An Object-Oriented, PHIGS-based Internal Layout Module for Aircraft Design

by

Shahab Hasan

thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Mechanical Engineering

APPROVED:

Dr. Arvid Myklebust, Chairman

Dr. J. H. Bøhn

Dr. W.H. Mason

December 13, 1993

Blacksburg, Virginia
An Object-Oriented, PHIGS-based Internal Layout Module for Aircraft Design

by Shahab Hasan

Committee Chairman: Dr. Arvid Myklebust, Professor of Mechanical Engineering

ABSTRACT

During the conceptual design phase, aircraft designers require tools to be able to quickly and accurately produce concepts that meet given requirements. These tools often take the form of computer programs and computer-aided design systems which model the shape of the airplane concept and perform various analyses. Often, a common element in these analyses is that only the exterior of the aircraft is considered. Thus a major component of aircraft design is largely overlooked: the internal layout of the airplane.

This thesis describes the design and implementation of a new computer software, the Internal Layout Module, designed to enhance the ability of the aircraft designer to arrange, modify, view, and analyze the internal components of an airplane. By considering the internal layout concurrently with the external shape and size, the configurator can effectively design the smallest airplane that meets specifications.

The module is coded in C++ entirely under the object-oriented paradigm to ease integration with existing code and to ensure future maintainability and extensibility. New components as varied as seats, galleys, lavatories and cargo containers have been designed for use in the Internal Layout Module and to provide a foundation for future object-oriented geometry for aircraft design.

The module has been successfully integrated with the aircraft conceptual design code ACSYNT (AirCraft SYNThesis). The process of adding this new object-oriented module to existing procedural code is discussed in detail.
Acknowledgments

The Family
Throughout my life I have received as much love and support from my family as anyone could hope for. Nowhere has this been as evident as in the area of my educational endeavors. I thank my parents, Zamir and Shaheen Hasan, for instilling in me the love of learning and an appreciation for education. And, while I readily admit that at heart I am a very lazy person, my parents also helped me develop a strong work ethic. I am often forced to fall back upon that strong work ethic after I am overpowered by an even stronger habit of procrastination. I also thank my two sisters, Hena and Saba Hasan, for their support. They have always been generous with their advice and opinions to me and even listen to what I have to say once in a while (just kidding!).

The Faculty
When virtually the entire university seemed to have given up on me and my own confidence was waveri ng, I did have some powerful allies in my corner. I thank my advisor Dr. Arvid Myklebust for believing in me in that fateful summer of 1991, helping me get accepted into the graduate program and for his continued guidance since then. I thank Dr. Sankar Jayaram for initially giving me the opportunity to prove myself by working in the CAD Lab (based on a recommendation by Eric Shrock, now at Lockheed) and I wish Dr. Jayaram and his family all the best at Washington State University. And I thank Dr. William Mason for all he has done for me ever since I was a student in his Aircraft Design class. Dr. Mason has the dubious honor of first introducing me to the wonderful world of ACSYNT.
It is no coincidence, of course, that Dr. Myklebust, Dr. Jayaram, and Dr. Mason served on my advisory committee but I would be deeply indebted to them even if this were not the case. They have given me friendship, guidance, and a tremendous education. Finally, I would be remiss if I did not thank Dr. Jan Helge Bøhn for agreeing to serve on my committee so close to my program's completion.

The Financiers

My parents deserve credit in this department as well; they made sure I never wanted for anything. I appreciate and acknowledge the financial support I received from Dr. Myklebust in the form of a Graduate Project Assistantship and his willingness to send me to several conferences and short courses over the last few years. Paul Gelhausen also is due thanks for providing me a position at NASA Ames Research Center through the graduate cooperative education program during the latter half of 1993.

The Friends

I think that by now I might have gone insane without all the friends I have made through the years. Of course, those very same friends might argue that they weren't enough to stave off mental illness entirely.

Scott Woyak: Thanks for listening to me ramble on about sports and other topics when you wanted to get back to what you were working on. Thanks for all the help with the Interface Framework, linked lists, PHIGS, and numerous other things that you understand so much better than me (or anyone else). Thanks also to Laura Woyak for her friendship and hospitality. I hope you two will continue to consider me a good friend no matter how far apart our paths may diverge in the future.
Brett Malone: A fellow "airplane guy" -- there seem to be so few of us left. Thanks for exchanging endless movie quotes with me and struggling to identify all those old Q-99 songs (we really need to get a stereo in the lab that picks up some good stations). Thanks also for discussing NASCAR and fishing (can I consider myself an honorary redneck?), and arguing the merits of The Washington Post vs. The Roanoke Times & World News. By the way, I hope I never see another "Family Circus" comic strip in my life.

Alan Jacobson: A former roommate. Thanks for putting up with my company for a whole year. We shared a lot at Virginia Tech, I'll never forget it. Good luck in all your future endeavors, I'm sure they will be many and varied.

Russell Conger: Another former roommate. Russ, we both know you are a guitarist extraordinaire and engineer par excellence (no, I am not being sarcastic). Don't worry, sooner or later, the rest of the world will realize it too. You and I went through so many good and bad times together, I wouldn't trade them for anything. I will never forget those times when we couldn't help laughing at what only you and I found funny.

And I'd like to thank everyone else who worked in the lab during my tenure there: Jan Aarsaether, Marti Altizer, Scott Angster, Tom Arledge, David Coe, Darrell Early, Michele Grieshaber, Uma Jayaram, John Kelly, Krishnan Kolady, Kerry McClure, Dee (and Amy and Grace) Meier, John Nafziger, Parasuram Narayanan, Augustin Reyna, Francisco Rivera, Doug Squire, Andreas Steude, Jie Wen Tjiang, and Steve Uhorchak. I consider myself very fortunate to have been associated with so many talented and personable individuals.
Table of Contents

Acknowledgments........................................................................................................... iii
    The Family ............................................................................................................. iii
    The Faculty ......................................................................................................... i
    The Financiers....................................................................................................... iv
The Friends................................................................................................................... iv
Table of Contents........................................................................................................ vi
List of Illustrations...................................................................................................... ix
Introduction .................................................................................................................. 1
    Aircraft Design .................................................................................................... 1
    Internal Layout Design ....................................................................................... 6
    ACSYNT ............................................................................................................... 7
    PHIGS .................................................................................................................. 8
Object-Oriented Design and Programming ................................................................. 10
    Graphical User Interface .................................................................................. 13
Literature Review ......................................................................................................... 16
    Aircraft Conceptual Design .............................................................................. 16
    Aircraft Conceptual Design Programs ............................................................. 17
    Object-Oriented Design and Programming ..................................................... 22
Thesis Objectives ......................................................................................................... 24
Overview of Program Design ..................................................................................... 26
    Design for Ease of Integration ......................................................................... 26
    User Interaction .................................................................................................. 27
    Screen Layout ..................................................................................................... 28
Class Structure and Brief Class Descriptions .............................................................. 30
    ACSYNT_Component class ............................................................................. 36
    Aft.Body class .................................................................................................. 36
    Canard class ...................................................................................................... 36
    Component_Manager class ............................................................................. 36
    File_Manager class ......................................................................................... 37
    Galley class ........................................................................................................ 37
    Galley_Manager class ...................................................................................... 37
    Horizontal.Tail class ....................................................................................... 37
    Input_Field class ............................................................................................... 38
    Internal.Layout class ........................................................................................ 38
    Lavatory class ................................................................................................... 38
    Lavstory_Manager class ................................................................................... 38
    LD3 class ............................................................................................................ 38
    Mid.Section class ............................................................................................... 39
    Nose class ............................................................................................................ 39
"Move/Modify Lavatories" pushbutton ...........................................64
"Next Section" pushbutton ..........................................................63
"Previous Section" pushbutton ......................................................63
"Goto Section" field .................................................................63

Results and Conclusions ................................................................65
Tools Used ..................................................................................69
References ..................................................................................71
Appendix A: Detailed Class Definitions ...........................................78
Appendix B: Source code file "acs_seat.vpi" .....................................148
Vita .............................................................................................175
List of Illustrations

Figure 1  Breakdown of the Phases of Aircraft Design from [Raym89] ..........................4
Figure 2  Breakdown of the Phases of Aircraft Design from [Ohsu89] ..........................5
Figure 3  Photograph of the Screen Layout ..................................................................29
Figure 4  Key to Booch Notation .................................................................................32
Figure 5  Class Structure Using Booch Notation - 1 ....................................................33
Figure 6  Class Structure Using Booch Notation - 2 ....................................................34
Figure 7  Class Structure Using Booch Notation - 3 ....................................................35
Figure 8  The Seat Component ....................................................................................51
Figure 9  The Parametric Box Component ....................................................................52
Figure 10 Photograph of the Totals Display ..................................................................64
Figure 11 Sample Results ............................................................................................68
Introduction

The Internal Layout Module is concerned with several different topics. This section provides some background into these topics, specifically: aircraft design, internal layout design, ACSYNT, PHIGS, object-oriented design and programming, and graphical user interfaces.

Aircraft Design

Aircraft design is commonly viewed as occurring in three phases: conceptual, preliminary, and detail [Raym89]. This notion is echoed by Ohsuga [Ohsu89] who also adds a fourth phase - manufacturing and testing - that Raymer calls simply "fabrication" but does not include as a phase of design.

Conceptual design is the stage when the airplane is first being conceived. The requirements for the design are determined (or given, as in a Request for Proposal), and the design team goes through a brainstorming process to come up with several possible designs. These initial designs undergo first-order analysis and are modified and/or eliminated. At this point, basic characteristics of the aircraft are determined: an approximate takeoff weight, wing shape (swept or straight, aspect ratio, span, etc.), tail geometry and configuration (e.g. aft horizontal tail or canard, single or twin vertical tails), number and thrust class of the engines, and many others. However, while these type of basic decisions are consequently adhered to, many details are still free to change. So items such as the location of the vertical
tain, the exact degree of wing sweep, and size of control surfaces are not fixed at this step. The conceptual design phase is marked by the airplane being drawn and re-drawn, sized and re-sized, and analyzed and re-analyzed. Concepts are rapidly considered and thrown out and the designer is free to explore many ideas because he or she has a "clean sheet of paper". It is important for the design team to have available tools that allow for rapid turn-around time with analysis that accurately predicts the airplane's characteristics and performance.

Preliminary design begins once the major decisions about the concept have been made. There is much less re-draw of the airplane although minor revisions are still allowed. Toward the end of preliminary design all changes to the geometry are stopped and the design is considered "frozen". The analysis is now of a higher order and more people are involved. Specialists in the various systems of the airplane, such as structures, landing gear, and controls, proceed in their areas of expertise. Because the analysis is now more in-depth, the aircraft geometry must be accurately modeled and this mathematical model must be usable by the detail design CAD systems used later in the design process. Therefore, an important activity is "lofting" - the mathematical modeling of the outside skin of the aircraft. Finally, before proceeding to detail design, an accurate cost estimation must be made. It is imperative that the company is confident in their cost estimates. Raymer elaborates [Raym89]:

"...the end of preliminary design usually involves a full-scale development proposal. In today's environment, this can result in a situation jokingly referred to as "you-bet-your-company." The possible loss on an overrun contract or from lack of sales can exceed the net worth of the company!"

Detail design involves the most personnel and the most time in the design process. The actual pieces to be fabricated are designed. For instance, the wing box will be broken down
into ribs, spars, and skins, and each must be designed and analyzed. A subset of detail design is production design. Production design involves the determination of manufacturing processes, fabrication techniques and more. Finally, detail design ends with the actual fabrication of the aircraft.

Figures 1 and 2 summarize how Raymer [Raym89] and Ohsuga [Ohsu89] breakdown the various components of design.

The Internal Layout Module created in this study is meant to aid in the conceptual phase of aircraft design. It allows the designer to quickly explore many configuration options for the internal components and therefore to choose the optimum size and shape of the airplane early in the design process.
Figure 1 - Breakdown of the Phases of Aircraft Design - from [Raym89]
Conceptual Design:
- initial plan
- performance estimation
- stability and maneuverability
- aerodynamic characteristics and load estimation
- rough structural analysis
- principal dimension

Preliminary Design:
- aerodynamic design (lift distribution, airfoil design, three-dimensional wing design, aerodynamic performance estimation/confirmation)
- airplane motion analysis and simulation
- range of pilot's vision
- structural analysis

Detail Design:
- Master-Dimension (MD) system
- various engineering computations
- various drawings (assemblies, parts, metal plates, piping and tubing, wire harnesses, circuit diagrams, etc.)
- and so forth

Manufacturing and Testing:
- numerical control machine data generation and production
- quality assurance data generation and test
- process control
- and so forth

Figure 2 - Breakdown of the Phases of Aircraft Design - from [Ohsu89]
Internal Layout Design

As part of the conceptual phase of aircraft design, the configuration designer faces the task of determining the internal layout of the airplane. This includes arranging the components that must be carried internally in a fashion that maximizes use of the available space. Pending the results of this layout, decisions can be made regarding the adequacy of the design. The designer may come to one of three conclusions:

1. It is necessary to alter the exterior shape of the design to either eliminate any wasted space or to provide more room.
2. The payload requirements should be reconsidered (if they are not yet fixed).
3. The ideal shape has been achieved and the design requires no further modification.

This is often an iterative process as the designer considers various internal arrangements and exterior shapes.

For commercial transport aircraft, the primary payload is, of course, passengers. When laying out this type of airplane, the designer is concerned with placing the seats throughout the cabin as well as lavatories and galleys. The parameters used include seat width, seat pitch (defined as the distance from the back of one seat to the back of the next, including the length of the seat itself and leg room), aisle width, number of passengers per lavatory, and galley volume per passenger. The actual arrangement is often dictated by marketing, such as when a certain seating capacity has been promised or a passenger comfort level (in terms of parameters like seat pitch and seat width) has been agreed upon [Raym89].
ACSNT

The ACSYNT (AirCraft SYNthesis) program is a multidisciplinary design and analysis tool for conceptual aircraft design. NASA Ames Research Center first began development of ACSYNT in the early 1970's and stressed the importance of modular, flexible code. Thus the building blocks of the analysis are modules for geometry, aerodynamics, propulsion, weights, trajectory, sonic boom, stability and control, takeoff and landing, and economics. Another objective of the program from the outset was the incorporation of non-linear optimization to enable constrained optimization and sensitivity studies.

Since 1986, NASA has worked with the Computer-Aided Design Laboratory at Virginia Tech to develop a transportable, interactive, three-dimensional computer-aided design version of ACSYNT which adds a graphical user interface and generates parametric 3-D surface models. To ensure portability, all graphics are performed using only PHIGS (discussed in detail below), the International Standards Organization (ISO) standard for 3-D graphics, thus making the program machine and graphics-device independent.

Development of ACSYNT is ongoing and is administered through the ACSYNT Institute, which was formed in 1990 and is comprised of representatives from government, academia, and U.S. industry. The history of the development of ACSYNT is well-documented in references [Wamp88], [Jaya92], and [Myki93].

The Internal Layout Module is one of the most recent additions to ACSYNT and provides new functionality to both the analysis and computer-aided design sides of the program. Other recent additions to ACSYNT include a graphical user interface based on C++
[Woya92, Woya93], redesigned mission input specification [Rive93], rule-based fuselage
design [Kell93], improved data graphing [Uhor93], pilot's eye view analysis [McCl93], and
integration of an interface to the Navy/NASA Engine Program (NNEP) [Steu93].

**PHIGS**

PHIGS is important enough to the Internal Layout Module to warrant a place in the title of
this thesis. It therefore also requires a section to describe its general features and advantages.

The definitive reference on PHIGS is *The PHIGS Programming Manual* by Tom Gaskins
[Gask92]. The description given here is culled mainly from the Preface and Introduction
sections of that extraordinary book.

PHIGS (Programmer's Hierarchical Interactive Graphics System) is a programming library
for 3D graphics consisting of over 400 functions. It is commonly called an Applications
Programming Interface (API) which conveys the fact that PHIGS is not an application unto
itself, rather it is a specification for a collection of routines meant to facilitate the addition of
graphics to programs. PHIGS is available on a wide variety of computing platforms from
mainframes to personal computers which makes it appealing for applications that must run
identically on different computers. It was approved by the American National Institute of
Standards (ANSI) and subsequently chosen as the standard for 3D graphics by the
The PHIGS library contains functions for many varied tasks. Among these are functions for creating graphics primitives (such as a line), setting attributes (such as color), and controlling the display (such as updating the screen). By providing such a large library of functions, PHIGS can be used to support a range of applications from the very simple (static line drawings) to the very complex (animation with sophisticated lighting and shading -- using PHIGS+).

PHIGS is a high-level library and thus is concerned with abstractions like geometric objects and color. This is in contrast to low-level libraries which require the programmer to manipulate entities such as pixels on the display hardware. This distinction between low-level and high-level libraries extends to the handling of input devices. PHIGS provides logical input devices which are idealized ways of viewing and controlling real, physical input devices such as a keyboard or mouse. PHIGS implementations then use real input devices to support the logical devices. The application (and therefore the applications programmer) does not need to be concerned with which real device is used; it deals with the logical one.

Another important feature of PHIGS is that it is a "display list" system. The programmer builds the geometric primitives which are then stored internally by PHIGS. The primitives can then be displayed on the screen, edited, or discarded. This is in contrast to the other common type of graphics library, the "immediate mode" system in which primitives are sent to the display device immediately and then essentially forgotten by the system. To redisplay, or modify the primitive(s), the entire picture must be re-created and re-sent. There are advantages and disadvantages to both approaches. Recent extensions have resulted in limited "immediate mode" capability in PHIGS.
The Internal Layout Module (and ACSYNT overall) utilizes PHIGS primarily because of the portability gained. For example, initial development of the new module was carried out on IBM workstations using the IBM implementation "graPHIGS". The bulk of the work was then shifted to Silicon Graphics machines running an implementation from Template Graphics Software, "Figaro+". Recently development continued on the IBM platforms. Because the implementations more or less follow the PHIGS and PHIGS+ standards, very few modifications were necessary. This is not to say that the process was completely painless, but the problems that were encountered arose when the vendors of the different implementations chose different means of meeting the standard. Even these problems could probably be eliminated if the standard were more stringent in outlining requirements instead of leaving many options open to the vendors.

Object-Oriented Design and Programming

"Give a power drill to a carpenter who knows nothing about electricity, and he would use it as a hammer. He will end up bending quite a few nails and smashing several fingers, for a power drill makes a lousy hammer." [Booc91]

The quote above is intended to emphasize the importance of understanding the philosophy and power of the object-oriented paradigm before attempting to apply it. The analogy is to a programmer attempting to use C++ in a traditional, procedural manner.
This section gives an overview of object-oriented design and programming mainly through descriptions of its basic concepts.

To start with, here are two definitions:

"Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships." [Booc91]

"[Object-oriented programming] asks first about the intent of the program: the what, not the how. Its goals are to find the objects and their connections; to do so, it ascertains what operations need to be performed and what information results from those operations. It then apportions responsibility for those operations and that information to objects. Each object knows how to perform its own operations and remember its own information. Object-oriented design decomposes a system into entities that know how to play their roles with the system; they know how to be themselves." [Wirf90]

The remainder of this section discusses the main concepts of object-oriented programming.

**Classes / Objects / Instances**

The basic building blocks in object-oriented programming (OOP) are, not surprisingly, what are called objects. This is in contrast to procedural programming (in a language like FORTRAN) in which the basic building blocks are functions (subroutines in FORTRAN syntax). But in actual practice, when using OOP the programmer does not define objects, rather he or she defines classes. A class is a collection of related data and functions grouped together for a purpose. For example, to model a tree, the programmer defines a class called "tree" in which data such as age, diameter, height, and color would be stored as well as functions such as "perform photosynthesis" and "shed leaves". Then to use the class, the programmer creates an instance of the class which is termed an object. Thus an object of the "tree" class would be instantiated and called something like "my_tree". The object
"my_tree" could then be used as necessary by setting its data, asking it to reveal its data, and telling it to perform its various functions.

A program is made up of as many classes and as many instances (objects) of those classes as necessary to model the problem domain. The key then becomes how the objects interact with each other which in the literature is often referred to as message passing.

**Encapsulation / Data-Hiding**

As discussed above, a class groups together data and functions. This is the essence of encapsulation. As Wirfs-Brock et al. put it, "it allows you to draw a circle around related ideas and operations, and say, 'These things belong together.'" [Wirf90]. Data-hiding is the precept that all objects should not know everything about each other. That is, there must be a public interface and a private representation for each class. The public interface is what is used when a "tree" object is asked to "perform_photosynthesis". The private representation contains the actual function and the details of how that process is carried out. The requesting object is not concerned about these details, it just wants results. This trait of classes greatly adds to the maintainability of a system. If a particular method or function in a class is to be improved or replaced, the public interface can remain the same and no changes are necessary to the rest of the system.

**Inheritance**

Inheritance refers to the relationship between classes that allows hierarchies to be built. A derived class inherits all the data and functions from a base class and then adds any additional data and functions specific to that derived class. For instance, at a marina a program might be developed to keep track of all the various vessels. The base class could be called "ship" or "boat" or "vessel" or a similar descriptive term. This class would contain data and functions relevant to all water-traveling vehicles. Then derived classes would
inherit from the base class and be called, for example, "battleship", "yacht", "bass boat", etc. There can be as many layers in the hierarchy as necessary. So, there may be a class between the "ship" class and the "battleship" class called "military ship". At each level of the hierarchy, additional information and methods may be added as appropriate. Also possible is multiple inheritance where a derived class inherits from more than one base class. The derived class then inherits all the data and functions from all the base classes.

**Polymorphism**

Polymorphism refers to the ability of two or more classes to contain functions of the same name that perform different (but usually similar) actions. In the preceding example involving ships, each derived class could have a function called "calculate_weight". Then the program could sum the weight of all the ships in the marina by asking each type of ship to "calculate_weight" and adding it to the total. Each class may have different methods to actually calculate its weight but the program is not burdened with this fact; it makes the request identically to each ship.

**Graphical User Interface**

The graphical user interface (GUI) for the Internal Layout Module uses a PHIGS-based "Motif-like" object-oriented interface framework developed by Scott Woyak at Virginia Tech [Woya92], [Woya93]. This section explains what Motif is, lists Motif's limitations, and describes Woyak's interface framework.
Motif is a widely used interface tool kit for Unix workstations that runs on top of the X windowing system. Since X is available on many platforms, Motif gains the portability and other benefits of X. Motif provides 2D windows and menus. Widgets (push buttons, sliders, radio buttons, etc.) and callbacks used for user input. Because of the popularity of Motif, new program applications that use it for the GUI have a familiar "look-and-feel" that users do not have to re-learn.

However, there are strong limitations to using Motif, particularly with programs that feature 3D graphics. This is the case with many engineering programs and certainly with CAD systems. Specifically, some of the major drawbacks to Motif are [Woya92]:

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

Motif shares the X limitation of working in a 2D environment and creating displays in terms of pixels and bitmaps. Three-dimensional graphics are extremely difficult to implement in this manner.

Woyak's interface framework offers an appealing alternative to Motif. It was chosen for use in this project because it is designed to support 3D graphics and because it is extensible. The framework is written using PHIGS (discussed in detail above) so 3D graphics are naturally supported and portability is ensured. It is also important that the framework is fully object-oriented. This means that if the exact feature desired is not provided by the framework, it can be added by the applications developer. For instance, for the Internal Layout Module, an
"input_field" class was created which combined the existing framework classes "text_input", "label", and "frame" to meet specific needs of the application.

A complete documentation of the interface framework is contained in [Woya92] and [Woya93].
Literature Review

This thesis was undertaken after a thorough review of previously published work in several fields. This section briefly examines books, papers, user's guides, and other sources that influenced the thinking and development of the Internal Layout Module.

Aircraft Conceptual Design

A number of references were consulted regarding aircraft conceptual design, with special attention paid to information on internal layout.

The book Aircraft Design: A Conceptual Approach is used as a textbook in many college design courses [Raym89]. It distinguishes between design layout and design analysis. First-order analytical methods are presented covering virtually all phases of conceptual design including configuration layout, payload considerations, aerodynamics, propulsion, structure and loads, weights, stability and control, handling qualities, performance, cost analysis, tradeoff studies, and optimization.

A section is included on designing the passenger compartment which explains the various parameters that are typically used. Also given is a table of data with representative values for those parameters.

Another commonly used design textbook is Fundamentals of Aircraft Design [Nico84]. This book also presents several first-order methods covering the various phases of conceptual
design. Included are typical internal components that influence fuselage size and internal configuration.

The series of books, *Airplane Design, Volumes I - VIII*, originating from the University of Kansas offer a comprehensive look at the design process from conceptual design techniques all the way to fabrication including a series of steps to follow [Rosk89]. They include vast amounts of data and statistics gleaned from historical trends of existing aircraft and thus the books have somewhat of a handbook flavor.

Finally, the book *Synthesis of Subsonic Airplane Design* gives an introduction to the preliminary design of subsonic general aviation and transport aircraft, with emphasis on layout, aerodynamic design, propulsion and performance [Tore82]. This book gives a good overview of the design process and the pros and cons of different ideas. Particularly valuable were discussions of passenger comfort level in terms of volume per passenger and of specifications in terms of Federal Aviation Regulations (FAR).

**Aircraft Conceptual Design Programs**

The aerospace industry is recognized as being among the first to actively use CAD professionally. The first major program to be used was developed at Lockheed in 1965, the Computer graphics Augmented Design And Manufacturing system (CADAM) [Bil88]. Subsequently, around 1980, the Avions Marcel Dassault-Breguet Aviation company developed the Computer-graphics Aided Three-dimensional Interactive Application system (CATIA) [Bil88]. These two programs remain the most widely used CAD systems in
aerospace and are used primarily for the detail design phase. They are both, however, not specifically tailored for or limited to aircraft design; they are general purpose CAD systems. As such, there are no direct analysis capabilities in either program.

A step toward a CAD system specifically for aircraft configuration design and layout was taken in 1982 by Rockwell International. The Configuration Design System (CDS) allows the creation of three-dimensional lofted surface aircraft components which may have up to 45 lines of numerical data associated with them [Raym82a]. CDS is now a part of the Integrated Design and Analysis System (IDAS) and has been renamed the Configuration Definition Module (CDM) [Raym82b]. CDM prepares detailed models for analysis by using 3-D solids defined by stacked parallel cross sections. This model then interfaces to the analysis modules of IDAS. However, the model and the components are not parametrically defined and the interface is command driven, thus making it more difficult for the aircraft designer to modify the concept.

HESCOMP is a preliminary design tool for helicopter sizing and performance originally developed at Boeing Vertol. In 1986, an interface, named HESCAD, was developed to create geometric models from the numeric output of HESCOMP [Mykl87]. By using this system, the user can make an assessment of mission equipment packages (such as avionics) on the geometry, weight, and performance of the helicopter. Orthographic views and a 3-D wireframe are produced and weights, volumes, and mass moments and products of inertia are computable from the model for feedback to HESCOMP.

General Dynamics developed the Advanced Computer Aided Design program (ACAD) which is a three-dimensional drafting system that focuses on the wireframe modeling needs
of the aircraft designer [Lock92]. Once the surface model is created, it is prepared by ACAD to be exported to analysis routines outside of ACAD. There are no inherent analysis capabilities.

The Advanced Aircraft Analysis (AAA) program was developed by Jan Roskam based on his series of Airplane Design books [Rosk92]. The emphasis is on first-order methods (as in his books) with the computer automating the calculations. There is no 3-D geometry capability and the program is intended primarily for educational use.

RDS-Student by Daniel P. Raymer bears similarity to Roskam's AAA in that it is based on Raymer's design textbook and also contains mostly first-order analysis methods [Raym92]. However, there is also a 3-D CAD module for design layout included which interfaces to the analysis methods. The CAD module is limited in several ways. There is no component library; each component must be created by manually defining one side of a cross section and then reflecting it to complete the drawing. The components are not surfaced which precludes exporting the model to other, higher order analysis programs. Finally, the components are defined by a very limited set of parameters which hamper modification. An enhanced version for industry usage is also available called RDS-Professional.

The Aircraft Design and Analysis System (ADAS) was developed at Delft University, the Netherlands in 1988 [Bil88]. This system allows the placement of seats throughout the aircraft cabin but it is important to note that the seats are represented only in 2-D, they are not 3-D surfaced components such as the Internal Layout Module of ACSYNT described in this thesis. No parameters can be associated with the seats in ADAS either.
The Douglas Aircraft Company of McDonnell Douglas Corporation has developed a Configuration Layout System which addresses many of the same problems as this thesis work [Ande88]. The Configuration Layout System was developed for use in the conceptual and preliminary phases of commercial aircraft design and is meant to bridge the gap between a CAD data base and several analytical programs. The program allows the designer to enter the number of seats and the aisle width(s) and select the type of seats from a library. The seats are then translated into place and the program creates a minimum clearance arc around them. A unit load device (ULD) is similarly selected and placed and the program creates a second arc. The two arcs meet at the fuselage floor and if they differ in diameter, a "double bubble" shape is created. The designer may smooth the resulting cusp at the intersection.

In comparing this Configuration Layout System with the Internal Layout Module of ACSYNT described in this thesis, the Douglas system offers more capability in cross section design. The capabilities, and even the process, of seat layout are very similar between the two. There are important differences, however, in the geometry. The Douglas system only creates wireframe geometry and the Unigraphics II (UGII) CAD system (developed and marketed by McDonnell Douglas) is used in conjunction with the Configuration Layout System for non-parametric component creation and for all surfacing of models. The possibility of 3-D surfaced seats is mentioned as a future goal of the system. The ACSYNT Internal Layout Module offers fully surfaced parametric components, including seats. The other important distinction, perhaps the most important, is that the Douglas Configuration Layout System extracts the geometric data and prepares it for transfer to outside analysis programs. There is no analysis done by the system itself. In this respect, it is very similar to ACAD of General Dynamics and CDM of Rockwell (both discussed above) in that analysis takes place outside of the CAD system. One of the primary features of ACSYNT is that it integrates CAD capabilities with analysis routines in one system.
Nonetheless, the Configuration Layout System of Douglas Aircraft offers many powerful features. It was an important and valuable reference for this thesis work.

The V/STOL Aircraft Sizing and Performance Computer Program (VASCOMP) was developed by the Boeing Vertol Company, originally in 1966, and has undergone several revisions since then [Scho89]. The program can be used to define design requirements such as weight breakdown, required propulsive power, and physical dimensions of aircraft which are designed to meet specified mission requirements. It is also capable of aiding in sensitivity studies. VASCOMP is a batch mode program and there is therefore no associated CAD system or graphical user interface.

The program has options to allow the user to specify number of passengers, seat width, number of aisles, aisle width and number of seats abreast. These parameters are set for tourist class and first class service. Data on galley size and lavatory size is also entered and the program is able to return the required size of the fuselage based on all the entered information. Cross section shaping is semi-automated by a system of control points that define clearances; it is limited to circular cross sections.

The General Aviation Synthesis Program (GASP) provides a method of cross section shaping similar to VASCOMP but uses fewer control points and is less flexible [Wyat77].

The AirCraft SYNThesis (ACSYNT) program has been described in detail in the "Introduction" section of this thesis. More information is available in [Wamp88], [Jaya92a], [Jaya92b], and [Mykl93].
Object-Oriented Design and Programming

The "Introduction" section of this thesis gave a detailed look at object-oriented design and programming. Here, the reader is provided with references for further study and background information.

The book Object Oriented Design with Applications by Grady Booch [Booc91] is rapidly becoming a standard in this field. Booch spends much of his time traveling the country conducting seminars and short courses and is in heavy demand. The book provides a solid explanation of what object oriented design is all about and a notation for documenting object-oriented systems. This notation is used in this thesis due to its rising popularity. The book and the notation are independent of any particular programming language. This is at once a boon and a bane. It makes the book and notation useful to anyone practicing object-oriented programming but because the examples in the book alternate between five popular languages, most readers can only relate to one-fifth of them.

The book Designing Object-Oriented Software by Rebecca Wirfs-Brock [Wirf90] offers an alternative to Booch or it can be used as a complement as was the case for this thesis project. This book too is independent of any one programming language but because the examples are in pseudocode, no reader is disadvantaged. The book features a notation that is much less rigid and comprehensive than the Booch notation. However, it provides a much more detailed method of creating a system. The authors guide the reader through each step and prescribe lengthy documentation throughout. This book is best for the beginner in the field as it is much easier reading than Booch.
Finally, the *C++ Primer Plus* by Stephen Prata [Prat91] provides a reference for C++ syntax as well as an overview of object-oriented programming in general. This book was referred to most often during actual coding of the Internal Layout Module. The book is exceptionally readable due to a liberal sprinkling of humor.
Thesis Objectives

This thesis work was undertaken to fill a perceived need: to enhance the ability of ACSYNT to perform internal layout design. This section lists the specific goals that were hoped to be accomplished. The goals are not in any particular order.

Create new components necessary for internal layout

ACSYNT did not provide seats, lavatories, galleys, or cargo containers or a means for designing the aircraft interior with these components. The aim was to create these components, parametrically defined, following the existing ACSYNT geometry routines.

Model components as objects

The goal was to take a first step toward the much larger, long-term objective of object-oriented geometry for aircraft design, especially in ACSYNT. By creating classes and a class hierarchy for the current ACSYNT components, the advantages and disadvantages, as well as the potential difficulties of the object-oriented approach could be realized. The scope was limited to creating classes for individual components; the larger problem of treating the collected components as an "airplane" class was only examined peripherally.

Demonstrate the feasibility and usefulness of Woyak's GUI framework

This was not a major goal for the Internal Layout Module but it was recognized as important. The GUI framework will likely be a major part of the ACSYNT of the future and its development at this early stage is critical. The advantages gained from the object-oriented nature of the framework were looked at carefully.
Provide easy to use methods of placing seats throughout the aircraft cabin

It was recognized that the Internal Layout Module would primarily be used to layout seats for passenger aircraft. The goal was to make this process flexible and easy.

Demonstrate integration of new C++ code with existing ACSYNT code

The goal was to identify the problems associated with multiple programming languages compiled together to combine procedural code with object-oriented code and to solve those problems.
Overview of Program Design

This section describes the general philosophy behind the design of the Internal Layout Module. Included are discussions of integration with aircraft design programs, user interaction, and screen layout.

Design for Ease of Integration

The Internal Layout Module was designed such that it can integrated with any aircraft design program. The sole stipulation is that the 3D graphics of the program must utilize PHIGS. If this requirement is met, the module is able to retrieve the PHIGS structures used to draw the various aircraft components. The Internal Layout Module itself does not create geometry "from scratch". However, once the PHIGS structures have been retrieved from the invoking program, the module is then capable of modifying the geometry in any fashion that the original program provides. The changed geometry is then sent back to the calling program where the changes are reflected visually and in that program's data structures.

To enable the module to retrieve the PHIGS structures and the information associated with the aircraft components, the only modification necessary to the Internal Layout Module is to the data transfer functions ("bridge" functions) used to retrieve information about the components. These data transfer functions must be tailored to the data structures used by the aircraft design program. These functions are also responsible for cross-language translation of data, i.e. FORTRAN to C++.
The section "ACSYNT Integration", later in this thesis, discusses the integration of the Internal Layout Module with the aircraft design program ACSYNT. A further explanation of data transfer ("bridge") functions is included in that section.

User Interaction

It was recognized early in the design of the Internal Layout Module that it would likely be used primarily for designing the internal arrangement of commercial transport aircraft. When laying out this type of aircraft with seats, lavatories, and galleys, the designer generally divides the cabin into various classes, e.g. First Class, Business Class, and Economy Class. To emulate this kind of division, the Internal Layout Module allows the user to work on one "Section" of the airplane at a time. In a particular Section, the user can specify a name (e.g. First Class), input the starting and ending locations of the Section (or the starting location and a length), and fill the Section with seats, lavatories, and galleys. The number of Sections that can be defined by the user is not limited. This is important because to achieve a high degree of detail, the user may need more than the three Sections suggested above (First Class, Business Class, and Economy Class); the user can subdivide those Sections as necessary.

Of the three components types typically used in laying out the passenger cabin (i.e. seats, lavatories, and galleys), the seats are the most important. Therefore, all the pertinent parameters relating to seat placement -- number of aisles, aisle width, configuration (e.g. 2-3-
2), pitch, number of seats -- are on-screen at all times. There are not many levels of menus that must be traversed to modify these characteristics.

Finally, attention was paid to the visual feedback provided to the user. For internal layout, the designer needs an overhead view of the passenger cabin as well as a cross-sectional view. These are furnished by the Internal Layout Module along with an isometric view. Furthermore, functions are provided to modify the views including scaling, rotating, translating, and "rubber-band" zooming of the geometry displayed. The last item, "rubber-band" zooming refers to using the mouse to stretch a box around a portion of the displayed geometry. The geometry within the box is then enlarged to fill the view.

**Screen Layout**

Figure 3 shows a photograph of the Internal Layout Module. There are three views available for the aircraft geometry (no geometry is displayed in the photograph). The top view is on the left side of the screen and is the largest of the three views. Across the top right side of the screen are the front view (cross section) and an isometric view. The rest of the screen is occupied by the "Workspace" which contains the menu items that allow the user to interact with the program. The upper left corner of the Workspace contains pushbuttons for file input/output, exiting the program, and help. Also included in this area is a label identifying the data file currently in use. The upper right corner of the Workspace contains labels identifying how many sections and decks have been defined. The center portion of the Workspace contains the menu items relating to sections of the aircraft. The data for one Section is displayed here at any one time.
Figure 3 - Photograph of the Screen Layout
Class Structure and Brief Class Descriptions

This section describes, primarily through diagrams, how the class structure of the Internal Layout Module is organized. Also given are brief descriptions of each class; complete definitions are included in Appendix A.

Booch notation [Booc91] was chosen to illustrate the relationships between the various classes created. A key to this notation is given in Figure 4. The actual Booch diagrams are shown in Figures 5, 6, and 7. Following are descriptions of how the classes interact among each other. Later in this section, brief descriptions are given for each class itself.

Figure 5 shows a top-level diagram of the major classes and the relationships between them. When the Internal Layout Module is invoked, the first action that takes place is the instantiation or creation of an object of type "Internal/Layout". When this object is created, it automatically sets up the screen layout including the views of the geometry. It then creates objects of type "Workspace" and "Component_Manager". The Workspace object derives from the "Totals_Manager" and "File_Manager" classes which are responsible for maintaining total statistics and managing file input/output, respectively. The Workspace object also manages the "Sections", this is shown in the next Booch diagram and is discussed below. The Component_Manager object queries ACSYNT to determine which components make up the current airplane model. It then creates objects corresponding to each component type.
Figure 6 breaks down how the "Workspace" class works. The Workspace object creates and destroys "Sections" as requested by the user. A "Section" is a delimitation of a portion of the aircraft fuselage that is to be worked upon. Thus if the user wants to work on the very front of an airplane, he or she can create a "Section" that starts at location 0.0 (the nose of the airplane) and ends at whatever position is specified. If the "Section" is to filled with seats, galleys, and lavatories, it is designated a "Passenger_Section". If it will be filled with cargo containers, it is designated a "Cargo_Section". Only the "Passenger_Section" has been implemented thus far. General information about a "Section" is displayed by the "Section_Template" class (e.g. starting and end locations, and length). A "Passenger_Section" derives from the "Seat_Manager", "Galley_Manager", and "Lavatory_Manager" classes which are responsible for creating and placing their respective component types.

Figure 7 depicts the hierarchy among the component classes. The common base class is the "ACSYNT_Component" class. This class retrieves information from ACSYNT's data structures that is common to each component type. This includes color, number of cross sections, number of points, and other geometric information. It also draws a PHIGS structure to actually display the component. Every subsequent class that defines a component then derives from the "ACSYNT_Component" class. Another layer in the hierarchy is added by the "Wing" and "Parametric_Box" classes. The "Wing" class serves as a base class for the other "wing-like" components: the strake, canard, vertical tail and horizontal tail. Similarly, the "Parametric_Box" class serves as a base class for components that are various shapes of "Parametric_Box": the LD3, 727-200C, Galley, and Lavatory.
The "Booch bubble" - denotes a C++ class. The class name labels the bubble.

A class that has not yet been implemented in the Internal Layout Module. Included to show extensibility.

Class A inherits class B; class B is the base class of class A.

Class A uses class B; class B is created by class A.

Figure 4 - Key to Booch Notation
Figure 6 - Class Structure Using Booch Notation - 2
Figure 7 - Class Structure Using Booch Notation - 3
ACSINT_Component class
This class is the fundamental building block in encapsulating information and functions relating to ACSYNT components. It contains three PHIGS structures, one for each view of the Internal Layout Module (top, front [cross section], and isometric) and functions to display each of them individually or all together. It also contains basic information common to all component types including color, location, rotations, number of cross sections, and much more. Every other component class (nose, mid-section, wing, etc.) derives from this class, thus inheriting all the data and functions.

Aft_Body class
This class models the rear portion of the aircraft fuselage. The parameters include length, two radii for both the front and rear cross sections, and two angles. This class derives from the ACSYNT_Component class.

Canard class
This class models the canard horizontal control surface of the aircraft. It derives from the Wing class and inherits all the data and functions associated with that class. As in ACSYNT, a canard acts just like a wing for most purposes.

Component_Manager class
This class coordinates all external components of the aircraft. It is responsible for determining what type of components make up the ACSYNT model and for creating an instance (object) of each type. Thus if the ACSYNT model consists of a fuselage (nose, mid-section, aft-body), wing, horizontal tail and vertical tail, then, when the Internal Layout
Module is invoked the Component_Manager creates objects for each of those components. The only components the Component_Manager is not responsible for are the seat and the parametric box, as they are managed by other classes (Seat_Manager, Lavatory_Manager, and Galley_Manager). The Component_Manager class also performs "auto-centering" when the module is started to properly orient and position the airplane in the three views: top, front (cross section), and isometric.

**File_Manager class**

This class handles file input and output for the Internal Layout Module. It is one of the two base classes of the Workspace class (the Totals_Manager class is the other).

**Galley class**

This class models an aircraft galley. It derives from the Parametric_Box class.

**Galley_Manager class**

This class is responsible for keeping track of galleys in each Passenger_Section (a class described below) of the aircraft. This is one of the base classes of the Passenger_Section class. The number of galleys is not limited.

**Horizontal_Tail class**

This class models the horizontal control surface of the aircraft. It derives from the Wing class and inherits all the data and functions associated with that class. As in ACSYNT, a horizontal tail acts just like a wing for most purposes.
**Input_Field class**

This class was constructed as an addition to Woyak's interface framework [Woya92]. It combines the interface framework classes of Text_Input, Frame, and Label to create a consistent menu item for entering data.

**Internal_Layout class**

This class controls the Internal Layout Module. It sets up the views and background menus and is responsible for relaying user input to other classes. It also contains the main event loop for the module. This class is the entry and exit point of the module in relation to ACSYNT.

**Lavatory class**

This class models an aircraft lavatory. It derives from the Parametric_Box class.

**Lavatory_Manager class**

This class responsible for keeping track of lavatories in each Passenger_Section (a class described below) of the aircraft. This is one of the base classes of the Passenger_Section class. The number of lavatories is not limited.

**LD3 class**

This class models an LD3 cargo container which is a specific type of Unit Load Device (ULD). This class derives from the Parametric_Box class.
**Mid Section class**

This class models the middle portion of the aircraft fuselage. The parameters include length, and two radii for both the front and rear cross sections. This class derives from the ACSYNT_Component class.

**Nose class**

This class models the front portion of the aircraft fuselage. The parameters include length, tip angle, shoulder angle, dive angle and two radii for the rear cross sections. This class derives from the ACSYNT_Component class.

**Parametric_Box class**

This class is a flexible component type created to model several components. By setting parameters, the shape collapses to a regular three-dimensional "brick" shape. But the class contains many parameters in addition to just length, width, and height to allow very unusual shapes to be created. Volume, weight, and density are other parameters that can be calculated or set by the user (volume is only calculated from the geometry; it cannot be set by the user). The component can also be linearly scaled. See the section "New ACSYNT Components" later in this thesis for a pictorial representation of the Parametric_Box which shows all associated parameters.
**Passenger Section class**

This class manages a section of the aircraft filled with seats, lavatories, and galleys. It derives from the classes Section, Seat_Manager, Lavatory_Manager, and Gailey_Manager. Thus, it inherits the data and functions associated with all of those classes. The class contains data and functions dealing with the number of seats, the number of lavatories, the number of galleys, the aisle width, the number of aisles, the seat pitch, and the seat configuration.

**Seat class**

This class models an aircraft seat. See the section "New ACSYNT Components" earlier in this thesis for a pictorial representation of the Seat which shows all associated parameters.

**Seat_Manager class**

This class is responsible for keeping track of seats in each Passenger_Section (a class described above) of the aircraft. This is one of the base classes of the Passenger_Section class. The number of seats is not limited. The class contains functions to add and subtract seats and place the seats automatically based on the parameters specified by the user.

**Section class**

This is an abstract class used to delineate a portion of the aircraft. A Section has a starting and ending location and a length (measured along the fuselage, with the nose at 0.0). The ending location and length can be either fixed, in which case the user enters either parameter and the other is automatically calculated. The other option is to allow both parameters to be calculated based upon how many components are specified by the user to be contained in the
Section. For example, in a Passenger_Section (derived from the Section class) the user would enter how many seats are to be contained and the associated variables such as pitch and aisle width. And the ending location and length of the Section would automatically be calculated.

Section_Template class

This is a class dedicated to displaying the data of the Section class. The class does not contain any data of its own; it contains only graphical entities (mostly from Woyak's interface framework). The reason these graphics are not handled directly by the Section class is to reduce computational and graphical overhead. The user will typically divide the aircraft into many Sections (between three to ten sections is common, although the number is not limited). But there will only be one Section_Template. As the user changes the Section being worked on, the Section_Template changes the data being displayed. When the user moves on to another Section or adds a Section, the Section_Template always (and automatically) displays the data of the current Section.

Strake class

This class models the leading edge extensions to the wing of the aircraft. It derives from the Wing class and inherits all the data and functions associated with that class. As in ACSYNT, a strake acts just like a wing for most purposes.

Totals_Manager class

This class is responsible for keeping track of the running totals for the Internal Layout Module and displaying those totals on demand. The statistics that are tracked include
number of seats, number of lavatories, number of galleys, and more. This is one of the two base classes of the Workspace class (the File_Manager class is the other).

**Undefined_Component class**

This class is used to ensure that all the components created in ACSYNT are correctly transferred to the Internal Layout Module, even if no classes have been explicitly defined for them. For example, the module does not have classes defined for inlets. However, if the ACSYNT model contains inlets, they will correctly appear in the Internal Layout Module as well. They will not be able to be modified, but they will appear in the correct locations and with the same graphic properties as in ACSYNT, such as color, number of cross sections, etc.

**Vertical_Tail class**

This class models the vertical control surface(s) of the aircraft. It derives from the Wing class and inherits all the data and functions associated with that class. As in ACSYNT, a vertical tail acts just like a wing for most purposes.

**Wing class**

This class models the primary lifting surface of the aircraft. The parameters include span, leading edge sweep, quarter chord sweep, aspect ratio, area, dihedral, twist, taper ratio, root chord, airfoil type, root airfoil number, and tip airfoil number. Functions are provided to query these parameters and to set them.
**Workspace class**

This class can be thought of as the "Section Manager". It is responsible for keeping track of all the Sections, adding and removing Sections, moving between Sections and more. Provides the interface between the user and the Sections. Thus when a parameter is changed in a Section, the Workspace relays the change and displays the results. Also manages file input/output (delegated to the File_Manager class, from which the Workspace derives) and the totals (delegated to the Totals_Manager class, from which the Workspace also derives).

**727_200C class**

This class models an 727-200C cargo container which is a specific type of Unit Load Device (ULD). The class derives from the Parametric_Box class.
New Components for Internal Layout

When the Internal Layout Module was ready to be integrated with ACSYNT, an examination was made of ACSYNT geometry. ACSYNT supports many varied types of aircraft components. These include nose, mid-section, aft-body, wing, horizontal tail, vertical tail, canard, strake, multi-section wing, fuselage engine, wing engine, canopy, and gen-pod (generic pod). To effectively carry out internal layout, it was found that additional components were needed. As discussed in the section "Overview of Program Design", the module itself does not generate geometry from scratch; it retrieves PHIGS structures from any existing PHIGS program such as ACSYNT. Thus to add the new components, they were programmed within the existing framework for ACSYNT geometry. This section discusses the ACSYNT geometry structure that was used to create these new components and provides pictorial descriptions of the components.

ACSYNT Geometry Routines

A complete description of the details of ACSYNT geometry can be found in [Wamp88a] and in [Jaya92b]. Described here are the type of routines that must be coded for each new component added to ACSYNT. This section is included primarily to help future programmers attempting to add new components to ACSYNT. However, explanations of many of the fundamental concepts of ACSYNT geometry are also included within the routine descriptions. These explanations are intended for readers who are not intimately familiar with or interested in the guts of the code but do desire a working knowledge of the principles of ACSYNT geometry. Also, the source code file "acs_seat.vpi" is included as
Appendix B. Each type of routine described in this section has a concrete example in the seat component. Commenting is quite extensive.

**Load Component**

In the ACSYNT vernacular, this is referred to as simply the "load" routine. The subroutine is responsible for loading the cross section data structure with the cross section type and the characteristic points that lie on each cross section. The corresponding routine for the seat component in "acs_seat.vpi" is LDSEAT (LoaD SEAT).

A note about the cross section types is in order here. Prior to this thesis work, ACSYNT provided elliptic, conic, and airfoil types of cross sections. The elliptic and conic section types are used for the fuselage components (nose, mid-section, and aft-body), and the airfoil type is used for the wing and wing-like components (wing, horizontal and vertical tails, canard, and strake). Neither the seat nor the parametric box could be adequately constructed using any of these cross section types. Consequently, two new cross section types, the "seat type" and the "box type", were developed. They were designed to be used only for the seat and the parametric box, respectively. The "seat type" cross section is constructed with straight line segments and conic curve segments while the "box type" cross sections is constructed with only straight line segments.

**Find and Store Surface Points**

This is a high level routine that invokes other subroutines to eventually calculate and store all the surface points along all the cross sections of a component. The corresponding routine for the seat component in "acs_seat.vpi" is SEATPTS (SEAT PoinTS).
Retrieve Cross Section Points on a Cross Section

This routine retrieves the cross section points on a particular cross section which are stored in the cross section data structure. The corresponding routine for the seat component in "acs_seat.vpi" is SEATPNT (SEAT PoiNT).

The terms "characteristic points" and "surface points" have heretofore been used without any accompanying explanations. Some discussion is due. ACSYNT geometry is defined by a set of cross sections. To define a cross section, the programmer draws a set of characteristic points outlining the desired shape. The characteristic points are then connected by either conic curve segments or straight line segments to form the cross section. The programmer must then define a set of equations to connect the cross sections. For example, the nose component uses equations that form a parabolic shape when the cross sections are connected. The cross sections for both the seat component and the parametric box component are connected linearly, thus the equations are particularly simple. The characteristic points are also used to define the parameters of the component. The relationships between the various characteristic points yield the parameters.

Surface points lie on the cross sections of each component and are calculated using the characteristic points. These surface points are then used to loft bicubic Hermite surface patches through the cross sections to generate the surfaced model. The surface points are also used as the basis for the creation of non-uniform bicubic B-spline surface models which ensure curvature continuity except where tangent or positional continuity is required. For further detail, the reader should consult [Wamp88a], [Jaya92a], and [Jaya92b].
Find Surface Points on a Cross Section

This routine calculates the surface points on a particular cross section. The surface points are found by varying the parametric variable to move along the cross section and finding the points between the appropriate characteristic points. The corresponding routine for the seat component in "acs_seat.vpi" is SEATSRF (SEAT SuRFace).

Draw Component Shape Template

This routine draws a template on the screen to allow the user to modify the parameters of a component. It should be noted that the parameters are not actually stored; they are calculated from the characteristic points (which are stored) each time the template is drawn. This situation is remedied in the C++ classes which store all parameters associated with a component as well as the characteristic points. The corresponding routine for the seat component in "acs_seat.vpi" is SEATTMP (SEAT TeMPlate).

Get Characteristic Points

This routine retrieves the characteristic points of a component which are stored in the characteristic points data structure. The corresponding routine for the seat component in "acs_seat.vpi" is SEATGP (SEAT Get Points).

Store Characteristic Points

This routine saves the characteristic points of a component which are stored in the characteristic points data structure. The corresponding routine for the seat component in "acs_seat.vpi" is SEATSP (SEAT Store Points).
**Process Data**

This routine accepts the user input in response to the component shape template. The new value for the modified parameter is used to recalculate any other parameters that may be affected and all the value in the template are updated. The corresponding routine for the seat component in "acs_seat.vpi" is SEATPD (SEAT Process Data).

**Return Parameter Value**

This routine is used to form the prompt for the user when a parameter is selected to be changed. The old value of the parameter is used as the prompt. The corresponding routine for the seat component in "acs_seat.vpi" is SEATRT (SEAT ReTurn data).

**Convert Points to Parameters**

This routine uses the characteristic points of a component to calculate parameters. The programmer defines relationships through equations. The parameter is on the left hand side of the equals sign and a mathematical expression involving characteristic points is on the right hand side. The corresponding routine for the seat component in "acs_seat.vpi" is SEAT12.

**Convert Parameters to Points**

This routine performs the conversion in the opposite direction as the routine described immediately above and is invoked when a parameter has been changed by the user. As noted previously, the parameters for a component are not stored by ACSYNT. This routine is invoked from the routine that displays the component template (described above). That routine first uses the characteristic points to calculate and display parameters and then it passes all the parameters (changed and unchanged) to this routine, along with the unchanged
characteristic points. This routine in turn uses the parameters that have been passed to it to calculate the new characteristic points. It is a somewhat circuitous sequence that is necessary only because the parameters are not stored along with the characteristic points. The corresponding routine for the seat component in "acs_seat.vpi" is $\text{SEAT}12$. 
The Seat and Parametric Box Components

The new components created are the seat and the parametric box. A written description for each is not given; instead a detailed illustration of each component is included which conveys all the relevant information and parameters. This approach was deemed the most concise and informative. Figures 8 and 9 show the seat and the parametric box, respectively.
NOTES:

Point 19 (not shown) is located on the last cross section, displaced in the X direction from point 1 by the value of "width".

The "seatback angle" parameter is not shown.

Figure 8 - The Seat Component
Figure 9 - The Parametric Box Component
ACSYNT Integration

The Internal Layout Module was added to the existing ACSYNT program as an entirely new module. This section discusses how the module was added to ACSYNT and some of the issues that were important during integration.

Entry/Exit

A new menu item was added under the ACSYNT geometry menu called "INTERNAL LAYOUT". This is within the Geometry Module of ACSYNT. Selecting this menu item invokes the Internal Layout Module. At this point, different actions occur depending on the type of workstation the program is being run on because of differences in PHIGS implementations. On IBM machines (running graPHIGS), a new PHIGS workstation is opened in which the Internal Layout Module appears. The existing ACSYNT workstation remains open - there are two PHIGS workstations open simultaneously. By having two PHIGS workstations on the screen at the same time, the idea that the Internal Layout Module is a separate software is reinforced. However, since the module has been integrated as part of ACSYNT, both windows share a common title bar which displays the name of the program, "ACSYNT V3.0". Also, by having two separate windows open, it is readily apparent to the user when changes made within the Internal Layout Module are updated in the original ACSYNT program. On the Silicon Graphics machines (running Figaro+), a new PHIGS workstation cannot be opened because the implementation only provides for one
open workstation at any one time. Therefore, in this case, the ACSYNT workstation is closed and a new one is opened for the Internal Layout Module.

Upon exiting the Internal Layout Module (by pressing the "CLOSE" push button), the results vary slightly, again owing to the differences in PHIGS implementations. On IBM machines, the separate PHIGS workstation used for the Internal Layout Module is closed and control is restored to the still-open ACSYNT workstation. On SGI machines, the one open workstation being used for the internal Layout Module is closed and a new workstation is opened in which ACSYNT is re-started. All the PHIGS structures are retained. The main difference for the user is that ACSYNT re-starts outside of the Geometry module. This is only a slight inconvenience.

A technique has been investigated which eliminates the need for separate PHIGS workstations and results in a smoother transition to and from the Internal Layout Module. The new approach is to use one workstation with overlapping views. ACSYNT currently only uses eight views. Most PHIGS implementations provide the capacity to use 16 or even 32 views. Using this method, one view would be used to "cover up" ACSYNT. Then the remaining views would be used for the Internal Layout Module or any other module using Woyak's interface framework. Woyak's interface framework facilitates this method by allowing the programmer to designate certain views as "off limits". The views currently used by ACSYNT would be so designated. This technique was tried briefly and showed promise. The overlapping views functioned as expected, however, the program crashed when the Internal Layout Module was exited. Time constraints did not permit further investigation since, at that time, the approach of using separate PHIGS workstations was working satisfactorily. This overlapping views approach deserves further attention because
it is smoother; it is felt that the multiple PHIGS workstations method is inherently unstable and is not well-supported by PHIGS implementations.

This problem will be obviated in the future when a completely object-oriented ACSYNT is developed. This future version of ACSYNT will rely entirely upon Woyak's interface framework for the graphical user interface and there will be no reason for either multiple PHIGS workstations or overlapping views.

**FORTRAN / C / C++ Data Exchange**

Prior to the Internal Layout Module, ACSYNT consisted of code written in the FORTRAN and C programming languages. Since the new module was to be object-oriented, it was written entirely in C++. This presented a challenge in exchanging data between the three languages and making the transition from procedural code to object-oriented code. The solution was to use "bridge" functions. By isolating the transfer of data to a limited set of functions, maintainability is greatly increased.

The data required to be transferred are component attributes and parameters. This information is stored by ACSYNT in a large FORTRAN data structure. This data structure manages the information for all the components in an ACSYNT model. When a piece of data is required, the programmer requests the information from the data structure by invoking a function which passes the component number and an integer index identifying what attribute or parameter is desired. The data structure then returns the requested data and an error indicator to report success or failure. Similar functions are used to insert
information into the data structure. Object-oriented philosophy precludes the use of these type of large data structures that are responsible for maintaining information for other "objects". The Internal Layout Module defines classes, using C++, for each ACSYNT component type, for example, nose, mid-section, aft-body, wing, etc. These classes are responsible for maintaining their own information and for providing functions to get and set the data.

There are two types of bridge functions used to transfer the data between the ACSYNT FORTRAN data structure and the C++ classes. As soon as the Internal Layout Module is invoked, a query is made to determine what components comprise the model. For each component, the first type of bridge function is called. This first type is used to transfer general data common to each type of component. This is mostly information for the geometric rendering of the component such as color, linetype, number of cross sections, number of lines in the U and W parametric directions, etc. This information is referred to in ACSYNT as "general parameter data". There are two bridge functions of this type: one to retrieve the data from the FORTRAN data structure into the C++ classes, and another to transfer the data in the other direction. All the different component types use the same two functions.

The second type of bridge function is specific to each component type, i.e. the wing has its own functions distinct from those of the nose and from those of any other component type. This type of bridge function is used to exchange component parameters between the FORTRAN data structure and the C++ classes. Each component has two bridge functions of this type: one to transfer data in each direction.
Application to ACSYNT / User's Guide

The thesis has so far discussed the background, objectives, and design of the new Internal Layout Module. The module has been successfully integrated with ACSYNT. This section gives a comprehensive look at the features of the module including a description of the screen layout, user interaction, and all the tasks the module is capable of performing. The section is organized as a User's Guide.

"Close" pushbutton

Click on this pushbutton to exit the Internal Layout Module. Caution: Remember to save your work; no prompt is displayed when this button is pushed, exit is immediate.

"Help" pushbutton

Click on this pushbutton to display a help screen. This feature has not yet been implemented.

"Load/Save Data File" pushbutton

Click on this pushbutton to read or write a data file. A pop-up menu appears which contains radio buttons to select the operation to be performed: reading or writing. Choose the desired button. The pop-up menu also contains a text input field for the file name. Enter the name of the file. Finally, choose the "OK" button to proceed with the file read or write, or choose the "CANCEL" button to leave the menu with no operation performed.
"Section Name" field

Use this text input field to enter a descriptive name for the current section. An example would be "First Class Cabin".

"Section Type" pushbutton

Click on this pushbutton to change the type of section for the current section. For example, the default type is "Passenger". Pushing the button changes the type to "Cargo". Only "Passenger" section type has thus far been implemented.

"Deck" pushbutton

Click on this pushbutton to move the section to another deck of the aircraft. For example, the default deck is the "Main Passenger Deck". This button would be used to move the section to, for example, a "Cargo Deck". Changing the deck changes the Z location of all components contained in a section.

"Options" pushbutton

Click on this pushbutton to bring up a pop-up menu that displays options for modifying components, adding and removing sections, and hiding and showing the totals menu. The totals menu is shown in Figure 10.
"Start Location" field

Use this text input field to enter the starting location for the section. The nose of the aircraft is located at position 0.0. After entering the starting location, the "APPLY" button must be pushed to initiate the action.

"End Location" field

Use this text input field to enter the ending location for the section. User input of the ending location or the length is only required when the "Fixed End Location / Length" radio button is selected. If the "Calculated End Location and Length" radio button is selected, the ending location for the section is automatically calculated and displayed. The two radio buttons are explained below.

"Length" field

Use this text input field to enter the length of the section. User input of the length or the ending location is only required when the "Fixed End Location or Length" radio button is selected. If the "Calculated End Location and Length" radio button is selected, the length of the section is automatically calculated and displayed. The two radio buttons are explained below.

"Fixed End Location or Length" radio button

In a set of two or more "radio buttons", only one button can be selected at a time; i.e. selecting one button causes all the other buttons to be un-selected. Choose this radio button to manually enter either the ending location or length of the section. This also allows you to use the "Auto Fill" option (explained below).
"Calculated End Location and Length" radio button

In a set of two or more "radio buttons", only one button can be selected at a time; i.e. selecting one button causes all the other buttons to be un-selected. Choose this radio button to let the program automatically calculate and display the ending location and length of the section. The calculation is based on how many components you have specified are to be contained in the section. For example, if you have specified 25 seats in the section, the program will use this information along with other parameters such as seat width, aisle width, etc. to determine the length and ending location of the section. Use of this method disables the "Auto Fill" option (explained below).

"Number of Seats" field

Use this text input field to enter how many seats are to be placed in the section. The program then automatically places the seats depending on the parameters specified such as number of aisles, aisle width, pitch, etc. After entering the number of seats, the "APPLY" button must be pushed to initiate the action.

"Auto Fill" pushbutton

Click on this pushbutton to automatically fill the section with as many seats as the specified parameters (number of aisles, aisle width, pitch, etc.) will allow. The "Fixed End Location or Length" radio button must be selected to enable this option. After pushing the "Auto Fill" button, the "APPLY" button must be pushed to initiate the action.
"Seat Pitch" field

Use this text input field to enter the seat pitch for the section. After entering the seat pitch, the "APPLY" button must be pushed to initiate the action.

"Aisle Width" field

Use this text input field to enter the aisle width for the section. The same aisle width is applied to each aisle. After entering the aisle width, the "APPLY" button must be pushed to initiate the action.

"Number of Aisles" number box

Use the small up arrow and down arrow buttons to increment or decrement the number of aisles in the section. Up to four aisles may be specified. After the desired number of aisles is displayed, the "APPLY" button must be pushed to initiate the action.

"Configuration" fields

Enter the configuration desired using these text input boxes. The number of boxes available is automatically updated depending upon the number of aisles specified. For example, if there are two aisles, then three boxes will be available to enter the configuration (e.g. 2 - 3 - 2). After entering the desired configuration, the "APPLY" button must be pushed to initiate the action.
"Modify Seats" pushbutton
Click on this pushbutton to bring up a pop-up menu that allows you to change the parameters for all the seats in the current section. The parameters include width, height, length, bottom thickness, back thickness, and seatback angle.

"Add/Remove Galleys" pushbutton
Click on this pushbutton to bring up a pop-up menu which allows you to add or remove galleys to the current section.

"Move/Modify Galleys" pushbutton
Click on this pushbutton to bring up a pop-up menu which allows you to move or modify the shape of any of the galleys defined in the section. If no galleys have been defined in the current section, an error message to that effect is displayed.

"Add/Remove Lavatories" pushbutton
Click on this pushbutton to bring up a pop-up menu which allows you to add or remove lavatories to the current section.

"Move/Modify Lavatories" pushbutton
Click on this pushbutton to bring up a pop-up menu which allows you to move or modify the shape of any of the lavatories defined in the section. If no lavatories have been defined in the current section, an error message to that effect is displayed.
"Next Section" pushbutton

Click on this pushbutton to edit the next defined section of the aircraft. If the current section is the last defined section, the data for the first section is displayed. If the current section is the only defined section, no action takes place.

"Previous Section" pushbutton

Click on this pushbutton to edit the previous defined section of the aircraft. If the current section is the first defined section, the data for the last section is displayed. If the current section is the only defined section, no action takes place.

"Goto Section" field

Use the text input field to jump to a particular section of the aircraft by entering the desired section's name. If any of the defined sections have a name matching the entered name, that section's data is displayed and becomes the current section to be modified. If two or more sections have the same names, the first section found with the matching name becomes the current section. If none of the defined sections have a name matching the entered name, no action takes place.
Figure 10 - Photograph of Totals Display
Results and Conclusions

The importance of an aircraft design program, such as ACSYNT, to be able to perform internal layout design is becoming more pronounced in projects such as the High Speed Civil Transport (HSCT). This is pointed out in a report by the Association of European Airlines which states, "the fundamental issue for this aircraft will be the interior flexibility" [AEA91]. The Internal Layout Module addresses this issue and allows designers to investigate the internal arrangement quickly and easily.

A completely new module has been designed for internal layout and added to ACSYNT to facilitate internal layout design. The module has been designed and implemented using object-oriented principles and the C++ programming language, the preeminent computer language for object-oriented programming.

New, parametrically defined, fully surfaced components were created including seats, lavatories, galleys, and cargo containers. These components were created as classes and are a first step toward object-oriented geometry for aircraft design.

New methods of arranging the aircraft interior by placing seats throughout the cabin were designed and implemented. These methods greatly facilitate the design of passenger transport aircraft.
To validate the work, the module was applied to the aircraft conceptual design code, ACSYNT. To integrate the new object-oriented module with the existing ACSYNT code, FORTRAN and C data transfer routines were designed and written.

The wireframe geometry graphics for the module use the international standard for 3-D graphics, PHIGS, to ensure portability across platforms. In addition, the user interface of the module was produced using the GUI Interface Framework previously created by Woyak [Woya93][Woya92] to provide Motif-like look and feel while retaining the portability and 3-D geometry support provided by PHIGS.

Figure 11 shows an example of the type of results that the Internal Layout Module is capable of producing.

While the new module greatly extends ACSYNT's capability to perform internal layout, other features could be added to add even more functionality. One such feature is the ability to perform conflict resolution / cross section design. The methods used by programs such as VASCOMP and GASP could be emulated and improved upon. These programs use a system of control points that determine clearances between the window seat passenger and the sidewall of the fuselage. Based on parameters specified by the user, the programs determine if there is any conflict. The Internal Layout Module could incorporate a similar method which would not only determine conflict but also assist the designer to resolve the conflict. The capabilities of the two programs mentioned would have to be exceeded because they handle only circular cross sections. The Internal Layout Module would have to support any shape cross section produced by ACSYNT geometry. This makes conflict resolution more difficult, of course, but it would make the program that much more flexible.
Another possible extension is the placement of doors and emergency exits. Federal Aviation Regulations (FAR) could be built into the module to check safety requirements such as how many seats access the same exit. Escape routes would automatically be generated and drawn on the top view and the number of seats that access each door would be reported to the user.

These types of extensions to the Internal Layout Module are greatly facilitated by the object-oriented design of the program. New classes can be easily added as well as new methods to existing classes.
Figure 11 - Sample Results
Tools Used

This section discusses the specific tools used to carry out the thesis work.

Computers:
All program development was carried out on UNIX workstations. To demonstrate the portability of the code written, including graphics, several different platforms were used. The code was developed and run on International Business Machines (IBM) RS/6000 models 350, 520, 530 workstations and model 980 server; Silicon Graphics Incorporated (SGI) Iris model 4D/80GT, Power Series, and Indigo workstations. It is anticipated that in the future, the code will be ported to Hewlett-Packard (HP) and Sun workstations as well.

Compilers:
As noted in the Introduction section, C++ was chosen as the programming language for the vast majority of coding done. Some work was also done in FORTRAN for the programming of new ACSYNT components to emulate pre-existing routines and techniques established for ACSYNT geometry. A mix of C/C++/FORTRAN was used to transfer data between ACSYNT's FORTRAN data structures and the C++ classes. The specific compilers used were "xlc" (C and C++) and "xlf" (FORTRAN) on the IBM workstations and "cc" (C and C++) and "f77" (FORTRAN) on the SGI workstations.
**PHIGS Implementations:**

The reasons for choosing PHIGS for all graphics programming were discussed in the Introduction section. On the IBM workstations, the IBM product "graPHIGS" was used and on the SGI workstations, "Figarò+" from Template Graphics Software (TGS) was used. SGI does not provide their own implementation of PHIGS.
References


Appendix A: Detailed Class Definitions

This section contains complete definitions of all classes used in the Internal Layout Module. In general, the names of both variables (data members) and functions (member functions) are long and verbose, thus making extensive descriptions superfluous. Where needed, however, explanations are given to clarify the purpose of certain variables and functions.

It should be noted that these are class definitions - that is, complete documentation of all the variables and functions that comprise the class, inheritance relationships, "friend" relationships to other classes, and other information. However, the actual source code of the functions is not included here due to excessive length. The information contained in these definitions should be sufficient to illustrate the purpose of each class and together with the Booch diagrams of the class structure (in the section "Class Structure"), the reader is able to gain a complete understanding of the Internal Layout Module. For readers of this thesis who may be following this work and adding to it, access may be required to the actual source code (the implementation details - the so-called "secrets" of the methods). These readers are referred directly to the author or to the major advisor, Dr. Myklebust, to make arrangements to access the source code.

The class definitions are listed in alphabetical order.
// ******************************************************************************************************************
// ACSYNT_Component class header file
// Programmer: Shahab Hasan
// Original coding: 4/93
// Last modification: 10/93
// ******************************************************************************************************************
#ifndef _ACSYNT_COMPONENT
#define _ACSYNT_COMPONENT

#include <string.h>
#include "scott/PHIGS_structure_id.h"
#include "scott/simple_view.h"
#include "scott/colors.i"
#include "scott/dll_element.h"
#include "fortran.h"
#include "shahab_constants.h"

class ACSYNT_Component : public DLL_Element< ACSYNT_Component >
{
    protected:
        char name[20];
        PHIGS_Structure_ID top_view_structure;
        PHIGS_Structure_ID front_view_structure;
        PHIGS_Structure_ID isometric_view_structure;
        int class_name[1];

        struct point_3D
        {
            float x,y,z;
        };

        int acsynt_structure_id;
        int existence_flag;
        int component_number;
        int acsynt_color_index;
        int acsynt_line_type;
        int global_symmetry_flag;
        int component_type;
        int cross_section_type;
        int number_of_characteristic_points;
        int number_of_points_per_x_section;
        int number_of_x_sections;
        int number_of_U_surface_lines;

    // ...
int number_of_w_surface_lines;
point_3D **cross_section_points;
point_3D **surface_points;  // NOT CURRENTLY USED
float x_rotation;
float y_rotation;
float z_rotation;
float x_translation;
float y_translation;
float z_translation;

// internal functions

void get_initial_data_from_acsynt();
void generate_initial_structures();
void allocate_memory_for_cross_section_points();
void free_memory_for_cross_section_points();
void implement_global_symmetry();
int translate_acsynt_color_index();

public:

// constructor

ACSYNT_Component( int _component_number );

// destructor (virtual so that a derived destructor can be called)

virtual ~ACSYNT_Component();

// member functions

// Functions to transfer data to/from ACSYNT; in essence,
// to transfer data between FORTRAN and C++.

void update_component_in_acsynt();
void get_rendering_data_from_acsynt();
void set_rendering_data_into_acsynt();
void get_rotation_and_translation_data_from_acsynt();
void set_rotation_and_translation_data_into_acsynt();
// Views and PHIGS structures functions.

void update_PHIGS_structures( int update_type = FULL );
void update_translations_rotations_or_color_only();
void post_to_all_views( Interface_Manager *iman );
void post_to_isometric_view( Interface_Manager *iman );
void post_to_top_view( Interface_Manager *iman );
void post_to_front_view( Interface_Manager *iman );

int get_top_view_structure_id();
int get_front_view_structure_id();
int get_isometric_view_structure_id();

// Functions to get information about the component.
//
// Note: These functions are very simple; they simply return
// the data that they are asked to give. The philosophy
// is that it should be easy to query data from an object.

char* get_name();
int get_existence();
int get_component_number();
int get_acsorient_color_index();
int get_acsorient_line_type();
int get_global_symmetry_flag();
int get_component_type();
int get_cross_section_type();
int get_number_of_characteristic_points();
int get_number_of_points_per_x_section();
int get_number_of_x_sections();
int get_number_of_u_surface_lines();
int get_number_of_w_surface_lines();
float get_x_rotation();
float get_y_rotation();
float get_z_rotation();
float get_x_translation();
float get_y_translation();
float get_z_translation();
// Functions to set information about the component.

// Note 1: These functions should be designed and implemented such that the integrity of the data is preserved.
// This means that an inept programmer (and further downstream, an inept user) should not be able to enter "Hello, world!" when an integer is called for. So, there should be error-trapping, range-checking, and the like to ensure that the object's data is uncorrupted.

// Note 2: After climbing on my soapbox and preaching in Note 1, I should also make it known that these functions *DO NOT* always take the precautions prescribed so strongly.

int set_name( char* new_name );
int set_existence_on();
int set_existence_off();
int set_acsyt_color_index( int new_color_index );
int set_acsyt_line_type( int new_line_type );
int set_global_symmetry_off();
int set_global_symmetry_about_yz_plane();
int set_global_symmetry_about_xz_plane();
int set_global_symmetry_about_xy_plane();
int set_cross_section_type( int new_cross_section_type );
int set_number_of_points_per_x_section( int new_numpts );
int set_number_of_x_sections( int new_number_of_x_secs );
int set_number_of_U_surface_lines( int new_num_U_lines );
int set_number_of_W_surface_lines( int new_num_W_lines );
int set_x_rotation( float new_x_rotation );
int set_y_rotation( float new_y_rotation );
int set_z_rotation( float new_z_rotation );
int set_x_translation( float new_x_translation );
int set_y_translation( float new_y_translation );
int set_z_translation( float new_z_translation );

// virtual functions

// These functions must be defined for each specific type of ACSYNT_component because the data varies from component to component.

virtual void change_parameter( int index, float* value );
virtual void get_component_data_from_acsyt();
virtual void set_component_data_into_acsyt();
// **********************************************
//
// Aft_Body class header file
//
// Programmer: Shahab Hasan
// Original coding: 4/93
// Last modification: 10/93
//
// **********************************************

#ifndef _AFT_BODY
#define _AFT_BODY

#include "ecsnt_component.h"

class Aft_Body: public ACSYNT_Component
{

protected:
    float  length, radius1, radius2, radius3, radius4;
    float  alpha1, alpha2;

public:

    //************
    // constructor
    //************
    Aft_Body( int _component_number );

    //-----------------------------------------------------------------------------------
    // destructor {virtual so that a derived destructor can be called}
    //-----------------------------------------------------------------------------------
    virtual ~Aft_Body();

    //************
    // member functions
    //************
    float get_length();
    float get_radius1();
    float get_radius2();
    float get_radius3();
    float get_radius4();
    float get_alpha1();
    float get_alpha2();

83
int set_length ( float new_length );
int set_radius1 ( float new_radius1 );
int set_radius2 ( float new_radius2 );
int set_radius3 ( float new_radius3 );
int set_radius4 ( float new_radius4 );
int set_alpha1 ( float new_alpha1 );
int set_alpha2 ( float new_alpha2 );
int update_mid_section();

//--------------------------------------------------------------------
// redefined (inherited) virtual functions
//--------------------------------------------------------------------
void change_parameter ( int index, float* value );
void get_component_data_from_acsyt();
void set_component_data_into_acsyt();

});

#endif
Canard class header file

Programmer: Shahab Hasan
Original coding: 5/93
Last modification: 9/93

ifndef _CANARD
#define _CANARD

#include "acsynt_component.h"
#include "wing.h"

class Canard : public Wing
{

protected:

    // Place any additional data members associated with
    // the canard here.

public:

    //------------------
    // constructor
    //------------------
    Canard(int _component_number);

    //-----------------------------------------------
    // destructor (virtual so that a derived destructor can be called)
    //-----------------------------------------------
    virtual ~Canard();

    //----------------------
    // member functions
    //----------------------

    // Place any additional member functions associated with
    // the canard here.

    //----------------------
    // redefined virtual functions
    //----------------------

};

#endif
Component_Manager class header file

Description: This class creates C++ objects derived from the ACSYNT_Component class. Each component in ACSYNT that is used in the interior layout module has a corresponding C++ class and an instance of each is created here. The class allocates the necessary memory using "new" and frees it using "delete".

Programmer: Shahab Hasan
Original coding: 4/93
Last modification: 10/93

#ifndef _COMPONENT_MANAGER
#define _COMPONENT_MANAGER

#include "scott/interface_manager.h"
#include "acsynt_component.h"
#include "aft_body.h"
#include "canard.h"
#include "fortran.h"
#include "horizontal_tail.h"
#include "mid_section.h"
#include "nose.h"
#include "shahab_constants.h"
#include "strake.h"
#include "undefined_component.h"
#include "vertical_tail.h"
#include "wing.h"

class Component_Manager
{

private:

    Interface_Manager *iman;
    int number_of_components;

    //------------------------------
    // The pointer for the linked list of components.
    //------------------------------
    ACSYNT_Component* first_component;
// These pointers are used for autocentering.

Nose* nose;
Mid_Section* mid_section;
Aft_Body* aft_body;
Wing* wing;

// internal functions

void add_to_list( ACSYNT_Component* new_component );
int get_number_of_components_from_acsynt();
int determine_component_type( int component_number );

public:

-------------
// constructor
-------------

Component_Manager( Interface_Manager *_iman );

-------------
// destructor
-------------

-Component_Manager();

-------------
// member functions
-------------

void create_components();
void delete_components();

void update_acsynt();
void test_function();

int allow_autocenter();
void autocenter();
float get_fuselage_length();
float get_fuselage_max_radius();

};

#endif
#ifndef _FILE_MANAGER
#define _FILE_MANAGER

#include <fstream.h>
#include "scott/interface_manager.h"

class Passenger_Section;

class File_Manager
{

protected:

    Interface_Manager* _iman;

    char file_name[20];

    fstream* file_pointer;

    //-------------------
    // internal functions
    //-------------------

public:

    //----------
    // constructor
    //----------

    File_Manager( Interface_Manager* _iman );

    //-----------------------------------------------
    // destructor (virtual so that a derived destructor can be called)
    //-----------------------------------------------

    virtual ~File_Manager();

};
// member functions

void set_data_file_for_reading( const char* new_file_name );
void set_data_file_for_writing( const char* new_file_name );
void skip();
void read_header_data( char* _number_of_sections_string );
void write_header_data( char* _number_of_sections_string );
void read_passenger_section_from_file( Passenger.Section* in_section );
void write_passenger_section_to_file( Passenger.Section* out_section );

};

#endif
#ifndef _FORTRAN
#define _FORTRAN

// "gtdata1", "gtdata2", "gtdata3", "stdata2", and "stdata3"
// are used in the class "acsyt_component".
// These routines pass general geometric data relating to components
// back and forth between ACSYNT and the C++ side. The data are
// parameters such as line color, cross section type, translations and
// rotations and many more. See the file "acsyt_component.C" for a
// complete list of the arguments.

#ifdef WORKSTATION==SGI
  extern "C" void gtdata1_( int &, int &, int &, int & );
  extern "C" void gtdata2_( int &, int &, int &, int &, int &,
                           int &, int &, int & );
  extern "C" void gtdata3_( int &, float &, float &, float &,
                           float &, float & );
  extern "C" void stdata2_( int &, int &, int &, int &, int &,
                           int &, int &, int & );
  extern "C" void stdata3_( int &, float &, float &, float &,
                           float &, float & );
#else
  extern "FORTRAN" void gtdata1( int &, int &, int &, int & );
  extern "FORTRAN" void gtdata2( int &, int &, int &, int &, int &,
                               int &, int &, int & );
  extern "FORTRAN" void gtdata3( int &, float &, float &, float &,
                               float &, float & );
  extern "FORTRAN" void stdata2( int &, int &, int &, int &, int &,
                               int &, int &, int & );
  extern "FORTRAN" void stdata3( int &, float &, float &, float &,
                               float &, float & );
#endif
// "tmupds" and "vuwk" are used in the class "component_manager".
// "tmupds" updates the ACSYN component templates.
// "vuwk" updates the ACSYN workstation.

#ifdef WORKSTATION==SGI
    extern "C" void tmupds_( int & );
    extern "C" void vuwk_( int &, int & );
#else
    extern "FORTRAN" void tmupds( int & );
    extern "FORTRAN" void vuwk( int &, int & );
#endif

// The following functions are used in the class "aft_body".
// "aftgpe" gets the characteristic points that define an elliptic aft_body.
// "aftspe" stores the characteristic points that define an elliptic aft_body.
// "aft12e" converts from points to parameters for an elliptic aft_body.
// "aft21e" converts from parameters to points for an elliptic aft_body.
// "aftpde" processes data for an elliptic aft_body.

#ifdef WORKSTATION==SGI
    extern "C"
    {
        void aftgpe_( int &, float *, float &, float & );
        void aftspe_( int &, float *, float &, float & );
        void aft12e( float*, float &, float &, float &, float &, float & );
        void aft21e( float*, float &, float &, float &, float & );
        void aftpde_( int &, int &, float & );
    }
#else
    extern "FORTRAN" void aftgpe( int &, float *, float &, float & );
    extern "FORTRAN" void aftspe( int &, float *, float &, float & );
    extern "FORTRAN" void aft12e( float*, float &, float &, float &, float & );
    extern "FORTRAN" void aft21e( float*, float &, float &, float &, float & );
    extern "FORTRAN" void aftpde( int &, int &, float & );
#endif

91
void seatgp_( int &, float*, float[6] );
void seatsp_( int &, float*, float[6] );
void seat12_( float*, float &, float &, float &, float &,
float &, float &, float & );
void seat21_( float*, float &, float &, float &,
float &, float &, float & );
void seatpd_( int &, int &, float & );

void nosgp_( int &, float *, float &, float & );
void nosspe_( int &, float *, float &, float & );
void nos12e_( float*, float &, float &, float &,
float &, float &, float & );
void nos21e_( float*, float &, float &, float &,
float &, float &, float & );
void nospde_( int &, int &, float & );
#else
extern "FORTRAN" void nosgpe( int & , float * , float & , float & );
extern "FORTRAN" void nosspe( int & , float * , float & , float & );
extern "FORTRAN" void nos12e( float* , float & , float & , float & ,
 float & , float & , float & );
extern "FORTRAN" void nos21e( float* , float & , float & , float & ,
 float & , float & , float & );
extern "FORTRAN" void nospde( int & , int & , float & );
#endif

// The following functions are used in the class 'mid_section'.
// "midgpe" gets the characteristic points that define an elliptic mid_section.
// "midspe" stores the characteristic pts. that define an elliptic mid_section.
// "mid12e" converts from points to parameters for an elliptic mid_section.
// "mid21e" converts from parameters to points for an elliptic mid_section.
// "midpde" processes data for an elliptic mid_section.

#if WORKSTATION==SGI
 extern "C"
 {
  void midgpe_( int &, float * );
  void midspe_( int &, float * );
  void mid12e_( float*, float &, float & , float & , float & , float & , float & );
  void mid21e_( float*, float &, float & , float & , float & , float & , float & );
  void midpde_( int &, int & , float & );
 }
#else
 extern "FORTRAN" void midgpe( int & , float * );
 extern "FORTRAN" void midspe( int & , float * );
 extern "FORTRAN" void mid12e( float* , float & , float & , float & ,
 float & , float & );
 extern "FORTRAN" void mid21e( float* , float & , float & , float & ,
 float & , float & );
 extern "FORTRAN" void midpde( int & , int & , float & );
#endif
// The following functions are used in the class "wing".
// "wnggpa" gets the characteristic points that define a wing.
// "wngspa" stores the characteristic points that define a wing.
// "wp12" converts from points to parameters for a wing.
// "wp21" converts from parameters to points for a wing.
// "wngpda" processes data for a wing.
//-------------------------------------------------------------------------------

#if WORKSTATION==SGI
    extern "C"
    {
    void wnggpa_( int &, float * );
    void wngspa_( int &, float * );
    void wp12_( int &, float*, float &, float &, float &,
    float &, float &, int &, int &, float & );
    void wp21_( int &, float*, float &, float &, float &,
    float &, float &, int &, int &, float & );
    void wngpda_( int &, int &, float & );
    }
#else
    extern "FORTRAN" void wnggpa( int &, float * );
    extern "FORTRAN" void wngspa( int &, float * );
    extern "FORTRAN" void wp12( int &, float*, float &, float &,
    float &, float &, int &, int &, float & );
    extern "FORTRAN" void wp21( int &, float*, float &, float &,
    float &, float &, int &, int &, float & );
    extern "FORTRAN" void wngpda( int &, int &, float & );
#endif

//-------------------------------------------------------------------------------

// The following functions are used in the class "Parametric_Box".
// "boxgp" gets the characteristic points that define a box.
// "boxsp" stores the characteristic points that define a box.
// "box12" converts from points to parameters for a box.
// "box21" converts from parameters to points for a box.
// "boxpd" processes data to change a parameter for a box.
//-------------------------------------------------------------------------------

#if WORKSTATION==SGI
    extern "C"
    {
    void boxgp_( int &, float *, float & );
    void boxsp_( int &, float *, float & );
    void box12_( float*, float & float &, float &, float &,
    float &, float &, float &, float &,
    float &, float &, float &, float & );
    void box21_( float*, float & float &, float &, float &,
    float &, float & float &, float &,
    float &, float & float &, float & );
    void boxpd_( int &, int &, float & );
    }

94
#else
extern "FORTRAN" void boxgp( int &, float *, float & );
extern "FORTRAN" void boxsp( int &, float *, float & );
extern "FORTRAN" void box12( float*, float &, float &,
    float &, float &,
    float &, float &,
    float &, float &,
    float &, float &,
    float &, float &,
    float &, float & );
extern "FORTRAN" void box21( float*, float &, float &,
    float &, float &,
    float &, float &,
    float &, float &,
    float &, float &,
    float &, float & );
extern "FORTRAN" void boxpd( int &, int &, float & );
#endif

// The following functions are used in the class "Component_Manager".
// "gticmp" gets an integer parameter from the ACSYNT data structure.
// "gtncmp" gets the number of components in the ACSYNT model.
#endif

#if WORKSTATION==SGI
  extern "C"
  {
    void gticmp( int &, int &, int & );
    void gtncmp( int & );
  }
#else
  extern "FORTRAN" void gticmp( int &, int &, int & );
  extern "FORTRAN" void gtncmp( int & );
#endif

#endif
// *******************************************************************************

// Galley class header file

// Programmer: Shahab Hasan
// Original coding: 10/93
// Last modification: 10/93

// *******************************************************************************
#ifndef _GALLEY
#define _GALLEY

#include "scott/radio_button_manager.h"
#include "scott/pop_up_menu.h"
#include "acsynt_component.h"
#include "parametric_box.h"

class Galley : public Parametric_Box
{

protected:

    // Place any additional data members associated with the Galley here.

public:

    // constructor
    Galley( int _component_number );

    // destructor (virtual so that a derived destructor can be called)
    virtual ~Galley();

    // member functions

    // Place any additional member functions associated with
    // the Galley here.

    // redefined virtual functions

};

#endif

96
#ifndef _GALLEY_MANAGER
#define _GALLEY_MANAGER

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "scott/pop_up_menu.h"
#include "scott/push_button.h"
#include "scott/radio_button.h"
#include "scott/radio_button_manager.h"
#include "scott/seperator.h"
#include "galley.h"
#include "shahab_constants.h"

class Galley_Manager
{

protected:

    // The linked list pointers.
    Galley* first_galley;
    Galley* current_galley;

    // Data stored by the Galley_Manager class.
    int number_of_galleys;
    char number_of_galleys_string[3];
    int number_of_galleys_made;
    float total_volume_of_galleys;
    float initial_x_location;
    char new_name_string[20];
// The following chunk of strings are used when reading
// in the data for galleys from a data file. These
// strings are filled once for each galley defined in the
// file and the contents of the strings are then processed
// to fill the data members for that galley.
// The process is reversed when writing to a data file.

char  gal_name[20];
char  gal_color_index_string[2];
char  gal_left_width_string[10];
char  gal_center_width_string[10];
char  gal_right_width_string[10];
char  gal_top_step_height_string[10];
char  gal_bottom_step_height_string[10];
char  gal_main_left_height_string[10];
char  gal_main_right_height_string[10];
char  gal_depth_string[10];
char  gal_angle1_string[10];
char  gal_angle2_string[10];
char  gal_angle3_string[10];
char  gal_angle4_string[10];
char  gal_weight_string[10];
char  gal_x_location_string[10];
char  gal_y_location_string[10];
char  gal_z_location_string[10];
char  gal_x_rotation_string[10];
char  gal_y_rotation_string[10];
char  gal_z_rotation_string[10];

// internal functions

void initialize_current_galley();
void remove_all_galleys();
void read_galley_from_file( Interface_Manager* iman );
void write_galley_to_file();

public:

// constructor

Galley_Manager();
/**
// destructor (virtual so that a derived destructor can be called)
//
-Galley_Manager();

/**
// member functions
/**

int get_number_of_galleys();
float get_total_volume_of_galleys();
void invoke_add_remove_galley_menu( Interface_Manager* iman );
void invoke_remove_galley_menu( Interface_Manager* iman );
void invoke_move_modify_galley_menu( Interface_Manager* iman );
void invoke_error_menu( Interface_Manager* iman );
void add_galley( Interface_Manager* iman );
void remove_galley();
void modify_galley();

*/

#endif
### Horizontal_Tail class header file

Programmer: Shahab Hasan
Original coding: 5/93
Last modification: 9/93

```cpp
#ifndef __HORIZONTAL_TAIL
#define __HORIZONTAL_TAIL

#include "acsyst_component.h"
#include "wing.h"

class Horizontal_Tail : public Wing {

protected:

    // Place any additional data members associated with
    // the horizontal tail here.

public:

    // constructor
    Horizontal_Tail(int _component_number);

    // destructor (virtual so that a derived destructor can be called)
    virtual ~Horizontal_Tail();

    // member functions

    // Place any additional member functions associated with
    // the horizontal tail here.

    // redefined virtual functions

};

#endif
```
#ifndef _INPUT_FIELD
#define _INPUT_FIELD

#include "scott/static_menu.h"
#include "scott/pop_up_menu.h"
#include "scott/label.h"
#include "scott/text_input.h"
#include "scott/frame.h"
#include "scott/color_group.h"
#include "scott/colors.i"
#include "scott/interface.h"
#include <string.h>

class Input_Field
{

protected:

    Label *label;
    Text_Input *text_input;
    Frame *frame;

    // internal functions
    // -----------------------
    void make_label( float x, float y, float character_height, char *label_string );
    void make_text_input( int event_id, float x, float y, float character_height, char *text_input_string );
    void make_frame( float x, float y, float frame_width, float frame_height );
    void add_to_menu( Static_Menu *menu );
    void add_to_menu( Pop_Up_Menu *menu );

};
public:

// constructors

Input_Field( Static_Menu *menu, float x, float y, float frame_width,
float frame_height, float character_height,
char *label_string, char *text_input_string,
int event_id );

Input_Field( Static_Menu *menu, float x, float y, float frame_width,
float frame_height, float label_character_height,
char *label_string, float text_input_character_height,
char *text_input_string,
int event_id );

Input_Field( Pop_Up_Menu *menu, float x, float y, float frame_width,
float frame_height, float character_height,
char *label_string, char *text_input_string,
int event_id );

Input_Field( Pop_Up_Menu *menu, float x, float y, float frame_width,
float frame_height, float label_character_height,
char *label_string, float text_input_character_height,
char *text_input_string,
int event_id );

// destructor

~Input_Field();

// member functions

void set_string( const char *text_string,
int _perform_flag = PERFORM );

const char* get_string();

void change_text_color( Color_Group* _color );

void set_event_time_to_delayed();

void set_event_time_to_immediate();

};
// "internal_layout" header file

// Description: This is the main class for the internal layout
design module. The screen layout is drawn here
which includes a title bar, geometry areas for
top view, front view (cross section), and
isometric view, and layout workspace area.

// Tools Used: This module is coded in C++ under the object-oriented
design paradigm. Graphics utilize standard PHIGS when
they originate from within this module. Geometry that
is transferred from ACSYNT to this module is generated
within ACSYNT using the established ACSYNT geometry
pipeline. This pipeline uses characteristic points
and parameters to create a surface definition and then
displays the B-spline representation. Once the geometry
is fully defined in ACSYNT, the PHIGS structures
are copied into this interface. The interface for this
module is driven by SuperScott Woyak's GUI framework.

//

// Programmer: Shahab Hasan
// Original coding: 3/93
// Last modification: 9/93

//***************************************************************************

#ifndef _INTERNAL_LAYOUT
#define _INTERNAL_LAYOUT

#include <iostream.h> // the new C++ I/O library
#include <fstream.h> // the C++ file I/O library
#include <stdio.h> // the old C I/O library
#include <stdlib.h> // a basic library for C

/*-----------------------------------------------------------------------------
// include SuperScott's header files
//-------------------------------------------------------------------------------
#include "scott/PHIGS_resource_manager.h"
#include "scott/interface.h"
#include "scott/interface_manager.h"
#include "scott/open.h"
#include "scott/color.h"
#include "scott/colors.i"
#include "scott/event.h"
#include "scott/push_button.h"
#include "scott/seperator.h"
#include "scott/frame.h"

}}}
#include "scott/simple_view.h"
#include "scott/static_menu.h"
#include "scott/title_bar.h"

// -------------------------------------
// include my header files
// -------------------------------------
#include "component_manager.h"
#include "shahab_constants.h"
#include "workspace.h"

// Declare the classes invoked from interior_layout.C.
// -------------------------------------
class Component_Manager;
class Workspace;

class Internal_Layout
{
private:

    //----------
    // data members
    //----------
    Interface_Manager *interface_manager;
    Title_Bar *title_bar;
    Static_Menu *front_iso_view_background, *top_view_background, *workspace_background;
    Frame *front_view_frame, *top_view_frame, *iso_view_frame;
    Simple_View *front_view, *top_view, *iso_view;
    Simple_View *full_screen_view;
    Separator *view_separator;
    Workspace *workspace;
    Component_Manager *component_manager;
// internal functions

void open_workstation_and_interface_manager();
void create_views();
void create_backgrounds();
void add_views_and_backgrounds();
void create_workspace();
void create_external_components();
void start_main_event_loop();
void clean_up_and_exit();

public:

// constructor
Internal_Layout();

// destructor
-Internal_Layout();

// member functions

// void create_full_screen_view( PHIGS_Structure_ID* structure );
// void destroy_full_screen_view();

};

#endif


class Lavatory : public Parametric_Box
{

protected:

    // Place any additional data members associated with the Lavatory here.

public:

    //-----------
    // constructor
    //-----------
    Lavatory( int _component_number );

    //.CheckedChanged
    // destructor (virtual so that a derived destructor can be called)
    virtual ~Lavatory();

    //--------------
    // member functions
    //--------------

        // Place any additional member functions associated with
        // the Lavatory here.

    //--------------
    // reDefined virtual functions
    //--------------

};
#endif
// Lavatory_Manager class header file

// Programmer: shahab Hasan
// Original coding: 8/93
// Last modification: 10/93

#ifndef _LAVATORY_MANAGER
#define _LAVATORY_MANAGER

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "scott/pop_up_menu.h"
#include "scott/push_button.h"
#include "scott/radio_button.h"
#include "scott/radio_button_manager.h"
#include "scott/seperator.h"
#include "lavatory.h"
#include "shahab_constants.h"

class Lavatory_Manager
{

protected:

// The linked list pointers.

Lavatory* first_lavatory;
Lavatory* current_lavatory;

// Data stored by the Lavatory_Manager class.

int number_of_lavatories;
char number_of_lavatories_string[3];
int number_of_lavatories_made;
float initial_x_location;
char new_name_string[20];

```
The following chunk of strings are used when reading
in the data for lavatories from a data file. These
strings are filled once for each lavatory defined in the
file and the contents of the strings are then processed
to fill the data members for that lavatory.
The process is reversed when writing the data file.

char lav_name[20];
char lav_color_index_string[2];
char lav_left_width_string[10];
char lav_center_width_string[10];
char lav_right_width_string[10];
char lav_top_step_height_string[10];
char lav_bottom_step_height_string[10];
char lav_main_left_height_string[10];
char lav_main_right_height_string[10];
char lav_depth_string[10];
char lav_angle1_string[10];
char lav_angle2_string[10];
char lav_angle3_string[10];
char lav_angle4_string[10];
char lav_weight_string[10];
char lav_x_location_string[10];
char lav_y_location_string[10];
char lav_z_location_string[10];
char lav_x_rotation_string[10];
char lav_y_rotation_string[10];
char lav_z_rotation_string[10];

internal functions

void initialize_current_lavatory();
void remove_all_lavatories();
void read_lavatory_from_file( Interface_Manager* iman );
void write_lavatory_to_file();

public:

constructor

Lavatory_Manager();
// destructor (virtual so that a derived destructor can be called)
- Lavatory_Manager();

//----------
// member functions
//----------
int get_number_of_lavatories();
void invoke_add_remove_lavatory_menu(Interface_Manager* iman);
void invoke_remove_lavatory_menu(Interface_Manager* iman);
void invoke_move_modify_lavatory_menu(Interface_Manager* iman);
void invoke_error_menu(Interface_Manager* iman);
void add_lavatory(Interface_Manager* iman);
void remove_lavatory();
void modify_lavatory();

};

#endif
#ifndef _LD3
#define _LD3

#include "acsynt_component.h"
#include "parametric_box.h"

class LD3 : public Parametric_Box
{

protected:

    // Place any additional data members associated with the LD3 here.

public:

    //-------
    // constructor
    //-------
    LD3();

    // destructor (virtual so that a derived destructor can be called)
    //-------------------------
    virtual ~LD3();

    //-------
    // member functions
    //-------

    // Place any additional member functions associated with
    // the LD3 here.

    //-------------------
    // redefined virtual functions
    //-------------------

};
#endif
// **************************************************************************
//
// "main_shell" header file
//
// **NOTE**: This is **not** a class. "main_shell" is a so-called
// free subroutine (in Booch's terms).
//
// All this function does is provide an entry point into ACSYNT from
// from C++. This is necessary on the Iris because their
// implementation is, well, deficient, if you're polite.
// If you're not inclined to be polite about it (like me), it could
// be called pathetic, non-robust, and primitive.
// The problem is that this implementation (a translator, by the way)
// insists that C++ must have control over the whole program. This
// is achieved by giving it a function "main".
//
// Another interesting (ridiculous and unintuitive?) "feature" is
// that to link to FORTRAN subroutines, it is necessary to use the
// "extern" keyword to tell C++ that the function is written in
// another language. In itself, there is nothing wrong with this;
// C++ needs to be told not to mangle the function name. What *is*
// troublesome is that extern "C" must be used whether it is a C
// function or a FORTRAN subroutine. There is no such thing as an
// extern "FORTRAN" which is what the IBM implementation provides.
//
// **NOTE**: Neither of the above problems/solutions were resolved by
// SGI documentation or technical phone support. They are
// of the opinion that linkage between C++ and FORTRAN is
// inherently impossible - this was told to me by a SGI
// representative. Therefore, if there are future problems
// associated with the approach I have taken, do not count
// on help from Silicon Graphics.
//
// Programmer: Shahab Hasan
// Original coding: 7/93
//
// **************************************************************************

#ifndef _MAIN_SHELL
#define _MAIN_SHELL

#include <iostream.h>
#include "scott/configuration.dat"

void main();
#if WORKSTATION==SGI
    extern "C"
    {
        void main_();
        void abort_();
    }
#else
    extern 'FORTRAN'
    {
        void main();
    }
#endif

#endif
```cpp
// **********************************************************************************************
// Mid_Section class header file
// Programmer: Shahab Hasan
// Original coding: 4/93
// Last modification: 10/93
// **********************************************************************************************

#ifndef _MID_SECTION
#define _MID_SECTION

#include "acsynt_component.h"

class Mid_Section: public ACSYNT_Component
{

protected:

    float length, radius1, radius2, radius3, radius4;

public:

    //---------------------
    // constructor
    //---------------------
    Mid_Section( int _component_number );

    //----------------------------------------
    // destructor (virtual so that a derived destructor can be called)
    //----------------------------------------
    virtual ~Mid_Section();

    //---------------------
    // member functions
    //---------------------
    float get_length();
    float get_radius1();
    float get_radius2();
    float get_radius3();
    float get_radius4();
```
int set_length( float new_length );
int set_radius1( float new_radius1 );
int set_radius2( float new_radius2 );
int set_radius3( float new_radius3 );
int set_radius4( float new_radius4 );
int update_nose();
int update_aft_body();

// redefined virtual functions

void change_parameter( int index, float* value );
void get_component_data_from_acsynt();
void set_component_data_into_acsynt();
// **************************************************************************
// Nose class header file
//
// Programmer: Shahab Hasan
// Original coding: 4/93
// Last modification: 10/93
//
// ***************************************************************************

#ifndef _NOSE
#define _NOSE

#include "acsynt_component.h"

class Nose: public ACSYNT_Component
{

protected:
    float    length, tip_angle, shoulder_angle, dive_angle;
    float    radius1, radius2;

public:

    // constructor
    Nose( int _component_number );

    // destructor (virtual so that a derived destructor can be called)
    virtual ~Nose();

    // member functions
    //
    float    get_length();
    float    get_tip_angle();
    float    get_shoulder_angle();
    float    get_dive_angle();
    float    get_radius1();
    float    get_radius2();

};
int     set_length( float new_length );
int     set_tip_angle( float new_tip_angle);
int     set_shoulder_angle( float new_shoulder_angle );
int     set_dive_angle( float new_dive_angle );
int     set_radius1( float new_radius1 );
int     set_radius2( float new_radius2 );
int     update_mid_section();

//-----------------------------
// redefined virtual functions
//-----------------------------
void     change_parameter( int index, float* value );
void     get_component_data_from_acsynt();
void     set_component_data_into_acsynt();

);
#ifndef _PARAMETRIC_BOX
#define _PARAMETRIC_BOX

#include "acsynt_component.h"

class Parametric_Box : public ACSYNT_Component
{

protected:

    float main_left_height, total_left_height, main_right_height,
        total_right_height, total_width, left_width,
        center_width, right_width, depth,
        bottom_step_height, top_step_height;

    float lower_left_angle, upper_left_angle,
        upper_right_angle, lower_right_angle;

    float volume, weight, density, scale;

public:

    //-----------------
    // constructor
    //-----------------

    Parametric_Box( int _component_number );

    //-------------------------------
    // destructor (virtual so that a derived destructor can be called)
    //-------------------------------

    virtual ~Parametric_Box();
// -------------------------
// member functions
// -------------------------

float get_main_left_height();
float get_total_left_height();
float get_main_right_height();
float get_total_right_height();
float get_total_width();
float get_center_width();
float get_left_width();
float get_right_width();
float get_depth();
float get_bottom_step_height();
float get_top_step_height();
float get_lower_left_angle();
float get_upper_left_angle();
float get_upper_right_angle();
float get_lower_right_angle();
float get_volume();
float get_weight();
float get_density();

int set_main_left_height(float height);
int set_total_left_height(float height);
int set_main_right_height(float height);
int set_total_right_height(float height);
int set_total_width(float width);
int set_center_width(float width);
int set_left_width(float width);
int set_right_width(float width);
int set_depth(float new_depth);
int set_bottom_step_height(float height);
int set_top_step_height(float height);
int set_lower_left_angle(float angle);
int set_upper_left_angle(float angle);
int set_upper_right_angle(float angle);
int set_lower_right_angle(float angle);
int set_volume(float new_volume);
int set_weight(float new_weight);
int set_density(float new_density);
int scale_by(float scale_factor);

// -------------------------------
// redefined (inherited) virtual functions
// -------------------------------

void change_parameter(int index, float* value);
void get_component_data_from_acsynt();
void set_component_data_into_acsynt();

#endif
Passenger_Section class header file

Programmer: Shahab Hasan
Original coding: 8/93
Last modification: 10/93

ifndef _PAASSENGER_SECTION
#define _PAASSENGER_SECTION

#include "scott/number.h"
#include "scott/colors.1"
#include "scott/color_group.h"
#include "scott/label.h"
#include "scott/number_box.h"
#include "scott/push_button.h"
#include "internal_layout.h"
#include "input_field.h"
#include "section.h"
#include "seat_manager.h"
#include "galley_manager.h"
#include "lavatory_manager.h"

class File_Manager;

class Passenger_Section : public Section, public Seat_Manager,
                                public Galley_Manager, public Lavatory_Manager
{

protected:
// There is no real data stored by this class.
// The following chunk of data members consists of the 'cosmetics'
// applied to the actual data. The actual data is stored by the
// base classes that this class derives from. By "cosmetics" I mean
// this is the stuff for the interface: labels, buttons, and the
// like to make everything look good.

Interface_Manager *iman;
Static_Menu *menu;
Label *aisles_label, *lavatories_label,
     *galleys_label, *configuration_label;
Input_Field *number_of_seats_field, *aisle_width_field,
         *pitch_field, *configuration_field;
Push_Button *fill_button, *modify_seats_button,
         *lavatories_button1, *lavatories_button2,
         *galleys_button1, *galleys_button2;
Number *lavatories_number, *galleys_number;
Number_Box *aisles_number_box, *lavatories_number_box,
           *galleys_number_box;
Separator *separator2, *separator3,
          *separator4;

// internal functions
//
void process_input_data();

public:

//
// constructor
//
Passenger_Section( Interface_Manager *iman, Static_Menu *menu );

// destructor (virtual so that a derived destructor can be called)
//
virtual ~Passenger_Section();

120
// member functions
//
//
void    print_data();
void    turn_ETC_off();
void    create_configuration_menu_items();
void    destroy_configuration_menu_items();
void    update_number_of_seats( int TURN_YELLOW = NO );
void    update_number_of_galleys_and_lavatories();

// friend functions
//
friend void    File_Manager::read_passenger_section_from_file
               ( Passenger_Section* in_section );
friend void    File_Manager::write_passenger_section_to_file
               ( Passenger_Section* out_section );

// inherited virtual functions
//
void    initialize_the_section();
void    initialize_blank_section();
void    create_menu_items();
void    destroy_menu_items();
float   calculate_length();
float   calculate_end_location();

);
ifndef _SECTION_TEMPLATE
#define _SECTION_TEMPLATE

#include <stdlib.h>
#include "scott/interface_manager.h"
#include "scott/static_menu.h"
#include "scott/push_button.h"
#include "scott/label.h"
#include "scott/radio_button.h"
#include "scott/radio_button_manager.h"
#include "scott/number.h"
#include "scott/colors.i"
#include "input_field.h"
#include "section.h"

class Section;

class Section_Template
{

protected:
The data members of this class exist only to support the display of data from other classes. More specifically, this class displays the data of the "section" class and the classes derived from "section" ("passenger_section", "cargo_section" and "fuel_section"). It is a classic "using" relationship between classes: the "section" class "uses" this class.

-----------------------------------------------

Interface_Manager
Static_Menu
Label

Input_Field

RadioButton_Manager
RadioButton
PushButton
Separator

-----------------------
// internal functions
-----------------------

void create_menu_items();
void free_memory();

public:

-------------------
// constructor
-------------------
Section_Template( Interface_Manager* _iman, Static_Menu* _menu );

-------------------
// destructor
-------------------
~Section_Template();

-----------------------
// member functions
-----------------------

void update_template( Section* section );
void turn_ETC_off();
// ******************************************************
// Seat class header file
// Programmer: Shahab Hasan
// Original coding: 4/93
// Last modification: 10/93
// ******************************************************

#ifndef __SEAT
#define __SEAT

#include "acsynt_component.h"

class Seat : public ACSYNComponent
{

protected:
    float height, width, length, angle;
    float bottom_thickness, back_thickness;
    float rho_values[t];

public:

    //-------------
    // constructor
    //-------------
    Seat( int _component_number );

    //-----------------------------------------------
    // destructor (virtual so that a derived destructor can be called)
    //-----------------------------------------------
    virtual ~Seat();
// member functions

float get_height();
float get_width();
float get_length();
float get_angle();
float get_bottom_thickness();
float get_back_thickness();
float get_rho1();
float get_rho2();
float get_rho3();
float get_rho4();
float get_rho5();
float get_rho6();

int set_height( float new_height );
int set_width( float new_width );
int set_length( float new_length );
int set_angle( float new_angle );
int set_bottom_thickness( float new_bottom_thickness );
int set_back_thickness( float new_back_thickness );
int set_rho1( float new_rho1 );
int set_rho2( float new_rho2 );
int set_rho3( float new_rho3 );
int set_rho4( float new_rho4 );
int set_rho5( float new_rho5 );
int set_rho6( float new_rho6 );

// redefined virtual functions

void change_parameter( int index, float* value );
void get_component_data_from_acsynt();
void set_component_data_into_acsynt();

};

#endif
// ******************************************************
// // Seat_Manager class header file
// // Programmer: Shahab Hasan
// // Original coding: 8/93
// // Last modification: 10/93
// // ******************************************************

#ifndef _SEAT_MANAGER
#define _SEAT_MANAGER

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "seat.h"

class Seat_Manager
{

protected:

    //---------------------
    // Linked list pointer.
    //---------------------
    Seat* first_seat;

}
// Shahab Hasan note: What are all these strings for?
//
// 1) Most are necessary because the user interaction with the
data is through text. Therefore, the data must be stored in its
natural form and also redundantly as a string. For example, the
'aisle_width' is of type float, but for the user to change it,
he/she changes a text string.
// 2) File storage. I store everything as a text string, so that's
the format I need the data in. Why is everything stored as a
text string? Well, it just seems to work better and the file
is more readable.
//
// One or both of these reasons apply to all the strings.

char number_of_seats_string[4];
char aisle_width_string[5];
char pitch_string[5];
char configuration_strings[5][2];
char number_of_aisles_string[2];
char seat_color_index_string[2];
char seat_width_string[5];
char seat_height_string[5];
char seat_length_string[5];
char seatback_angle_string[5];
char back_thickness_string[5];
char bottom_thickness_string[5];
char armrest_width_string[5];
char dual_armrest_width_string[5];
char seat_x_location_string[6];
char seat_y_location_string[6];
char seat_z_location_string[6];
char seat_x_rotation_string[6];
char seat_y_rotation_string[6];
char seat_z_rotation_string[6];

// This is the real data.

int number_of_seats, number_of_aisles, *configuration,
    number_of_columns, number_of_rows, seat_color_index;
float aisle_width, pitch, section_width, seatback_angle,
    seat_width, seat_length, seat_height, back_thickness,
    bottom_thickness, armrest_width, dual_armrest_width,
    seat_x_location, seat_y_location, seat_z_location,
    seat_x_rotation, seat_y_rotation, seat_z_rotation,
x_location, y_location, z_location;
internal functions

void add_seats( Interface_Manager* _iman, int delta_seats );
void remove_seats( int delta_seats );
void move_the_seat( Seat* seat );

public:

constructor
Seat_Manager();

destructor (virtual so that a derived destructor can be called)
~Seat_Manager();

member functions

void initialize_the_seats( Interface_Manager* iman,
float _start_location );
void place_the_seats( float _start_location );
void auto_fill( Interface_Manager* iman,
float _start_location, float _end_location );
void modify_the_seats();
int get_number_of_seats();
void set_number_of_seats( Interface_Manager* _iman,
const char* new_number_of_seats_string );
float get_pitch();
void set_pitch( const char* new_pitch_string );
float get_aisle_width();
void set_aisle_width( const char* new_aisle_width_string );
int get_number_of_aisles();
void set_number_of_aisles( int new_number_of_aisles );
int get_configuration_value1();
int get_configuration_value2();
int get_configuration_value3();
int get_configuration_value4();
int get_configuration_value5();

128
void set_configuration_value1(const char* value_string);
void set_configuration_value2(const char* value_string);
void set_configuration_value3(const char* value_string);
void set_configuration_value4(const char* value_string);
void set_configuration_value5(const char* value_string);

#endif
// *******************************************************
// Section class header file
// Programmer: Shahab Hasan
// Original coding: 5/93
// Last modification: 10/93
//***************************************************************

#ifdef _SECTION
#define _SECTION

#include <stdlib.h>
#include "scott/dll_element.h"

class Section : public DLL_Element< Section >
{

protected:

﻿// The following chunk of data members consists of the actual
// data managed by the section class. By "actual data" I mean this
// is the good stuff: the hard numbers that need to be stored.
//---------------------------------------------------------------
    char    section_type[9];
    int     section_type_toggle;
    char    section_name[32];
    char    deck_name[22];
    char    start_location_string[7];
    float   start_location;
    char    end_location_string[7];
    float   end_location;
    char    length_string[7];
    float   length;
    int     end_location_flag;
    char    end_location_flag_string[2];

    // internal functions
    //---------------

public:

}
// constructor

Section();

// destructor (virtual so that a derived destructor can be called)

virtual ~Section();

// member functions

char* get_section_name();
void set_section_name( const char* _section_name );

char* get_section_type();
void change_section_type();

char* get_deck_name();
void set_deck_name( const char* _deck_name );
void change_deck();

char* get_start_location_string();
float get_start_location();
void set_start_location( const char* _start_location_string );

char* get_end_location_string();
float get_end_location();
void set_end_location( const char* _end_location_string );

void change_to_fixed_end_location();
void change_to_calculated_end_location();
char* get_end_location_flag_string();
int get_end_location_flag();

char* get_length_string();
float get_length();
void set_length( const char* _length_string );
// virtual functions
virtual void create_menu_items();
virtual void destroy_menu_items();
virtual float calculate_length();
virtual float calculate_end_location();
virtual void initialize_the_section();
virtual void initialize_blank_section();
// Strake class header file
//
// Programmer: Shahab Hasan
// Original coding: 5/93
// Last modification: 9/93
//
// ****************************************************************************

#ifndef _STRAKE
#define _STRAKE

#include "acsynt_component.h"
#include "wing.h"

class Strake : public Wing
{

protected:

    // Place any additional data members associated with
    // the strake here.

public:

    //-------------------
    // constructor
    //-------------------

    Strake( int _component_number );

    //-------------------------------------------------------------------------------
    // destructor (virtual so that a derived destructor can be called)
    //-------------------------------------------------------------------------------

    virtual ~Strake();

    //-------------------
    // member functions
    //-------------------

    // Place any additional member functions associated with
    // the strake here.

    //-------------------------------------------------------------------------------
    // redefined virtual functions
    //-------------------------------------------------------------------------------

};

#endif


/*
 * This file contains all the constants I use in the
 * internal_layout module.
 */

Programmer: Shahab Hasan
Original coding: 9/93
Last modifications: 10/93

.getOrElse(shahab_constants.h

 ifndef _SHAHAB_CONSTANTS
 define _SHAHAB_CONSTANTS

#define WORKSTATION

#elif defined(WORKSTATION
#endif

#else

#endif

/*
 * Define the workstation id constant.
 */

Shahab Hasan note: For the SGI, the wsid can only be 1 (only one
workstation is allowed). For other implementations,
the wsid should be different from 1 because that
is the wsid for the ACSYNT workstation. It must be
equal to or less than 5, however; this restriction
is imposed by Scott "the Viking" Woyak's framework.

 if WORKSTATION==SGI
   const int wsid = 1;
 else
   const int wsid = 3;

#endif

/*
 * Define constants to facilitate the placement
 * menus, views, and other entities.
 */

const float MARGIN = 0.10/12.0;
const float WIDTH = 0.30;

/*
 * Define constants so I can update the PHISS structures
 * of only translations, rotations, or color changes.
 */

const int FULL = 1;
const int PARTIAL = 2;
// Define constants for the end location of sections.
//
const int FIXED = 1;
const int CALCULATED = 2;

// Define some font sizes.
//
#if WORKSTATION==SGI
  const float FONT_SIZE1 = 1.4;
  const float FONT_SIZE2 = 1.2;
  const float BUTTON_HEIGHT = 3.25;
  const float RADIO_BUTTON_HEIGHT = 2.4;
#else
  const float FONT_SIZE1 = 1.25;
  const float FONT_SIZE2 = 1.2;
  const float BUTTON_HEIGHT = 2.85;
  const float RADIO_BUTTON_HEIGHT = 2.05;
#endif

// Define constants for ACSYNT component types.
//
const int NOSE = 1;
const int MID_SECTION = 2;
const int AFT_BODY = 3;
const int WING = 4;
const int HORIZONTAL_TAIL = 5;
const int VERTICAL_TAIL = 6;
const int CANARD = 7;
const int STRAKE = 8;
const int SEAT = 31;
const int BOX = 32;
// Define constants for all the menu items on the screen.

const int CLOSE_BUTTON = 1;
const int HELP_BUTTON = 2;
const int FILE_BUTTON = 3;
const int OPTIONS_BUTTON = 5;
const int SECTION_NAME_FIELD = 6;
const int DECK_BUTTON = 7;
const int SECTION_TYPE_BUTTON = 8;
const int START_LOCATION_FIELD = 9;
const int END_LOCATION_FIELD = 10;
const int LENGTH_FIELD = 11;
const int FIXED_END_BUTTON = 12;
const int CALCULATED_END_BUTTON = 13;
const int NUMBER_OF_SEATS_FIELD = 14;
const int AUTO_FILL_BUTTON = 15;
const int PITCH_FIELD = 16;
const int AISLE_WIDTH_FIELD = 17;
const int AISLES_NUMBER_BOX = 18;
const int CONFIGURATION_FIELD1 = 19;
const int CONFIGURATION_FIELD2 = 20;
const int CONFIGURATION_FIELD3 = 21;
const int CONFIGURATION_FIELD4 = 22;
const int CONFIGURATION_FIELD5 = 23;
const int MODIFY_SEATS_BUTTON = 24;
const int GALLEY_BUTTON1 = 25;
const int GALLEY_BUTTON2 = 26;
const int LAVATORY_BUTTON1 = 27;
const int LAVATORY_BUTTON2 = 28;
const int APPLY_BUTTON = 29;
const int NEXT_SECTION_BUTTON = 30;
const int PREVIOUS_SECTION_BUTTON = 31;
const int GOTO_SECTION_FIELD = 32;
const int ADD_SECTION_BUTTON = 33;
const int DELETE_SECTION_BUTTON = 34;

#endif
#ifndef _TOTALS_MANAGER
#define _TOTALS_MANAGER

#include "scott/frame.h"
#include "scott/interface_manager.h"
#include "scott/label.h"
#include "scott/number.h"
#include "scott/static_menu.h"
#include "input_field.h"
#include "shahab_constants.h"

class Passenger_Section;

class Totals_Manager
{
protected:

   // ------------------------------------------------------------------------
   // The following chunk of data members consists of the actual
   // data managed by the workspace. By 'actual data' I mean this is the
   // good stuff: the hard numbers that need to be stored.
   // ------------------------------------------------------------------------
   int total_seats;
   int total_lavatories;
   int total_galleys;
   float passengers_per_lavatory;
   float total_galley_volume;
   float galley_volume_per_passenger;

   /*
    int total_LD3s;
   int total_727s;
       float total_container_volume;
   */
The following chunk of data members consists of the cosmetics.
By "cosmetics" I mean this is the stuff for the interface:
labels, buttons, and the like to make everything look good.

Interface_Manager* iman;
Static_Menu* menu;
Label
  *title_label,
  *passenger_accomodations_label, *cargo_label,
  *fuel_label, *seats_label,
  *lavatories_label, *pass_per_lav_label,
  *galleys_label, *gal_vol_per_pass_label,
  *LD3s_label, *temp_label1,
  *total_727s_label, *temp_label2,
  *other_boxes_label, *temp_label3,
  *vol_of_other_boxes_label, *temp_label4,
  *total_box_vol_label, *temp_label5,
  *fuel_tanks_label, *temp_label6,
  *fuel_tanks_vol_label, *temp_label7,
  *fuel_tanks_weight_label, *temp_label8,
  *fuel_volume_label, *temp_label9,
  *fuel_weight_label, *temp_label10;
Input_Field
  *density_of_add_fuel,
  *add_fuel_vol,
  *add_fuel_weight;
Number
  *seats_number, *lavatories_number,
  *pass_per_lav_number, *galleys_number,
  *gal_vol_per_pass_number;
Frame
  *menu_frame, *passenger_accomodations_frame,
  *cargo_frame, *fuel_frame;

// internal functions
void initialize_totals();
void calculate_totals( Passenger_Section* first_section );
void create_totals_menu();

public:

// constructor
Totals_Manager( Interface_Manager* _iman, Passenger_Section* first_section );
// destructor (virtual so that a derived destructor can be called)
virtual ~Totals_Manager();

// member functions

void hide();
void show();
void update_totals( Passenger_Section* first_section );

};

#endif
#ifndef _UNDEFINED_COMPONENT
#define _UNDEFINED_COMPONENT

#include "acsysnt_component.h"

class Undefined_Component : public ACSYNT_Component
{

protected:

public:

    // constructor
    Undefined_Component( int _component_number );

    // destructor (virtual so that a derived destructor can be called)
    virtual ~Undefined_Component();

};

#endif
/* Vertical_Tail class header file */

Programmer: Shahab Hasan
Original coding: 5/93
Last modification: 9/93

/* Vertical_Tail */

#ifndef _VERTICAL_TAIL
#define _VERTICAL_TAIL

#include "acsynt_component.h"
#include "wing.h"

class Vertical_Tail : public Wing
{
protected:
    // Place any additional data members associated with
    // the vertical tail here.

public:
    //------------------
    // constructor
    ------------------
    Vertical_Tail( int _component_number );

    // destructor (virtual so that a derived destructor can be called)
    virtual ~Vertical_Tail();

    // member functions
    //------------------
    // Place any additional member functions associated with
    // the vertical tail here.

    // redefined virtual functions
    //-----------------------------
};
#endif
// **********************************************************************************************
// Wing class header file
//
// Programmer: Shahab Hasan
// Original coding: 5/93
// Last modification: 10/93
//
// **********************************************************************************************

#ifndef _WING
#define _WING

#include "acsynt_component.h"

class Wing : public ACSYNT_Component
{

protected:

    float    span, leading_edge_sweep, quarter_chord_sweep,
             aspect_ratio, area, dihedral, twist, taper_ratio,
             root_chord;

    int      airfoil_type, root_airfoil_number, tip_airfoil_number;

public:

    //--------
    // constructor
    //--------
    Wing( int _component_number );

    //---------------------------------------------------------------------
    // destructor (virtual so that a derived destructor can be called)
    //---------------------------------------------------------------------
    virtual ~Wing();
// member functions
=-=-=-=-=

float get_span();
float get_leading_edge_sweep();
float get_quarter_chord_sweep();
float get_aspect_ratio();
float get_area();
float get_dihedral();
float get_twist();
float get_taper_ratio();
float get_root_chord();
int get_airfoil_type();
int get_root_airfoil_number();
int get_tip_airfoil_number();

int set_span( float new_span );
int set_leading_edge_sweep( float new_leading_edge_sweep );
int set_quarter_chord_sweep( float new_quarter_chord_sweep );
int set_aspect_ratio( float new_aspect_ratio );
int set_area( float new_area );
int set_dihedral( float new_dihedral );
int set_twist( float new_twist );
int set_taper_ratio( float new_taper_ratio );
int set_root_chord( float new_root_chord );
int set_airfoil_type( int new_airfoil_type );
int set_root_airfoil_number( int new_airfoil_number );
int set_tip_airfoil_number( int new_airfoil_number );

// redefined virtual functions
=-=-=-=-=

void change_parameter( int index, float* value );
void get_component_data_from_acsyt();
void set_component_data_into_acsyt();

);
// Workpace class header file
//
// Programmer: Shahab Hasan
// Original coding: 4/93
// Last modification: 10/93
//
// *****************************************************************************

#ifndef _WORKSPACE
#define _WORKSPACE

#include "scott/colors.i"
#include "scott/interface_manager.h"
#include "scott/label.h"
#include "scott/number.h"
#include "scott/pop_up_menu.h"
#include "scott/push_button.h"
#include "scott/radio_button.h"
#include "scott/radio_button_manager.h"
#include "scott/static_menu.h"

#include "file_manager.h"
#include "input_field.h"
#include "parametric_box.h"
#include "passenger_section.h"
#include "section.h"
#include "section_template.h"
#include "shahab_constants.h"
#include "totals_manager.h"

class Section;
class Passenger_Section;
class FileManager;

class Workspace : public FileManager
{

protected:

    //-------------------------------
    // The pointers for the linked list of sections.
    //-------------------------------
    Section *first_section;
    Passenger_Section *current_section;

};
// There is one 'section_template' used to display information
// that all sections have in common.

Section_Template* section_template;

// The *totals_manager* displays a menu with all the totals.

Totals_Manager* totals_manager;
int totals_on_screen;

// The following chunk of data members consists of the actual
// data managed by the workspace. By "actual data" I mean this is the
// good stuff: the hard numbers that need to be stored.

char number_of_sections_string[3];
int number_of_sections;
int number_of_decks;

// The following chunk of data members consists of the cosmetics.
// By "cosmetics" I mean this is the stuff for the interface:
// labels, buttons, and the like to make everything look good.

Static_Menu *background;
Pop_Up_Menu *file_io_menu;
Radio_Button_Manager *read_write_button_manager;
Radio_Button *read_button, *write_button;
Push_Button *close_button, *help_button,
*file_button, *options_button,
*next_section_button,
*previous_section_button, *apply_button;
Input_Field *current_file_field, *goto_section_field;
Separator *separator5;
Frame *file_buttons_frame, *section_template_frame,
*sections_and_decks_frame;
Label *number_of_sections_label,
*number_of_decks_label,
*data_file_label, *current_file_label;
Number *number_of_sections_number,
*number_of_decks_number;
// internal functions

void initialize();
void create_file_operations_area();
void create_sections_and_decks_area();
void create_options_button();
void create_section_buttons();
void create_totals_area();
void add_section_to_list( Section* section );
void update_end_location_and_length();
void free_memory();

public:

// constructor

Workspace( Interface_Manager* _iman, Static_Menu* _background );

// destructor (virtual so that a derived destructor can be called)

virtual ~Workspace();

// member functions

// File management functions.
// (most of this is handled by the *File_Manager*
// class which this class inherits from)

void invoke_file_menu();
void read_data_file();
void write_data_file();

// General section functions.

void change_section_name( Event* event );
void change_section_type();
void change_start_location( Event* event );
void change_end_location( Event* event );
void change_to_fixed_end_location();
void change_to_calculated_end_location();
void change_length( Event* event );
// Passenger section functions.

void change_number_of_seats( Event* event );
void auto_fill_current_section();
void change_pitch( Event* event );
void change_aisle_width( Event* event );
void change_number_of_aisles( Event* event );
void change_configuration_field1( Event* event );
void change_configuration_field2( Event* event );
void change_configuration_field3( Event* event );
void change_configuration_field4( Event* event );
void change_configuration_field5( Event* event );
void modify_seats();
void add_or_remove_galley();
void move_or_modify_galley();
void add_or_remove_lavatory();
void move_or_modify_lavatory();

// Section management functions.

void remove_all_sections();
void remove_section();
void add_blank_passenger_section();
int get_number_of_sections();
void set_number_of_sections( int _number_of_sections );
void increment_number_of_sections();
void decrement_number_of_sections();
goto_next_section();
goto_previous_section();
goto_section( const char* section_name );

// Deck management functions.

int get_number_of_decks();
void set_number_of_decks( int _number_of_decks );
void increment_number_of_decks();
void decrement_number_of_decks();

// Other functions.

void invoke_options_menu();
void turn_ETC_off();
void test_function();

};

#endif
Appendix B: Source code file "acs_seat.vpi"

The actual source code for the generation of the seat component is contained in this appendix. It is included as a guide for future programmers adding new components to ACSYNT.
SUBROUTINE LDSEAT (ICOMP)

INTEGER ICOMP, ITEM, XSTYPE, IERR, NPOINT, NXSECT, ISECT, IPOINT
REAL PT(3), WIDTH

C*** Item seventeen is the cross section type in the data structure,
C*** item five is the number of cross sections in the component.
PARAMETER (ITEM1 = 17, ITEM2 = 5)

C GET THE CROSS SECTION TYPE AND THE NUMBER OF CROSS SECTIONS
CALL GTICMP(ITEM1, ICOMP, XSTYPE, IERR)
CALL GTICMP(ITEM2, ICOMP, NXSECT, IERR)

C LOAD CROSS SECTION TYPE INTO THE CROSS SECTION DATA STRUCTURE
CALL LDXSTP(ICOMP, NXSECT, XSTYPE)

C GET THE NUMBER OF POINTS REQUIRED TO DEFINE THE CROSS SECTION
CALL GTXSNP(XSTYPE, NPOINT, IERR)

C GET THE WIDTH OF THE SEAT USING THE FIRST (#1) AND LAST (#19)
C CHARACTERISTIC POINTS
    CALL GTCPPM(ICOMP, 19, PT, IERR)
    WIDTH = PT(1)
    CALL GTCPPM(ICOMP, 1, PT, IERR)
    WIDTH = WIDTH - PT(1)
C LOAD THE DEFINING POINTS FOR EACH CROSS SECTION
C INTO THE CROSS SECTION DATA STRUCTURE
C**** Shahab Hasan note:  There are nineteen characteristic points
C**** defining the seat but the last point
C**** is only used to define the width. The first
C**** eighteen points are all on the first cross-
C**** section. Once the width is determined (done
C**** above), the points for the subsequent cross
C**** sections aren't calculated; they are
C**** just offset from the points on the first
C**** cross section.
DO 200 ISECT = 1, NXSECT
   DO 100 IPOINT = 1, NPOINT

C Get the defining point on the first cross section.
CALL GTCPPM(ICOMP, IPOINT, PT, IERR)

C Adjust the X location of the point depending on which
C cross section it lies upon.
PT(1) = PT(1) +
   > ((REAL(ISECT) - 1.0)/(REAL(NXSECT) - 1.0) * WIDTH)

C Store the point in the cross section data structure
CALL PTRXS(ICOMP, ISECT, IPOINT, PT, IERR)

100 CONTINUE
200 CONTINUE
RETURN
END
**C** SEATPTS: SEAT Points

Shahab Hasan note: The corresponding routines for other cross-sections are CNCPUT, ELIPPUT, AF1PUT, and GCNCPUT. I didn't use the "PUT" extension; I felt that PTS more accurately reflects the purpose of the routine, that is to find the points on the cross section. The old extension of "PUT" probably meant "Points and U Tangents" but after the B-spline integration, tangents are not needed.

**C** INPUT PARAMETERS:

- COMPN - INTEGER, COMPONENT NUMBER
- XSN - INTEGER, CROSS SECTION NUMBER
- NPNTS - INTEGER, NUMBER OF POINT PER CROSS SECTION
- LOCSYM - INTEGER, LOCAL SYMMETRY INDICATOR
- XA, YA, ZA - INTEGER, AXIS INDICES FROM CROSS SECTION NORMAL

**C** COMMON OUTPUT PARAMETERS:

- P(3,*) - REAL, 3 D POINTS (into the surface data structure)

**C** FUNCTIONAL DESCRIPTION:

Shahab Hasan note: This routine is called from SGDPUT located in "ageom4.vpi" and is invoked for each cross section of the component.

**C** CODED BY: SHAHAB HASAN

**C** DATE: 2/5/93

**C** DEVELOPMENT SITE:

- VIRGINIA TECH CAD LAB

SUBROUTINE SEATPTS(COMPN, XSN, NPNTS, LOCSYM, XA, YA, ZA, P)

REAL P1(3), P2(3), P3(3), P4(3), P5(3), P6(3), P7(3), P8(3)
REAL P17(3), P18(3), P19(3)
REAL RHO(6), P(3,*);
INTEGER COMPN, XSN, NXSECT, NPNTS, LOCSYM, XA, YA, ZA
PARAMETER (IPOINT=1)

C RETRIEVE THE NINETEEN POINTS DEFINING THE CROSS SECTION
CALL SEATPNT(COMPN, XSN, P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, > P11, P12, P13, P14, P15, P16, P17, P18, P19)

C RETRIEVE THE RHO'S FOR THE CROSS SECTION
There are six corners for the seat; each has an associated rho value. The rho's do not vary between cross sections.

CALL SEATRHO(COMPN, RHO)

FIND AND STORE SURFACE POINTS ALONG THE CROSS SECTION

The corresponding routines for other cross-sections are CNCDIV, ELIDIV, AF1DIV, and GCNCDIV. I didn't use the *DIV* extension; I felt that SRF more accurately reflects the purpose of the routine, that is to find surface points.

CALL SEATSRF(P1, P2, P3, P4, P5, P6, P7, P8, P9, P10, P11, P12, P13, P14, P15, P16, P17, P18, P19, RHO, XSN, NPNTS, XA, YA, ZA, P)

STORE THE POINT IN THE SURFACE DATA STRUCTURE

DO 100 INODE = 1, NPNTS
   CALL PTRSFP(XSN, INODE, IPOINT, COMPN, F(1, INODE), IERR)
100 CONTINUE

RETURN

END
C===============================================================================
C SEATPNT: SEAT POINT
C**** Shahab Hasan note: Do not be confused by the naming of this
C**** routine and the routine that it is called
C**** from, "SEATPTS". This routine retrieves
C**** the points on the cross section which are
C**** stored in the cross section data structure.
C**** The calling routine, "SEATPTS" oversees the
C**** entire process of finding all the surface
C**** points for the seat and this routine is
C**** invoked for each cross section. The
C**** limitations of FORTRAN naming convention
C**** precludes a more descriptive routine name.
C===============================================================================
C INPUT PARAMETERS:
C COMPN - INTEGER, COMPONENT NUMBER
C XSN - INTEGER, CROSS SECTION NUMBER
C===============================================================================
C OUTPUT PARAMETERS:
C P1()...P19() - REAL - THE NINETEEN POINTS DEFINING THE
C FIRST CROSS SECTION OF THE SEAT
C===============================================================================
C FUNCTIONAL DESCRIPTION:
C RETRIEVE NINETEEN POINTS DEFINING A SEAT CROSS SECTION
C===============================================================================
C CODED BY: SHAHAB HASAN
C DATE: 2/5/93
C===============================================================================
C DEVELOPMENT SITE:
C VIRGINIA TECH CAD LAB
C===============================================================================

SUBROUTINE SEATPNT(COMPN,XSN,P1,P2,P3,P4,P5,P6,P7,P8,P9,
> P10,P11,P12,P13,P14,P15,P16,P17,P18,P19)

REAL P1(3),P2(3),P3(3),P4(3),P5(3),P6(3),P7(3),P8(3),P9(3)
REAL P17(3),P18(3),P19(3)
INTEGER COMPN, XSN, IERR

C RETRIEVE NINETEEN POINTS DEFINING SEAT CROSS SECTION
C**** Shahab Hasan note: Yes, this is an extremely ugly, inelegant
C**** way to do this.
CALL GTRXS(COMPN, XSN, 1, P1, IERR)
CALL GTRXS(COMPN, XSN, 2, P2, IERR)
CALL GTRXS(COMPN, XSN, 3, P3, IERR)
CALL GTRXS(COMPN, XSN, 4, P4, IERR)
CALL GTRXS(COMPN, XSN, 5, P5, IERR)
CALL GTRXS(COMPN, XSN, 6, P6, IERR)
CALL GTRXS(COMPN, XSN, 7, P7, IERR)
CALL GTRXS(COMPN, XSN, 8, P8, IERR)
CALL GTRXS(COMP, XSN, 9, P9, IERR)
CALL GTRXS(COMP, XSN, 10, P10, IERR)
CALL GTRXS(COMP, XSN, 11, P11, IERR)
CALL GTRXS(COMP, XSN, 12, P12, IERR)
CALL GTRXS(COMP, XSN, 13, P13, IERR)
CALL GTRXS(COMP, XSN, 14, P14, IERR)
CALL GTRXS(COMP, XSN, 15, P15, IERR)
CALL GTRXS(COMP, XSN, 16, P16, IERR)
CALL GTRXS(COMP, XSN, 17, P17, IERR)
CALL GTRXS(COMP, XSN, 18, P18, IERR)
CALL GTRXS(COMP, XSN, 19, P19, IERR)

IF (IERR .EQ. 1) THEN
  CALL CONERR('SEATPNT','ERROR IN RETRIEVING X-SECTION POINTS')
ENDIF

RETURN
END
SUBROUTINE SEATRHO(COMPN, RHO)

INTEGER COMPN
REAL RHO(*)

GET THE RHOS FROM THE DATABASE

C*** Shahab Hasan note: The rho values are stored in slots 50 through 55 in the data structure. Why such high numbers? Jus' felt like it, bro!

CALL GTRCMP(50, 6, COMPN, RHO, IERR)

RETURN
END
SUBROUTINE SEATSRF(P1,P2,P3,P4,P5,P6,P7,P8,P9,P10,P11,P12,P13,
    >     P14,P15,P16,P17,P18,P19,RHO,XSN,NPNTS,
    >     XA,YA,ZA,P)

REAL     P1(3),P2(3),P3(3),P4(3),P5(3),P6(3),P7(3),P8(3),P9(3)
REAL     P17(3),P18(3),P19(3)
REAL     P(3,*), RHO(*)
INTEGER   XSN,NPNTS,XA,YA,ZA,INODE
REAL     U, U1, PP(3)
INTEGER   IPOINT,IBERR
PARAMETER (IPOINT=1)

C LOOP TO FIND POINTS
DO 100 INODE=1,NPNTS

C FIND PARAMETRIC VARIABLE VALUE
U = FLOAT( NPNTS-INODE )/FLOAT( NPNTS-1 )

C* FIND THE SURFACE POINT FOR THE GIVEN U VALUE
C*** Shahab Hasan note: In each IF and ELSSIF chunk of code,
C****
C**** the value of U is scaled so that it
C**** is between 0.0 and 1.0 because the
C**** routines that calculate the point
C**** (CONICP and LINEP) act on individual
C**** segments of curves. The scaled value
C**** is called U1.
C**** Also, CONICP is located in "ageox4.vpi"
C**** while LINEP is located in this file.
C****
IF (U .GE. 0.0) .AND. U .LE. 1.0/12.0) THEN
  U1 = 12.0 * U
  CALL CONICP(P1,P3,P2,RHO(1),U1,PP)
ELSEIF (U .GT. 1.0/12.0) .AND. U .LT. 1.0/6.0) THEN
  U1 = 12.0 * (U - 1.0/12.0)
  CALL LINEP(P3,P4,U1,PP)
ELSEIF (U .GE. 1.0/6.0) .AND. U .LE. 0.25) THEN
  U1 = 12.0 * (U - 1.0/6.0)
  CALL CONICP(P4,P6,P5,RHO(2),U1,PP)
ELSEIF (U .GT. 0.25 .AND. U .LT. 1.0/3.0) THEN
  U1 = 12.0 * (U - 0.25)
  CALL LINEP(P6,P7,U1,PP)
ELSEIF (U .GE. 1.0/3.0) .AND. U .LE. 5.0/12.0) THEN
  U1 = 12.0 * (U - 1.0/3.0)
  CALL CONICP(P7,P9,P8,RHO(3),U1,PP)
ELSEIF (U .GT. 5.0/12.0) .AND. U .LT. 0.5) THEN
  U1 = 12.0 * (U - 5.0/12.0)
  CALL LINEP(P9,P10,U1,PP)
ELSEIF (U .GE. 0.5 .AND. U .LE. 7.0/12.0) THEN
  U1 = 12.0 * (U - 0.5)
  CALL CONICP(P10,P12,P11,RHO(4),U1,PP)
ELSEIF (U .GT. 7.0/12.0) .AND. U .LT. 2.0/3.0) THEN
  U1 = 12.0 * (U - 7.0/12.0)
  CALL LINEP(P12,P13,U1,PP)
ELSEIF (U .GE. 2.0/3.0) .AND. U .LE. 3.0/4.0) THEN
  U1 = 12.0 * (U - 2.0/3.0)
  CALL CONICP(P13,P15,P14,RHO(5),U1,PP)
ELSEIF (U .GT. 3.0/4.0) .AND. U .LT. 5.0/6.0) THEN
  U1 = 12.0 * (U - 3.0/4.0)
  CALL LINEP(P15,P16,U1,PP)
ELSEIF (U .GE. 5.0/6.0) .AND. U .LE. 11.0/12.0) THEN
  U1 = 12.0 * (U - 5.0/6.0)
  CALL CONICP(P16,P18,P17,RHO(6),U1,PP);
ELSEIF (U .GT. 11.0/12.0 .AND. U .LE. 1.0) THEN
    U1 = 12.0 * (U - 11.0/12.0)
    CALL LINEP(P18,P1,U1,PP)
ENDIF

C
SET THE POINT IN THE PROPER ARRAY
P(XA,INODE) = PP(XA)
P(YA,INODE) = PP(YA)
P(ZA,INODE) = PP(ZA)

100 CONTINUE

RETURN
END
LINEP : LINE Point

INPUT PARAMETERS:
- P0(3) - REAL - STARTING POINT OF THE LINE SEGMENT
- P1(3) - REAL - ENDING POINT OF THE LINE SEGMENT
- U - REAL - VALUE OF THE PARAMETRIC VARIABLE; THE INTERPOLATION PARAMETER (e.g. if U = 0.5, the midpoint of the line will be found)

OUTPUT PARAMETERS:
- PT(3) - REAL - POINT ON THE LINE SEGMENT

LOCAL VARIABLES:
- I - INTEGER - LOOP CONTROL VARIABLE

FUNCTIONAL DESCRIPTION:
FINDS A POINT ON A LINE SEGMENT USING LINEAR INTERPOLATION

CODED BY: SHAHAB HASAN
DATE: 2/14/93 (VALENTINE'S DAY!)

DEVELOPMENT SITE:
VIRGINIA TECH CAD LAB

SUBROUTINE LINEP (P0, P1, U, PT)

REAL P0(3), P1(3), PT(3), U
INTEGER I

DO 10 I = 1,3
   PT(I) = P1(I) - (P1(I) - P0(I)) * (1.0 - U)
10 CONTINUE
RETURN
END
SEATMP : SEAT Template

INPUT PARAMETERS:
COMPN - INTEGER, COMPONENT IDENTIFIER NUMBER

OUTPUT PARAMETERS:
TITLE - CHARACTER - TITLE TO BE PUT ON TOP OF THE TEMPLATE
LINES - CHARACTER - CONTENT OF EACH LINE
NLINES - INTEGER - NO. OF LINES IN THE TEMPLATE
PLINES - INTEGER - IDENTIFIER OF EACH LINE

FUNCTIONAL DESCRIPTION:
BRING UP TEMPLATE OF THE SEAT

CODED BY: SHAHAB HASAN
DATE: 2/14/93 (VALENTINE'S DAY!)

DEVELOPMENT SITE:
VIRGINIA TECH CAD LAB

SUBROUTINE SEATMP(COMPN, TITLE, LINES, NLINES, PLINES)

INTEGER COMPN, NLINES, PLINES(*)
CHARACTER(*) TITLE, LINES(*)

INTEGER IERR
REAL P(3,19), RHO(6)
REAL HEIGHT, LENGTH, WIDTH, THICK1, THICK2, ANGLE
CHARACTER*(40) CDAT

NLINES = 17

GET SEAT POINTS
CALL SEATUP(COMPN, P, RHO)

CONVERT POINTS TO PARAMETERS
CALL SEAT12(P, HEIGHT, LENGTH, WIDTH, THICK1, THICK2, ANGLE)

SET TITLE
TITLE = 'SEAT TEMPLATE'

LINE WITH COMPONENT NAME
CALL GTCOMP(1, COMPN, CDAT, IERR)
LINES (1) = 'COMPONENT NAME:'
LINES (1) (17:35) = CDAT
PLINES(1) = 0

LINES (2) = ''
PLINES( 2) = 0

C
LINE WITH SEAT HEIGHT
LINES ( 3) = 'SEAT HEIGHT:    ###### inches'
CALL RLBNWM(HEIGHT, LINES( 3))
PLINES( 3) = 3

C
LINE WITH SEAT LENGTH
LINES ( 4) = 'SEAT LENGTH:    ###### inches'
CALL RLBNWM(LENGTH, LINES( 4))
PLINES( 4) = 4

C
LINE WITH SEAT WIDTH
LINES ( 5) = 'SEAT WIDTH:    ###### inches'
CALL RLBNWM(WIDTH, LINES( 5))
PLINES( 5) = 5

LINES ( 6) = ' '
PLINES( 6) = 0

C
LINE WITH SEAT BOTTOM THICKNESS
LINES ( 7) = 'SEAT BOTTOM THICKNESS:    ###### inches'
CALL RLBNWM(THICK1, LINES( 7))
PLINES( 7) = 7

C
LINE WITH SEAT BACK THICKNESS
LINES ( 8) = ' SEAT BACK THICKNESS:    ###### inches'
CALL RLBNWM(THICK2, LINES( 8))
PLINES( 8) = 8

C
LINE WITH SEAT BACK ANGLE
LINES ( 9) = '  SEAT BACK ANGLE:    ###### deg.'
CALL RLBNWM(ANGLE, LINES( 9))
PLINES( 9) = 9

LINES (10) = ' '
PLINES(10) = 0

LINES (11) = ' '
PLINES(11) = 0

C
FIRST CORNER
LINES (12) = ' FIRST RHO:    #---##'
CALL RLBNWM(RHO(1), LINES(12))
PLINES(12) = 12

C
SECOND CORNER
LINES (13) = 'SECOND RHO:    #---##'
CALL RLBNWM(RHO(2), LINES(13))
PLINES(13) = 13
C THIRD CORNER
  LINES (14) = 'THIRD RHO:  #---#
  CALL RLBWNM(RHO(3),LINES(14))
  PLINES(14)=14

C FOURTH CORNER
  LINES (15) = 'FOURTH RHO:    #---#
  CALL RLBWNM(RHO(4),LINES(15))
  PLINES(15)=15

C FIFTH CORNER
  LINES (16) = 'FIFTH RHO:   #---#
  CALL RLBWNM(RHO(5),LINES(16))
  PLINES(16)=16

C SIXTH CORNER
  LINES (17) = 'SIXTH RHO:  #---#
  CALL RLBWNM(RHO(6),LINES(17))
  PLINES(17)=17

RETURN
END
SUBROUTINE SEATGP(COMPN, P, RHO)

INTEGER COMPN, IERR
REAL P(3,*), RHO(*)

C GET POINTS
DO 100 I=1,19
    CALL GTCPFPM(COMPN, I, P(I,:), IERR)
100 CONTINUE

C GET RHO VALUES
CALL GTRCmp(50, 6, COMPN, RHO, IERR)

RETURN
END
C=======================================================================
C SEATS P : SEAT Store Points
C=======================================================================
C INPUT PARAMETERS:
C    COMPN - INTEGER - COMPONENT IDENTIFIER NUMBER
C    P   - REAL(3,*)  - ARRAY OF CROSS SECTION POINTS
C    RHO - REAL(*)    - ARRAY OF CORNER SHARPNESS PARAMETERS
C=======================================================================
C OUTPUT PARAMETERS:
C    NONE
C=======================================================================
C FUNCTIONAL DESCRIPTION:
C    STORE THE CROSS SECTION POINTS AND THE RHOS
C=======================================================================
C CODED BY: SHAHAB HASAN
C    DATE: 9/14/93
C=======================================================================
C DEVELOPMENT SITE:
C    VIRGINIA TECH CAD LAB
C=======================================================================

SUBROUTINE SEATSP(COMPN, P, RHO)

INTEGER COMPN, IERR
REAL P(3,*), RHO(*)

C    STORE POINTS
DO 100 I=1,19
    CALL PTCPPM(COMPN, I, P(I,:), IERR)
100 CONTINUE

C    STORE RHO VALUES
CALL PTRCMP(50, 6, COMPN, RHO, IERR)

RETURN
END
C=====================================================================================================
C  S E A T P D :  SEAT Process Data
C=====================================================================================================
C INPUT PARAMETERS:
C    COMPN - INTEGER - COMPONENT IDENTIFIER NUMBER
C    CFUNC - INTEGER - IDENTIFIER OF EACH LINE IN THE TEMPLATE
C    NEWVAL - REAL(*) - NEW VALUES ENTERED
C=====================================================================================================
C OUTPUT PARAMETERS:
C    NONE
C=====================================================================================================
C FUNCTIONAL DESCRIPTION:
C    UPDATES THE TEMPLATE BASED ON CHANGED VALUES GIVEN BY USER AND
C    PUTS BACK THE NEW VALUES
C=====================================================================================================
C CODED BY: SHAHAB HASAN
C    DATE: 2/15/93
C=====================================================================================================
C DEVELOPMENT SITE:
C    VIRGINIA TECH CAD LAB
C=====================================================================================================
SUBROUTINE SEATPD(COMPN,CFUNC,NEWVAL)

INTEGER COMPN, CFUNC
REAL NEWVAL(*)

INTEGER IERR
REAL P(3,19),RHO(6),HEIGHT,LENGTH,WIDTH,THICK1,THICK2,ANGLE

C GET SEAT POINTS
CALL SEATGP(COMPN,P,RHO)

C CONVERT POINTS TO PARAMETERS
CALL SEAT12(P,HEIGHT,LENGTH,WIDTH,THICK1,THICK2,ANGLE)

C UPDATE THE APPROPRIATE PARAMETER
IF (CFUNC .EQ. 3) THEN
   SEAT HEIGHT
   HEIGHT = NEWVAL(1)
ELSE IF (CFUNC .EQ. 4) THEN
   SEAT LENGTH
   LENGTH = NEWVAL(1)
ELSE IF (CFUNC .EQ. 5) THEN
   SEAT WIDTH
   WIDTH = NEWVAL(1)
ELSE IF (CFUNC .EQ. 7) THEN

165
C SEAT BOTTOM THICKNESS
THICK1 = NEWVAL(1)

ELSE IF (CFUNC .EQ. 8) THEN
C SEAT BACK THICKNESS
THICK2 = NEWVAL(1)

ELSE IF (CFUNC .EQ. 9) THEN
C SEAT BACK ANGLE
ANGLE = NEWVAL(1)

ELSE IF (CFUNC .EQ. 12) THEN
C RHO 1
RHO(1) = NEWVAL(1)

ELSE IF (CFUNC .EQ. 13) THEN
C RHO 2
RHO(2) = NEWVAL(1)

ELSE IF (CFUNC .EQ. 14) THEN
C RHO 3
RHO(3) = NEWVAL(1)

ELSE IF (CFUNC .EQ. 15) THEN
C RHO 4
RHO(4) = NEWVAL(1)

ELSE IF (CFUNC .EQ. 16) THEN
C RHO 5
RHO(5) = NEWVAL(1)

ELSE IF (CFUNC .EQ. 17) THEN
C RHO 6
RHO(6) = NEWVAL(1)
ENDIF

C CONVERT PARAMETERS TO POINTS
CALL SEAT21(P,HEIGHT,LENGTH,WIDTH,THICK1,THICK2,ANGLE)

C STORE SEAT POINTS
CALL SEATSP(COMPN,P,RHO)

RETURN
END
C=============================================================================
C SEATRT : SEAT Return data (???)
C=============================================================================
C INPUT PARAMETERS:
C COMPN - INTEGER - COMPONENT IDENTIFIER NUMBER
C CFUNC - INTEGER - IDENTIFIER OF EACH LINE IN THE TEMPLATE
C=============================================================================
C OUTPUT PARAMETERS:
C OLDVAL - REAL(*) - NEW VALUES ENTERED
C NO - INTEGER - NUMBER OF VALUES (!)
C=============================================================================
C FUNCTIONAL DESCRIPTION:
C RETURNS A VALUE TO BE USED FOR THE PROMPT WHEN CHANGING VALUES
C=============================================================================
C CODED BY: SHAHAB HASAN
C DATE: 5/23/93
C=============================================================================
C DEVELOPMENT SITE:
C NASA AMES RESEARCH CENTER
C=============================================================================
SUBROUTINE SEATRT(COMPN, CFUNC, NO, OLDVAL)

INTEGER COMPN, CFUNC, NO
REAL OLDVAL(*)

INTEGER IERR
REAL P(3,19),RHO(6),HEIGHT,LENGTH,WIDTH,THICK1,THICK2,ANGLE

C GET SEAT POINTS
CALL SEATGP(COMPN, P, RHO)

C CONVERT POINTS TO PARAMETERS
CALL SEAT12(P, HEIGHT, LENGTH, WIDTH, THICK1, THICK2, ANGLE)

C RETURN THE APPROPRIATE PARAMETER
    IF (CFUNC .EQ. 3) THEN
        SEAT HEIGHT
        OLDVAL(1) = HEIGHT
    ELSE IF (CFUNC .EQ. 4) THEN
        SEAT LENGTH
        OLDVAL(1) = LENGTH
    ELSE IF (CFUNC .EQ. 5) THEN
        SEAT WIDTH
        OLDVAL(1) = WIDTH
    ELSE IF (CFUNC .EQ. 7) THEN
        SEAT BOTTOM THICKNESS

167
OLDVAL(1) = THICK1

ELSE IF (CFUNC .EQ. 8) THEN
  SEAT BACK THICKNESS
  OLDVAL(1) = THICK2

ELSE IF (CFUNC .EQ. 9) THEN
  SEAT BACK ANGLE
  OLDVAL(1) = ANGLE

ELSE IF (CFUNC .EQ. 12) THEN
  RHO 1
  OLDVAL(1) = RHO(1)

ELSE IF (CFUNC .EQ. 13) THEN
  RHO 2
  OLDVAL(1) = RHO(2)

ELSE IF (CFUNC .EQ. 14) THEN
  RHO 3
  OLDVAL(1) = RHO(3)

ELSE IF (CFUNC .EQ. 15) THEN
  RHO 4
  OLDVAL(1) = RHO(4)

ELSE IF (CFUNC .EQ. 16) THEN
  RHO 5
  OLDVAL(1) = RHO(5)

ELSE IF (CFUNC .EQ. 17) THEN
  RHO 6
  OLDVAL(1) = RHO(6)
END IF

NO = 1
RETURN
END
SUBROUTINE SEAT12 (P, HEIGHT, LENGTH, WIDTH, THICK1, THICK2, ANGLE)

REAL P(3,*)
REAL HEIGHT, LENGTH, WIDTH, THICK1, THICK2, ANGLE

HEIGHT = SQRT ( (P(3,14) - P(3,17))**2 +
             (P(2,17) - P(2,14))**2 )

LENGTH = ABS( P(2,2) - P(2,17) )

WIDTH = ABS( P(1,19) - P(1,1) )

THICK1 = ABS( P(3,5) - P(3,2) )

THICK2 = SQRT ( (P(3,10) - P(3,15))**2 +
                (P(2,10) - P(2,15))**2 )

ANGLE = ATAN2 ( (P(2,8) - P(2,9)), (P(3,9) - P(3,8)) )
       = ANGLE * 180.0 / ACOS(-1.0)

POINTS ARE STORED IN UNITS OF FEET; CONVERT PARAMETERS TO INCHES
HEIGHT = HEIGHT * 12.0
LENGTH = LENGTH * 12.0
WIDTH = WIDTH * 12.0
THICK1 = THICK1 * 12.0
THICK2 = THICK2 * 12.0

RETURN
END
C==============================================================================
C SEAT 2 1: SEAT 2 (parameters) to 1 (points)
C==============================================================================
C INPUT PARAMETERS:
C HEIGHT   - REAL, SEAT HEIGHT
C LENGTH   - REAL, SEAT LENGTH
C WIDTH    - REAL, SEAT WIDTH
C THICK1   - REAL, SEAT BOTTOM THICKNESS
C THICK2   - REAL, SEAT BACK THICKNESS
C ANGLE    - REAL, SEAT BACK ANGLE
C==============================================================================
C OUTPUT PARAMETERS:
C P(I,*)  - REAL, ARRAY OF POINTS
C==============================================================================
C LOCAL VARIABLE:
C HYPOTMSE - REAL, HYPOTENUSE, USED IN TRIGONOMETRIC CALCULATIONS
C==============================================================================
C FUNCTIONAL DESCRIPTION:
C CONVERT FROM SEAT PARAMETERS TO POINTS
C*** Shahab Hasan note: This routine is necessarily a little
C*** cryptic. The reason is that the x, y, and
C*** z locations are difficult to visualize
C*** without a figure that depicts all the
C*** characteristic points of the seat, properly
C*** numbered and labeled with axes orientation.
C==============================================================================
C CODED BY: SHAHAB HASAN
C DATE: 2/15/93
C==============================================================================
C DEVELOPMENT SITE:
C VIRGINIA TECH CAD LAB
C==============================================================================
SUBROUTINE SEAT21(P,HEIGHT,LENGTH,WIDTH,THICK1,THICK2,ANGLE)

REAL P(I,*)
REAL HEIGHT, LENGTH, WIDTH, THICK1, THICK2, ANGLE
REAL HYPOTMSE

C PARAMETERS HAVE BEEN ENTERED IN INCHES; CONVERT TO FEET FOR
C POINTS STORAGE.
C
HEIGHT = HEIGHT / 12.0
LENGTH  = LENGTH / 12.0
WIDTH   = WIDTH / 12.0
THICK1  = THICK1 / 12.0
THICK2  = THICK2 / 12.0

C SEAT BACK ANGLE HAS BEEN ENTERED IN DEGREES; CONVERT TO
C RADIANS FOR CALCULATIONS
ANGLE = ANGLE * ACOS(-1.0) / 180.0
X LOCATIONS OF ALL THE POINTS

THE X LOCATION FOR THE FIRST EIGHTEEN POINTS IS 0.0

DO 10 I = 1, 18
   P(1,I) = 0.0
10 CONTINUE

THE X LOCATION FOR THE LAST (NINETEENTH) POINT IS THE WIDTH
   P(1,19) = WIDTH

POINT 1
   P(2,1) = 0.5*LENGTH - THICK1/3.0
   P(3,1) = 0.0

POINT 2
   P(2,2) = 0.5*LENGTH
   P(3,2) = 0.0

POINT 3
   P(2,3) = 0.5*LENGTH
   P(3,3) = THICK1/3.0

POINT 4
   P(2,4) = 0.5*LENGTH
   P(3,4) = THICK1 + 2.0/3.0

POINT 5
   P(2,5) = 0.5*LENGTH
   P(3,5) = THICK1

POINT 6
   P(2,6) = 0.5*LENGTH - THICK1/3.0
   P(3,6) = THICK1

POINT 7
   P(2,7) = -0.5*LENGTH + THICK2 + THICK2/3.0
   P(3,7) = THICK1

POINT 8
   P(2,8) = -0.5*LENGTH + THICK2
   P(3,8) = THICK1

POINT 9
   HYPTNSE = THICK2/3.0
   P(2,9) = -0.5*LENGTH + THICK2 - SIN(ANGLE)*HYPTNSE
   P(3,9) = THICK1 + COS(ANGLE)*HYPTNSE

POINT 10
   HYPTNSE = HEIGHT
   P(2,10) = -0.5*LENGTH - SIN(ANGLE)*HYPTNSE
   P(3,10) = COS(ANGLE)*HYPTNSE
   HYPTNSE = THICK2
   P(2,10) = P(2,10) + COS(ANGLE)*HYPTNSE
\[ P(3,10) = P(3,10) + \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ \text{HYPTNSE} = \text{THICK2}/3.0 \]
\[ P(2,10) = P(2,10) + \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,10) = P(3,10) - \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]

C POINT 11
\[ \text{HYPTNSE} = \text{HEIGHT} \]
\[ P(2,11) = -0.5 \cdot \text{LENGTH} - \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,11) = \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ \text{HYPTNSE} = \text{THICK2} \]
\[ P(2,11) = P(2,11) + \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,11) = P(3,11) + \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]

C POINT 12
\[ \text{HYPTNSE} = \text{HEIGHT} \]
\[ P(2,12) = -0.5 \cdot \text{LENGTH} - \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,12) = \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ \text{HYPTNSE} = \text{THICK2} - \text{THICK2}/3.0 \]
\[ P(2,12) = P(2,12) + \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,12) = P(3,12) + \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]

C POINT 13
\[ \text{HYPTNSE} = \text{HEIGHT} \]
\[ P(2,13) = -0.5 \cdot \text{LENGTH} - \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,13) = \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ \text{HYPTNSE} = \text{THICK2}/3.0 \]
\[ P(2,13) = P(2,13) + \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,13) = P(3,13) + \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]

C POINT 14
\[ \text{HYPTNSE} = \text{HEIGHT} \]
\[ P(2,14) = -0.5 \cdot \text{LENGTH} - \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,14) = \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]

C POINT 15
\[ \text{HYPTNSE} = \text{HEIGHT} - \text{THICK2}/3.0 \]
\[ P(2,15) = -0.5 \cdot \text{LENGTH} - \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,15) = \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]

C POINT 16
\[ \text{HYPTNSE} = \text{THICK2}/3.0 \]
\[ P(2,16) = -0.5 \cdot \text{LENGTH} - \sin(\text{ANGLE}) \cdot \text{HYPTNSE} \]
\[ P(3,16) = \cos(\text{ANGLE}) \cdot \text{HYPTNSE} \]

C POINT 17
\[ P(2,17) = -0.5 \cdot \text{LENGTH} \]
\[ P(3,17) = 0.0 \]

C POINT 18
\[ P(2,18) = -0.5 \cdot \text{LENGTH} + \text{THICK2}/3.0 \]
\[ P(3,18) = 0.0 \]
C POINT 19
P(2,19) = 0.5*LENGTH - THICK1/3.0
P(3,19) = 0.0

RETURN

END
Vita

Shahab Hasan was born in Washington, D.C. on November 5, 1969. He has lived in Fairfax County, Virginia for most of his life but has traveled extensively throughout the United States and the rest of the world. Shahab chose to attend Virginia Tech for both his undergraduate and graduate studies based on the combination of the school's outstanding reputation in the engineering field, proximity to home, and beautiful surroundings. Shahab is a devout follower of professional football (especially the Pittsburgh Steelers) and also enjoys other sports, music, television and movies.

Shahab Hasan