment of the profile curve.

Also note that if a segment is added before the current first segment of the profile curve, the first point of the current first segment is automatically adjusted to be equal to the last point of the added segment.

The profile curve should now be displayed with the added segment.

**Between Segments**

To add a segment between two other segments of a profile curve, select Between Segments from the current menu. The following message will be displayed in the lower left-hand corner:

    Select 1st Segment

Use the mouse to select one of the segments between which the new segment is to be added (note: the order in which the segments is selected is not important). The following message will be displayed in the lower left-hand corner:

    Select 2nd Segment
Use the mouse to select the other of the two segments between which the new segment is to be added. The following menu options should now be displayed:

- Linear Segment
- Cubic Segment
- CANCEL

Select the type of segment to be added from the current menu and create the segment (see Create Segments). The endpoints of the selected segments will automatically adjust to be equal to the endpoints of the added segment.

The profile curve should now be displayed with the added segment.

Modify Segment

To modify the points of a profile curve segment, select Modify Segment from the current menu. The following message will be displayed in the lower left-hand corner:

Select Segment to Modify

Use the mouse to select the segment to be modified. The segment will be highlighted and all of the segment points should now be displayed. Depending on the type of segment (cubic or linear), one of the following menus will appear:

Modify Linear Segment

- Move Point
- Change Angle/Length
Move Point

To move the point of a linear segment to a new location, select Move Point from the current menu. The following message will be displayed in the lower-left hand corner:

Select Point to Move

Use the mouse to select the segment point to be moved. Depending on the current input method (see INPUT), the following should occur:

Keyboard Input:

A menu with the following should appear:

Enter X: X.X
Enter Y: X.X
Enter Z: X.X

DONE  CANCEL

The X's represent the current values of the point to be moved. Enter the new XYZ values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until all XYZ values are set to their desired values.

Select DONE when finished entering the new XYZ values. The segment should now be updated and the profile curve segments should now be displayed with all updates due to the modification. should now be updated and displayed with the point moved to the new location.
Grid Input:

The following message will be displayed in the lower-left hand corner:

Select Point from Grid

Use the mouse to select the new point location from the grid. The segment should now be updated and the profile curve segments should now be displayed with all updates due to the modification.

Change Angle/Length

To change the angle or length of a linear segment, select Change Angle/Length from the current menu. The following menu should now appear:

<table>
<thead>
<tr>
<th>Angle</th>
<th>X.X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>X.X</td>
</tr>
<tr>
<td>DONE</td>
<td>CANCEL</td>
</tr>
</tbody>
</table>

The X's represent the current segment angle and length values. Enter the new angle and/or length by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until the segment angle and/or length are set to their desired values.

Once the segment angle and length are set to their desired values, select DONE from the current menu. The profile curve segments should now be displayed with all updates due to the modification.
Modify Cubic Segment

Delete Point
Add Point
Move Point
Change Tangents

Note that the Delete Point option is only available if the cubic segment has 3 or more points.

Delete Point

To delete a point from a cubic segment, select Delete Point from the current menu. The following message will be displayed in the lower left-hand corner:

Select Point to Delete

Use the mouse to select the point to delete. The following menu will be displayed:

Delete Selected Point?
OK CANCEL

To delete the point, select OK from the current menu. The profile curve segments should now be displayed with all updates due to the modification.

Add Point

To add a point to the cubic segment, select Add Point from the current menu. The following menu options should now be displayed:

End Point
Between Points
DONE

End Point

To add a point to the end of the cubic segment, select End Point from the current menu. The following message will be displayed in the lower left-hand corner:

Select End Point

Use the mouse to select the endpoint of the cubic segment to indicate to which end of the segment the point is to be added. Depending on the current input method (see INPUT), one of the following should appear on the screen:

Keyboard Input

A menu with the following should appear:

Enter X: 0.0
Enter Y: 0.0
Enter Z: 0.0
DONE CANCEL

Enter the XYZ values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value. Continue this process until all XYZ values are set to their desired values.

Select DONE when finished entering all XYZ values (note that the default for each value is always zero). The profile curve segments should now be displayed with all updates due to the added cubic segment point.
Grid Input

The following message should appear at the bottom left-hand corner:

Select Point from the Grid

Use the mouse to select the desired point from the grid. The profile curve segments should now be displayed with all updates due to the added cubic segment point.

Between Points

To add a point between two other points of a cubic segment, select Between Points from the current menu. The following message will be displayed in the lower left-hand corner:

Select 1st Point

Use the mouse to select one of the points between which the new point is to be added (note: the order in which the points is selected is not important).

The following message will be displayed in the lower left-hand corner:

Select 2nd Point

Use the mouse to select the other of the two points between which the new point is to be added. The new point is created following the same procedure outlined in End Point (see above). The profile curve segments should now be displayed with all updates due to the added cubic segment point.
Move Point

To move a point of a cubic segment to a new location, select Move Point from the current menu. Follow the procedures described in the Move Point section under Modify Linear Segment. The profile curve segments should now be displayed with all updates due to the added cubic segment point.

Change Tangents

To change the values of the input tangents (slopes) of a cubic segment, select Change Tangents from the current menu. The following message will be displayed in the lower left-hand corner:

Select Point to Modify

Use the mouse to select the point for which the input tangent values (slopes) are to be changed. Depending on the current input method (see INPUT), one of the following should appear on the screen:

Keyboard Input

A menu with the following should appear:

Enter DX: X.X
Enter DY: X.X
Enter DZ: X.X

DONE            CANCEL

The X's represent the current input tangent values of the selected point. Change the input tangent values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the
menu should now display the entered value. Continue this process until all input tangent values are set to their desired values.

Select DONE when finished entering all input tangent values. The profile curve segments should now be displayed with all updates due to the modified cubic segment point.

**Grid Input**

The following message should appear at the lower left-hand corner:

Select Tangent from the Grid

Use the mouse to select the desired point from the grid. The input tangent values of the point are calculated from the slope of the line between the cubic segment point and the point selected from the grid. The profile curve segments should now be displayed with all updates due to the modified cubic segment point.

**Move Segment**

To move a profile curve segment, select Move Segment from the current menu. The following message will be displayed in the lower left-hand corner:

Select Segment to Move

Use the mouse to select the segment to be modified. The segment will be highlighted and the following menu options should now appear:

Point to Point
Enter Amount
CANCEL

*Move Segment by Reference Point*

To move the segment by using a reference point from the segment, select the Point to Point option from the current menu. The segment points should now be displayed and the following message will be displayed in the lower left-hand corner:

**Select Reference Point from the Segment**

Use the mouse to select a reference point from the segment to be moved. If no grid currently exists, the Create Grid menu will appear and a grid must be created (see GRID). The following message should now be displayed in the lower left-hand corner:

**Select Reference Point from the Grid**

Use the mouse to select a reference point from the grid. All of the segment points will be moved by the difference between this point and the segment reference point. The profile curve segments should now be displayed with all updates due to the moved segment.

*Move Segment by Amount*

To move a segment of a profile curve by entering the amount by which the segment is to be moved, select Enter Amount from the current menu. The following menu should now appear:

- Enter X-delta: 0.0
- Enter Y-delta: 0.0
- Enter Z-delta: 0.0

**DONE**
Enter the amount by which the segment is to be moved by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the profile curve segments should now be displayed with all updates due to the moved segment. Enter another delta amount or select DONE to finish moving the segment.

Scale Segment

To scale a segment of a profile curve, select Scale Segment from the current menu. The following message will be displayed in the lower left-hand corner:

Select Segment to Scale

Use the mouse to select the segment to be modified. The segment will be highlighted and the following menu options should now appear:

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>X-scale</td>
<td>0.0</td>
</tr>
<tr>
<td>Y-scale</td>
<td>0.0</td>
</tr>
<tr>
<td>Z-scale</td>
<td>0.0</td>
</tr>
<tr>
<td>XYZ-scale</td>
<td>0.0</td>
</tr>
</tbody>
</table>

DONE

Enter the amount by which the segment is to be scaled by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the profile curve segments should now be displayed with all updates due to the scaled segment. Enter another scaling amount or select DONE to finish scaling the segment.
Create Cross-Sections

The cross-sections of a component may be created once all necessary profile curves have been created. The cross-sections may be created individually or automatically from a previously defined cross-section, referred to as the scaling cross-section.

Cross-Section spacing methods:

- Manual Spacing: user specifies cross-section depth for each cross-section.
- Linear Spacing: user specifies initial cross-section depth and the distance between each cross-section.
- Geometric Spacing: user specifies initial cross-section depth, geometric spacing factor, and initial distance between first two cross-sections. Distance between other cross-sections equals the distance between the two previous cross-sections multiplied by the geometric spacing factor.

The cross-section spacing method is used to determine the depth of the cross-section to be created (x-value of cross-section points).

To create a cross-section, the current menu must be the Create <component> Component menu (<component> is the component type, Rule-Based Fuselage or Spine and Cross-Sections). The last 3 options of the menu should be as follows:

Create Cross-Sections
Auto Cross-Sections
Modify Cross-Sections
DONE

Note that the Modify Cross-Sections option is not available if no cross-sections have been created.
Create Individual Cross-Sections

To create individual cross-sections of the component, select Create Cross-Sections from the current menu. The following menu options should appear:

Simple Cross-Section
Piecewise Cross-Section
Change Symmetry
Change Spacing
DONE

Create Simple Cross-Section

To create a simple cross-section, select Simple Cross-Section from the current menu. If the cross-section to be created is the first cross-section of the component, the following menu options should appear:

Manual Spacing
Linear Spacing
Geometric Spacing
CANCEL

Select the cross-section spacing method from the current menu. Based on the current cross-section and the spacing method, one of the following should occur:

Manual Spacing - any cross-section:

The following menu should appear:

Enter Cross-Section Depth: 0.0
CANCEL

Linear Spacing - 1st cross-section:

The following menu should appear:

Enter Cross-Section Depth: 0.0

CANCEL

If the depth is entered properly, the following menu should now appear:

Enter Linear Spacing: 0.0

CANCEL

Geometric Spacing - 1st cross-section:

The following menu should appear:

Enter Cross-Section Depth: 0.0

CANCEL

If the depth is entered properly, the following menu should now appear:

Enter Geometric Spacing: 0.0

CANCEL

If the geometric spacing factor is entered properly, the following menu should now appear:

Enter Initial Spacing: 0.0

CANCEL
If all prompted values are entered properly, the following menu options should appear:

Rectangle
Circle
Ellipse
CANCEL

**Create Rectangular Cross-Section**

To create a rectangular cross-section, select Rectangle from the current menu. Half of a rectangle should be displayed and the following menu should now appear:

Width: 2.0
Height: 2.0
DONE

Enter the cross-section parameter values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value and the cross-section half should be redrawn with the new parameters. Continue this process until all cross-section parameter values are set to their desired values.

Select DONE when finished entering all cross-section parameter values. The full cross-section should now be displayed in the 4-view display oriented on the component.
Create Circular Cross-Section

To create a circular cross-section, select Circle from the current menu. Half of a circle should be displayed and the following menu should now appear:

Radius: 2.0
DONE

Enter the cross-section parameter values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the value was entered properly, the menu should now display the entered value and the cross-section half should be redrawn with the new parameters. Continue this process until all cross-section parameter values are set to their desired values.

Select DONE when finished entering all cross-section parameter values. The full cross-section should now be displayed in the 4-view display oriented on the component.

Create Elliptical Cross-Section

To create an elliptical cross-section, select Ellipse from the current menu. Half of an ellipse should be displayed and the following menu should now appear:

Horizontal Radius: 2.0
Vertical Radius: 3.0
DONE

Enter the cross-section parameter values by selecting the value with the mouse or by tabbing to the value with the keyboard and then enter the desired value. Hit enter after entering the value. If the
value was entered properly, the menu should now display the entered value and the cross-section half should be redrawn with the new parameters. Continue this process until all cross-section parameter values are set to their desired values.

Select DONE when finished entering all cross-section parameter values. The full cross-section should now be displayed in the 4-view display oriented on the component.

Create Piecewise Cross-Section

To create a piecewise cross-section, select Piecewise Cross-Section from the current menu. If the cross-section to be created is the first cross-section of the component, the following menu options should appear:

Y-Symmetry
Z-Symmetry
YZ-Symmetry
No Symmetry
CANCEL

Note: if the component for which the cross-section to be created is Rule-Based Fuselage type, only Z-Symmetry and YZ-Symmetry are available.

Cross-Sections are defined in the YZ-plane. It is important to make full use of the cross-section symmetry. It is also important that the cross-section be created around the origin to achieve the correct symmetry (if any) and enable the cross-section to be properly oriented. For this reason, it is suggested that a Center Grid be used (see GRID). Note that a grid may be displayed and keyboard input can still be used (see INPUT).
Select the cross-section symmetry from the current menu. The following menu should now appear:

   Manual Spacing
   Linear Spacing
   Geometric Spacing
   CANCEL

The spacing options and corresponding menu options and input are described above in the section Create Simple Cross-Section. If the cross-section symmetry and spacing are properly entered, the following menu options should now appear:

   Linear Segment
   Cubic Segment
   DONE

Create the piecewise cross-sections using the same procedure outlined for creating profile curve segments (see Create Spine Curve), keeping in mind the cross-section symmetry. Once all of the cross-section segments are created, select DONE from the Create Segment menu. The full cross-section should now be displayed in the 4-view display oriented on the component.

**Auto Cross-Sections**

To create a specified number of cross-sections from a previously defined cross-section, referred to as the scaling cross-section, select Auto Cross-Sections from the current menu. The following menu should now appear:

   Number of Cross-Sections: 1
   CANCEL
Enter the number of cross-sections to be created. If no cross-sections have been created, the following menu options should appear:

Simple Cross-Section
Piecewise Cross-Section
Change Symmetry
Change Spacing
DONE

Since there are no cross-sections currently defined, the scaling cross-section must first be created. To create the scaling cross-section, follow the procedures outlined above in the Create Cross-Sections section.

If any cross-sections have already been created, the following message should appear in the lower left-hand corner:

Select Scaling Cross-Section

Use the mouse to select the cross-section which will be used to create the auto cross-sections (i.e. select the scaling cross-section).

If the component for which the cross-sections are to be created is of Spine and Cross-Section type, the scaling factor must be entered for each cross-section. The scaling factor is the amount by which the scaling cross-section is to be scaled to create each individual cross-section.

If the cross-section spacing method is manual, the depth for each cross-section must be entered. As each cross-section is created, the cross-section is drawn oriented to the component.
Modify Cross-Sections

To modify the cross-sections of a component, select Modify Cross-Sections from the current menu. Only the component cross-sections should now be displayed and the following menu options should appear:

- Modify Cross-Section
- Delete Cross-Section
- Delete All Cross-Sections

Modify Cross-Section

To modify the parameters of an individual cross-section, select Modify Cross-Section from the current menu. Based on the cross-section type the following will occur:

Modify Simple Cross-Section

Based on the type of simple cross-section, the menu that is used to create that type of simple cross-section is displayed. To modify the cross-section parameters, follow the procedures outlined in the section Create <simple cross-section type> Cross-Section (<simple cross-section type> - rectangle, circle, or ellipse).

Modify Piecewise Cross-Section

The following menu options should appear:

- Delete Segment
- Add Segment
- Modify Segment
- Move Segment
- Scale Segment
DONE

The menu options work exactly the same as for modifying the segments of a profile curve. To modify the piecewise cross-section segments, refer to the Modify Profile Curve Segments section.

Modify Component

To modify a component, the current menu must be the Edit Components menu (select EDIT from the main menu). The following menu items should be displayed:

Create RBF Component
Create Spine Component
Modify Component
Delete Component

DONE

Note that the first two options are not available. To modify the component, select Modify Componènt from the Edit Components menu. The menu options for creating the component should now be displayed.
Delete Component

To delete a component, the current menu must be the Edit Components menu (select EDIT from the main menu). Select Delete Component from the Edit Components menu. The component is highlighted and the following message is displayed in the lower left-hand corner:

Delete Component?

OK CANCEL

To delete the selected component, select OK. The component disappears.

Viewing Options

The system provides options which allow the user to change the orientation of the view in which component geometry is displayed. These options may be accessed from any open control pop-up menu by double-clicking the middle button of the mouse outside of the menu in the view for which the orientation is to be changed. The following menu options should appear:

Options

None
Zoom
Spin
Scale
Translate
Rotate
Cancel
None

Selecting None from the Options menu turns off the current view orientation operation.

Zoom

To zoom in on the component geometry of the selected view, select Zoom from the Options menu and drag the mouse to create a box enclosing the geometry to be zoomed. The view should zoom in to display only the enclosed geometry.

Spin

To spin the component geometry of the selected view (2D rotation), select Spin from the Options menu and select the geometry to spin. Drag the mouse to spin the selected geometry. Release the mouse button when finished spinning the geometry.

Scale

The Scale option allows the user to zoom in and out of the selected view by dragging the mouse within the selected view.
Translate

To move the component geometry of the selected view, select Translate from the Options menu and select the geometry to translate. Drag the mouse to move the selected geometry. Release the mouse button when finished translating the geometry.

Rotate

To rotate the geometry of the selected view (3D rotation), select Rotate from the Options menu. Select the geometry and drag the mouse to rotate the geometry.

To reset the selected view, select AUTOCENTER from the main menu. To return to the pop-up menu from which the Options menu was displayed by double-clicking, select MENU from the main menu.

It is important to note that the Options menu options do not change the xyz values of the geometry. The Options menu options change the view orientation in which the geometry is displayed by changing the viewing reference of the user, giving the appearance that the geometry has been moved.

ACSYNT User Guide

This section describes how to create and modify Rule-Based Fuselage and Spine and Cross-Section components within ACSYNT. For a more detailed description of how to use ACSYNT,
the reader should refer to [ACSY93].

Create Component

The current menu should be the ACSYNT main menu. The following menu options should be displayed:

    GEOMETRY
    .
    FILE

To create a new component, select GEOMETRY from the current menu. The following menu options should now be displayed:

    RETURN
    COMPONENT
    .

Select COMPONENT from the current menu. The following menu options should now be displayed:

    RETURN
    ADD
    MODIFY
    DELETE
    COPY
    .

Select ADD from the current menu. A template containing the list of available components should appear. Select the appropriate component to create (Rule-Based Fuselage or Spine and Cross-Section). Follow the procedures outlined in the previous sections of this user guide to create the
Modify Component

The current menu should be the component menu and the following menu options should be displayed:

```
RETURN
ADD
MODIFY
DELETE
COPY
```

Select MODIFY from the current menu. Select the component to be modified with the mouse. The following menu options should now appear:

```
RETURN
COMP SHAPE
COMP LOCATION
GENERAL PARAMS
```

Modify System Component Data

To modify the component data within the system, select COMP SHAPE from the current menu. Modify the component following the procedures outlined in the previous sections of this user guide.

Modify Component Location

To modify the xyz location and orientation of the component, select COMP LOCATION from the
current menu. A template will appear which displays the current xyz translations and rotations. Use the mouse to select the desired translation or rotation to change and enter the new value.

**Modify ACSYNT Component Parameters**

To modify the surface rendering, local and global symmetry, and line attributes of the component, select GENERAL PARAMS from the current menu. A template will appear which displays the component parameters which may be modified. Note that the number of cross-sections and number of points per cross-section cannot be modified for a Rule-Based Fuselage or Spine and Cross-Section component.

**Save Component**

To save the geometric data, select FILE from the main menu. Select SAVE GEOMETRY from the current menu and enter the filename to be saved.

**Load Component**

To retrieve an ACSYNT geometry file, select FILE from the main menu and enter the filename to be retrieved.
Appendix B. Detailed Class Descriptions
Overview

Included are detailed descriptions of the variables and functions contained within each class of the system. These descriptions are provided as a reference for any persons who wish to modify the classes. Such individuals must have knowledge of both object-oriented programming and the C++ programming language as well as an understanding of standard PHIGS. Also, familiarity with the “motif-like” interface framework is required [Woya92].
Component Class

Class Description:
Base class for the Rule-Based Fuselage and Spine and Cross-Section derived classes.

Class Inheritance:
None.

Private Variables:

PHIGS_Structure_ID*  PHIGS_struct_id[8]
Array of PHIGS structure ids for the given component. 8 structure ids are maintained to display the component in all 8 PHIGS views at once - it is necessary to maintain 8 copies of the PHIGS structure because a PHIGS structure can be associated with only 1 view at a time. The type PHIGS_Structure_ID is a standard PHIGS structure type developed by Scott Woyak [Woya92].

PHIGS_Structure_ID*  surface_structure[8]
Array of PHIGS structure ids containing the cross-section surface structures. These surface structures do not represent the actual component surface. Instead, the structures indicate the cross-section data points which are sent to ACSYNT to create the component surface model.

int  component_number
ACSYNT component identifier.

int  pick_id
PHIGS pick id of the component.

int  component_type
Type of component:
1 = Rule-Based Fuselage
2 = Spine and Cross-Section

int  component_definition
Flag indicating the status of the component:
0 = not defined
1 = defined

int  highlight_flag
Flag which indicates whether or not the component PHIGS structure is to be displayed with highlighting color:
0 = do not highlight component
1 = highlight component

Component*  previous
Pointer to the previous component in the components linked list.

Component*  next
1 Pointer to the next component in the components inked list.
float* surface_points
Dynamically allocated array of cross-section points which are sent to ACSYNT to create the component surface model and to create the PHIGS surface_structure (see above).

int symmetry_flag
Flag which indicates if the user wants to change the current symmetry of the component cross-sections:
0 = do not change current cross-section symmetry
1 = change current cross-section symmetry

int spacing_flag
Flag which indicates if the user wants to change the current spacing of the component cross-sections:
0 = do not change current cross-section spacing
1 = change current cross-section spacing

int cross_section_symmetry
Component cross-section symmetry:
1 = symmetry about the Y-axis.
2 = symmetry about the Z-axis.
3 = symmetry about the Y and Z axes.
4 = asymmetric

int cross_section_spacing
The component cross-section spacing method is used by the system to set the current cross-section depth:
1 = manual spacing
2 = linear spacing
3 = geometric spacing

int number_of_cross_sections
Number of cross-sections that the component has.

Cross_Section* first_cross_section
Pointer to the first component cross-section in the cross-sections linked list.

Cross_Section* last_cross_section
Pointer to the last component cross-section in the cross-sections linked list.

Protected Functions:

void set_cross_section_parameters(
   Input_Display_Manager* manager,
   Cross_Section* cross_section,
   Cross_Section* scaling_cross_section )

Function Description:
Sets the cross-section parameters of the new cross-section to be created equal to the parameters of the scaling cross-section.
**Argument Description:**
manager

cross_section

scaling_cross_section

pointer to the Input/Display Manager class.
cross-section to be created by scaling a
previously defined cross-section, referred to
as the scaling cross-section.
cross-section to be scaled to create the new
cross-section.

```c
void auto_cross_sections( Input_Display_Manager* manager )
```

**Function Description:**
Allows the user to create a specified number of cross-sections from a
previously defined cross-section, referred to as the scaling cross-section.

**Argument Description:**
manager

pointer to the Input/Display Manager class.

```c
void change_symmetry( Input_Display_Manager* manager )
```

**Function Description:**
Allows the user to change the current cross-section symmetry.

**Argument Description:**
manager

pointer to the Input/Display Manager class.

```c
void change_spacing( Input_Display_Manager* manager )
```

**Function Description:**
Allows the user to change the current cross-section spacing.

**Argument Description:**
manager

pointer to the Input/Display Manager class.

```c
void modify_cross_sections( Input_Display_Manager* manager )
```

**Function Description:**
Allows the user to add/delete cross-sections, modify a selected
cross-section, or delete all of the cross-sections.

**Argument Description:**
manager

pointer to the Input/Display Manager class.

```c
void delete_cross_sections()
```

**Function Description:**
Deletes all of the cross-sections of a component.

**Argument Description:**
None.

```c
void delete_one_cross_section( Input_Display_Manager* manager )
```

**Function Description:**
Allows the user to delete a selected cross-section of a component.

**Argument Description:**
manager

pointer to the Input/Display Manager class.
void update_delete_cross_section( Cross_Section* cross_section )

Function Description:
Updates the cross-sections linked list after a cross-section has been deleted and reduces the number of cross-sections by one.

Argument Description:
cross_section which was deleted.

void add_cross_section( Input_Display_Manager* manager, Cross_Section* cross_section )

Function Description:
Updates the cross-sections linked list by adding the new cross-section to the list and increases the number of cross-sections by one. The position of the new cross-section in the linked list is determined by comparing the cross-section depths.

Argument Description:
manager pointer to the Input/Display Manager class.
cross_section newly created cross-section.

void link_new_cross_section( Cross_Section* cross_section )

Function Description:
Updates the cross-sections linked list by adding the new cross-section to the linked list if the new cross-section is between two previously defined cross-sections.

Argument Description:
cross_section newly created cross-section.

void modify_cross_section( Input_Display_Manager* manager )

Function Description:
Allows the user to modify an individual cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.

Cross_Section* get_cross_section_to_modify( Input_Display_Manager* manager, char message[] )

Function Description:
Displays a prompt to the user and returns a pointer to the cross-section selected by the user.

Argument Description:
manager pointer to the Input/Display Manager class.
message character array containing a prompt to the user.

void display_cross_sections( Input_Display_Manager* manager )

Function Description:
Displays the PHIGS structures of all of the cross-sections of a component.
autocentered in the current view.

**Argument Description:**
manager pointer to the Input/Display Manager class.

**void** remove_cross_sections( Input_Display_Manager* manager )

**Function Description:**
Removes the PHIGS structures of all of the cross-sections of a component from the current view.

**Argument Description:**
manager pointer to the Input/Display Manager class.

**void** delete_cross_sections_structures()

**Function Description:**
Deletes the PHIGS structures of all of the cross-sections of a component.

**Argument Description:**
None.

**void** highlight_cross_sections( int highlighting )

**Function Description:**
Draws the PHIGS structures of all of the component cross-sections with or without highlighting depending on the highlighting flag. Used to highlight a component.

**Argument Description:**
highlighting flag which indicates whether or not the component cross-sections are to be highlighted.

**Public Functions:**

**void** set_component_pick_id()

**Function Description:**
Sets the pick id of the component PHIGS structure using the interface framework [Woya92].

**Argument Description:**
None.

**void** display_component( Input_Display_Manager* manager, int autocenter_flag )

**Function Description:**
Displays the component PHIGS structure in the current view. The component is autocentered in the current view if the autocenter_flag is AUTOCENTER. Typically, if one component is displayed, the component is autocentered. If more than one component is displayed, the component is not autocentered since all of the components are autocentered together in this case.

**Argument Description:**
manager pointer to the Input/Display Manager class.
void remove_component(Input_Display_Manager* manager)

**Function Description:**
Removes the component PHIGS structure from the current view.

**Argument Description:**
manager: pointer to the Input/Display Manager class.

void autocenter_component(Input_Display_Manager* manager)

**Function Description:**
Autocenters the component PHIGS structure within the current view by determining the minimum and maximum point values of the component by checking the minimum and maximum point values of the component profile curve(s) and any existing cross-sections.

**Argument Description:**
manager: pointer to the Input/Display Manager class.

void link_cross_sections(Cross_Section* previous, Cross_Section* current)

**Function Description:**
Links together the previous and current cross-sections by setting the previous pointer of the current cross-section equal to the previous cross-section and setting the next pointer of the previous cross-section equal to the current cross-section.

**Argument Description:**
previous: pointer to the cross-section which is to become the previous cross-section of the current cross-section.
current: pointer to the current cross-section.

void copy_cross_sections(Component* component)

**Function Description:**
Creates the cross-sections of a new component by duplicating the cross-sections of a previously defined component.

**Argument Description:**
component: pointer to the new component to be created.

void draw_cross_sections(Input_Display_Manager* manager)

**Function Description:**
Draws the PHIGS structures of all of the cross-sections of a component.

**Argument Description:**
manager: pointer to the Input/Display Manager class.
void create_surface( Input_Display_Manager* manager, int display_flag )

Function Description:
Creates the cross-section surfaces of a component (i.e. loads all of the cross-section points into an array of points).

Argument Description:
manager   pointer to the Input/Display Manager class.
display_flag   flag indicating whether or not to display the cross-section surface structures (see above):
          0 = do not display
          1 = display

void create_cross_section_surface(Cross_Section* cross_section, int xsect )

Function Description:
Creates the cross-section surface of an individual cross-section (i.e. loads all of the cross-section points into an array of points) and sends the array of surface points to ACSYNT to create the component surface model.

Argument Description:
cross_section   pointer to the cross-section for which the surface is to be created.
xsect   id indicating for which cross-section the surface is to be created (1st, 2nd, etc.).

Component Manager Class

Class Description:
Maintains a linked list of components created within the system and provides functions to create and modify Rule-Based Fuselage and Spine and Cross-Section components.

Class Inheritance:
None.

Private Variables:

int component_number   ACSYNT component identifier.
int number_of_components   Number of components currently in the components linked list.
Component* first_component   Pointer to the first component in the components linked list.
Component* last_component   Pointer to the last component in the components linked list.
Component* current_component     Pointer to the component which is currently being created/modified within the system.

Input_Display_Manager* input_display_manager
Pointer to the Input/Display Manager class. Provides access to the viewing and menu functions contained within the Input/Display Manager class by inheritance from the View Manager and Menu Manager classes. This pointer is passed as an argument to all functions which require access to these functions.

Private Functions:

void modify_components()

Function Description:
Allows the user to create a new component. If any components are currently defined, the user may delete or modify a selected component or delete all of the components.

Argument Description:
None.

void create_component( int component_type )

Function Description:
Calls the constructor of the component to be created based on the component type and creates the component. Updates the component manager.

Argument Description:
component_type type of component to be created.

void link_components
( Component* previous, Component* current )

Function Description:
Links together the previous and current components by setting the previous pointer of the current component equal to the previous component and setting the next pointer of the previous component equal to the current component.

Argument Description:
previous pointer to the component which is to become the previous component of the current component.
current pointer to the current component.

void delete_components()

Function Description:
Deletes all existing components. Called when the Component Manager destructor is called.

Argument Description:
None.
void delete_one_component()

Function Description:
Allows the user to delete a selected component.

Argument Description:
None.

void update_component_manager_delete_component ( Component* component )

Function Description:
Updates the components linked list after a component has been deleted and reduces the total number of components by one.

Argument Description:
    component    deleted component.

Public Functions:

Component_Manager()

Function Description:
Constructor for the Component Manager class. Initializes the Input/Display Manager class which, in turn, initializes the View Manager and Menu Manager classes.

Argument Description:
None.

void process_main_menu()

Function Description:
Interface main menu. Program execution begins with the main menu process loop. Program execution ends when this loop is exited. Processes the main menu options.

Argument_Description:
None.

void update_component_manager( Component* component )

Function Description:
If the component was properly defined (i.e. all profile curve(s) created), the new component is added to the components linked list and the number of components increased by one.

Argument Description:
    component    pointer to the component that was just created.

void copy_component( int comp_number, int new_comp_number )

Function Description:
Creates a new component by copying a previously defined component.
Argument Description:
comp_number    ACSYNT component number of the previously
defined component to be copied.
new_comp_number ACSYNT component number of the new component
to be created.

void draw_components()

Function Description:
Creates the PHIGS structures of all existing components by creating the
profile curve(s) PHIGS structures and any existing cross-sections PHIGS
structures of each component.

Argument Description:
None.

void create_surfaces()

Function Description:
Creates the cross-sections surfaces for all existing components.

Argument Description:
None.

void save_acsynt_component( char* file_name )

Function Description:
Appends the system geometric data of a component to the ACSYNT
gallery file.

Argument Description:
file_name    character pointer array containing the name of the ACSYNT
gallery file to which the data is to be appended.

void load_acsynt_component( char* file_name )

Function Description:
From within ACSYNT, this function reads the system geometric data
which was appended to the ACSYNT geometry file and creates the
component objects from the data.

Argument Description:
file_name    character pointer array containing the name of the ACSYNT
geometry file from which the data is to be read.

Cross-Section Class

Class Description:
Base class for the Simple Cross-Section and Piecewise Cross-Section derived classes.
Class Inheritance:
None.

Private Variables:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>pick_id Cross-section PHIGS structure pick id.</td>
</tr>
<tr>
<td>int</td>
<td>cross_section_type Type of cross-section: 1 = simple cross-section, 2 = piecewise cross-section</td>
</tr>
<tr>
<td>int</td>
<td>highlight_flag Flag which indicates whether or not the cross-section PHIGS structure is to be displayed with highlighting color: 0 = do not highlight cross-section, 1 = highlight cross-section</td>
</tr>
<tr>
<td>int</td>
<td>cross_section_definition Flag which indicates the status of a cross-section which is being created: 0 = not defined, 1 = defined, 2 = ready to define</td>
</tr>
<tr>
<td>Cross_Section*</td>
<td>previous Pointer to the previous cross-section in the cross-sections linked list.</td>
</tr>
<tr>
<td>Cross_Section*</td>
<td>next Pointer to the next cross-section in the cross-sections linked list.</td>
</tr>
<tr>
<td>int</td>
<td>cross_section_symmetry Component cross-section symmetry: 1 = symmetry about the Y-axis, 2 = symmetry about the Z-axis, 3 = symmetry about the Y and Z axes, 4 = asymmetric</td>
</tr>
<tr>
<td>int</td>
<td>cross_section_spacing The cross-section spacing method is used by the system to set the current cross-section depth: 1 = manual spacing, 2 = linear spacing, 3 = geometric spacing</td>
</tr>
<tr>
<td>int</td>
<td>symmetry_flag Flag which indicates if the user wants to change the current symmetry of the component cross-sections: 0 = do not change current cross-section symmetry, 1 = change current cross-section symmetry</td>
</tr>
<tr>
<td>int</td>
<td>spacing_flag Flag which indicates if the user wants to change the</td>
</tr>
</tbody>
</table>
current spacing of the component cross-sections:
\[ 0 = \text{do not change current cross-section spacing} \]
\[ 1 = \text{change current cross-section spacing} \]

float \text{cross_section_linear_spacing} \quad \text{Distance between adjacent cross-sections which are linearly spaced.}

float \text{cross_section_geometric_spacing} \quad \text{Factor by which the distance between the two previous cross-sections is multiplied to determine the depth of a new cross-section (for geometric spacing).}

float \text{cross_section_initial_spacing} \quad \text{The distance between the first two adjacent cross-sections using geometric spacing.}

float \text{cross_section_depth} \quad \text{Cross-section depth which represents the x-value of all points of the cross-section.}

Cross\_Section* \quad \text{scale_pointer} \quad \text{Pointer to the scaling cross-section.}

float \text{scale_factor} \quad \text{Factor by which a cross-section, referred to as the scaling cross-section, is to be scaled in order to create a new cross-section.}

PHIGS\_Structure\_ID* \quad \text{YZ\_cross\_section\_structure\_id[8]} \quad \text{PHIGS structure id for a non-oriented cross-section. Used to display the previous cross-section, if it exists, of a new cross-section to be created.}

Protected Variables:

Point* \text{crown_point} \quad \text{Pointer to the point at which the cross-section intersects the crown profile curve (for Rule-Based Fuselage components only).}

Point* \text{keel_point} \quad \text{Pointer to the point at which the cross-section intersects the keel profile curve (for Rule-Based Fuselage components only).}

Point* \text{plan_point} \quad \text{Pointer to the point at which the cross-section intersects the plan profile curve (for Rule-Based Fuselage components only).}

Point* \text{orient_point} \quad \text{Pointer to the point about which a cross-section of a Spine and Cross-Section component is oriented normal to the spine profile curve.}
float orientation_matrix[4][4] matrix used to position a cross-section of a Rule-Based Fuselage component to the proper depth along the fuselage or to orient a cross-section of a Spine and Cross-Section component normal to the spine profile curve.

Segment* first_surface_segment Pointer to the first segment of a cross-section which contains the points data required to generate component surface models from the cross-section.

Segment* last_surface_segment Pointer to the last segment of a cross-section which contains the points data required to generate component surface models from the cross-section.

int number_of_cross_section_points Number of points defined on the cross-section used to generate component surface models.

Protected Functions:

void manual_spacing( Input_Display_Manager* manager )

Function Description:
Sets the cross-section depth manually by prompting the user for each cross-section depth.

Argument Description:
manager pointer to the Input/Display Manager class.

void linear_spacing( Input_Display_Manager* manager )

Function Description:
Sets the cross-section depth linearly. Initially, the user must enter a depth for the first cross-section (or if the current cross-section spacing method has just been changed to linear spacing) and a linear spacing value. The remaining cross-section depths are evenly spaced by that value.

Argument Description:
manager pointer to the Input/Display Manager class.

void geometric_spacing( Input_Display_Manager* manager )

Function Description:
Sets the cross-section depth geometrically. Initially, the user must enter a depth for the first cross-section (or if the current cross-section spacing method has just been changed to geometric spacing), the depth for the next cross-section, and a geometric spacing factor. The remaining cross-section depths are calculated by multiplying the distance between the two previous cross-sections by the spacing factor.

Argument Description:
manager pointer to the Input/Display Manager class.
void  **mirror_segments** ( float matrix[4][4] )

**Function Description:**
Determines the points for the segments of a cross-section that must be mirrored based on the cross-section symmetry.

**Argument Description:**
matrix  
4 x 4 PHIGS transformation matrix used to transform the segment points.

void  **set_number_of_cross_section_points()**

**Function Description:**
Determines the number of cross-section points that are along the cross-section required to generate the cross-section surface geometry.

**Argument Description:**
None.

void  **delete_surface_segments()**

**Function Description:**
Deletes the cross-section surface segments and releases the allocated memory. Called when the destructor of the cross-section is called.

**Argument Description:**
None.

**Public Functions:**

void  **set_cross_section_depth** ( Input_Display_Manager* manager )

**Function Description:**
Sets the cross-section depth based on the current cross-section spacing method.

**Argument Description:**
manager  
pointer to the Input/Display Manager class.

void  **display_cross_section_YZ** ( Input_Display_Manager* manager )

**Function Description:**
Creates and displays the PHIGS structure of a non-oriented cross-section autocentered in the current view.

**Argument Description:**
manager  
pointer to the Input/Display Manager class.

void  **remove_cross_section_YZ()**

**Function Description:**
Removes the PHIGS structure of a non-oriented cross-section from the current view.
Argument Description:
None.

void display_cross_section( Input_Display_Manager* manager )

Function Description:
Displays the PHIGS structure of an oriented cross-section autocentered in the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void remove_cross_section( Input_Display_Manager* manager )

Function Description:
Removes the PHIGS structure of an oriented cross-section autocentered from the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void delete_cross_section_structure()

Function Description:
Deletes the PHIGS structure of an oriented cross-section.

Argument Description:
None.

void set_cross_section_pick_id()

Function Description:
Sets the pick id of the cross-section PHIGS structure and adds it to the cross-section class name (for use with the pick filter).

Argument Description:
None.

void autocenter_cross_section( Input_Display_Manager* manager )

Function Description:
Autocenters the cross-section PHIGS structure by determining the minimum and maximum point values of the cross-section based on the cross-section type.

Argument Description:
manager pointer to the Input/Display Manager class.

Crown Curve Class

Class Description:
Contains functions to create and modify the crown profile curve of a Rule-Based Fuselage
component.

Class Inheritance:
RBF Profile Curve Manager class.

Private Functions:

void draw_crown_curve_auto_points(Input_Display_Manager* manager)

Function Description:
Creates the PHIGS structure containing the points automatically generated
by the crown curve parameters if the parameters were entered by the user.

Argument Description:
manager pointer to the Input/Display Manager class.

void check_nose_crown_angle()

Function Description:
Ensures that the nose crown angle input by the user is between 0 and 90
degrees.

Argument Description:
None.

Public Functions:

Crown_Curve()

Function Description:
Constructor. Initializes crown curve parameters.

Argument Description:
None.

void set_rbf_parameters(Profile_Curve* keel_curve, Profile_Curve* plan_curve)

Function Description:
Sets the Rule-Based Fuselage parameters of the crown profile curve equal
to those of the keel curve or plan curve if either exists.

Argument Description:
keel_curve pointer to the keel profile curve.
plan_curve pointer to the plan profile curve.

void create_profile_curve(Input_Display_Manager* manager,
Component* component)

Function Description:
Allows user to enter profile curve parameters for the crown curve. Also
used to modify the profile curve parameters of the crown curve.
Argument Description:
manager pointer to the Input/Display Manager class.
component pointer to the Rule-Based Fuselage component for which the crown curve is to be created.

void set_last_point_rbf_first_linear_segment
(Input_Display_Manager* manager, Point* point)

Function Description:
If the first segment of the crown profile curve is linear, this function automatically sets the last point of the segment based on the crown nose angle if the angle is non-zero and the user has entered the crown profile curve parameters.

Argument Description:
manager pointer to the Input/Display Manager class.
point pointer to the first point of the linear segment.

void set_last_rbf_linear_segment_point
(Input_Display_Manager* manager,
Segment* segment, Point* point)

Function Description:
This function checks to see if the last point of the linear segment of the crown profile curve has an x-value equal to the x-value of the end of the fuselage. If so, the point is automatically adjusted based on the aft offset if the user entered the crown profile curve parameters.

Argument Description:
manager pointer to the Input/Display Manager class.
segment pointer to the last linear segment of the crown profile curve.
point pointer to the last point of the linear segment.

void set_first_point_rbf_first_cubic_segment
(Input_Display_Manager* manager, Point* point)

Function Description:
If the first segment of the crown profile curve is cubic, this function automatically sets the input tangent values (slopes) of the first point of the segment based on the crown nose angle if the angle is non-zero and the user has entered the crown profile curve parameters.

Argument Description:
manager pointer to the Input/Display Manager class.
point pointer to the first point of the cubic segment.

void check_last_point_cubic_segment
(Input_Display_Manager* manager, Point* point)

Function Description:
This function checks to see if the last point of the cubic segment of the crown profile curve has an x-value equal to the x-value of the end of the fuselage. If so, the point is automatically adjusted based on the aft offset if the user entered the crown profile curve parameters.

Argument Description:
manager pointer to the Input/Display Manager class.
point  pointer to the last point of the cubic segment.

void modify_curve_parameters( Input_Display_Manager* manager, Component* component )

Function Description:
Allows the user to enter the profile curve parameters of the crown profile curve. This function is used when the crown profile curve is first created and when the user modifies the crown curve parameters.

Argument Description:
manager  pointer to the Input/Display Manager class.
component  pointer to the Rule-Based Fuselage component to which the crown curve belongs.

void update_curve_segments( Input_Display_Manager* manager )

Function Description:
Automatically updates the segments of the crown profile curve after the crown curve parameters have been modified.

Argument Description:
manager  pointer to the Input/Display Manager class.

void update_curve_parameters()

Function Description:
Automatically updates the parameters of the crown profile curve after the crown curve segments have been modified.

Argument Description:
None.

Cubic Segment Class

Class Description:
Contains data and functions used to create cubic segments of profile curves and piecewise cross-sections of Rule-Based Fuselage and Spine and Cross-Section components.

Class Inheritance:
Segment class.

Private Variables:

float* cubic_points  Dynamically allocated array of intermediate cubic segment points.

int tangent_specification  Tangent specification for the cubic segment:
1 = no tangents specified
2 = all tangents specified
3 = first tangent specified
4 = last tangent specified
5 = first and last tangents specified
6 = any tangents specified

Indicates for which points of the cubic segment the user must input tangent values (slopes).

Private Functions:

void set_number_of_cubic_segment_points( Input_Display_Manager* manager )

Function Description:
Prompts the user for the number of points for the cubic segment to be created.

Argument Description:
manager pointer to the Input/Display Manager class.

void set_cubic_segment_tangent_specification
( Input_Display_Manager* manager )

Function Description:
Prompts the user for the tangent specification of the cubic segment to be created.

Argument Description:
manager pointer to the Input/Display Manager class.

void set_point_tangent_specification( Point* point, int pt )

Function Description:
Sets the tangent specification of the given point based on which point is being set and the cubic segment tangent specification. The point tangent specification is used by the create_point() function to determine whether or not the user must be prompted to enter an input tangent value (slope) for the point being created.

Argument Description:
point pointer to the point for which the point tangent specification is to be set.
pt indicates which point of the cubic segment is to be set (1st, 2nd, etc.).

void calculate_tangents() 

Function Description:
Initializes the calculation of the cubic segment tangents based on the tangent specification method. The cubic segment is broken into several cubic segment portions, each with one of the following tangent specifications: (1) no tangents specified, (2) first tangent specified, (3) last tangent specified, or (4) first and last tangents specified. The tangents are calculated for each cubic segment portion based on this tangent specification using the cubic_segment_tangents() functions (Appendix E). These functions only
allow the preceding 4 tangent specification methods, thus requiring the
original cubic segment to be broken into portions which satisfy one of these
tangent specifications.

**Argument Description:**
None.

```c
void calculate_chord_lengths()
```

**Function Description:**
Calculates the chord length between each pair of cubic segment points.
Used by the `cubic_segment_tangents()` functions which are based on a
chord length parametrization scheme (Appendix E).

**Argument Description:**
None.

```c
float set_point_chord_length( float x[], float y[], float z[] )
```

**Function Description:**
Returns the chord length between the two points given by the x, y, and z
arrays.

**Argument Description:**

- x: Array containing the x-values of the chord endpoints.
- y: Array containing the y-values of the chord endpoints.
- z: Array containing the z-values of the chord endpoints.

```c
int check_first_point_tangents( Point* point )
```

**Function Description:**
Checks to see if the first point of the cubic segment has any non-zero
tangents specified and sets the tangent specification of the first cubic
segment portion accordingly. Returns the first cubic segment portion
tangent specification.

**Argument Description:**

- point: pointer to the first point of the cubic segment.

```c
int check_tangents( Point* point )
```

**Function Description:**
Returns a flag indicating whether or not the intermediate point of the cubic
segment has any non-zero tangents specified. If so, this point becomes the
last point of the current cubic segment portion and the tangents are then
calculated for that cubic segment portion.

**Argument Description:**

- point: pointer to an inner point of the cubic segment.

```c
int check_last_point_tangents( Point* point, int spec )
```

**Function Description:**
Checks to see if the last point of the cubic segment has any non-zero
tangents specified and sets the tangent specification of the last cubic segment
portion accordingly. Returns the last cubic segment portion tangent
specification.

**Argument Description:**
- point  pointer to the last point of the cubic segment.
- spec  tangent specification of the current cubic segment portion.

```c
void cubic_tangents( Point* point, int n )
```

**Function Description:**
Determines which overloaded function to use to calculate the cubic segment portion tangents based on the number of points in the current cubic segment portion, n.

**Argument Description:**
- point  pointer to the first point of the current cubic segment portion.
- n  number of points in the current cubic segment portion.

```c
void cubic_segment_tangents( Point* point, int spec )
```

**Function Description:**
Overloaded function which calculates the tangents for a cubic segment portion which has only two points.

**Argument Description:**
- point  pointer to the first point of the current cubic segment portion.
- spec  tangent specification of the current cubic segment portion.

```c
void cubic_segment_tangents( Point* point, int N, int spec )
```

**Function Description:**
Overloaded function which calculates the tangents for a cubic segment portion which has more than two points.

**Argument Description:**
- point  pointer to the first point of the current cubic segment portion.
- N  number of points in the current cubic segment portion.
- spec  tangent specification of the current cubic segment portion.

```c
void calculate_intermediate_points()
```

**Function Description:**
Calculates the intermediate points of the cubic segment (i.e. the points between the user-specified segment points) using cubic Hermite interpolation of the cubic segment points.

**Argument Description:**
None.

```c
void calculate_points( Point* point, float u, int i )
```

**Function Description:**
Calculates the intermediate cubic segment points between the given point and the next point in the points linked list using cubic Hermite blending functions (Appendix D).

**Argument Description:**
- point  cubic segment point; the intermediate points are to be calculated.
between this point and the next point in the cubic segment points
linked list.

u    local Hermite parameter in the interval [0,1].
i    array index for the intermediate cubic segment points array.

void delete_one_point( Input_Display_Manager* manager )

Function Description:
Allows the user to delete a point of a cubic segment (cubic segment must
have at least 2 points). Segment points linked list automatically updated and
PHIGS structure updated to reflect the modification. Number of points
reduced by one.

Argument Description:
manager    pointer to the Input/Display Manager class.

void add_point( Input_Display_Manager* manager )

Function Description:
Allows user to add a point to the end of a cubic segment or between two
other points of the segment. Segment PHIGS structure updated to reflect
the modification and number of points increased by one.

Argument Description:
manager    pointer to the Input/Display Manager class.

void end_point( Input_Display_Manager* manager )

Function Description:
Determines to which end of the cubic segment the new point is to be added.
Segment points linked list automatically updated.

Argument Description:
manager    pointer to the Input/Display Manager class.

void between_points( Input_Display_Manager* manager )

Function Description:
Determines between which two points of the cubic segment the new point is
to be added.

Argument Description:
manager    pointer to the Input/Display Manager class.

void update_add_point( Point* point, Point* point_1, Point* point_2 )

Function Description:
Adds a point to the cubic segment points linked list which was added
between two other points of the cubic segment. The new point is inserted
into the points linked list by checking the pick ids of point_1 and point_2.

Argument Description:
point    pointer to the new point to be added to points linked list.
point_1    pointer to the 1st point selected by user in the
            between_points function.
point_2    pointer to the 2nd point selected by user in the
between_points function.

void change_tangents( Input_Display_Manager* manager )

Function Description:
Allows the user to change the input tangent values (slopes) of a cubic
segment point. Segment PHIGS structure updated to reflect the
modification.

Argument Description:
manager pointer to the Input/Display Manager class.

void adjust_tangents()

Function Description:
If the automatic tangents update is used (default), the input tangents of the
endpoints of the cubic segment are automatically set equal to the slope of
any existing adjacent linear segments. This function is always called before
the tangents and intermediate points of the cubic segment are calculated.

Argument Description:
None.

void display_cubic_segment( Input_Display_Manager* manager )

Function Description:
Creates the PHIGS structure of a cubic segment and displays the structure
in the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

Public Functions:

Cubic_Segment()

Function Description:
Constructor. Initializes the cubic segment parameters.

Argument Description:
None.

void create_segment( Input_Display_Manager* manager )

Function Description:
Allows the user to create a cubic segment. User specifies number of points
and tangent specification.

Argument Description:
manager pointer to the Input/Display Manager class.

void create_segment( Input_Display_Manager* manager, Profile_Curve* curve )

Function Description:
Overloaded function which allows the user to create a cubic segment for a
Rule-Based Fuselage profile curve using the profile curve parameters.

**Argument Description:**
- **manager:** pointer to the Input/Display Manager class.
- **curve:** pointer to the Rule-Based Fuselage profile curve for which the cubic segment is to be created.

```c
void modify_segment( Input_Display_Manager* manager )
```

**Function Description:**
Allows the user to modify the points of a cubic segment: delete a point, add a point, move a point, or change the tangent values of a point.

**Argument Description:**
- **manager:** pointer to the Input/Display Manager class.

```c
void draw_segment( Input_Display_Manager* manager )
```

**Function Description:**
Executes the functions required to create the PHIGS structure of a cubic segment: calculate chord lengths, calculate tangents, and calculate the intermediate cubic segment points.

**Argument Description:**
- **manager:** pointer to the Input/Display Manager class.

```c
void draw_segment_structure()
```

**Function Description:**
Creates the cubic segment PHIGS structure which is inserted into an open PHIGS structure (profile curve or piecewise cross-section).

**Argument Description:**
None.

```c
Point* linear_orient_tangents( Point* orient_point, Point* point )
```

**Function Description:**
Creates the points linked list for a dummy cubic segment consisting of two points (orient_point and point). This dummy cubic segment is used to calculate the tangent vector of the orientation point (see Appendix C).

Returns a pointer to a point containing the orientation point tangent vector.

**Argument Description:**
- **orient_point:** pointer to the orientation point (see Cross-Section class).
- **point:** pointer to the first or last point of the linear segment on which the orientation point lies.

```c
Point* cubic_orient_point( Cross_Section* cross_section, Point* point )
```

**Function Description:**
Determines the xyz values of the orientation point by iteratively comparing the x-values of the cubic segment intermediate points (only for the interval in which the orientation point lies) with the cross-section depth (see Appendix C). Returns a pointer to the orientation point.
Argument Description:
cross_section cross-section for which the orientation point is to be
determined.
point pointer to the point preceding the orientation point.

Point* cubic_orient_tangents( Point* orient_point, Point* point )

Function Description:
Creates the points linked list for a dummy cubic segment consisting of the
original cubic segment on which the orientation point lies and the orientation
point. The dummy segment is used to calculate the tangent vector of the
orientation point (see Appendix C). Returns a pointer to a point containing
the orientation point tangent vector.

Argument Description:
orient_point pointer to the orientation point (see Cross-Section class).
point pointer to the point preceding the orientation point.

File Manager Class

Class Description:
Contains functions used by the Component Manager class to perform file input/output.

Class Inheritance:
None.

Private Functions:

void save_component( Input_Display_Manager* manager,
Component_Manager* comp_manager )

Function Description:
Writes the data of all existing components to an archive file.

Argument Description:
manager pointer to the Input/Display Manager class.
comp_manager pointer to the Component Manager class.

void save_rule_based_fuseage( char* file_name, Component* component )

Function Description:
Saves the component data for a Rule-Based Fuselage component.

Argument Description:
file_name character array containing the name of the file to which the
component data is to be written.
component pointer to the component for which the data is to be saved.
void save_spine_and_cross_section( char* file_name, Component* component )

Function Description:
Saves the component data for a Spine and Cross-Section component.

Argument Description:
- file_name character array containing the name of the file to which the
  component data is to be written.
- component pointer to the component for which the data is to be saved.

void save_cross_sections( char* file_name, Component* component )

Function Description:
Writes the data of all component cross-sections to a file.

Argument Description:
- file_name character array containing the name of the file to which the
  cross-sections data is to be written.
- component pointer to the component for which the data is to be saved.

void save_cross_section( char* file_name, Cross_Section* cross_section, int xsect )

Function Description:
Writes the data for the given cross-sections to a file.

Argument Description:
- file_name character array containing the name of the file to which the
  cross-section data is to be written.
- cross-section pointer to the cross-section for which the data is to be saved.
- xsect indicates the component cross-section for which the data is to be saved.

void save_segment( char* file_name, Segment* segment, int seg, char string[] )

Function Description:
Writes the data for the given segment to a file (profile curve or piecewise
cross-section segment).

Argument Description:
- file_name character array containing the name of the file to which the
  segment data is to be written.
- segment pointer to the segment for which the data is to be saved.
- seg indicates the segment for which the data is to be saved.
- string character array containing string written to file to indicate
  whether segment is a profile curve or piecewise
cross-section segment.

void save_point( char* file_name, Point* point, int pt )

Function Description:
Writes the data for the given point to a file.

Argument Description:
- file_name character array containing the name of the file to which the
  point data is to be written.
point pointer to the point for which the data is to be saved.
pt indicates the point for which the data is to be saved.

char* get_save_name( Input_Display_Manager* manager )

Function Description:
Returns the name of the file entered by the user to which the components
data is to be written. Checks to make sure if the file does not already exist.
If the file already exists, the user has the option to overwrite the file.

Argument Description:
manager pointer to the Input/Display Manager class.

void load_component( Input_Display_Manager* manager,
Component_Manager* comp_manager )

Function Description:
Reads components data from an archive file and creates the components.

Argument Description:
manager pointer to the Input/Display Manager class.
comp_manager pointer to the Component Manager class.

void load_rule_based_fuselage
(char* file_name, Component* component, int comp )

Function Description:
Reads the component data for a Rule-Based Fuselage component from an
archive file.

Argument Description:
file_name character array containing the name of the file from which
the component data is to be read.
component pointer to the component for which the data is to be read.
comp indicates the component for which the data is to be read.

void load_spine_and_cross_section
(char* file_name, Component* component, int comp )

Function Description:
Reads the component data for a Spine and Cross-Section component from an
archive file.

Argument Description:
file_name character array containing the name of the file from which
the component data is to be read.
component pointer to the component for which the data is to be read.
comp indicates the component for which the data is to be read.

void set_number_of_profile_curve_segments
(char* file_name, Profile_Curve* curve, char string[], int comp )

Function Description:
Sets the number of segments for the given profile curve from the data
contained in the file which is being read.
Argument Description:

file_name character array containing the name of the file from which the profile curve data is to be read.

curve pointer to the profile curve for which the data is to be read.

String character array which allows the function to read the profile curve segments data from the proper profile curve (crown, keel, plan, or spine).

comp indicates the component for which the data is to be read.

void load_profile_curve
( char* file_name, Profile_Curve* curve, char string[], int comp )

Function Description:

Creates the segments linked list of the profile curve from the data contained in the file which is being read.

Argument Description:

file_name character array containing the name of the file from which the profile curve data is to be read.

curve pointer to the profile curve for which the data is to be read.

String character array which allows the function to read the profile curve segments data from the proper profile curve (crown, keel, plan, or spine).

comp indicates the component for which the data is to be read.

Segment* load_profile_curve_segment
( char* file_name, int seg, char string[], int comp )

Function Description:

Creates a segment of a profile from the data contained in the file which is being read and returns a pointer to the segment.

Argument Description:

file_name character array containing the name of the file from which the segment data is to be read.

seg indicates the segment for which the data is to be read.

String character array which allows the function to read the profile curve segments data from the proper profile curve (crown, keel, plan, or spine).

comp indicates the component for which the data is to be read.

void load_cross_sections( char* file_name, Component* component, int comp )

Function Description:

Creates the cross-sections linked list of the given component from the data contained in the file which is being read.

Argument Description:

file_name character array containing the name of the file from which the component data is to be read.

component pointer to the component for which the data is to be read.

comp indicates the component for which the data is to be read.
Cross_Section* load_cross_section( char* file_name, int xsect, int comp )

Function Description:
Creates a component cross-section from the data in the file which is being read and returns a pointer to that cross-section.

Argument Description:
file_name character array containing the name of the file from which the component data is to be read.
xsect indicates the cross-section for which the data is to be read.
comp indicates the component for which the data is to be read.

Segment* load_cross_section_segment( char* file_name, int seg, char string[], int xsect, int comp )

Function Description:
Creates a segment of a piecewise cross-section from the data contained in the file which is being read and returns a pointer to the segment.

Argument Description:
file_name character array containing the name of the file from which the segment data is to be read.
seg indicates the segment for which the data is to be read.
string character array which allows the function to read the cross-section data from the proper cross-section (simple or piecewise).
xsect indicates the cross-section for which the data is to be read.
comp indicates the component for which the data is to be read.

char* get_load_name( Input_Display_Manager* manager )

Function Description:
Returns the filename entered by the user to be read. Checks to make sure the file exists. If it does not, the user is prompted to enter a new filename or return.

Argument Description:
manager pointer to the Input/Display Manager class.

Public Functions:

void process_files_menu( Input_Display_Manager* manager,
Component_Manager* comp_manager )

Function Description:
Allows user to save components data (if any components currently exist) or to read components data from an archive file.

Argument Description:
manager pointer to the Input/Display Manager class.
comp_manager pointer to the Component Manager class.
void save_acsynt_component
(    char* file_name, Input_Display_Manager *manager,
    Component_Manager* comp_manager )

Function Description:
Appends the geometric data of a component to the ACSYNT geometry file.

Argument Description:
file_name character pointer array containing the filename of the
ACYST geometry file.
manager pointer to the Input/Display Manager class.
comp_manager pointer to the Component Manager class.

void load_acsynt_component
(    char* file_name, Input_Display_Manager *manager,
    Component_Manager* comp_manager )

Function Description:
Reads the geometric data of a component from the ACSYNT geometry file
and creates the component objects from the data. The data is sent to
ACYST to create the surface model of the component within ACSYNT.

Argument Description:
file_name character pointer array containing the filename of the
ACYST geometry file.
manager pointer to the Input/Display Manager class.
comp_manager pointer to the Component Manager class.

Fillet Segment Class

Class Description:
Creates a fillet between two adjacent linear segments.

Class Inheritance:
Segment class.

Private Variables:
float  fillet_radius  Fillet radius entered by user.
Point* fillet_start_point Starting point of the fillet arc.
Point* fillet_end_point End point of the fillet arc.
Point* fillet_center_point Center of the fillet arc.
float  trim_length Equal length trimmed off of the linear segments to be
filleted by the fillet.
float theta
Angle between the linear segments to be filleted.

float fillet_arc_angle
Angle swept by the fillet arc.

float start_angle
Starting angle for the fillet arc.

float angle_step
Fillet arc angle increment used when the fillet arc points are calculated.

Private Functions:

void calculate_start_angle()

Function Description:
Calculates the starting angle of the fillet arc.

Argument Description:
None.

float euclidean_distance( Point* pt_1, Point* pt_2 )

Function Description:
Calculates the Euclidean distance between two points and returns the value.

Argument Description:
pt_1 pointer to the first point.
pr_1 pointer to the second point.

Public Functions:

Fillet_Segment( Segment* segment_1, Segment* segment_2, float radius )

Function Description:
Constructor. Initializes the fillet segment parameters, including the given fillet radius.

Argument Description:
segment_1 pointer to the first linear segment to be filleted.
segment_2 pointer to the second linear segment to be filleted.
radius fillet radius.

void create_segment( Input_Display_Manager* manager )

Function Description:
Creates a fillet between two linear segments given the fillet radius.

Argument Description:
manager pointer to the Input/Display Manager class.

void draw_segment( Input_Display_Manager* manager)
Function Description:
Creates and displays the PHIGS structure of the fillet segment.

Argument Description:
manager pointer to the Input/Display Manager class.

void draw_segment_structure()

Function Description:
Creates the fillet segment PHIGS structure which is inserted into an open
PHIGS structure (profile curve or piecewise cross-section).

Argument Description:
None.

Grid Class

Class Description:
Contains data and functions used to create grids in the XY, YZ, and XZ planes.

Class Inheritance:
None.

Private Variables:

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHIGS_Structure_ID*</td>
<td>PHIGS_struct_id[8]</td>
</tr>
<tr>
<td>int</td>
<td>grid_type</td>
</tr>
<tr>
<td>int</td>
<td>grid_plane</td>
</tr>
<tr>
<td>float</td>
<td>horizontal_minimum</td>
</tr>
<tr>
<td>float</td>
<td>horizontal_maximum</td>
</tr>
<tr>
<td>float</td>
<td>vertical_minimum</td>
</tr>
<tr>
<td>float</td>
<td>vertical_maximum</td>
</tr>
<tr>
<td>float</td>
<td>horizontal_spacing</td>
</tr>
</tbody>
</table>

Array of grid PHIGS structure ids.

Type of grid:
1 = auto grid
2 = new grid
3 = center grid

Plane in which grid is to be oriented:
1 = XY plane
2 = YZ plane
3 = XZ plane

Minimum horizontal value of the grid.

Maximum horizontal value of the grid.

Minimum vertical value of the grid.

Maximum vertical value of the grid.

Spacing between grid points in the horizontal direction.
float  **vertical_spacing**  
Spacing between grid points in the vertical direction.

float*  **grid_points**  
Dynamically allocated array of grid points. The array width is 3 (corresponding to x, y, and z values of the grid point).

**Private Functions:**

```c
void  **grid_spacing** ( Input_Display_Manager* manager )
```

**Function Description:**
Determines the grid horizontal and vertical spacing based on the type of grid and grid parameters.

**Argument Description:**
- manager  pointer to the Input/Display Manager class.

```c
int  **grid_points_space** ( )
```

**Function Description:**
Determines the amount of memory space which must be allocated for the grid points array based on the grid size and spacing and returns the value to the *draw_grid* function().

**Argument Description:**
None.

```c
void  **set_grid_point_pick_id** ( int pick_id )
```

**Function Description:**
Sets the pick id of a grid point to the given pick id and adds the pick id to the grid pick filter class for use with a pick filter.

**Argument Description:**
- pick_id  pick id to be set for the current grid point.

```c
void  **display_grid** ( Input_Display_Manager* manager )
```

**Function Description:**
Displays the grid PHIGS structure in the current view.

**Argument Description:**
- manager  pointer to the Input/Display Manager class.

**Public Functions:**

```c
Grid ( Input_Display_Manager* manager, int type_of_grid )
```

**Function Description:**
Constructor. Initializes grid horizontal minimum and maximum values based on the current view size. Initially sets the horizontal and vertical grid points spacing to 1/20th the total grid length, always making sure that the spacing is an even number.
Argument Description:
manager pointer to the Input/Display Manager class.
type_of_grid type of grid (auto, new, or center).

float get_grid_point( int index )

Function Description:
Returns the x, y, or z value of a grid point based on the given array index.

Argument Description:
index array index of the grid point whose value is to be returned

void draw_grid( Input_Display_Manager* manager )

Function Description:
Calculates the grid points based on the grid parameters and the plane in which the grid is to be oriented and creates the grid PHIGS structure.

Argument Description:
manager pointer to the Input/Display Manager class.

Input/Display Manager Class

Class Description:
Provides other classes with access to menu and viewing functions contained in the Menu Manager and View Manager classes. Allows the user to change the input methods and linear segment input methods.

Class Inheritance:
View Manager class.

Private Variables:

int input_method Points input method:
                          1 = keyboard
                          2 = grid

int linear_segment_input Linear segment input method:
                           1 = points only
                           2 = first point, angle, and length

Grid* XY_grid_pointer Pointer to a grid defined in the XY-plane.
Grid* YZ_grid_pointer Pointer to a grid defined in the YZ-plane.
Grid* XZ_grid_pointer Pointer to a grid defined in the XZ-plane.
Profile_Curve* crown_pointer Pointer to the crown profile curve of a
Profile_Curve*  \texttt{keel\_pointer}  
Pointer to the keel profile curve of a Rule-Based Fuselage component.

Profile_Curve*  \texttt{plan\_pointer}  
Pointer to the plan profile curve of a Rule-Based Fuselage component.

\textbf{PHIGS\_Structure\_ID*} \quad \textbf{rbf\_curve\_auto\_points\_structure\_id[8]}  
Array of structure ids for the PHIGS structure containing the three (sometimes only two) points which are automatically generated from the Rule-Based Fuselage profile curve parameters.

\textbf{PHIGS\_Structure\_ID*} \quad \textbf{rbf\_cross\_section\_points\_structure\_id[8]}  
Array of structure ids for the PHIGS structure containing the points at which a cross-section intersects the crown, keel, and plan profile curves of a Rule-Based Fuselage component.

\textbf{Public Functions:}

\textbf{Input\_Display\_Manager()}

\textbf{Function Description:}
Constructor. Calls the constructor of the View Manager class.

\textbf{Argument Description:}
None.

\textbf{void process\_input\_method\_menu()}

\textbf{Function Description:}
Processes the INPUT main menu option which allows the user to change the current input method and the current linear segment input method using the \texttt{process\_linear\_segment\_input\_menu()}.

\textbf{Argument Description:}
None.

\textbf{void process\_linear\_segment\_input\_menu()}

\textbf{Function Description:}
Allows the user to change the current linear segment input method.

\textbf{Argument Description:}
None.

\textbf{void process\_grid\_menu()}

\textbf{Function Description:}
Processes the GRID main menu option which allows the user to create a grid in the XY, YZ, or XZ plane or to turn off a previously created grid.

\textbf{Argument Description:}
None.
void set_grid_off_button( int button_active_state[] )

Function Description:
Sets the button_active_state of the GRID OFF push button for the GRID menu options. If no grids currently exist, this option is not available.

Argument Description:
button_active_state array of flags which indicate whether or not a push button may be selected.

void grid_off()

Function Description:
Turns off the selected grid.

Argument Description:
None.

void check_delete_grid( int grid_plane )

Function Description:
Checks to see if a grid exists in the given plane to see if it can be deleted.

Argument Description:
grid_plane plane which must be checked to see if a grid exists.

void delete_grid()

Function Description:
Deletes the grid from the current plane.

Argument Description:
None.

Point* get_grid_point( char message[] )

Function Description:
Returns a pointer to the point selected from the grid. This point may also be a Rule-Based Fuselage profile curve auto point or cross-section intersection point.

Argument Description:
message character array containing a prompt to the user.

int regular_menu
( char* button_labels, int number_of_items, int button_active_flag[] )

Function Description:
Creates an open control menu (i.e. user can select outside of the menu before closing the menu) from the input arguments using functions from the Menu Manager.

Argument Description:
button_labels array of character strings containing the push button labels.

number_of_items number of menu items for the menu to be created.
button_active_flag array of flags which indicate whether or not a push button may be selected.

void process_main_menu()

Function Description:
Processes the main menu options (MENU, INPUT, TANGENTS, GRID, AUTOCENTER, VIEWS) when the user selects outside of an open control menu.

Argument Description:
None.

void process_views_menu()

Function Description:
Processes the VIEWS main menu option which allows the user to toggle between the full display views and the 4-view display.

Argument Description:
None.

Keel Curve Class

Class Description:
Contains functions to create and modify the keel profile curve of a Rule-Based Fuselage component.

Class Inheritance:
RBF Profile Curve Manager class.

Private Functions:

void draw_keel_curve_auto_points( Input_Display_Manager* manager )

Function Description:
Creates the PHIGS structure containing the points automatically generated by the keel curve parameters if the parameters were entered by the user.

Argument Description:
manager pointer to the Input/Display Manager class.

void check_nose_keel_angle()

Function Description:
Ensures that the nose keel angle input by the user is between 0 and 90 degrees.
Argument Description:
  None.

void check_aft_body_upsweep_angle()

Function Description:
  Ensures that the aftbody upsweep angle input by the user is between 0 and 90 degrees.

Argument Description:
  None.

Public Functions:

Keel_Curve()

Function Description:
  Constructor. Initializes keel curve parameters.

Argument Description:
  None.

void set_rbf_parameters( Profile_Curve* crown_curve, Profile_Curve* plan_curve )

Function Description:
  Sets the Rule-Based Fuselage parameters of the keel profile curve equal to those of the crown curve or plan curve if either exists.

Argument Description:
  crown_curve pointer to the crown profile curve.
  plan_curve pointer to the plan profile curve.

void create_profile_curve( Input_Display_Manager *manager,
                           Component* component )

Function Description:
  Allows user to enter profile curve parameters for the keel curve. Also used to modify the profile curve parameters of the keel curve.

Argument Description:
  manager pointer to the Input/Display Manager class.
  component pointer to the Rule-Based Fuselage component for which the keel curve is to be created.

void set_last_point_rbf_first_linear_segment
( Input_Display_Manager* manager, Point* point )

Function Description:
  If the first segment of the keel profile curve is linear, this function automatically sets the last point of the segment based on the keel nose angle if the angle is non-zero and the user has entered the keel profile curve parameters.

Argument Description:
  manager pointer to the Input/Display Manager class.
point: pointer to the first point of the linear segment.

void set_last_rbf_linear_segment_point
(Input_Display_Manager* manager, Segment* segment, Point* point)

Function Description:
This function checks to see if the last point of the linear segment of the keel profile curve has an x-value equal to the x-value of the end of the fuselage. If so, the point is automatically adjusted based on the aft offset if the user entered the keel profile curve parameters.

Argument Description:
manager: pointer to the Input/Display Manager class.
segment: pointer to the last linear segment of the keel profile curve.
point: pointer to the last point of the linear segment.

void update_previous_keel_segment
(Input_Display_Manager* manager, Segment* segment, Point* point)

Function Description:
If the last user-defined segment of the keel profile curve is linear, this function adjusts the previous segment by setting the last point of the previous segment equal to the first point of the last user defined linear segment.

Argument Description:
manager: pointer to the Input/Display Manager class.
segment: pointer to the last segment of the keel profile curve.
point: pointer to the first point of the last user defined segment.

void set_first_point_rbf_first_cubic_segment
(Input_Display_Manager* manager, Point* point)

Function Description:
If the first segment of the keel profile curve is cubic, this function automatically sets the input tangent values (slopes) of the first point of the segment based on the keel nose angle if the angle is non-zero and the user has entered the keel profile curve parameters.

Argument Description:
manager: pointer to the Input/Display Manager class.
point: pointer to the first point of the cubic segment.

void check_last_point_cubic_segment
(Input_Display_Manager* manager, Point* point)

Function Description:
This function checks to see if the last point of the cubic segment of the keel profile curve has an x-value equal to the x-value of the end of the fuselage. If so, the point is automatically adjusted based on the aft offset if the user entered the keel profile curve parameters.

Argument Description:
manager: pointer to the Input/Display Manager class.
point: pointer to the last point of the cubic segment.
void modify_curve_parameters
    ( Input_Display_Manager* manager, Component* component )

Function Description:
    Allows the user to enter the profile curve parameters of the keel profile
curve. This function is used when the keel profile curve is first created and
when the user modifies the keel curve parameters.

Argument Description:
    manager          pointer to the Input/Display Manager class.
    component        pointer to the Rule-Based Fuselage component to which the
                     keel curve belongs.

void update_curve_segments( Input_Display_Manager* manager, Point* point )

Function Description:
    Automatically updates the segments of the keel profile curve after the keel
curve parameters have been modified.

Argument Description:
    manager          pointer to the Input/Display Manager class.

void update_curve_parameters()

Function Description:
    Automatically updates the parameters of the keel profile curve after the keel
curve segments have been modified.

Argument Description:
    None.

Linear Segment Class

Class Description:
    Contains data and functions used to create linear segments of profile curves and piecewise
    cross-sections of Rule-Based Fuselage and Spine and Cross-Section components.

Class Inheritance:
    Segment class.

Private Variables:
    float linear_segment_length         Length of the linear segment.
    float linear_segment_angle          Angle of the linear segment measured
c                                         counterclockwise from the horizontal to the linear
                                         segment.
Private Functions:

void set_linear_segment_point( Input_Display_Manager* manager, Point* point )

Function Description:
Calculates the point values of the second point of a linear segment from the
first point, angle, and length of the segment.

Argument Description:
manager pointer to the Input/Display Manager class.
point pointer to the first point of the linear segment.

void set_first_point_rbf_first_linear_segment
( Input_Display_Manager* manager, Profile_Curve* curve, Point* point )

Function Description:
Sets the first point xyz values of the first segment of a Rule-Based
Fuselage profile curve equal to the xyz values of the nose position if the
profile curve is created using the parameters.

Argument Description:
manager pointer to the Input/Display Manager class.
curve pointer to the profile curve for which the segment is to be
created.
point pointer to the first point of the linear segment.

void set_last_rbf_linear_segment_points
( Input_Display_Manager* manager, Profile_Curve* curve, Point* point )

Function Description:
If the last user-defined segment of a Rule-Based Fuselage profile curve
created using parameters is linear, this function sets the xyz values of the
last point based on the parameters (aft offset for the crown and keel curves
and aft radius for the plan curve).

Argument Description:
manager pointer to the Input/Display Manager class.
curve pointer to the profile curve for which the segment is to be
created.
point pointer to the last point of the linear segment.

void change_angle_length( Input_Display_Manager* manager )

Function Description:
Allows the user to change the angle and/or length of a linear segment and
updates the linear segment PHIGS structure. Any existing adjacent
segments are automatically updated as necessary.

Argument Description:
manager pointer to the Input/Display Manager class.
Public Functions:

**Linear_Segment()**

*Function Description:* Constructor. Initializes the linear segment parameters.

*Argument Description:* None.

void **create_segment**( Input_Display_Manager* manager )

*Function Description:* Allows the user to create a linear segment using either points only or the first point, angle and length of the segment.

*Argument Description:* manager pointer to the Input/Display Manager class.

void **create_segment** ( Input_Display_Manager* manager, Profile_Curve* curve )

*Function Description:* Overloaded function which allows the user to create a linear segment for a Rule-Based Fuselage profile curve using the profile curve parameters.

*Argument Description:* manager curve pointer to the Input/Display Manager class. pointer to the Rule-Based Fuselage profile curve for which the linear segment is to be created.

void **set_linear_segment_angle_length** ( Input_Display_Manager* manager )

*Function Description:* Calculates the angle and length of a linear segment from the segment endpoints.

*Argument Description:* manager pointer to the Input/Display Manager class.

void **modify_segment** ( Input_Display_Manager* manager )

*Function Description:* Allows the user to modify the parameters of a linear segment: move endpoints, change angle, and change length.

*Argument Description:* manager pointer to the Input/Display Manager class.

void **draw_segment** ( Input_Display_Manager* manager )

*Function Description:* Creates the PHIGS structure of a linear segment and displays the structure in the current view.

*Argument Description:* manager pointer to the Input/Display Manager class.
void draw_segment_structure()

Function Description:
Creates the linear segment PHIGS structure which is inserted into an open
PHIGS structure (profile curve or piecewise cross-section).

Argument Description:
None.

Point* linear_orient_point( Cross_Section* cross_section, Point* point )

Function Description:
Determines the orientation point if it lies between two points of a linear
segment (see Appendix C). The xyz values of the orientation point are
determined from the slope of the linear segment. Returns a pointer to the
orientation point.

Argument Description:
cross_section pointer to the cross-section for which the orientation
point is to be determined.

Point* special_linear_orient_point( Point* point )

Function Description:
Determines the orientation point for the special case when it lies exactly on
the endpoint of a linear segment (see Appendix C). Returns a pointer to the
orientation point.

Argument Description:
point pointer to the first point of the linear segment on which the
orientation point lies.

Menu Manager Class

Class Description:
Contains closed control menus (i.e. menus which must be processed before the user can
select outside of the menu). Initializes the interface manager which is inherited by the View
Manager class and indirectly inherited by the Input/Display Manager class.

Class Inheritance:
None.

Private Variables:

Interface_Manager* interface_manager Pointer to the interface manager which
controls the raising and lowering of windows [Woya92].
Static_Menu*  main_menu  Pointer to the main menu. Used to raise and lower the main menu.

Static_Menu*  slab_menu  Pointer to the slab menu which defines the user interface background. Used to raise and lower the background frames for the 4-view display.

Pop_Up_Menu*  current_menu  Pointer to the current menu. Used to raise and lower the current menu when the user selects outside of the menu to process the main menu options (for open control menus only).

Public Functions:

Menu_Manager()

Function Description:
Constructor. Initializes the interface manager (i.e. calls the constructor).

Argument Description:
None.

int  input_menu( char* prompt[], char* default[], char* input, int number_of_items, int num_buttons )

Function Description:
Creates a pop-up menu which is used to allow the user to enter text input. Returns the id of the menu selection.

Argument Description:
prompt  array of character strings containing the input prompts.
default  array of character strings containing the default values for the input items.
input  character array containing the input string entered by the user (NULL if no input entered).
number_of_items  number of input items that the menu is to create.
num_buttons  number of push buttons to be created at the bottom of the menu.

int  message_menu( char* message, int number_of_items )

Function Description:
Creates a pop-up menu which displays a message to the user. This menu is a confirm menu with either one or two push buttons at the bottom. Returns the id of the menu selection.

Argument Description:
message  character array containing the message to be displayed to the user.
number_of_buttons  number of push buttons to be created at the bottom of the menu.
float set_pop_up_menu_width( char* button_labels[], int number_of_items )

Function Description:
Dynamically calculates the width of a pop-up menu based on the maximum string length of the array of character strings (button_labels). Returns the value of the pop-up menu width.

Argument Description:
button_labels array of character strings containing the menu push button labels.
number_of_items number of menu items to be created.

float set_pop_up_menu_height( int number_of_items )

Function Description:
Dynamically calculates the height of a pop-up menu based on the number_of_items. Returns the value of the pop-up menu height.

Argument Description:
number_of_items number of menu items to be created.

float set_input_menu_width( char* prompt[], int number_of_items )

Function Description:
Dynamically calculates the width of an input menu based on the maximum string length of the array of character strings (prompt). Returns the value of the input menu width.

Argument Description:
prompt array of character strings containing the input prompts.
number_of_items number of menu items to be created.

float set_input_menu_height( int number_of_items )

Function Description:
Dynamically calculates the height of an input menu based on the number of items. Returns the value of the input menu height.

Argument Description:
number_of_items number of menu items to be created.

float set_frame_x( char* prompt[], int number_of_items )

Function Description:
Dynamically sets the x-location of a text input frame of an input menu based on the maximum string length of the array of character strings (prompt). Returns the value of the frame x-location.

Argument Description:
prompt array of character strings containing the input prompts.
number_of_items number of menu items to be created.
float set_frame_y( float input_menu_height )

Function Description:
Dynamically sets the y-location of a text input frame of an input menu based on the input menu height. Returns the value of the frame y-location.

Argument Description:
input_menu_height    height of the input menu (dynamically set by the set_input_menu_height() function).

float set_frame_width()

Function Description:
Dynamically sets the width of a text input frame of an input menu. Returns the value of the text input frame width.

Argument Description:
None.

float set_max_string_length( char* prompt[], int number_of_items )

Function Description:
Determines the maximum string length of an array of character strings and returns that string length value.

Argument Description:
prompt array of character strings containing the input prompts.
number_of_items number of menu items to be created.

Piecewise Cross-Section Class

Class Description:
Contains functions to create, modify, display, and orient piecewise cross-sections which consist of linear and/or cubic segments.

Class Inheritance:
Cross-Section class.
Segment Manager class.

Private Functions:

void set_scale_segment_points( Segment* segment, Segment* scale_segment )

Function Description:
Creates a segment of a piecewise cross-section from a segment of the scaling cross-section.
void set_scale_segment_points(
    Segment* segment, Segment* scale_segment, float delta_crown,
    float delta_keel, float delta_plan)

Function Description:
Overloaded function. Creates a segment of a piecewise cross-section from a
segment of the scaling cross-section for Rule-Based Fuselage components.

void mirror_y_and_z_plane()

Function Description:
Mirrors the segments of a piecewise cross-section with YZ symmetry
(symmetrical about the Y and Z axes).

Argument Description:
None.

void mirror_y_plane()

Function Description:
Mirrors the segments of a piecewise cross-section with Y symmetry
(symmetrical about the Y axis).

Argument Description:
None.

void mirror_z_plane()

Function Description:
Mirrors the segments of a piecewise cross-section with Z symmetry
(symmetrical about the Z axis).

Argument Description:
None.
Public Functions:

Piecewise_Cross_Section( int symmetry_flag, int spacing_flag )

Function Description:
Constructor. Initializes the cross-section symmetry and spacing method.

Argument Description:
symmetry_flag flag indicating the cross-section symmetry.
spacing_flag flag indicating the cross-section spacing.

void create_cross_section( Input_Display_Manager* manager )

Function Description:
Allows the user to create the segments (linear and/or cubic) of a piecewise cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.

void scale_cross_section( Input_Display_Manager* manager )

Function Description:
Creates a piecewise cross-section by scaling a previously defined cross-section, referred to as the scaling cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.

void scale_cross_section( Input_Display_Manager* manager,
                        Profile_Curve* crown_curve,
                        Profile_Curve* keel_curve,
                        Profile_Curve* plan_curve )

Function Description:
Overloaded function. Creates a piecewise cross-section by scaling a previously defined cross-section, referred to as the scaling cross-section for a Rule-Based Fuselage component.

Argument Description:
manager pointer to the Input/Display Manager class.
crown_curve pointer to the crown profile curve of a Rule-Based Fuselage component.
keel_curve pointer to the keel profile curve of a Rule-Based Fuselage component.
plan_curve pointer to the plan profile curve of a Rule-Based Fuselage component.

Cross_Section* copy_cross_section()

Function Description:
Creates a new piecewise cross-section by copying the segments of a previously defined cross-section. Returns a pointer to the new cross-section.
Argument Description:
None.

```c
void modify_cross_section( Input_Display_Manager* manager,
                          Component* component )
```

Function Description:
Allows the user to modify the segments of a piecewise cross-section.

Argument Description:
- manager: pointer to the Input/Display Manager class.
- component: pointer to the component for which the cross-section is to be modified.

```c
void draw_cross_section()
```

Function Description:
Creates the PHIGS structure of a piecewise cross-section by creating the PHIGS structures of the cross-section segments.

Argument Description:
None.

```c
void display_cross_section_segments( Input_Display_Manager* manager )
```

Function Description:
Displays the segments of a piecewise cross-section autocentered in the current view. Used to modify the cross-section segments.

Argument Description:
- manager: pointer to the Input/Display Manager class.

```c
void remove_cross_section_segments( Input_Display_Manager* manager )
```

Function Description:
Removes the segments of a piecewise cross-section from the current view.

Argument Description:
- manager: pointer to the Input/Display Manager class.

```c
void set_min_max_extrema( Input_Display_Manager* manager )
```

Function Description:
Sets the minimum and maximum point values of the piecewise cross-section by checking the minimum and maximum values of each cross-section segment. The minimum and maximum values must be modified depending on the cross-section symmetry. These values are used to autocenter the cross-section.

Argument Description:
- manager: pointer to the Input/Display Manager class.

```c
void check_cross_section_symmetry()
```

Function Description:
Checks the piecewise cross-section symmetry to determine if any of the cross-section segments need to be mirrored about any axes of symmetry.
Argument Description:
None.

void create_surface_segments()

Function Description:
Creates the segments which define the cross-section surface and contains the cross-section points which are sent to ACSYNT to create the component surface model. The segments are created based on the cross-section symmetry.

Argument Description:
None.

void check_rbf_cross_section(Input_Display_Manager* manager,
Profile_Curve* crown_curve,
Profile_Curve* keel_curve,
Profile_Curve* plan_curve)

Function Description:
Ensures that the piecewise cross-section of a Rule-Based Fuselage component intersects the crown, keel, and plan profile curves at the cross-section depth.

Argument Description:

manager pointer to the Input/Display Manager class.
crown_curve pointer to the crown profile curve of a Rule-Based Fuselage component.
keel_curve pointer to the keel profile curve of a Rule-Based Fuselage component.
plan_curve pointer to the plan profile curve of a Rule-Based Fuselage component.

Plan Curve Class

Class Description:
Contains functions to create and modify the plan profile curve of a Rule-Based Fuselage component.

Class Inheritance:
RBF Profile Curve Manager class.

Private Functions:

void check_nose_angle()

Function Description:
Ensures that the nose angle input by the user is between 0 and 90 degrees.
Argument Description:
None.

void draw_plan_curve_auto_points( Input_Display_Manager* manager )

Function Description:
Creates the PHIGS structure containing the points automatically generated by the plan curve parameters if the parameters were entered by the user.

Argument Description:
manager pointer to the Input/Display Manager class.

Public Functions:

Plan_Curve()

Function Description:
Constructor. Initializes plan curve parameters.

Argument Description:
None.

void set_rbf_parameters( Profile_Curve* crown_curve, Profile_Curve* keel_curve )

Function Description:
Sets the Rule-Based Fuselage parameters of the plan profile curve equal to those of the crown curve or keel curve if either exists.

Argument Description:
crown_curve pointer to the crown profile curve.
keel_curve pointer to the keel profile curve.

void create_profile_curve( Input_Display_Manager* manager,
Component* component )

Function Description:
Allows user to enter profile curve parameters for the keel curve. Also used to modify the profile curve parameters of the plan curve.

Argument Description:
manager pointer to the Input/Display Manager class.
component pointer to the Rule-Based Fuselage component for which the plan curve is to be created.

void set_last_point_rbf_first_linear_segment
( Input_Display_Manager* manager, Point* point )

Function Description:
If the first segment of the plan profile curve is linear, this function automatically sets the last point of the segment based on the nose radius and angle if the angle and radius are non-zero and the user has entered the plan profile curve parameters.

Argument Description:
manager pointer to the Input/Display Manager class.
point pointer to the first point of the linear segment.

**void set_last_rbf_linear_segment_point**
(Input_Display_Manager* manager, Segment* segment, Point* point)

**Function Description:**
This function checks to see if the last point of the linear segment of the plan profile curve has an x-value equal to the x-value of the end of the fuselage. If so, the point is automatically adjusted based on the aft radius if the user entered the plan profile curve parameters.

**Argument Description:**
- manager: pointer to the Input/Display Manager class.
- segment: pointer to the last linear segment of the plan profile curve.
- point: pointer to the last point of the linear segment.

**void set_first_point_rbf_first_cubic_segment**
(Input_Display_Manager* manager, Point* point)

**Function Description:**
If the first segment of the plan profile curve is cubic, this function automatically sets the input tangent values (slopes) of the first point of the segment based on the nose angle if the angle is non-zero and the user has entered the plan profile curve parameters.

**Argument Description:**
- manager: pointer to the Input/Display Manager class.
- point: pointer to the first point of the cubic segment.

**void set_second_point_rbf_first_cubic_segment**
(Input_Display_Manager* manager, Point* point)

**Function Description:**
If the first segment of the plan profile curve is cubic, this function automatically sets the value of the second point of the cubic segment based on the nose radius if the radius is non-zero and the user has entered the plan profile curve parameters.

**Argument Description:**
- manager: pointer to the Input/Display Manager class.
- point: pointer to the second point of the cubic segment.

**void check_last_point_cubic_segment**
(Input_Display_Manager* manager, Point* point)

**Function Description:**
This function checks to see if the last point of the cubic segment of the plan profile curve has an x-value equal to the x-value of the end of the fuselage. If so, the point is automatically adjusted based on the aft radius if the user entered the plan profile curve parameters.

**Argument Description:**
- manager: pointer to the Input/Display Manager class.
- point: pointer to the last point of the cubic segment.
void modify_curve_parameters
( Input_Display_Manager* manager, Component* component )

Function Description:
Allows the user to enter the profile curve parameters of the plan profile curve. This function is used when the plan profile curve is first created and when the user modifies the plan curve parameters.

Argument Description:
manager pointer to the Input/Display Manager class.
component pointer to the Rule-Based Fuselage component to which the plan curve belongs.

void update_curve_segments( Input_Display_Manager* manager, Point* point )

Function Description:
Automatically updates the segments of the plan profile curve after the plan curve parameters have been modified.

Argument Description:
manager pointer to the Input/Display Manager class.

void update_curve_parameters()

Function Description:
Automatically updates the parameters of the plan profile curve after the plan curve segments have been modified.

Argument Description:
None.

Point Class

Class Description:
Contains data and functions used to create, modify, and display points of linear and cubic segments as well as fillet segments.

Class Inheritance:
None.

Private Variables:

PHIGS_Structure_ID* PHIGS_struct_id[8]
Array of PHIGS structure ids.

int pick_id Pick id of the point PHIGS structure.

int point_definition Flag which indicates whether or not a segment point has been defined:
0 = not defined  
1 = defined

float x  
X-value of the segment point.

float y  
Y-value of the segment point.

float z  
Z-value of the segment point.

float dx  
X-value of the segment point tangent input by the user (slope at that point).

float dy  
Y-value of the segment point tangent input by the user (slope at that point).

float dz  
Z-value of the segment point tangent input by the user (slope at that point).

float x_tangent  
X-value of the segment point tangent calculated by the calculate_tangents() function. This tangent value is used to calculate the intermediate cubic segment points (see Cubic Segment class).

float y_tangent  
Y-value of the segment point tangent calculated by the calculate_tangents() function. This tangent value is used to calculate the intermediate cubic segment points (see Cubic Segment class).

float z_tangent  
Z-value of the segment point tangent calculated by the calculate_tangents() function. This tangent value is used to calculate the intermediate cubic segment points (see Cubic Segment class).

float chord_length  
Euclidean distance between adjacent points of a cubic segment. Used to calculate the cubic segment points tangents vectors (x_tangent, y_tangent, z_tangent). Note that a chord length of zero is always associated with the last point of a cubic segment since there is no next point.

int tangent_specification  
Flag which indicates whether or not the user must specify an input tangent for the point to be created (dx, dy, dz):

1 = user must specify tangent  
2 = user cannot specify tangent  
3 = user has option to specify tangent or not

Note that for linear segments, only option 2 is available. For cubic segments, this flag is set by the set_point_tangent_specification() function (see Cubic Segment class).

Point* previous  
Pointer to the previous point in the segment points
linked list.

Point* next  Pointer to the next point in the segment points linked list.

Public Functions:

Point()

Function Description:
Constructor. Initializes the point parameters.

Argument Description:
None.

void create_point( Input_Display_Manager* manager )

Function Description:
Creates the point of a linear or cubic segment.

Argument Description:
manager       pointer to the Input/Display Manager class.

void get_xyz_values( Input_Display_Manager* manager )

Function Description:
Gets the xyz values of the point to be created from the user either by
keyboard or grid input.

Argument Description:
manager       pointer to the Input/Display Manager class.

void get_xyz_tangents( Input_Display_Manager* manager )

Function Description:
Gets the xyz tangent values of the point to be created from the user either by
keyboard or grid input (cubic segments only).

Argument Description:
manager       pointer to the Input/Display Manager class.

void draw_segment_point( Input_Display_Manager* manager )

Function Description:
Creates the PHIGS structure of a segment point from the point xyz values.

Argument Description:
manager       pointer to the Input/Display Manager class.

void display_point( Input_Display_Manager* manager )

Function Description:
Displays the PHIGS structure of a segment point in the current view.

Argument Description:
manager       pointer to the Input/Display Manager class.
void remove_point(Input_Display_Manager* manager)

Function Description:
Removes the PHIGS structure of a segment point from the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void delete_point_structure()

Function Description:
Deletes the PHIGS structure of a segment point.

Argument Description:
None.

void set_point_pick_id()

Function Description:
Sets the pick id of the segment point PHIGS structure and adds the pick id
to the segment point pick filter class.

Argument Description:
None.

Profile Curve Class

Class Description:
Indirect base class of the Crown Curve, Keel Curve, and Plan Curve derived classes
(Rule-Based Fuselage components) and the Spine Curve derived class (Spine and
Cross-Section component).

Class Inheritance:
Segment Manager class.

Private Variables:

int profile_curve_type Type of profile curve:
1 = crown curve
2 = keel curve
3 = plan curve
4 = spine curve

int profile_curve_definition Flag which indicates the status of the profile curve:
0 = not defined
1 = defined

int highlight_flag Flag which indicates whether or not the profile curve
PHIGS structure is to be displayed with highlighting color:
  0 = highlighting off
  1 = highlighting on

Public Functions:

Profile_Curve* copy_profile_curve()

Function Description:
Copies a profile curve by duplicating the segments of a previously defined profile curve. Returns a pointer to the new profile curve.

Argument Description:
None.

void draw_profile_curve( Input_Display_Manager* manager )

Function Description:
Creates the profile curve PHIGS structure by creating the PHIGS structures of the profile curve segments.

Argument Description:
manager pointer to the Input/Display Manager class.

void display_profile_curve( Input_Display_Manager* manager )

Function Description:
Displays the profile curve PHIGS structure autocentered in the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void remove_profile_curve( Input_Display_Manager* manager )

Function Description:
Removes the profile curve PHIGS structure from the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void delete_profile_curve_structure()

Function Description:
Deletes the profile curve PHIGS structure and deletes the profile curve segments PHIGS structures.

Argument Description:
None.

void display_profile_curve_segments( Input_Display_Manager* manager )

Function Description:
Displays the PHIGS structures of the profile curve segments autocentered in...
the current view so the user may modify the segments.

**Argument Description:**
manager pointer to the Input/Display Manager class.

```c
void remove_profile_curve_segments( Input_Display_Manager* manager )
```

**Function Description:**
Removes the PHIGS structures of the profile curve segments from the current view.

**Argument Description:**
manager pointer to the Input/Display Manager class.

```c
void autocenter_profile_curve( Input_Display_Manager* manager )
```

**Function Description:**
Autocenters the profile curve PHIGS structure in the current view.

**Argument Description:**
manager pointer to the Input/Display Manager class.

```c
void set_min_max_extrema( Input_Display_Manager* manager )
```

**Function Description:**
Sets the minimum and maximum point values of the profile curve by checking the minimum and maximum values of each profile curve segment. These values are used to autocenter the profile curve.

**Argument Description:**
manager pointer to the Input/Display Manager class.

```c
void highlight_profile_curve( Input_Display_Manager* manager )
```

**Function Description:**
Highlights a profile curve by setting the highlighting flag of each profile curve segment to HIGHLIGHTING_ON.

**Argument Description:**
manager pointer to the Input/Display Manager class.

**Point** `orientation_point( Cross_Section* cross_section )`

**Function Description:**
Determines the segment and two segment points between which the cross-section orientation point lies (see Cross-Section class). Based on the segment type, calls functions to determine the orientation point along the profile curve. Returns a pointer to the orientation point (see Appendix C).

**Argument Description:**
cross_section pointer to the cross-section for which the orientation point is to be determined.
**RBF Profile Curve Manager**

**Class Description:**
This class contains data and functions that are used by the Rule-Based Fuselage profile curve classes - Crown Curve, Keel Curve, and Plan Curve.

**Class Inheritance:**
Profile Curve class.

**Protected Variables:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
</table>
| int    | last_rbf_segment | Flag which indicates if the current segment is the last user-defined segment (i.e. manually created):
                     0 = not the last user-defined segment  
                     1 = last user-defined segment
                     This flag is only used when the Rule-Based Fuselage profile curve is created using the profile curve parameters. In this case, a linear segment is automatically created by the system for the last segment of the profile curve from the parameters if the parameters are used to create the last segment (aft offset for the crown and keel curve, aft radius for the plan curve) are non-zero. |
| Point* | auto_point_1     | Pointer to the nose position point automatically generated by the Rule-Based Fuselage profile curve parameters.                           |
| Point* | auto_point_2     | Pointer to the offset point (aft offset for Crown and Keel curves, aft radius for Plan curve) automatically generated by the Rule-Based Fuselage profile curve parameters. |
| Point* | auto_point_3     | Pointer to the fuselage end point automatically generated by the Rule-Based Fuselage profile curve parameters.                           |
| Point* | intersection_point | Point at which a Rule-Based Fuselage profile curve intersects a cross-section.                                                              |
| Point* | nose_position    | Pointer to the fuselage nose position.                                                                                                      |
| float  | fuselage_length  | Length of the fuselage.                                                                                                                     |
float nose_radius  Fuselage nose radius. Plan profile curve parameter.
float nose_angle  Fuselage nose angle. Plan profile curve parameter.
float aft_radius  Fuselage aft radius. Plan profile curve parameter.
float nose_keel_angle  Fuselage nose keel angle. Keel profile curve parameter.
float aft_body_upsweep_angle  Fuselage aft-body upsweep angle. Keel profile curve parameter.

**Rule-Based Fuselage Class**

**Class Description:**
Contains functions to create and display a Rule-Based Fuselage component as well as functions to create, modify, and display the profile curves and cross-sections of a Rule-Based Fuselage component. The three profile curves which define the Rule-Based Fuselage component are the crown, keel, and plan profile curves.

**Class Inheritance:**
Component class.

**Private Variables:**

Profile_Curve*  crown_curve  pointer to the crown profile curve.
Profile_Curve*  keel_curve  pointer to the keel profile curve.
Profile_Curve*  plan_curve  pointer to the plan profile curve.

**Private Functions:**

```c
int check_create_component( Input_Display_Manager* manager )
```

**Function Description:**
When the user exits the create_component() function, this function checks
to make sure that the component is properly defined (i.e. the crown, keel, and plan profile curves have been created). If not, the user is prompted to return to create_component() or to cancel, deleting any profile curves which may have been created. Returns a flag indicating whether or not to exit the create_component() function.

**Argument Description:**

void **create_crown_curve**( Input_Display_Manager* manager )

**Function Description:**
Initializes the parameters required to create the crown profile curve.

**Argument Description:**
manager pointer to the Input/Display Manager class.

void **create_keel_curve**( Input_Display_Manager* manager )

**Function Description:**
Initializes the parameters required to create the keel profile curve.

**Argument Description:**
manager pointer to the Input/Display Manager class.

void **create_plan_curve**( Input_Display_Manager* manager )

**Function Description:**
Initializes the parameters required to create the plan profile curve.

**Argument Description:**
manager pointer to the Input/Display Manager class.

void **check_profile_curve**( Input_Display_Manager* manager, Profile_Curve* curve )

**Function Description:**
Checks to see if the profile curve selected by the user to create already exists. If so, the user is given the option to replace the profile curve or to cancel the selection.

**Argument Description:**
manager pointer to the Input/Display Manager class.
curve pointer to the curve to be created by the user.

void **create_profile_curve**( Input_Display_Manager* manager, Profile_Curve* curve )

**Function Description:**
Allows to create the segments of the crown, keel, or plan profile curve.

**Argument Description:**
manager pointer to the Input/Display Manager class.
curve pointer to the curve to be created by the user.
void create_last_segment( Input_Display_Manager* manager,
                        Profile_Curve* curve, Segment* segment )

Function Description:
Automatically creates the last segment of a Rule-Based Fuselage profile
curve if the curve is created using the profile curve parameters and the
parameters which are used to calculate the last segment are non-zero (aft
offset for the crown and keel curves, aft radius for the plan curve). Note
that this segment is always linear.

Argument Description:
manager     pointer to the Input/Display Manager class.
curve       pointer to the curve to be created by the user.
segment     pointer to the current last segment of the profile curve to be
            made the previous segment of the linear segment to be
            automatically created.

void modify_profile_curve( Input_Display_Manager* manager,
                            Profile_Curve* curve )

Function Description:
Determines if the user wants to modify the segments or the parameters of
the profile curve to be modified.

Argument Description:
manager     pointer to the Input/Display Manager class.
curve       pointer to the curve to be modified.

void check_modify_profile_curve( Profile_Curve* curve )

Function Description:
After a profile curve has been modified, this function checks to see if all of
the profile curve segments have been deleted. If so, the profile curve must
be deleted.

Argument Description:
curve       pointer to the modified profile curve.

void modify_profile_curve_segments( Input_Display_Manager* manager,
                                    Profile_Curve* curve )

Function Description:
Allows the user to modify the segments of the given profile curve.

Argument Description:
manager     pointer to the Input/Display Manager class.
curve       pointer to the curve to be modified.

void delete_profile_curve( Profile_Curve* curve )

Function Description:
Sets the pointer to the given profile curve equal to NULL and deletes the
curve (i.e. calls the destructor).

Argument Description:
curve       pointer to the profile curve which is to be deleted.
void remove_profile_curve_structures( Input_Display_Manager* manager )

Function Description:
Removes the PHIGS structures of all three Rule-Based Fuselage profile curves from the current view. Note that not all three profile curves need to be defined to use this function. Only those profile curves which are defined will be removed.

Argument Description:
manager pointer to the Input/Display Manager class.

void display_profile_curves( Input_Display_Manager* manager )

Function Description:
Displays the segments of all three Rule-Based Fuselage profile curves auto-centered in the current view to allow the user to modify the segments.

Argument Description:
manager pointer to the Input/Display Manager class.

void remove_profile_curves( Input_Display_Manager* manager )

Function Description:
Removes the segments of all three Rule-Based Fuselage profile curves from the current view.

Argument Description:
manager pointer to the Input/Display Manager class.
manager pointer to the Input/Display Manager class.

void highlight_profile_curves
( Input_Display_Manager* manager, Profile_Curve* curve, int highlighting )

Function Description:
When a Rule-Based Fuselage profile curve is modified, this function draws the PHIGS structures of any other existing profile curves with or without highlighting depending on the highlighting flag.

Argument Description:
manager pointer to the Input/Display Manager class.
curve pointer to the curve to be modified.
highlighting flag indicating whether or not any other existing profile curves are to be highlighted.

void display_rbf_cross_section_points(
 Input_Display_Manager* manager,
 Cross_Section* cross_section )

Function Description:
Creates the PHIGS structure containing the three points at which the crown, keel, and plan profile curve intersect the cross-section to be created. Displays the PHIGS structure auto-centered in the current view.

Argument Description:
manager pointer to the Input/Display Manager class.
void **create_cross_sections** (Input_Display_Manager* manager)

*Function Description:* Allows the user to create individual cross-sections of a Rule-Based Fuselage component.

*Argument Description:*
  - manager: pointer to the Input/Display Manager class.

Cross_Section* **create_cross_section_menu** (Input_Display_Manager* manager)

*Function Description:* Allows the user to select the type of cross-section to create or the change the current cross-section symmetry and/or spacing method. Returns a pointer to the cross-section to be created.

*Argument Description:*
  - manager: pointer to the Input/Display Manager class.

void **initialize_cross_section** (Input_Display_Manager* manager,
                                    Cross_Section* cross_section)

*Function Description:* Initializes the parameters of the given cross-section to be created (symmetry, spacing, and depth).

*Argument Description:*
  - manager: pointer to the Input/Display Manager class.
  - cross_section: pointer to the cross-section to be created.

void **auto_cross_section** (Input_Display_Manager* manager,
                              Cross_Section* cross_section)

*Function Description:* Creates a cross-section automatically by scaling a previously defined cross-section, referred to as the scaling cross-section.

*Argument Description:*
  - manager: pointer to the Input/Display Manager class.
  - cross_section: pointer to the cross-section to be created.

void **scale_cross_section** (Input_Display_Manager* manager,
                               Cross_Section* cross_section)

*Function Description:* Creates the given cross-section using the parameters of the scaling cross-section.

*Argument Description:*
  - manager: pointer to the Input/Display Manager class.
  - cross_section: pointer to the cross-section to be created.
void check_cross_section_depth( Input_Display_Manager* manager,
                                Cross_Section* cross_section )

Function Description:
Checks to see if the depth input/calculated for the current cross-section is
valid (i.e. the depth value lies on the fuselage profile curve and no
cross-section already exists at that depth). If the depth is invalid, the user
is informed and the cross-section is not created.

Argument Description:
manager pointer to the Input/Display Manager class.
cross_section pointer to the cross-section to be created.

void orient_cross_section( Input_Display_Manager* manager,
                          Cross_Section* cross_section )

Function Description:
Orients the given cross-section (i.e. the cross-section is created at the proper
position along the fuselage and is adjusted to intersect the crown, keel, and
plan profile curves).

Argument Description:
manager pointer to the Input/Display Manager class.
cross_section pointer to the cross-section to be oriented.

Public Functions:

Rule_Based_Fuselage( int component_number )

Function Description:
Constructor. Initializes the Rule-Based Fuselage component parameters.

Argument Description:
component_number ACSYNT component identifier.

void create_component( Input_Display_Manager* manager )

Function Description:
Allows the user to create or modify a Rule-Based Fuselage component
(i.e. create/modify profile curves and cross-sections).

Argument Description:
manager pointer to the Input/Display Manager class.

Component* copy_component( int component_number )

Function Description:
Creates a new Rule-Based Fuselage component by duplicating an existing
component. Returns a pointer to the new component.

Argument Description:
component_number ACSYNT component identifier.
void draw_component(Input_Display_Manager* manager)

Function Description:
Creates the component PHIGS structure by creating the PHIGS structures
of the profile curves and cross-sections (if they exist).

Argument Description:
manager pointer to the Input/Display Manager class.

void delete_component_structure()

Function Description:
Deletes the component PHIGS structure and deletes the PHIGS structures
of the profile curves and cross-sections (if they exist).

Argument Description:
None.

void set_min_max_extrema(Input_Display_Manager* manager)

Function Description:
Sets the minimum and maximum point values of the component by checking
the minimum and maximum values of each profile curve and any existing
cross-sections. These values are used to autocenter the component.

Argument Description:
manager pointer to the Input/Display Manager class.

void highlight_component(Input_Display_Manager* manager, int highlighting)

Function Description:
Turns the highlighting of a component on or off depending on the specified
highlighting flag.

Argument Description:
manager pointer to the Input/Display Manager class.
highlighting highlighting_flag:
0 = highlighting off
1 = highlighting on

void update_same_rbf_parameters(
    Input_Display_Manager* manager, Profile_Curve* curve)

Function Description:
When the parameters of a Rule-Based Fuselage component are modified,
this function sets the nose position and fuselage length of any other existing
profile curves equal to the nose position and fuselage length of the modified
profile curve.

Argument Description:
manager pointer to the Input/Display Manager class.
curve pointer to the modified profile curve.
void draw_profile_curves_structures(Input_Display_Manager* manager)

Function Description:
Draws the PHIGS structures of all three Rule-Based Fuselage profile curves
(or however many currently exist).

Argument Description:
manager pointer to the Input/Display Manager class.

Segment Class

Class Description:
Base class for the Linear Segment, Cubic Segment, and Fillet Segment derived classes.
Contains functions to create, modify, and display linear and cubic segments of profile
curves and piecewise cross-sections as well as functions to create, modify, and display the
points of linear and cubic segments.

Class Inheritance:
None.

Private Variables:

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int pick_id</td>
<td>Pick id of the segment PHIGS structure.</td>
</tr>
<tr>
<td>int segment_type</td>
<td>Type of segment: 1 = linear segment 2 = cubic segment</td>
</tr>
<tr>
<td>int segment_definition</td>
<td>Flag indicating whether or not a segment has been properly defined: 0 = segment not defined 1 = segment defined</td>
</tr>
<tr>
<td>int number_of_points</td>
<td>Number of points in the segment (always 2 for linear segments).</td>
</tr>
<tr>
<td>Point* first_point</td>
<td>Pointer to the first point of the segment.</td>
</tr>
<tr>
<td>Point* last_point</td>
<td>Pointer to the last point of the segment.</td>
</tr>
<tr>
<td>Segment* previous</td>
<td>Pointer to the previous segment in the segments linked list (maintained by a profile curve or piecewise cross-section).</td>
</tr>
<tr>
<td>Segment* next</td>
<td>Pointer to the next segment in the segments linked</td>
</tr>
</tbody>
</table>
int tangents_updating

Flag indicating automatic tangents updating:
0 = do not automatically update tangents
1 = automatically update tangents

int highlight_flag

Flag which indicates whether or not a segment PHIGS structure is to be drawn with highlighting color:
0 = highlighting off
1 = highlighting on

Protected Functions:

void delete_points()

Function Description:
Deletes the points of a segment. Used when the segment destructor is called.

Argument Description:
None.

void move_point( Input_Display_Manager* manager )

Function Description:
Allows the user to move the point of a segment to a new location.

Argument Description:
manager pointer to the Input/Display Manager class.

void update_move_point( Input_Display_Manager* manager, Point* point )

Function Description:
Updates the PHIGS structure of a segment after a point of the segment has been moved.

Argument Description:
manager pointer to the Input/Display Manager class.
point pointer to the segment point which was moved.

Public Functions:

void set_segment_first_point( Input_Display_Manager* manager, Point* point )

Function Description:
If the current segment to be created is not the first segment of the profile curve or piecewise cross-section, this function automatically sets the point values and tangents of the first point equal to the point values and tangents of the last point of the previous segment.
Argument Description:
manager pointer to the Input/Display Manager class.
point pointer to the first point of the segment to be created.

Segment* duplicate_segment( int reverse_flag )

Function Description:
Creates a new segment by duplicating a previously defined segment. The points linked list of the new segment may also be reversed depending on the given reverse flag. Returns a pointer to the new segment.

Argument Description:
reverse_flag flag indicating whether or not the points linked list of the new segment is to be reversed.

double move_segment_points_by_delta( float delta[] )

Function Description:
Changes the point values of all of the segment points by the specified delta amount. Used to move a segment to a new location.

Argument Description:
delta array of xyz delta amounts by which to segment point values are to be changed.

void scale_segment( float scaling_factor[] )

Function Description:
Scales all of the segment points by the specified scaling factor.

Argument Description:
scaling_factor array of xyz scaling factors by which the segment points are to be scaled.

void apply_matrix( float matrix[4][4] )

Function Description:
Applies a transformation matrix (scaling, rotation, or translation) to the segment.

Argument Description:
matrix 4 x 4 array of the transformation matrix.

void update_adjacent_segments()

Function Description:
After a segment has been modified, this function updates any existing adjacent segments to reflect the change in the modified segment. For example, if the first point of a modified segment has been moved, the last point of the previous segment (if it exists) must be moved to that same location.

Argument Description:
None.
void check_previous_segment( Input_Display_Manager* manager )

Function Description:
After a segment has been modified, this function checks to see if a previous segment exists. If so, the previous segment is updated to reflect the change in the modified segment and the PHIGS structure of the previous segment is modified.

Argument Description:
manager pointer to the Input/Display Manager class.

void check_next_segment( Input_Display_Manager* manager )

Function Description:
After a segment has been modified, this function checks to see if a next segment exists. If so, the next segment is updated to reflect the change in the modified segment and the PHIGS structure of the next segment is modified.

Argument Description:
manager pointer to the Input/Display Manager class.

void update_previous_segment()

Function Description:
After a segment has been modified, this function updates the previous segment to reflect the change in the modified segment.

Argument Description:
None.

void update_next_segment()

Function Description:
After a segment has been modified, this function updates the next segment to reflect the change in the modified segment.

Argument Description:
None.

void display_segment( Input_Display_Manager* manager )

Function Description:
Displays the segment PHIGS structure autocentered in the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void remove_segment( Input_Display_Manager* manager )

Function Description:
Removes the segment PHIGS structure from the current view.

Argument Description:
manager pointer to the Input/Display Manager class.
void delete_segment_structure()

Function Description:
Deletes the segment PHIGS structure and the PHIGS structures of the
segment points.

Argument Description:
None.

void set_segment_pick_id()

Function Description:
Sets the pick id of the segment PHIGS structure and adds the id to the
segment pick filter class.

Argument Description:
None.

void set_min_max_extrema(Input_Display_Manager* manager)

Function Description:
Sets the minimum and maximum point values of the segment by checking
the xyz values of each segment point. Used to autocenter profile curves and
piecewise cross-sections.

Argument Description:
manager pointer to the Input/Display Manager class.

void link_points(Point* previous, Point* current)

Function Description:
Links together the given points to update the segment points linked list. The
next pointer of the previous point is set equal to the current point and the
previous pointer of the current point is set equal to the previous point.

Argument Description:
previous pointer to the previous point of the current point.
current pointer to the current point.

Point* get_point_modify(Input_Display_Manager* manager, char message)

Function Description:
Returns a pointer to a segment point selected by the user.

Argument Description:
manager pointer to the Input/Display Manager class.
message character array containing a prompt to the user.

void draw_segment_points(Input_Display_Manager* manager)

Function Description:
Creates the PHIGS structures of all of the points of a segment.

Argument Description:
manager pointer to the Input/Display Manager class.
void display_segment_points( Input_Display_Manager* manager )

Function Description: Displays the PHIGS structures of all of the points of a segment in the current view to allow the segment points to be modified.

Argument Description: manager pointer to the Input/Display Manager class.

void remove_segment_points( Input_Display_Manager* manager )

Function Description: Removes the PHIGS structures of all of the points of a segment from the current view.

Argument Description: manager pointer to the Input/Display Manager class.

void delete_points_structures()

Function Description: Deletes the PHIGS structures of all of the points of a segment.

Argument Description: None.

void display_segment_end_points( Input_Display_Manager* manager )

Function Description: Displays the PHIGS structures of the endpoints of a segment in the current view.

Argument Description: manager pointer to the Input/Display Manager class.

void remove_segment_end_points( Input_Display_Manager* manager )

Function Description: Removes the PHIGS structures of the endpoints of a segment from the current view.

Argument Description: manager pointer to the Input/Display Manager class.

void display_segment_intermediate_points( Input_Display_Manager* manager )

Function Description: Displays the PHIGS structures of the intermediate points of a segment in the current view (cubic segments only).

Argument Description: manager pointer to the Input/Display Manager class.

void remove_segment_intermediate_points( Input_Display_Manager* manager )

Function Description: Removes the PHIGS structures of the intermediate points of a segment from
the current view (cubic segments only).

Argument Description:
manager pointer to the Input/Display Manager class.

Segment Manager Class

Class Description:
Maintains a linked list of segments for the Profile Curve and Piecewise Cross-Section classes. Contains data and functions used by these classes to create, modify, and display segments of profile curves and piecewise cross-sections.

Class Inheritance:
None.

Protected Variables:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>number_of_segments</td>
<td>Number of segments which define the profile curve or piecewise cross-section.</td>
</tr>
<tr>
<td>Segment*</td>
<td>first_segment</td>
<td>Pointer to the first segment of the profile curve or piecewise cross-section segments linked list.</td>
</tr>
<tr>
<td>Segment*</td>
<td>last_segment</td>
<td>Pointer to the last segment of the profile curve or piecewise cross-section segments linked list.</td>
</tr>
<tr>
<td>int</td>
<td>tangents_update</td>
<td>Flag indicating automatic tangents updating:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = do not automatically update tangents</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = automatically update tangents</td>
</tr>
</tbody>
</table>

Public Functions:

void link_segments( Segment* previous, Segment* current )

Function Description:
Links together the given segments to update the segments linked list. The next pointer of the previous segment is set equal to the current segment and the previous pointer of the current segment is set equal to the previous segment.

Argument Description:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>previous</td>
<td>pointer to the segment which is to be made the previous segment of the current segment.</td>
</tr>
<tr>
<td></td>
<td>current</td>
<td>pointer to the current segment.</td>
</tr>
</tbody>
</table>
void duplicate_segments( Segment* segment )

Function Description:
Duplicates the segments of a profile curve or piecewise cross-section to create a new profile curve or piecewise cross-section.

Argument Description:
segment pointer to the first segment of the profile curve or piecewise cross-section which is being duplicated.

void delete_segments()

Function Description:
Deletes all of the segments of a profile curve or piecewise cross-section, effectively deleting the curve or cross-section.

Argument Description:
None.

void delete_one_segment( Input_Display_Manager* manager )

Function Description:
Allows the user to delete a segment of a profile curve or piecewise cross-section. If the segment was the last segment, the curve or cross-section is effectively deleted. Note that if the deleted segment was between two other segments, the user must select which of the two segments is to be adjusted to close the gap created by the deleted segment.

Argument Description:
manager pointer to the Input/Display Manager class.

void add_segment( Input_Display_Manager* manager )

Function Description:
If the user wishes to add a segment to a profile curve or piecewise cross-section, this function determines if the segment is to be added to one end of the curve or cross-section or if the segment is to be added between two existing segments. If the curve or cross-section consists of only one segment, only the former option is allowable.

Argument Description:
manager pointer to the Input/Display Manager class.

void end_segment( Input_Display_Manager* manager )

Function Description:
Allows the user to add a segment to the end of a profile curve or piecewise cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.
int get_end_segment_flag( Segment* segment )

**Function Description:**
Determines to which end of the profile curve or piecewise cross-section the new segment is to be added if there is more than one segment. Returns a flag indicating which end.

**Argument Description:**
segment pointer to the first or last segment of the profile curve or piecewise cross-section.

---

int get_end_segment( Input_Display_Manager* manager )

**Function Description:**
Determines to which end of the profile curve or piecewise cross-section the new segment is to be added if there is only one segment. Returns a flag indicating which end.

**Argument Description:**
manager pointer to the Input/Display Manager class.

---

void between_segments( Input_Display_Manager* manager )

**Function Description:**
Allows the user to add a segment between two segments of a profile curve or piecewise cross-section.

**Argument Description:**
manager pointer to the Input/Display Manager class.

---

void move_segment( Input_Display_Manager* manager )

**Function Description:**
Allows the user to move a segment of a profile curve or piecewise cross-section to a new location either by moving a reference point on the segment or entering the xyz amount by which the segment is to be moved.

**Argument Description:**
manager pointer to the Input/Display Manager class.

---

void move_seg_pt_to_pt( Input_Display_Manager* manager, Segment* segment )

**Function Description:**
Allows the user to move a segment of a profile curve or piecewise cross-section by moving a reference point on the segment to a new location. The other segment points move by the amount that the reference point moved.

**Argument Description:**
manager pointer to the Input/Display Manager class.
segment pointer to the segment to be moved.
void \textbf{move\_seg\_by\_amount} \\
( \text{Input\_Display\_Manager* manager, Segment* segment} )

\textbf{Function Description:} \\
Allows the user to move a segment of a profile curve or piecewise cross-section by entering the xyz delta amount by which the segment is to be moved.

\textbf{Argument Description:} \\
manager \quad \text{pointer to the Input/Display Manager class.} \\
segment \quad \text{pointer to the segment to be moved.}

void \textbf{scale\_segments}( \text{float scaling\_factor[]} )

\textbf{Function Description:} \\
Scales all of the segments of a profile curve or piecewise cross-section by the specified scaling factor.

\textbf{Argument Description:} \\
scaling\_factor \quad \text{array containing the xyz scaling factors.}

void \textbf{scale\_one\_segment}( \text{Input\_Display\_Manager* manager} )

\textbf{Function Description:} \\
Allows the user to scale the points of a linear or cubic segment by entering the xyz scaling factors.

\textbf{Argument Description:} \\
manager \quad \text{pointer to the Input/Display Manager class.}

void \textbf{fillet\_segments}( \text{Input\_Display\_Manager* manager} )

\textbf{Function Description:} \\
Allows the user to create a fillet arc between two segments.

\textbf{Argument Description:} \\
manager \quad \text{pointer to the Input/Display Manager class.}

Segment* \textbf{get\_segment\_to\_modify}( \text{Input\_Display\_Manager* manager, char message[]} )

\textbf{Function Description:} \\
Returns a pointer to the segment selected by the user.

\textbf{Argument Description:} \\
manager \quad \text{pointer to the Input/Display Manager class.} \\
message \quad \text{character array containing a prompt to the user.}

void \textbf{apply\_matrix}( \text{Segment* segment, float matrix[4][4]} )

\textbf{Function Description:} \\
Applies a transformation matrix (scaling, rotation, or translation) to the segments of a profile curve or piecewise cross-section.

\textbf{Argument Description:} \\
segment \quad \text{pointer to the first segment of the profile curve or piecewise cross-section.}
matrix 4 x 4 array of the transformation matrix.

void update_segments_delete( Segment* segment, Segment* adjusted_segment )

Function Description:
Updates the segments linked list of a profile curve or piecewise cross-section after a segment has been deleted.

Argument Description:
segment pointer to the deleted segment.
adjusted_segment pointer to the segment which must be updated to fill the gap created by the deleted segment.

void update_add_segment( Segment* segment, Segment* segment_1, Segment* segment_2 )

Function Description:
Updates the segments linked list of a profile curve or piecewise cross-section after a segment has been added between two other segments by checking the pick ids of the other two segments.

Argument Description:
segment pointer to the added segment.
segment_1 pointer to the first of the other two segments.
segment_2 pointer to the second of the other two segments.

void draw_segments( Input_Display_Manager* manager )

Function Description:
Creates the PHIGS structures of all of the segments of a profile curve or piecewise cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.

void draw_segments_structures
( Input_Display_Manager* manager, int class_name )

Function Description:
Creates the PHIGS structures of all of the segments of a profile curve or piecewise cross-section to be inserted into the open structure of the curve or cross-section or into the open component structure.

Argument Description:
manager pointer to the Input/Display Manager class.
class_name pick filter of the class name to which the PHIGS structures pick ids are to be added.

void display_segments( Input_Display_Manager* manager )

Function Description:
Displays the PHIGS structures of all of the segments of a profile curve or piecewise cross-section autocentered in the current view.

Argument Description:
manager pointer to the Input/Display Manager class.
void remove_segments( Input_Display_Manager* manager )

Function Description:
Removes the PHIGS structures of all of the segments of a profile curve or piecewise cross-section from the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void delete_segments_structures()

Function Description:
Deletes the PHIGS structures of all of the segments of a profile curve or piecewise cross-section.

Argument Description:
None.

void display_segments_points( Input_Display_Manager* manager )

Function Description:
Displays the PHIGS structures of all of the points of all of the segments of a profile curve or piecewise cross-section in the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void remove_segments_points( Input_Display_Manager* manager )

Function Description:
Removes the PHIGS structures of all of the points of all of the segments of a profile curve or piecewise cross-section from the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void display_segments_end_points( Input_Display_Manager* manager )

Function Description:
Displays the PHIGS structures of the endpoints of all of the segments of a profile curve or piecewise cross-section in the current view.

Argument Description:
manager pointer to the Input/Display Manager class.

void remove_segments_end_points( Input_Display_Manager* manager )

Function Description:
Removes the PHIGS structures of the endpoints of all of the segments of a profile curve or piecewise cross-section from the current view.

Argument Description:
manager pointer to the Input/Display Manager class.
void process_tangents_update_menu()

Function Description:
Processes the TANGENTS main menu option which allows the user to change the automatic tangents updating to on or off.

Argument Description:
None.

Simple Cross-Section Class

Class Description:
Contains functions to create, modify, display, and orient simple cross-sections (rectangular, circular, and elliptical cross-sections).

Class Inheritance:
Cross-Section class.

Private Variables:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>simple_cross_section_type</td>
</tr>
<tr>
<td>float</td>
<td>cross_section_height</td>
</tr>
<tr>
<td>float</td>
<td>cross_section_width</td>
</tr>
<tr>
<td>float</td>
<td>rectangle_points[12]</td>
</tr>
<tr>
<td>float</td>
<td>cross_section_radius</td>
</tr>
<tr>
<td>float*</td>
<td>circle_points</td>
</tr>
<tr>
<td>float</td>
<td>cross_section_horizontal_radius</td>
</tr>
<tr>
<td>float</td>
<td>cross_section_vertical_radius</td>
</tr>
<tr>
<td>float*</td>
<td>ellipse_points</td>
</tr>
</tbody>
</table>

Type of simple cross-section:
1 = rectangle
2 = circle
3 = ellipse

Height of a rectangular cross-section.
Width of a rectangular cross-section.
Array of rectangular cross-section points.
Radius of a circular cross-section.
Dynamically allocated array of circular cross-section points.
Horizontal radius of an elliptical cross-section.
Vertical radius of an elliptical cross-section.
Dynamically allocated array of elliptical cross-section points.
Private Functions:

void create_rectangle_cross_section( Input_Display_Manager* manager )

Function Description:
Allows the user to create a rectangular cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.

void create_circle_cross_section( Input_Display_Manager* manager )

Function Description:
Allows the user to create a circular cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.

void create_ellipse_cross_section( Input_Display_Manager* manager )

Function Description:
Allows the user to create an elliptical cross-section.

Argument Description:

void draw_rectangle_cross_section()

Function Description:
Creates the PHIGS structure of a rectangular cross-section.

Argument Description:
None.

void draw_circle_cross_section()

Function Description:
Creates the PHIGS structure of a circular cross-section.

Argument Description:
None.

void draw_ellipse_cross_section()

Function Description:
Creates the PHIGS structure of an elliptical cross-section.

Argument Description:
None.

manager pointer to the Input/Display Manager class.

void mirror_z_plane()

Function Description:
Mirrors the simple cross-section about the Z axis.
Argument Description:
None.

Public Functions:

Simple_Cross_Section( int symmetry_flag, int spacing_flag )

Function Description:
Constructor. Initializes the cross-section symmetry and spacing method.

Argument Description:
symmetry_flag flag indicating the cross-section symmetry.
 spacing_flag flag indicating the cross-section spacing.

void create_cross_section( Input_Display_Manager* manager )

Function Description:
Allows the user to create a rectangular, circular, or elliptical cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.

void scale_cross_section( Input_Display_Manager* manager )

Function Description:
Creates a simple cross-section by scaling a previously defined cross-section, referred to as the scaling cross-section.

Argument Description:
manager pointer to the Input/Display Manager class.

void scale_cross_section( Input_Display_Manager* manager,
 Profile_Curve* crown_curve,
 Profile_Curve* keel_curve,
 Profile_Curve* plan_curve )

Function Description:
Overloaded function. Creates a simple cross-section by scaling a previously defined cross-section, referred to as the scaling cross-section for a Rule-Based Fuselage component.

Argument Description:
manager pointer to the Input/Display Manager class.
crown_curve pointer to the crown profile curve of a Rule-Based Fuselage component.
keel_curve pointer to the keel profile curve of a Rule-Based Fuselage component.
plan_curve pointer to the plan profile curve of a Rule-Based Fuselage component.

Cross_Section* copy_cross_section()

Function Description:
Creates a new simple cross-section by setting the simple cross-section parameters equal to the parameters of a previously defined cross-section.
based on the previous cross-section simple cross-section type. Returns a pointer to the new cross-section.

**Argument Description:**
None.

```c
void modify_cross_section( Input_Display_Manager* manager,
                          Component* component )
```

**Function Description:**
Allows the user to modify the parameters of a simple cross-section.

**Argument Description:**
- manager: pointer to the Input/Display Manager class.
- component: pointer to the component for which the cross-section is to be modified.

```c
void draw_cross_section()
```

**Function Description:**
Draws a simple cross-section based on the simple cross-section type.

**Argument Description:**
None.

```c
void set_min_max_extrema( Input_Display_Manager* manager )
```

**Function Description:**
Sets the minimum and maximum point values of the simple cross-section by checking the cross-section parameters. The minimum and maximum values must be modified based on the cross-section symmetry (always symmetrical about the y-axis). These values are used to autocenter the cross-section.

**Argument Description:**
- manager: pointer to the Input/Display Manager class.

```c
void check_cross_section_symmetry()
```

**Function Description:**
Checks the simple cross-section symmetry to ensure that the symmetry is about the y-axis.

**Argument Description:**
None.

```c
void create_surface_segments()
```

**Function Description:**
Overloaded function which creates the segments which define the cross-section surface and contain the cross-section points which are sent to ACSYNT to create the component surface model. The segments are created based on the simple cross-section type.

**Argument Description:**
None.
void create_surface_segments( int number_of_points, float points[] )

**Function Description:**
Overloaded function which creates linear segments to define the surface segments of a simple cross-section (see above).

**Argument Description:**
- number_of_points: number of cross-section points to be generated.
- points: array of points defining the simple cross-section.

void check_rbf_cross_section( Input_Display_Manager* manager,
                               Profile_Curve* crown_curve,
                               Profile_Curve* keel_curve,
                               Profile_Curve* plan_curve )

**Function Description:**
Ensures that the simple cross-section of a Rule-Based Fuselage component intersects the crown, keel, and plan profile curves at the cross-section depth.

**Argument Description:**
- manager: pointer to the Input/Display Manager class.
- crown_curve: pointer to the crown profile curve of a Rule-Based Fuselage component.
- keel_curve: pointer to the keel profile curve of a Rule-Based Fuselage component.
- plan_curve: pointer to the plan profile curve of a Rule-Based Fuselage component.

**Spine Curve Class**

**Class Description:**
Contains functions to create and modify the spine profile curve of a Spine and Cross-Section component.

**Class Inheritance:**
Profile Curve class.

**Public Functions:**

Spine_Curve()

**Function Description:**
Constructor. Initializes the spine profile curve parameters.

**Argument Description:**
None.
void create_profile_curve( Input_Display_Manager* manager,
                          Component* component )

Function Description:
Allows the user to create the segments of the spine profile curve.

Argument Description:
manager pointer to the Input/Display Manager class.
component pointer to the Spine and Cross-Section component for which
the spine profile curve is to be created.

void modify_profile_curve( Input_Display_Manager* manager )

Function Description:
Allows the user to modify the segments of the spine profile curve.

Argument Description:
manager pointer to the Input/Display Manager class.

Point* avg_spine_orient_point( Segment* segment, Point* point )

Function Description:
If the orientation point lies exactly on the endpoint of a linear segment
and the next point is also linear, this function calculates the tangent
vector by averaging the tangent vectors calculated using the last point of
the segment and the first point of the next segment (see Appendix C).
Returns a pointer to the orientation point.

Argument Description:
segment pointer to the linear segment on which the orientation point
lies.
point pointer to the point preceding the orientation point.

Spine and Cross-Section Class

Class Description:
Contains functions to create, modify, and display Spine and Cross-Section components
and functions to create and orient the cross-sections of Spine and Cross-Section
components.

Class Inheritance:
Component class.

Private Variables:
Profile_Curve* spine_curve pointer to the spine profile curve.
Private Functions:

```c
int check_create_component( Input_Display_Manager* manager )
```

**Function Description:**
When the user exits the create_component() function, this function checks to make sure that the component is properly defined (i.e. the spine profile curve has been created). If not, the user is prompted to return to create_component() or to cancel. Returns a flag indicating whether of not to exit the create_component() function.

**Argument Description:**
manager pointer to the Input/Display Manager class.

```c
void check_spine_curve( Input_Display_Manager* manager )
```

**Function Description:**
When the user selects the Create Spine Curve menu option in create_component(), this function checks to see if the spine profile curve of the current component has already been defined. If so, the user has the option of creating a new spine curve (thus deleting the current spine curve) or to cancel the menu selection.

**Argument Description:**
manager pointer to the Input/Display Manager class.

```c
void create_cross_sections( Input_Display_Manager* manager )
```

**Function Description:**
Allows the user to create the individual cross-sections of a Spine and Cross-Section component.

**Argument Description:**
manager pointer to the Input/Display Manager class.

```c
Cross_Section* create_cross_section_menu( Input_Display_Manager* manager )
```

**Function Description:**
Allows the user to select the type of cross-section to create or the change the current cross-section symmetry and/or spacing method. Returns a pointer to the cross-section to be created.

**Argument Description:**
manager pointer to the Input/Display Manager class.

```c
void initialize_cross_section( Input_Display_Manager* manager,
                              Cross_Section* cross_section )
```

**Function Description:**
Initializes the parameters of the given cross-section to be created (symmetry, spacing, and depth).

**Argument Description:**
manager pointer to the Input/Display Manager class.
cross_section pointer to the cross-section to be created.
void auto_cross_section( Input_Display_Manager* manager,
                       Cross_Section* cross_section )

Function Description:
Creates a cross-section automatically by scaling a previously defined
cross-section, referred to as the scaling cross-section.

Argument Description:
manager                  pointer to the Input/Display Manager class.
cross_section            pointer to the cross-section to be created.

void scale_cross_section( Input_Display_Manager* manager,
                          Cross_Section* cross_section )

Function Description:
Creates the given cross-section using the parameters of the scaling
cross-section.

Argument Description:
manager                  pointer to the Input/Display Manager class.
cross_section            pointer to the cross-section to be created.

void check_cross_section_depth( Input_Display_Manager* manager,
                                 Cross_Section* cross_section )

Function Description:
Checks to see if the depth input/calculated for the current cross-section is
valid (i.e. the depth value lies on the spine profile curve and no
cross-section already exists at that depth). If the depth is invalid, the user
is informed and the cross-section is not created.

Argument Description:
manager                  pointer to the Input/Display Manager class.
cross_section            pointer to the cross-section to be created.

void orient_cross_section( Input_Display_Manager* manager,
                          Cross_Section* cross_section )

Function Description:
Orients the given cross-section normal to the spine profile curve.

Argument Description:
manager                  pointer to the Input/Display Manager class.
cross_section            pointer to the cross-section to be oriented.

Public Functions:

Spine_and_Cross_Section( int component_number )

Function Description:
Constructor. Initializes the Spine and Cross-Section component
parameters.
Argument Description:  
component_number  ACSYNT component identifier.

void create_component( Input_Display_Manager* manager )

Function Description:  
Allows the user to create or modify a Spine and Cross-Section component  
(i.e. create/modify spine curve and cross-sections).

Argument Description:  
manager  pointer to the Input/Display Manager class.

Component* copy_component( int component_number )

Function Description:  
Creates a new Spine and Cross-Section component by duplicating an  
existing component.

Argument Description:  
component_number  ACSYNT component identifier.

void draw_component( Input_Display_Manager* manager )

Function Description:  
Creates the component PHIGS structure by creating the PHIGS structures  
of the spine curve and cross-sections (if they exist).

Argument Description:  
manager  pointer to the Input/Display Manager class.

void delete_component_structure()

Function Description:  
Deletes the component PHIGS structure and deletes the PHIGS structures  
of the spine curve and cross-sections (if they exist).

Argument Description:  
None.

void set_min_max_extrema( Input_Display_Manager* manager )

Function Description:  
Sets the minimum and maximum point values of the component by checking  
the minimum and maximum values of the spine profile curve and any  
existing cross-sections. These values are used to autocenter the component.

Argument Description:  
manager  pointer to the Input/Display Manager class.

void highlight_component( Input_Display_Manager* manager,  
int highlighting )

Function Description:  
Turns the highlighting of a component on or off depending on the specified  
highlighting flag.
**Argument Description:**

- **manager**: Pointer to the Input/Display Manager class.
- **highlighting_flag**:
  - 0 = highlighting off
  - 1 = highlighting on

**View Manager Class**

**Class Description:**
Contains the system viewing data and function required to display PHIGS structures.
Access to these functions is provided via pointer to the Input/Display Manager class.

**Class Inheritance:**
Menu Manager class.

**Private Variables:**

Note: Simple_View is a class defined in the interface framework which encapsulates a 3D PHIGS view [Woya92].

- **Simple_View**
  - **full_front_view**: Pointer to the full display of the front view.
  - **full_side_view**: Pointer to the full display of the side view.
  - **full_top_view**: Pointer to the full display of the top view.
  - **full_iso_view**: Pointer to the full display of the isometric view.

- **Simple_View**
  - **front_view**: Pointer to the front view of the 4-view display (lower right-hand corner).
  - **side_view**: Pointer to the side view of the 4-view display (lower left-hand corner).
  - **top_view**: Pointer to the top view of the 4-view display (upper left-hand corner).
  - **iso_view**: Pointer to the side view of the 4-view display (upper right-hand corner).

- **float**
  - **x_minimum**: Minimum viewport x-value.
  - **x_maximum**: Maximum viewport x-value.
  - **y_minimum**: Minimum viewport y-value.
float y_maximum  Maximum viewport y-value.
float z_minimum  Minimum viewport z-value.
float z_maximum  Maximum viewport z-value.
Point* view_translation  Pointer to the current viewing translation.
float grid_horizontal_minimum  Minimum horizontal grid value based on the current viewport dimensions. Used to initialize the parameters of a grid.
float grid_horizontal_maximum  Maximum horizontal grid value based on the current viewport dimensions. Used to initialize the parameters of a grid.

Protected Variables:

int current_view  Flag indicating the current active view:
  0 = 4-view display
  1 = front view
  2 = side view
  3 = top view
  4 = isometric view

int current_plane  Flag indicating the current active plane:
  0 = 4-view display plane
  1 = XY-plane
  2 = YZ-plane
  3 = XZ-plane
  4 = isometric plane

Public Functions:

View_Manager()

Function Description:
Constructor. Initializes the system PHIGS views.

Argument Description:
None.

void profile_curve_view( int active_flag )

Function Description:
Activates/deactivates the profile curve view (front view) depending on the active flag.

Argument Description:
active_flag flag:
  0 = deactivate view
  1 = activate view
void cross_section_view( int active_flag )

Function Description:
Activates/deactivates the cross-section view (side view) depending on the active flag.

Argument Description:
active_flag flag:
0 = deactivate view
1 = activate view

void plan_curve_view( int active_flag )

Function Description:
Activates/deactivates the plan profile curve view (top view) depending on the active flag.

Argument Description:
active_flag flag:
0 = deactivate view
1 = activate view

void isometric_view( int active_flag )

Function Description:
Activates/deactivates the isometric view depending on the active flag.

Argument Description:
active_flag flag:
0 = deactivate view
1 = activate view

void component_view( int active_flag )

Function Description:
Activates/deactivates the component view (4-view display) depending on the active flag.

Argument Description:
active_flag flag:
0 = deactivate view
1 = activate view

void set_view_parameters( Simple_View* view, int view_id )

Function Description:
Sets the viewing transformation and translation matrices of the given view.

Argument Description:
view pointer to the view for which the transformation and translation matrices are to be set.
view_id view id of the view for which the transformation and translation matrices are to be set.
void deactivate_current_view()

Function Description:
Deactivates the current active view. Used for toggling between different views.

Argument Description:
None.

void autocenter_display()

Function Description:
Based on the current active view, this function determines which views must be autocentered. If the current view is the 4-view display, for example, all 4 views must be autocentered.

Argument Description:
None.

void autocenter_view()

Function Description:
Autocenters the current active view based on the minimum and maximum point values of the geometry within that view. Also sets the grid horizontal minimum and maximum values.

Argument Description:
None.

void initialize_min_max_extrema()

Function Description:
Initializes the minimum and maximum point values of the geometry within the current view. Used in conjunction with the set_min_max_extrema() function for the geometry to be autocentered.

Argument Description:
None.

void display_structure( PHIGS_Structure_ID* struct_id[] )

Function Description:
Displays the given structure in all of the system views at once. The structure is displayed in all views in order to allow the user to toggle between the system views. Note that a copy of the PHIGS structure is required for each view since standard PHIGS does not allow a structure to be displayed in more than one view at a time.

Argument Description:
struct_id pointer to the array of PHIGS structure ids.

void remove_structure( PHIGS_Structure_ID* struct_id[] )

Function Description:
Removes the given structure from all of the system views.
Argument Description:
struct id pointer to the array of PHIGS structure ids.
Appendix C. Cross-Section Orientation Point
For a Spine and Cross-Section component, the orientation point is the point about which the cross-section is oriented normal to the spine profile curve. For a Rule-Based Fuselage component, the orientation point is used to determine the points at which the cross-section intersects the crown, keel, and plan profile curves. These intersection points are referred to as the crown point, keel point, and plan point, respectively.

The sequence of functions used to determine the xyz values and tangent vector of the orientation point are described in detail using pseudo code. A given profile curve determines the orientation point for the current cross-section as follows:

```c
Profile_Curve* curve; // Pointer to the profile curve.
Cross_Section* cross_section; // Pointer to the current cross-section.
Point* point; // Pointer to the orientation point.

point = curve->orientation_point( cross_section );
```

The orientation_point() function traverses the segments linked list of the profile curve and determines in which segment and between which two points of the segment the orientation point lies:

```c
Point* Profile_Curve::orientation_point( Cross_Section* cross_section )
{
    // Cross-Section depth
    float depth = cross_section->get_depth();

    // Pointer to the orientation point.
    Point* orient_pt;

    // Pointer to the point preceding the orientation point.
    Point* point;

    // Pointer to the 1st segment of the profile curve.
    Segment* segment = get_first_segment();

    // Traverse the segments linked list.
    while ( segment != NULL )
    {
        // Pointer to the 1st point of the segment.
        Point* point = segment->get_first_point();
```
// Traverse the points linked list.
while ( point != NULL )
{
    // Special case if the cross-section depth exactly equals the x-value of a
    // point on the segment:
    if ( point x-value = depth )
    {
        if ( current segment type is Linear and the current component
type is Rule-Based Fuselage )
        {
            orient_pt = special_linear_orient_point( point );
        }

        if ( current segment type is Linear and the current component
type is Spine and Cross-Section )
        {
            orient_pt = avg_spine_orient_point( segment, point );
        }

        if ( current segment type is Cubic )
        {
            orient_pt xyz values = point xyz values;
            orient_pt tangent vector = point tangent vector;
        }
    }

    // Cross-section depth lies between two segment points:
    else if ( point x-value > depth )
    {
        point = point->get_previous();

        if ( current segment type is Linear )
        {
            orient_pt = linear_orient_point( cross_section, point );
        }

        if ( current segment type is Cubic )
        {
            orient_pt =
            segment->cubic_orient_pt( cross_section, point );
        }

        point = point->get_next();
    }

    segment = segment->get_next();
}

return orient_pt;

As illustrated by the pseudo code, the orientation_point() function calls the appropriate function to
determine the orientation point. Following are the descriptions of each of these functions using
If the orientation point lies exactly on one of the points of a linear segment of a Rule-Based Fuselage profile curve, the following function is used:

```cpp
Point* Profile_Curve::special_linear_orient_point( Point* point )
{
    // Call the constructor for a new point to allocate memory for the orientation point.
    Point* orient_pt = new Point();

    // Create a dummy cubic segment consisting of the 1st point of the linear segment and
    // the orientation point.
    Segment* segment = new Cubic_Segment();

    // Calculate the tangent vector of the orientation point.
    Point* orient_tangent =
        segment->linear_orient_tangents( point, orient_point );

    // Set the values of the tangent vector of the orientation point.
    orient_pt->set_calculated_tangent(
        DX, orient_tangent->get_calculated_tangent( DX ) );

    return orient_pt;
}
```

If the orientation point lies exactly on one of the points of a linear segment of a Spine and Cross-Section profile curve, the avg_spine_orient_point function() is used:

```cpp
Point* Spine_Curve::avg_spine_orient_point( Segment* segment, Point* point )
{
    // Pointer to the next segment.
    Segment* next_segment = segment->get_next();

    // Pointer to the first point of the next segment.
    Point* first_pt = next_segment->get_first_point();

    // Call the constructor for a new point to allocate memory for the orientation point.
    Point* orient_pt = new Point();

    // If the point is the last point of the current segment and the next segment is also
    // linear, the tangent vector of the orientation point must be the average of the
    // tangent vectors calculated using the last point of the current segment and the first
    // point of the next segment.
    if ( (point->get_next() == NULL) && (next_segment->get_type() == LINEAR) )
    {
        // Calculate the two orientation points.
        Point* orient_pt_1 = special_linear_orient_point( point );
        Point* orient_pt_2 = special_linear_orient_point( first_pt );
    }
```
// Average the tangent vectors of the two orientation points.
orient_pt tangent vector =
( orient_pt_1 tangent vector + orient_pt_2 tangent vector ) / 2;
}

// Otherwise, use the special_linear_orient_point() function with the given point.
else
{
orient_pt = special_linear_orient_point( point );
}

return orient_pt;

If the orientation point lies exactly on one of the points of a cubic segment, the orientation xyz values and tangent vector are simply set equal to the corresponding values of that cubic segment point.

If the orientation point lies between two points of a linear segment, the following function is used:

Point* Profile_Curve::linear_orient_point
( Cross_Section* cross_section, Point* point )
{
    // Call the constructor for a new point to allocate memory for the orientation point.
    Point* orient_pt = new Point();

    // Get the xyz values of the 1st point.
    float x1 = point->get_value( X );
    .
    .

    point = point->get_next();

    // Get the xyz values of the 2nd point.
    float x2 = point->get_value( X );
    .
    .

    // Use these values to calculate the slope of the line.
    float x-slope = x-slope of linear segment;
    .
    .

    // Use the slope to determine the xyz values of the orientation point.
    point = point->get_first_point();

    orient_pt x-value = cross-section depth;
    orient_pt y-value = y-slope + point y-value;
    orient_pt z-value = z-slope + point z-value;
// Set the input tangents of the orientation point equal to the slope.
orient_pt->set_input_tangent( DX, x-value of slope );

// Create a dummy cubic segment consisting of the 1st point of the linear segment and
// the orientation point.
Segment* segment = new Cubic_Segment();

// Calculate the tangent vector of the orientation point.
Point* orient_tangent =
    segment->linear_orient_tangents( point, orient_point );

// Set the values of the tangent vector of the orientation point.
orient_pt->set_calculated_tangent
    ( DX, orient_tangent->get_calculated_tangent( DX ) );

return orient_pt;
}

If the orientation point lies between two points of a cubic segment, the following function is used:

Point* Cubic_Segment::cubic_orient_point
    ( Cross_Section* cross_section, Point* point )
{
    // The intermediate cubic segment points for the interval in which the orientation
    // point lies must be calculated (see Appendix C). The orientation point is determined
    // by comparing the x-value of each intermediate point with the cross-section depth.

    // Local cubic Hermite spline parameter (see Appendix D).
    float u;

    // Step size for the local cubic Hermite spline parameter.
    float step_size = 0.1;

    // Convergence criterion.
    float epsilon = 0.0001;

    // Number of points in the interval.
    int num_pts;

    // Call the constructor for a new point to allocate memory for the orientation point.
    Point* orient_pt = new Point();

    // Process loop until solution converges:
    while ( solution not converged )
    {
        u = 0;
        num_pts = 1/step_size + 1;

        // Calculate the intermediate points in the interval:
        for ( int i = 1; i <= num_pts; i++ )
        {
            Point* intermediate_point = calculate_points( point, u, ... );
// Check for convergence:
if ( abs( cross-section depth - intermediate_point x-value) < epsilon )
{
    // Solution converged.
    orient_pt = intermediate_point;
}
else
{
    // Solution not converged - reduce the step size by half, thereby
    // increasing the number of points in the interval.
    step_size = step_size / 2;
}
u = u + 0.1;

// Calculate the tangent vector of the orientation point.
Point* orient_tangent = cubic_orient_tangents( point, orient_point );

// Set the values of the tangent vector of the orientation point.
orient_pt->set_calculated_tangent
( DX, orient_tangent->get_calculated_tangent( DX ) );

return orient_pt;

The functions linear_orient_tangents() and cubic_orient_tangents() create the points linked list for
a dummy cubic segment which is used to calculate the tangent vector (using the
calculate_tangents() function) of the orientation point (see Appendix D, Cubic Segment class).

Note: the preceding code is only pseudo code. It is incomplete and is intended only for the
purpose of explaining the functions in greater detail than found in the class descriptions
of Chapter 10 and Appendix B.
The cubic segments created within the system consist of three-dimensional, planar cubic Hermite splines which pass through the cubic segment junction points specified by the user. The cubic splines are piecewise $C^2$ continuous if no input tangents (slopes) are specified by the user. The cubic spline is only $C^1$ continuous at a given junction point if the tangent (slope) is specified at that junction point.

The cubic splines use quadratic end conditions in which a parabola is passed through the first three points. The derivative of this parabola is the calculated tangent vector which is then normalized.

**Cubic Spline Parameters**

Figure 35 on page 320 shows the parameters of a cubic spline. The parameters are defined as follows:

- $X_n, Y_n, Z_n$: Coordinates of point $n$.
- $DX_n, DY_n, DZ_n$: Components of the tangent vector to the spline at point $n$. The magnitude of the vector is:

$$ (DX_n^2 + DY_n^2 + DZ_n^2)^{1/2} \times PL_n $$

Generally,

$$ DX_n^2 + DY_n^2 + DZ_n^2 = 1 $$

- $PL_n$: The chord length (Euclidean distance) between point $n$ and point $n + 1$. Note that for an $n$-bay spline, $n + 1$ points must be defined, with the last point of the spline having an associated chord length of zero.
Figure 35. Parametric Cubic Spline for Bay n.
Evaluating the Knot (Breakpoint) Sequence

The knot sequence \( t_1, t_2, ..., t_n \) of the cubic Hermite spline is the accumulated parametric length:

Let
\[
\begin{align*}
t_1 &= 0 \\
t_i &= t_{i-1} + PL_{i-1}, \quad i = 1, 2, 3, ..., n + 1
\end{align*}
\]

Evaluating the Spline

To evaluate the spline \( S \) at \( t \) (i.e. calculate the points between the given cubic spline points \( X_i, Y_i, Z_i \) and \( X_{i+1}, Y_{i+1}, Z_{i+1} \)) when \( t_i \leq t \leq t_{i+1} \):

Let \( u = (t-t_i) / PL_i \)

Then,
\[
\begin{align*}
S_x(t) &= a_0(u)X_i + a_1(u)X_{i+1} + (b_0(u)DX_i + b_1(u)DX_{i+1})PL_i \\
S_y(t) &= a_0(u)Y_i + a_1(u)Y_{i+1} + (b_0(u)DY_i + b_1(u)DY_{i+1})PL_i \\
S_z(t) &= a_0(u)Z_i + a_1(u)Z_{i+1} + (b_0(u)DZ_i + b_1(u)DZ_{i+1})PL_i
\end{align*}
\]

where
\[
\begin{align*}
a_0(u) &= 2u^3 - 3u^2 + 1 \\
a_1(u) &= -2u^3 + 3u^2 \\
b_0(u) &= u^3 - 2u^2 + u \\
b_1(u) &= u^3 - u^2
\end{align*}
\]

are the normalized cubic Hermite blending functions.

The values of \( u \) range from 0 to 1. The increment of \( u \) determines the number of points to be
calculated between each pair of cubic spline points.

The DX, DY, and DZ values at each point of the cubic spline are calculated using the cubic_segment_tangents() functions listed in Appendix E.
Appendix E. cubic_segment_tangents() Source Listing
The cubic_segment_tangents() function is an overloaded function which calculates the tangent vectors of the cubic segments used in this system. The first overloaded function calculates the tangent vectors for the special case of a cubic segment with only two points. If no input tangents (slopes) are specified by the user in this case, the cubic segment becomes a straight line. The second overloaded function calculates the tangent vectors for cubic segments with three or more points.

```cpp
void Cubic_Segment::cubic_segment_tangents( Point* p, int spec )
{
    float D[3][2], V[3][2], S[3];
    float DIST, T;
    float VSUM, V1SUM, VNSUM;
    int I, J, K, L, N = 1;

    Point* point = p;

    // Load the point arrays.
    for ( int pt = 0; pt <= N; pt++ )
    {
        S[pt] = p->get_chord_length();
        D[0][pt] = p->get_value(X);
        D[1][pt] = p->get_value(Y);
        D[2][pt] = p->get_value(Z);
        V[0][pt] = p->get_input_tangent(DX);
        V[1][pt] = p->get_input_tangent(DY);
        V[2][pt] = p->get_input_tangent(DZ);
        p = p->get_next();
    }
```
if ( ( spec == 1 ) || ( spec == 3 ) || ( spec == 4 ) )
{
    // If no tangents are specified, compute a straight line.
    //
    if ( spec == 1 )
    {
        for ( L = 0; L <= 1; L++ )
        {
            for ( I = 0; I <= 2; I++ )
            {
                V[I][L] = D[I][1] - D[I][0];
            }
        }
    }
    else
    {
        // Calculate a tangent if only one tangent is specified.
        //
        if ( spec == 3 )
        {
            J = 0;
            K = 1;
        }
        if ( spec == 4 )
        {
            J = 1;
            K = 0;
        }

        // Form chord dot end vector.
        //
        T = 0.0;
        DIST = 0.0;

        for ( I = 0; I <= 2; I++ )
        {
            T = T + ( D[I][1] - D[I][0] ) * V[I][J];
            DIST = DIST + ( D[I][1] - D[I][0] ) * ( D[I][1] - D[I][0] );
        }

        if ( fabs( (double) T ) >= EPSILON )
        {
            T = 0.05 * sqrt( (double) DIST ) / T;
            for ( I = 0; I <= 2; I++ )
            {
                V[I][K] = ( D[I][1] - D[I][0] );
            }
        }
    }
}
else
{
    for ( I = 0; I <= 2; I++ )
    {
        V[I][K] = -V[I][J];
    }
}

// Normalize end tangents.
//
V1SUM = 0.0;
VNSUM = 0.0;

for ( I = 0; I <= 2; I++ )
{
    V1SUM = V1SUM + V[I][0] * V[I][0];
    VNSUM = VNSUM + V[I][N] * V[I][N];
}

V1SUM = (float) sqrt( (double) V1SUM );
VNSUM = (float) sqrt( (double) VNSUM );

for ( I = 0; I <= 2; I++ )
{
    V[I][0] = V[I][0] / V1SUM;
    V[I][N] = V[I][N] / VNSUM;
}

// Set the tangent vectors of the cubic segment points.
//
for ( I = 0; I <= 1; I++ )
{
    point->set_calculated_tangent( DX, V[0][I] );
    point->set_calculated_tangent( DY, V[1][I] );
    point->set_calculated_tangent( DZ, V[2][I] );
    point = point->get_next();
}

// Variable Definitions:
//
// p         pointer to the first point of the cubic segment portion.
// spec      tangent specification of the cubic segment portion.
//            1 = no tangents specified
//            2 = all tangents specified
//            3 = first tangent specified
//            4 = last tangent specified
//            5 = first and last tangent specified
// N         number of cubic segment portion points.
void Cubic_Segment::cubic_segment_tangents(Point* p, int N, int spec)
{
    float DIST, F;
    float VSUM, V1SUM, VNSUM;
    int I, ID, J, K, L;
    Point* point = p;

    float **D = new float*[3];
    float **V = new float*[3];

    for (int i = 0; i <= 2; i++)
    {
        D[i] = new float[N+1];
        V[i] = new float[N+1];
    }

    float *S = new float[N+1];
    float *A = new float[N+1];
    float *B = new float[N+1];
    float *R = new float[N+1];

    for (int pt = 0; pt <= N; pt++)
    {
        S[pt] = p->get_chord_length();
        D[0][pt] = p->get_value(X);
        D[1][pt] = p->get_value(Y);
        D[2][pt] = p->get_value(Z);
        V[0][pt] = p->get_input_tangent(DX);
        V[1][pt] = p->get_input_tangent(DY);
        V[2][pt] = p->get_input_tangent(DZ);
        p = p->get_next();
    }

    int NM1 = N - 1;
    int NM2 = N - 2;
    float T = S[0] / (S[0] + S[1]);
    float TSQR = T * T;
    float TN = S[NM1] / (S[NM1] + S[NM2]);
    float TNSQR = TN * TN;
// If either or both end tangents are not specified, calculate their tangents here.

if ( spec != 5 )
{
    for ( I = 0; I <= 2; I++ )
    {
        if ( spec != 3 )
        {
            V[I][0] = ( D[I][2] * TSQR + D[I][0] * ( 1.0 - TSQR ) - D[I][1] ) / ( TSQR - T );
        }
        if ( spec != 4 )
        {
            V[I][N] = -(( D[I][NM2] * TNSQR + D[I][N] * ( 1.0 - TNSQR ) - D[I][NM1] ) / ( TNSQR - TN ));
        }
    }
}

// Normalize end tangents.
V1SUM = 0.0;
VNSUM = 0.0;

for ( I = 0; I <= 2; I++ )
{
    V1SUM = V1SUM + V[I][0] * V[I][0];
    VNSUM = VNSUM + V[I][N] * V[I][N];
}

V1SUM = (float) sqrt( (double) V1SUM );
VNSUM = (float) sqrt( (double) VNSUM );

for ( I = 0; I <= 2; I++ )
{
    V[I][0] = V[I][0] / V1SUM;
    V[I][N] = V[I][N] / VNSUM;
}

// Calculate intermediate tangents.
R[0] = 0.0;

for ( I = 0; I <= 2; I++ )
{
    for ( ID = 0; ID <= NM1; ID++ )
    {
        A[ID] = D[ID][ID+1] - D[ID][ID];
        B[ID] = 3.0 * A[ID];
    }
}
\[ B[NM1] = B[NM1] - V[I][N] \times S[NM1]; \]

for ( \( ID = 1; ID \leq NM1; ID++ \) )
{ 
    \[ R[ID] = S[ID-1] / S[ID]; \]
    \[ V[I][ID] = 2.0 \times B[ID-1] - 3.0 \times A[ID-1] + R[ID] \times R[ID] \times B[ID]; \]
}

\[ V[I][1] = V[I][1] = V[I][0] \times S[0]; \]

if ( \( NM1 \neq 1 \) )
{ 
    for ( \( ID = 1; ID \leq NM2; ID++ \) )
    { 
        \[ F = 2.0 \times R[ID] \times (1.0 + R[ID]) - R[ID-1]; \]
        for ( \( I = 0; I \leq 2; I++ \) )
        { 
            \[ V[I][ID] = V[I][ID] / F; \]
            \[ V[I][ID+1] = V[I][ID+1] - V[I][ID]; \]
        }
    }
    \[ R[ID] = R[ID] \times R[ID] \times R[ID+1] / F; \]
}

\[ F = (2.0 + 2.0 \times R[NM1]) \times R[NM1] - R[NM2]; \]

for ( \( L = 0; L \leq 2; L++ \) )
{ 
    \[ V[L][NM1] = V[L][NM1] / F; \]
}

if ( \( NM1 \neq 1 \) )
{ 
    for ( \( K = 1; K \leq NM2; K++ \) )
    { 
        \[ ID = NM2 + 1 - K; \]
        for ( \( I = 0; I \leq 2; I++ \) )
        { 
            \[ V[I][ID] = V[I][ID] \times V[I][ID+1] \times R[ID]; \]
        }
    }
// Apply the homogeneous coordinate to the tangents.  
// for ( I = 0; I <= 2; I++ )
{
    for ( ID = 1; ID <= NM1; ID++ )
    {
        V[I][ID] = V[I][ID] / S[ID];
    }
}

// Set the tangent vectors of the cubic segment points.  
// for ( I = 0; I <= 1; I++ )
{
    point->set_calculated_tangent( DX, V[0][I] );
    point->set_calculated_tangent( DY, V[1][I] );
    point->set_calculated_tangent( DZ, V[2][I] );

    point = point->get_next();
}
Vita

John Kelly was born December 26, 1969 in Washington, D.C. and currently lives in Gaithersburg, Maryland. John attended Virginia Tech for both his undergraduate and graduate studies because of the reputation that the university has in the field of engineering. Music is among one of his favorite hobbies and John has been a member of the Virginia Tech marching band, The Marching Virginians, for six seasons as a trombone player, tuba player, and manager. During his undergraduate career, he met a fellow Marching Virginian manager, Rebecca Kacenjar, whom he married on Halloween of 1992.

John H. Kelly